Roy Ernest
18 March 2015

72

77118 views

10

Roy Ernest 18 March 2015

77118 views 72 10

Introduction to SQL Server Spatial Data

> More and more applications require the handling of geospatial data. It is easy to store spatial data, but it takes rather more thought to retrieve and manipulate it. Tasks like searching neighborhoods, and calculating distances between points is often required from databases. But how do you start? Roy and Surenda take you through the basics.

With the introduction of so many handheld devices that support features such as GPS and maps, the need to store spatial data in a relational database is greater than ever. Database systems such as IBM, DB2, and Oracle have supported spatial data for some time. Microsoft added support in SQL Server 2008, with the introduction of native spatial data types to represent spatial objects. At the same time, Microsoft added the functionality necessary to access and index spatial data, provide cost-based optimizations, and support operations such as the intersection of two spatial objects.

What is spatial data?

Spatial data, also known as geospatial data or geographic information, is data that identifies the geographic location of features and boundaries on Earth. We

usually use spatial data to store coordinates, topology, or other data that can be mapped.

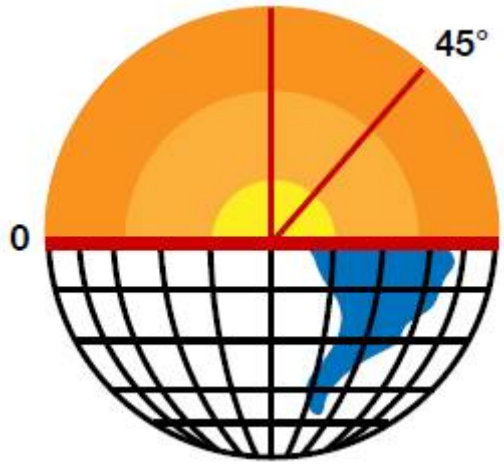
Because Earth is a sphere, we use latitude and longitude coordinates to define a location on its surface. Coordinates are measured in degrees and represent angular distances calculated from the earth's center. This figure provides an overview of how we obtain our coordinates in relationship to the earth's center:

What is latitude coordinate?

It is common knowledge that earth spins on an axis. The North and South poles lie at the ends of that axis. The Equator is an imaginary line on the earth's surface that circles the earth, bisecting the distance between the poles, as shown in the following figure:

Because the Equator lies at midpoint between the poles, it is considered to be at 0 degrees, which makes it the starting point for measuring latitude. Latitude values indicate the angular distance between the Equator and points north or south on the sphere.

A *line of latitude* is an imaginary line connecting all the points with same latitude value. The lines of latitude represent values in whole degrees. Because the lines are parallel to the Equator, they're also referred to as *parallels*. All parallels are equally spaced to each other.



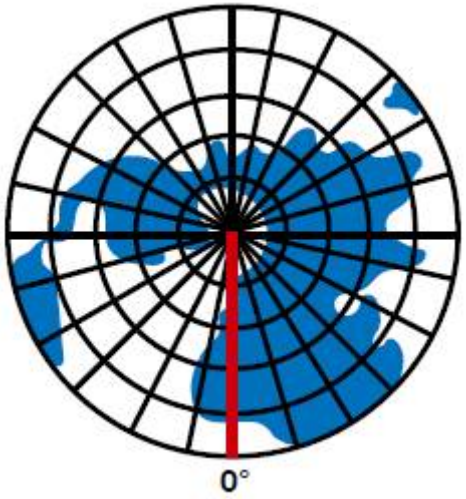
There are 90 degrees of latitude between the Equator and the North Pole. As a result, the North Pole is considered to be at *90 degrees north latitude*. Latitudes between the Equator and North Pole are marked accordingly. For example, the latitude halfway between the Equator and the North Pole is at 45 degrees, so it's considered to be at *45 degrees north latitude*, as shown in the following figure.

The South Pole works the same way as the North Pole. It is 90 degrees to the south of the Equator, or *90 degrees south latitude*.

When the directional designators are omitted, northern latitudes are given positive values and southern latitudes are given negative values.

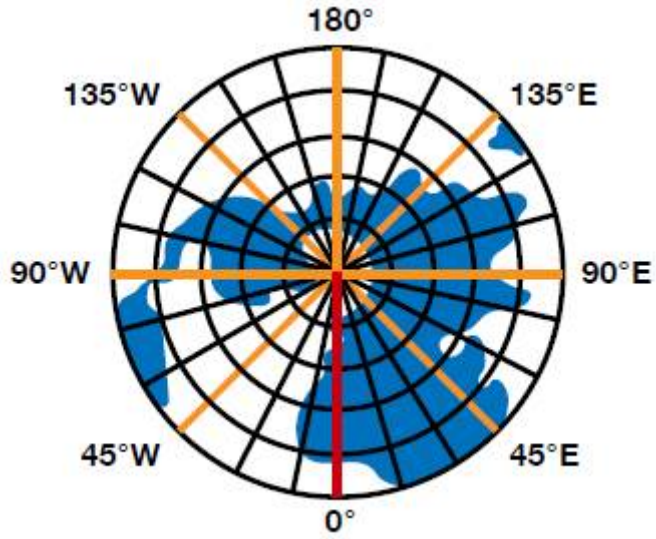
What is longitude coordinate?

