Volume : 5 | Issue : 10 | October 2016 • ISSN No 2277 - 8179 | IF : 3.508 | IC Value : 69.48

Original Research Paper

A Study on Spatial Database for Sql Server 2008 With Entity Framework 5.0

Computer Science

KEYWORDS : Spatial Data Objects, Framework, SharpMap, SRID, SSMS, TSQL.

Rama Krishna Thotapally	Senior Programmer, Centre on Geo-Informatics Application in Rural Development, NIRD&PR.
DSR Murthy	Senior Asst. Professor, Centre on Geo-Informatics Application in Rural Development, NIRD&PR
T Phanindra Kumar	Asst. Professor, Centre on Geo-Informatics Application in Rural Development, NIRD&PR.

ABSTRACT

While spatial features have been in SQL Server for a while using those features inside of .NET applications hasn't been as straight forward as could be, because .NET natively doesn't support spatial types. There are workarounds for this with a few custom projects like SharpMap or a hack using the Sql Server specific Geo types found in the Microsoft.SqlTypes assembly that ships with SQL server. These approaches work for manipulating spatial data from .NET code, they didn't work with database access if you're using Entity Framework. Other ORM vendors have been rolling their own versions of spatial integration. In Entity Framework 5.0 running on .NET 4.5 the Microsoft ORM finally adds support for spatial types as well.

INTRODUCTION

Before we look at how things work with Entity framework, let's take a look at how SQL Server allows you to use spatial data to get an understanding of the underlying semantics. The following SQL examples should work with SQL 2008 and forward. One of the best ways to learn is to play with the product. I've found that the Intellisense support in Visual Studio[1] (VS) is much more mature than that provided in SQL Server Management Studio (SSMS). This is especially true for the SQL Spatial Objects, where the TSQL people don't get offered a drop down list of all the valid methods (spelt & cased correctly) nor does it offer the parameters, their data types etc. So I'll describe how a DBA (non-Programmer) can still take advantage of VS to write TSQL queries.

In this paper I'll describe basic geography features that deal with single location and distance calculations which are probably the most common usage scenario.

CONTENT OVERVIEW AND DEFINITIONS

Before we begin I would like to make it clear that learning all the capabilities and limitations of SQL Server and Spatial SQL.

What is Microsoft Sql Server?

Microsoft SQL Server is a relational database system that stores data and provides several mechanisms to retrieve stored data. One of the beauties of relational database systems is that they allow you to effectively store a variety of data sets and they are both scalable and securable, and in my experience, much more trustworthy than any flat file.

What is Entity Framework (EF)?

Entity Framework is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write.

Entity framework is an Object/Relational Mapping (O/RM) framework. It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing & storing the data in the database.

Entity framework is useful in three scenarios. First, if you already have existing database or you want to design your database ahead of other parts of the application. Second, you want to focus on your domain classes and then create the database from your domain classes. Third, you want to design your database schema on the visual designer and then create the database and classes.

The following figure-1 illustrates the above scenarios.

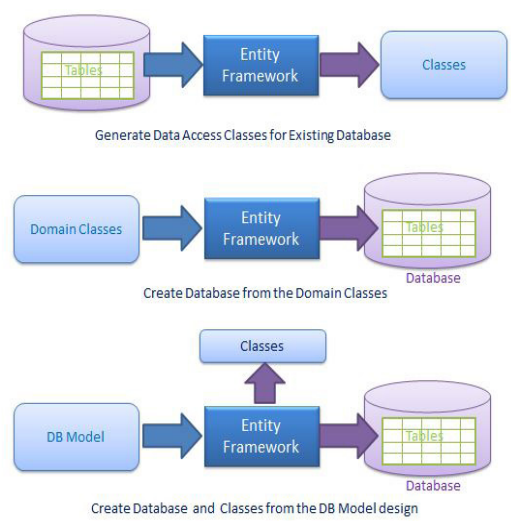


Figure-1: Entity Framework

What is O/RM?

ORM is a tool for storing data from domain objects to relational database like MS SQL Server, in an automated way, without much programming. O/RM includes three main parts: Domain class objects, Relational database objects and Mapping information on how domain objects map to relational database objects (tables, views & stored procedures). ORM allows us to keep our database design separate from our domain class design. This makes the application maintainable and extendable. It also automates standard CRUD operation (Create, Read, Update & Delete) so that the developer doesn't need to write it manually.

A typical ORM tool generates classes for the database interaction for your application as shown below.

Volume : 5 | Issue : 10 | October 2016 • ISSN No 2277 - 8179 | IF : 3.508 | IC Value : 69.48

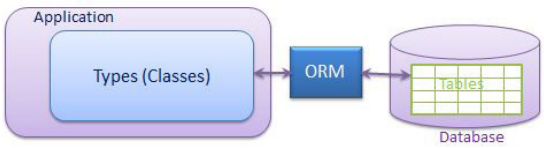


Figure-2: Object Relational Mapping

Entity Framework 5.0 API was distributed in two places, in NuGet package and in .NET framework. The .NET framework 4.0/4.5 included EF core API, whereas EntityFramework.dll via NuGet package included EF 5.0 specific features.