Longitude is a geographic coordinate that specifies the east-west position on the surface of Earth. The *lines of longitude*, also called *meridians*, run perpendicular (at right angles, or 90 degrees) to the lines of latitude and all pass through both poles. Each line of longitude is part of a great circle that wraps around the circumference of the earth. The degrees assigned to longitudes are different from what are used for latitudes.



The meridian that runs through Greenwich, England is considered to be 0 degrees of longitude. For this reason, it's referred to as the *prime meridian*. Longitude values indicate the angular distance between the prime meridian and points east or west of it on the surface of earth. This figure shows the prime meridian, as seen from the North Pole.

The Earth is divided equally into 360 degrees of longitude, with 180 degrees of longitude to the east of the prime meridian (+180) and 180 degrees of longitude to the west of the prime meridian (-180). Rather than being preceded with positive or negative signs, the lines of longitude are often designated in terms of their direction, as in *Longitude 45 degrees E* or *Longitude 45 degrees W,* for east and west, respectively.



The 180-degree longitude line is opposite the prime meridian and is the same whether traveling east or west. This figure to the right shows the main meridians, as seen from the North Pole.

To sum up how degrees are assigned to latitudes and longitudes, latitude values measured from the equator range from -90 degrees at the South Pole to +90 degrees at the North Pole, and longitude values are measured from the prime meridian and range from -180 degrees to +180 degrees.

Spatial Data Types

Spatial data describes the physical location and shape of geometric objects. These objects can be point locations or complex objects such as countries, roads, or lakes. SQL Server's spatial data types, which are based on latitude and longitude coordinates, let you store these objects and make them available to calling applications.

SQL Server supports two spatial data types:

- **`Geometry`:** Stores data based on a flat (Euclidean) coordinate system. The data type is often used to store the X and Y coordinates that represent lines, points, and polygons in two-dimensional spaces.
- **`Geography`:** Stores data based on a round-earth coordinate system. The data type is used to store the latitude and longitude coordinates that represent lines, polygons and points.

Each of the spatial data types has its own use. For example, the **`Geography`** type is often used to store an application's GPS data, while the **`Geometry`** type is often used to map a three-dimensional object, such as a building.

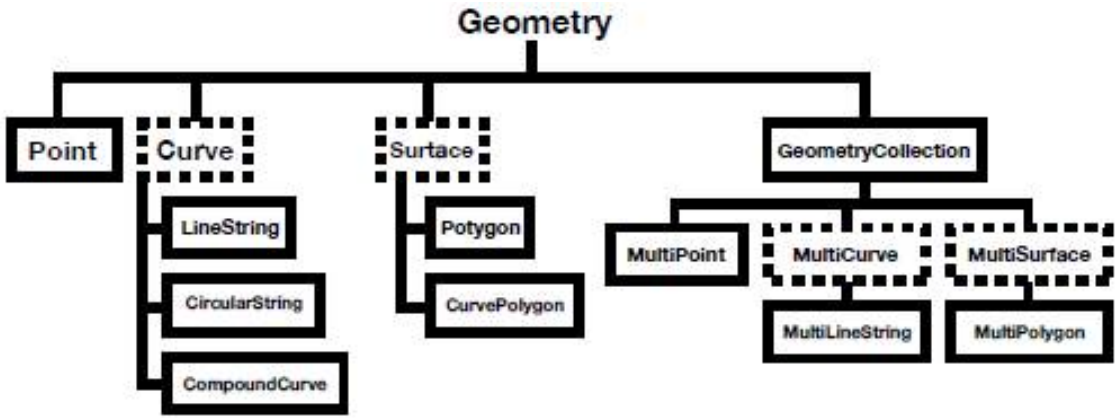
Together, the two spatial data types support 16 types of data objects. Of the 16, 11 are instantiable (can be represented) in a database. They derive certain properties from their parent data type, and it's these properties that distinguish them as objects such as points or linestrings.

Ten of the instantiable data objects are available to both the **`Geometry`** and **`Geography`** data types. The objects include Point, MultiPoint, LineString, CircularString, `MultiLineString` , CompoundCurve, Polygon, CurvePolygon, `MultiPolygon`, and GeometryCollection. The eleventh object type, FullGlobe, is instantiable only for the **`Geography`** data type.

The object types supported by the spatial data types can be divided in two groups:

- **Single geometries:** Single Geometries are types that can be stored in the database only in one way.
- **Geometry collection:** A geometry collection, as the name implies, is a collection of types of data objects.

The object types associated with a spatial data type form a relationship with each other. For example, the following diagram illustrates how the object types of the `Geometry` data type are related:



Although the figure is specific to the `Geometry` data type, it also applies to the `Geography` data type. For each data type, there are six single geometries and four collection geometries. The single geometries are as follows:

- Point
- LineString
- CircularString
- CompoundCurve

- Polygon
- CurvePolygon

Here are the collection geometries:

- MultiPoint
- `MultiLineString`
- `MultiPolygon`
- GeometryCollection

As mentioned above, the **`Geography`** data type also supports the **`FullGlobe`** data object. Now let's look at each object type in detail.

Point

A Point is the most important object type supported by the spatial data types. A Point represents a singular position in space. We can define the position by using an X-coordinate and Y-coordinate value-pair based on a planar coordinate system or on the latitude and longitude coordinates from a geographic coordinate system.

For example, suppose we want to specify a Point in the **`Geometry`** data type with the coordinates X = 5 and Y = 3. To define the Point, we can use the well-known text (WKT) format: **`POINT (5 3)`**. (WKT is a text markup language used to represent vector geometry objects.) The WKT format starts with the **`POINT`** keyword, followed by the coordinate values, contained within parentheses and separated by a space.

To define a Point in the **Geography** data type, we take a different approach because we're working with latitude and longitude coordinates. We first specify the longitude and then the latitude. For example, if we use the WKT format to locate a Point at a latitude of 30 degrees and longitude of 50 degrees, we write it as: **POINT (30 50)**.

The WKT syntax also lets us specify additional Z- and M-coordinate values so we can position a Point in four-dimensional space. The Z coordinate refers to the height or elevation of a Point, and the M coordinate represents the measure value, which is a user-defined value. For example, the slope of a line is a number that measures its steepness. This number is denoted by the letter *m*.

When referencing a four-dimensional Point in a **Geometry** data type, we can include the Z and M coordinates along with the X and Y coordinates, as shown in the following syntax:

```
POINT (x y z m)
```

For a **Geography** data type, we take a slightly different approach:

```
POINT (longitude latitude z m)
```

When working with points, keep in mind several characteristics:

- A Point is zero-dimensional; there is no length in any direction and no area contained within the point.
- A Point has no boundary.

- A Point is always classified as a *simple* geometry.

Let's look at several examples to better understand how to work with spatial data types and Points. In the first example, we create a **POINT** instance for the **Geometry** data type that uses the coordinates X = 4 and Y = 5:

```
Declare @g geometry;
SET @g= geometry::STGeomFromText('POINT(4 5)',0);
```

When we set the value of the variable, we use the **STGeomFromText** method available to the **Geometry** data type. The method's first argument is the Point's WKT representation. The second argument is the default spatial reference identifier (SRID). The SRID value corresponds to a spatial reference system based on the ellipsoid used for round-earth mapping or flat-earth mapping.

The next example also creates a Point instance, again for the **Geometry** type. As before, we use the coordinates X = 4 and Y = 5, as well as Z = 6 (elevation) and M = 3.5 (measure):

```
DECLARE @g geometry;
SET @g= geometry::Parse('POINT(4 5 6 3.5)');
```

Notice that this time we're using the **Parse** method. The method is equivalent to the **STGeomFromText** method, but it assumes that the value of SRID is 0.

After we set our value, we can retrieve the individual coordinates (X, Y, Z, and M) by calling the variable properties associated with those coordinates, as shown in the following **SELECT** statements:

```
SELECT@g.STX; -- Retrieves the X-coordinate property of a
SELECT@g.STY; -- Retrieves the Y-coordinate property of a
SELECT@g.Z; -- Retrieves the Z value (Elevation) of a Poir
SELECT@g.M; -- Retrieves the M value (measure) of a Point
```

Notice that the properties associated with the X and Y coordinates are preceded with ST, but the Z and M properties stand alone.

When using the Parse method you can also specify the Z and M values as **NULL** values, as shown in the following example:

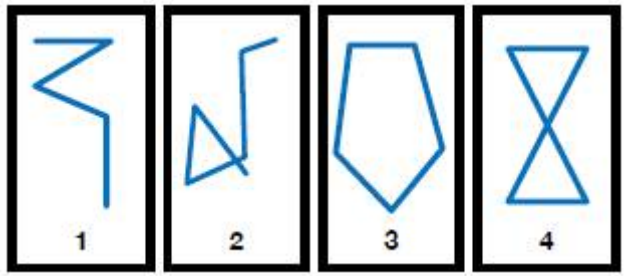
```
DECLARE@g geometry;
SET@g = geometry::Parse('POINT(4 5 NULL NULL)');
```

The **SET** statement in this example is equivalent to the one in the following example:

```
DECLARE@g geometry;
SET @g = geometry::STGeomFromText('POINT(4 5)',0);
```

LineString

After defining individual Points, we can create a series of two or more Points and connect one to the next to form a path. This path is a *LineString,* which you can use to represent entities such as roads and rivers.