Figure-3: Entity Framework Environment

FUNCTIONAL SCENERIO

Spatial Data Types: There are two types of spatial data[2]. The **geometry** data type supports planar, or Euclidean (flat-earth), data. The **geometry** data type both conforms to the Open Geospatial Consortium (OGC) Simple Features for SQL Specification version 1.1.0 and is compliant with SQL MM (ISO standard).

In addition, SQL Server supports the **geography** data type, which stores ellipsoidal (round-earth) data, such as GPS latitude and longitude coordinates.

Spatial Data Objects: The geometry and geography data types support sixteen spatial data objects, or instance types. However, only eleven of these instance types are instantiable; you can create and work with these instances (or instantiate them) in a database. These instances derive certain properties from their parent data types that distinguish them as Points, LineStrings, CircularStrings, CompoundCurves, Polygons, CurvePolygons or as multiple geometry or geography instances in a Geometry Collection. Geography type has an additional instance type, FullGlobe.

The figure below depicts the geometry hierarchy upon which the geometry and geography data types are based. The instantiable types of geometry and geography are indicated in blue.

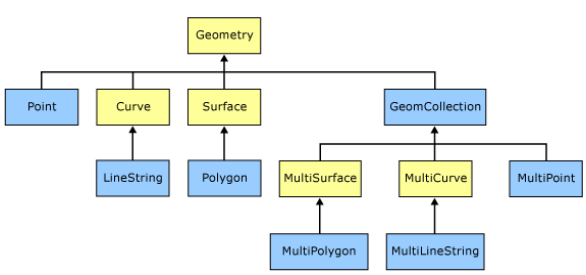


Figure-4: Geometry Hierarchy

The subtypes for geometry and geography types are divided into simple and collection types[3]. Some methods like `STNumCurves()` work only with simple types.

Simple types include:
Point
LineString
CircularString
CompoundCurve
Polygon
CurvePolygon

Collection types include:
MultiPoint
MultiLineString
MultiPolygon
GeometryCollection

I. Point is a 0-dimensional object representing a single location and may contain Z (elevation) and M (measure) values.

Example:

```
DECLARE @g geometry;

SET @g = geometry::STGeomFromText('POINT (3 4)', 0);
```

II. LineString is a one-dimensional object representing a sequence of points and the line segments connecting them.

Example(s):

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('LINESTRING(1 1, 2 4, 3 9)', 0);
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('LINESTRING(1 1 NULL 0, 2 4 NULL 12.3, 3 9 NULL 24.5)', 0);
```

Transact-SQL:

```
DECLARE @g geometry
SET @g = geometry::STGeomFromText('LINESTRING(1 3, 1 3)',0);
IF @g.STIsValid() = 1
BEGIN
SELECT @g.ToString() + ' is a valid LineString.';
END
ELSE
BEGIN
SELECT @g.ToString() + ' is not a valid LineString.';
SET @g = @g.MakeValid();
SELECT @g.ToString() + ' is a valid Point.';
END
```

The above Transact-SQL code snippet will return the following:

LINESTRING(1 3, 1 3) is not a valid LineString

POINT(1 3) is a valid Point.

III. CircularString is a collection of zero or more continuous circular arc segments. A circular arc segment is a curved segment defined by three points in a two-dimensional plane; the first point cannot be the same as the third point. If all three points of a circular arc segment are collinear, the arc segment is treated as a line segment.

Example(s):

```
DECLARE @g1 geometry = 'CIRCULARSTRING EMPTY';
DECLARE @g2 geometry = 'CIRCULARSTRING(1 1, 2 0, -1 1)';
```

```
DECLARE @g3 geometry = 'CIRCULARSTRING(1 1, 2 0, 2 0, 1 1, 0 1)';
DECLARE @g4 geometry = 'CIRCULARSTRING(1 1, 2 2, 2 2)';
SELECT @g1.STIsValid(), @g2.STIsValid(), @g3.STIsValid(),@ g4.STIsValid();
```

***Transact-SQL**: Instantiating a Geometry Instance with an Empty CircularString*

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

***Transact-SQL:** Instantiating a Geometry Instance Using a CircularString with One Circular Arc Segment*

```
DECLARE @g geometry;
SET @g = geometry:: STGeomFromText('CIRCULARSTRI NG(2 0, 1 1, 0 0)', 0);
SELECT @g.ToString();
```

***Transact-SQL:** Instantiating a Geometry Instance Using a CircularString with Multiple Circular Arc Segments*