In SQL Server, a LineString is a one-dimensional object that represents a sequence of points and the line segments connecting them. When using the WKT format, you specify each pair of coordinates, with a space between the coordinates themselves and a comma between the pairs, as in `LINESTRING (2 3, 4 6, 6 6, 10 4)`. This particular linestring will create a simple, unclosed linestring. (More on what "unclosed" means in a moment.)

Not surprisingly, you can create a wide range of LineStrings, such as the four shown here:

The LineStrings in the figures can be described as follows:

- Figure 1 is a simple, unclosed LineString.
- Figure 2 is a complex, unclosed LineString.
- Figure 3 is a simple, closed LineString and therefore is a ring.
- Figure 4 is a complex, closed LineString and therefore is not a ring.

To better understand the figures, it can help to be aware of several LineStrings characteristics:

- A simple LineString is one in which the path does not cross itself.

- A closed LineString is one that starts and ends at the same point.
- A LineString that is both simple and closed is known as a ring.
- A LineString's boundary consists of the two points that lie at the line's beginning and end.

In some cases, you can add a **LINESTRING** instance to a **Geometry** variable, but the instance might not be valid. For a **LINESTRING** instance to be valid, it must meet the following criteria:

- The **LINESTRING** instance must be accepted. This means that the instance must be formed from at least two points or it must be empty.
- If a **LINESTRING** instance is not empty, it must contain at least two distinct points.
- The **LINESTRING** instance cannot overlap itself over an interval of two or more consecutive points. For instance the starting point and the ending point cannot be the same. Linestrings with two identical points are invalid.

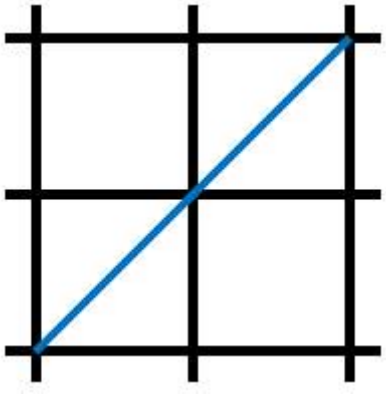
The following **LINESTRING** instance is valid:

```
DECLARE@g1 geometry='LINESTRING EMPTY';
```

The variable returns an empty grid. However, the next example populates the grid with a straight line:

```
DECLARE@g1 geometry='LINESTRING (1 1, 3 3)';
```

The following figure shows the grid now returned by the variable:

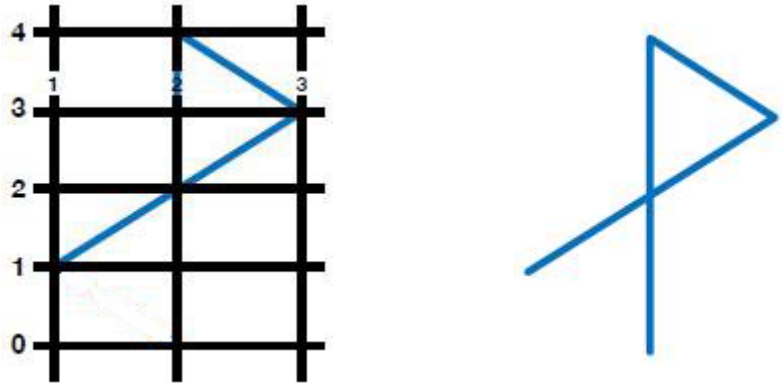


When you query spatial data in SQL Server Management Studio (SSMS), the results include the option to view the geometry on a grid, similar to what's shown in the figure. This can be useful in verifying the geometry definition.

Let's move on to the next example, in which we create a complex LineString:

```
DECLARE@g3 geometry='LINESTRING(1 1, 3 3, 2 4, 2 0)';
```

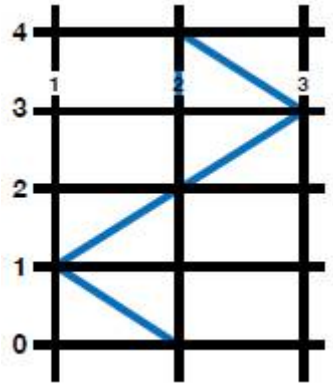
The following figures show the LineString now returned by the variable, the first figure with the grid lines, the second figure without:



Our last example adds one more segment to the previous LineString:

```
DECLARE@g4 geometry='LINESTRING (1 1, 3 3, 2 4, 2 0, 1 1)'
```

As the following figure shows, we've now closed our LineString.



CircularString

The CircularString object type is a new type of geometry introduced in SQL Server 2012. A CircularString is defined by a path segment that connects a series of ordered Points to form an arc (a segment of a circle). The arc is

defined by three points on a two-dimensional plane: the start point, end point, and an anchor point that lies between the other two. The first point cannot be the same as the third point, and the three points cannot be collinear (lie on the same line). If they are collinear, the path is treated as a line segment.

When using the WKT format to define a CircularString, you must specify the three points that make up the arc, as in `CIRCULARSTRING(1 3, 4 1, 9 4)`.

A valid CircularString must have an odd number of points and have at least three points, or it must be empty. Several characteristics define a CircularString object:

- A simple CircularString is one in which the path that connects the points does not cross itself. For example, an arc is a simple CircularString.
- A closed CircularString is one that starts and end at the same point. For example, a circle is a closed CircularString. For a closed CircularString, there must be at least five points.
- The boundary of a CircularString consists of the start and end points, except in the case of a closed CircularString, which has no boundary.
- Every CircularString must be defined by an odd number of points greater than one.

Now let's look at a few examples that demonstrate how to define CircularStrings. The first example creates an empty CircularString:

```
DECLARE @g geometry;
SET @g = geometry::Parse('CIRCULARSTRING EMPTY');
Select @G
```

All we've done here is pass in the argument **CIRCULARSTRING EMPTY** to the **Parse** function. As to be expected, if we run this statement, there will be no graphic for us to view. The next example, however, creates a simple CircularString shaped like an arc:

```
DECLARE@g geometry;
SET@g = geometry::STGeomFromText('CIRCULARSTRING(2 0, 1 1,
select@g
```

This time around we get our arc, as shown in the following figure:

Now let us look at another example of a simple CircularString:

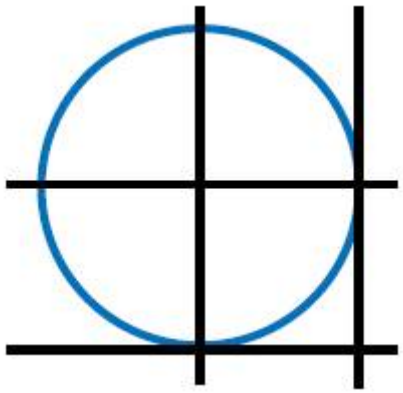
```
DECLARE@g2 geometry ='CIRCULARSTRING (1 1, 2 0, -1 1)';
select@g2
```

The following figure shows the results from our **SELECT** statement:

In the next example, we create a closed CircularString:

```
DECLARE @g geometry;
SET @g = geometry::Parse('CIRCULARSTRING(2 1, 1 2, 0 1, 1
Select @g
```

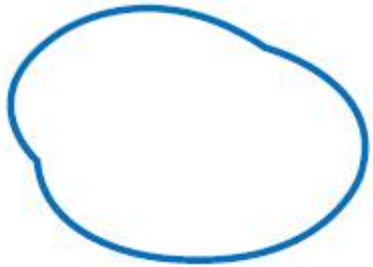
Notice that we've passed in five points to the **CIRCULARSTRING** object. As a result, a full circle is returned, as shown in the following figure:



The next example defines a CircularString on the **Geography** data type, using latitude and longitude coordinates:

```
DECLARE@g geography ='CIRCULARSTRING(-122.358 47.653, -122
```

As the following figure shows, this time our closed circle has a much more irregular shape:



CompoundCurve

A CompoundCurve is a single continues path that connects a set of Points. The segments joining each pair of Points can be linear (LineString) or curved (CircularString). SQL Server started supporting CompoundCurves in SQL Server 2012.

When using the WKT format to define a CompoundCurve object, you begin with the `COMPOUNDCURVE` keyword, followed by a set of parentheses. Within the parentheses, you specify the LineString and CircularString coordinates that form the CompoundCurve. Each LineString and CircularString segment must begin at the point where the previous segment ended so the CompoundCurve defines a single continuous path. In addition, you must precede the coordinates with the `CIRCULARSTRING` keyword if they're defining a CircularString.

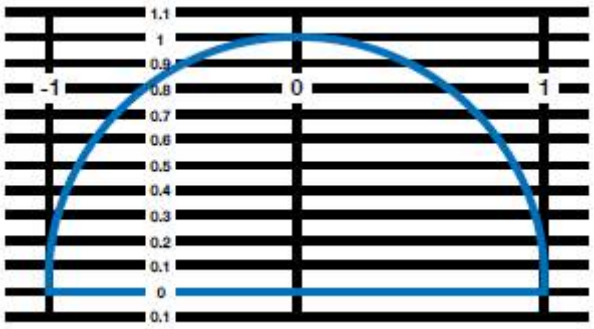
The specifics of how to define a CompoundCurve will be easier to understand after we look at a few examples. The following example shows how to create an empty CompoundCurve.

```
DECLARE@g1 geometry ='COMPOUNDCURVE EMPTY';
select@g1
```

Now let us take a look at how to create a CompoundCurve object that is a semicircle:

```
DECLARE@g2 geometry ='COMPOUNDCURVE(CIRCULARSTRING(1 0, 0
select@g2
```

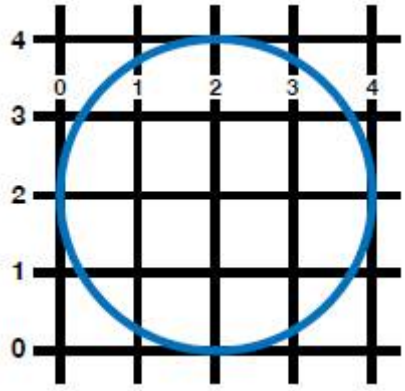
In the above example we are creating an arc with the CircularString(1 0, 0 1, -1 0) object and a line using (-1 0, 2 0). Note that, when creating a CompoundCurve that uses a CircularString and LineString, we do not need to use the keyword **`LINESTRING`**. The example creates a semicircle with a line closing if off at the bottom, as shown in the following figure:



Now let's look at how to instantiate an instance of a **`Geometry`** CompoundCurve with multiple CircularStrings:

```
DECLARE@g geometry;
SET@g = geometry::Parse('COMPOUNDCURVE(CIRCULARSTRING(0 2,
SELECT@g
```

In the above example, the CircularString objects create two arcs that start and end at the same points with the same radius. These two arcs together make a perfect circle:



Polygon

A Polygon is a two-dimensional geometry that describes an enclosed shape by defining the object's outer ring, which forms the Polygon's perimeter. The outer ring is based on a set of coordinates that start and end at the same Point and in the process enclose the space within the perimeter. Each segment of the Polygon's outer ring is a LineString. Together the segments form the outer ring that defines the perimeter. A Polygon can also contain inner rings that define spaces within the Polygon.

All Polygons share the following characteristics:

- A Polygon is constructed from series of one or more rings, which are made up of simple and closed LineStrings. As a result, all polygons are simple, closed geometries.

- Polygons are two-dimensional geometries with an associated length and area.

To define a Polygon using the WKT format, you begin with the **`POLYGON`** keyword, followed by the ring coordinates, enclosed in parentheses. You must use one set of parentheses to include all the coordinates and one or more inner sets of parentheses to enclose the individual ring coordinates. In addition, you should list the outer ring first, enclosed in parentheses, followed by any inner rings, enclosed in their own sets of parentheses. You should also separate each set of ring coordinates with a comma. (All this will become clearer as we work through the examples below.)

A Polygon instance must be accepted in order to be stored in a **`Geometry`** or **`Geography`** object. Otherwise, the database engine will throw an exception. SQL Server will accept the following Polygon instances:

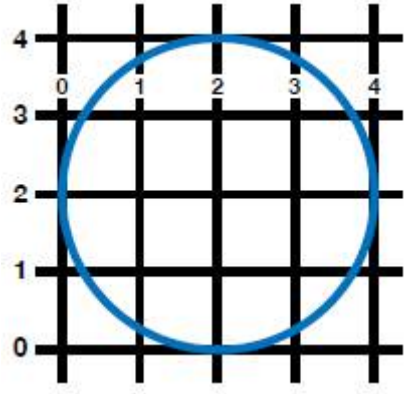
- An empty Polygon instance
- A Polygon instance with an acceptable exterior ring and zero or more acceptable interior rings.

Let us say that we want to find all the ATMs around a two kilometer radius of Punda(a city where I reside). For this we will need to put a boundary around the city. This can be represented as a Polygon and select all points (ATMs) that intersect this boundary.

Now let's look at a few examples. The following code demonstrates the WKT syntax for a rectangular Polygon:

```
DECLARE@g4 geometry ='POLYGON((1 1, 3 1, 3 7, 1 7, 1 1))';
select@g4;
```

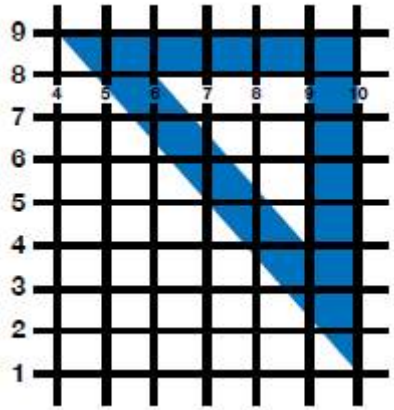
As you can see, this simple polygon starts and ends at the same point. The following figure shows the shape of the returned object:



The next example defines a triangular Polygon that contains an interior ring:

```
DECLARE@g5 geometry ='POLYGON((10 1, 10 9, 4 9, 10 1),(9 4
select@g5;
```

As you can see, the WKT specifies two rings: an external ring that is defined by the first set of coordinates (10 1, 10 9, 4 9, 10 1), and an internal ring that is defined by the other set (9 4, 9 8, 6 8, 9 4). Both these rings start and end at the same points, as shown in the following figure.



Now let's create a square polygon that contains a circle:

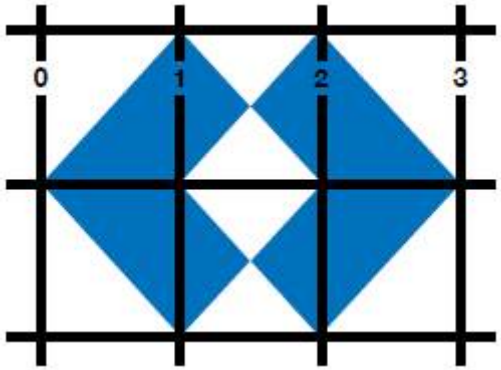
```
DECLARE@g2 geometry ='POLYGON((-20 -20, -20 20, 20 20, 20
select@g2
```

In this example, we create two simple polygons. The first set of values (-20 -20, -20 20, 20 20, 20 -20, -20 -20) creates the square and the second set (10 0, 0 10, 0 -10, 10 0) creates the triangle within square, as shown in the figure to the left:

Let us look at an example of a `MultiPolygon`, which is a collection of zero or more polygon instances. In the following example, we create two separate polygons that start and end at the same point:

```
DECLARE@g1 geometry ='MultiPolygon(((2 0, 3 1, 2 2, 1.5 1.
select@g1
```

Here's what the results look like this time around:



CurvePolygon

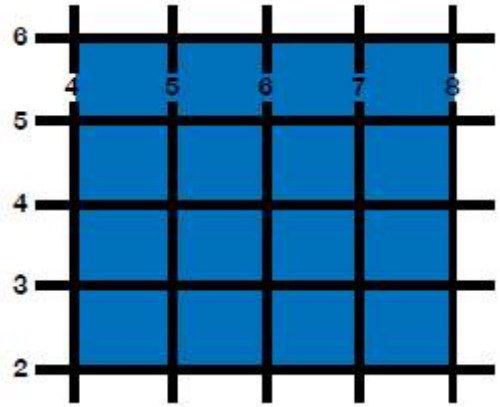
Like the Polygon, the CurvePolygon object is also a two-dimensional geometry. It is defined by one exterior ring and one or more interior rings. Each ring in a CurvePolygon can be any type of simple, closed geometry, including LineString, CircularString, or CompoundCurve. As a result, a CurvePolygon is itself a simple, closed geometry.

When using the WKT format to define a CurvePolygon, we follow the same general syntax used for a Polygon. Because the CurvePolygon allows rings to be defined as LineStrings, CircularStrings, or CompoundCurves, we must specify what type of curve is used for each ring. The LineString is the default curve type, so these rings do not need to be explicitly preceded by the **`LINESTRING`** keyword.

Let's move on to the examples to better understand how to define CurvePolygons. The first example defines a CurvePolygon with a linear ring made up of five Points:

```
DECLARE@g1 geometry ='CURVEPOLYGON ((4 2, 8 2, 8 6, 4 6, 4
select@g1
```

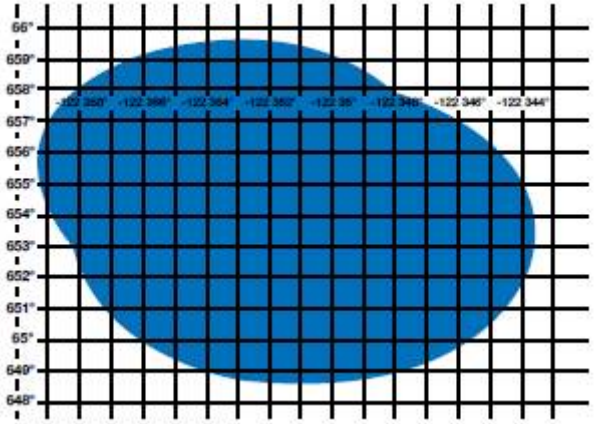
As the following figure shows, this time we end up with a square:



In the next example we will look at a CurvePolygon initialized with geography instance:

```
DECLARE @g geography
set @g = 'CURVEPOLYGON(CIRCULARSTRING(-122.358 47.653, -12
Select @g
```

In this case, we're initializing a **geography** instance with a CurvePolygon that is defined with a CircularString, giving us the object shown in the following figure:



MultiPolygon

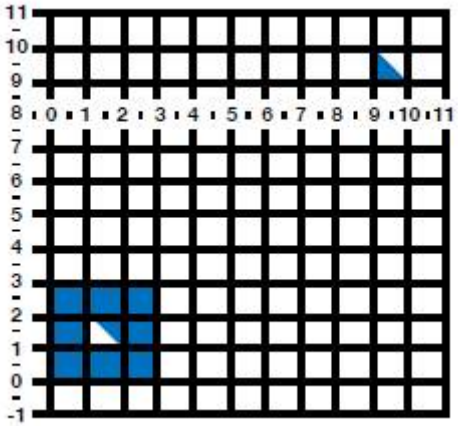
Until now we have been working with single geometries. Now let us look at collection geometries. The first of these is the `MultiPolygon`.

A `MultiPolygon`instance can be defined as a collection of zero or more polygons. A `MultiPolygon`allows us to define multiple distinct areas of space as part of the same element. For example countries like Russia and Germany can be represented as a Polygon. If we want to define a country like New Zealand, we would want to include the north and south islands as a single element. To do so, we can use a `MultiPolygon`.

Let's take a look at couple of examples of `MultiPolygon`s. The first is an example of a `MultiPolygon`with three Polygon instances:

```
Declare@g geometry;
SET@g = geometry::Parse('MultiPolygon(((0 0, 0 3, 3 3, 3 (
select@g
```

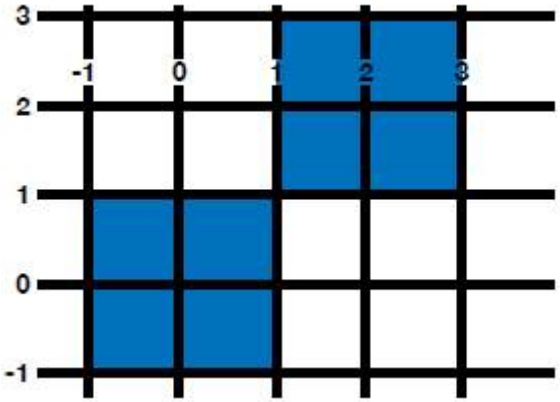
Each Polygon is defined with its own set of parentheses. For example, the second polygon is defined by the coordinates (1 1, 1 2, 2 1, 1 1). The following figure shows the final `MultiPolygon`shapes:



In the next example, we initiate a `MultiPolygon`object that contains two Polygon instances:

```
DECLARE@g2 geometry ='MultiPolygon(((1 1, 1 -1, -1 -1, -1
select@g2
```

Again, each set of Polygon coordinates are enclosed in their own parentheses, but are both part of the `MultiPolygon`instantiation, which is shown in the following figure:

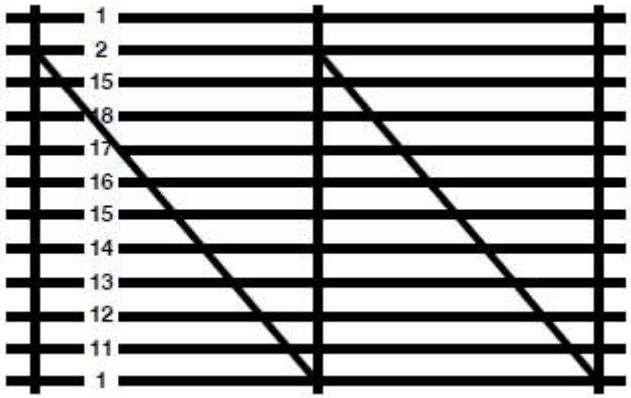


MultiLineString

A `MultiLineString` is a collection of zero or more LineString instances. A `MultiLineString` allows us to define multiple distinct LineStrings as part of a single element, as shown in the following example:

```
DECLARE @g geometry;
SET @g = geometry::Parse('MultiLineString ((0 2, 1 1), (2
Select @g
```

The above code snippet contains two distinct LineString elements defined as a single `MultiLineString` element, which gives us two parallel lines, as shown in the figure below:



MultiPoint

A multipoint is a collection of several Point geometries. A MultiPoint allows us to define two or more Point elements as one single MultiPoint object. The following example shows how to create the MultiPoint object:

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('MULTIPOINT((21 2), (12
Select @g
```

In this case, we have a MultiPoint that contains three Points, .

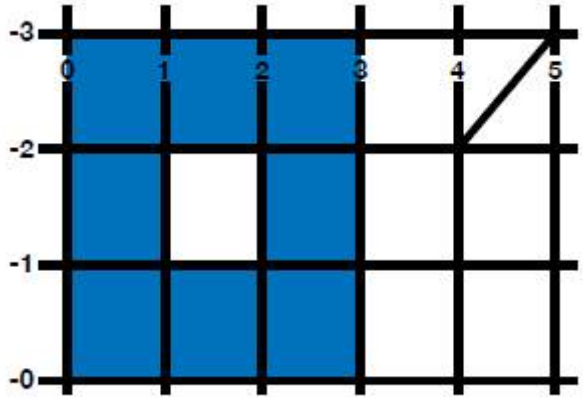
GeometryCollection

A GeometryCollection, as the name implies, is a collection of zero or more **`geometry`** or **`geography`** instances. We can also initialize a GeometryCollection as empty. A GeometryCollection can contain a combination of Point, LineString and Polygon objects.

Let us look at how we can initialize a GeometryCollection instance using Point, LineString and Polygon objects:

```
DECLARE @g geometry;
SET @g = 'GEOMETRYCOLLECTION (POINT (4 0), LINESTRING (4 2
Select @g
```

As you can see in the following figure, our GeometryCollection contains all three object types:



Conclusion

With so many handheld devices supporting applications that include GPS and mapping functionality, geographic data often needs to be stored in a relational database. Microsoft added support for this type of data in SQL Server 2008 with the introduction of the **`Geometry`** and **`Geography`** spatial data types. At the same time, Microsoft made sure that the spatial data could be easily retrieved and indexed to better support the new data types. In this article, we discussed various aspects of geographic data, provided an overview of the

spatial data types, and worked through a number of examples that demonstrated how to use the WKT format to define different types of geometries.