```
DECLARE @g geometry;
SET @g = geometry::Parse('CIRCULARSTRING(2 1, 1 2, 0 1, 1 0, 2 1)');
SELECT 'Circumference = ' + CAST(@g.STLength() AS NVARCHAR(10));
```

IV. CompoundCurve is a collection of zero or more continuous **CircularString** or **LineString** instances of either geometry or geography types.

CompoundCurve instance is accepted if it is an empty instance or meets the following criteria.

All the instances contained by CompoundCurve instance are accepted circular arc segment instances. For more information on accepted circular arc segment instances, see LineString and CircularString.

All of the circular arc segments in the CompoundCurve instance are connected. The first point for each succeeding circular arc segment is the same as the last point on the preceeding circular arc segment.

None of the contained instances are empty instances.

Example(s):

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

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

Transact-SQL:

```
DECLARE @g1 geometry;
DECLARE @g2 geometry;
SET @g1 = geometry::Parse('COMPOUNDCURVE(CIRCUL ARSTRING(0 2, 2 0, 4 2), (4 2, 2 4, 0 2))');
SELECT 'Circle One', @g1.STLength() AS Perimeter;
SET @g2 = geometry::Parse('COMPOUNDCURVE(CIRCU LARSTRING(0 2, 2 0, 4 2), CIRCULARSTRING(4 2, 2 4, 0 2))');
SELECT 'Circle Two', @g2.STLength() AS Perimeter;
```

Output:

```
Circle One11.940039…
Circle Two12.566370…
```

V. Polygon is a two-dimensional surface stored as a sequence of points defining an exterior bounding ring and zero or more interior rings.

Example(s):

```
DECLARE @g1 geometry = 'POLYGON EMPTY';
DECLARE @g2 geometry = 'POLYGON((1 1, 3 3, 3 1, 1 1))';
DECLARE @g3 geometry = 'POLYGON((-5 -5, -5 5, 5 5, 5 -5, -5 -5),(0 0, 3 0, 3 3, 0 3, 0 0))';
DECLARE @g4 geometry = 'POLYGON((-5 -5, -5 5, 5 5, 5 -5, -5 -5),(3 0, 6 0, 6 3, 3 3, 3 0))';
DECLARE @g5 geometry = 'POLYGON((1 1, 1 1, 1 1, 1 1))';
```

Transact-SQL:

```
DECLARE @g geometry
SET @g = geometry::Parse('POLYGON((1 3, 1 3, 1 3, 1 3))');
SET @g = @g.MakeValid();
SELECT @g.ToString()
```

The following criteria define attributes of a CurvePolygon instance:

- The boundary of the **CurvePolygon** instance is defined by the exterior ring and all interior rings.
- The interior of the **CurvePolygon** instance is the space between the exterior ring and all of the interior rings.

VI. CurvePolygon instance differs from a **Polygon** instance in that a **CurvePolygon** instance may contain the following circular arc segments: **CircularString** and **CompoundCurve**.

Example(s):

```
DECLARE @g1 geometry = 'CURVEPOLYGON EMPTY';
DECLARE @g2 geometry = 'CURVEPOLYGON((0 0, 0 0, 0 0, 0 0))';
DECLARE @g3 geometry = 'CURVEPOLYGON((0 0 1, 0 0 2, 0 0 3, 0 0 3))'
DECLARE @g4 geometry = 'CURVEPOLYGON(CIRCULARSTRING(1 3, 3 5, 4 7, 7 3, 1 3))';
DECLARE @g5 geography = 'CURVEPOLYGON((-122.3 47, 122.3 -47, 125.7 -49, 121 -38, -122.3 47))';
```

Transact-SQL:

```
DECLARE @g geometry;
SET @g = geometry::Parse('CURVEPOLYGON(CIRCULARS TRING(0 4, 4 0, 8 4, 4 8, 0 4), CIRCULARSTRING(2 4, 4 2, 6 4, 4 6, 2 4))');
SELECT @g.STArea() AS Area;
```

VII. MultiPoint is a collection of zero or more points. The boundary of a **MultiPoint** instance is empty.

The following example creates a `geometry MultiPoint` instance with SRID 23 and two points: one point with the coordinates (2, 3), one point with the coordinates (7, 8), and a Z value of 9.5.

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('MULTIPOINT((2 3), (7 8 9.5))', 23);
```

This `MultiPoint` instance can also be expressed using `STMPointFromText()` as shown below.

```
DECLARE @g geometry;

SET @g = geometry::STMPointFromText('MULTIPOINT((2 3), (7 8 9.5))', 23);
```

The following example uses the method `STGeometryN()` to retrieve a description of the first point in the collection.

```
SELECT @g.STGeometryN(1).STAsText();
```

Volume : 5 | Issue : 10 | October 2016 • ISSN No 2277 - 8179 | IF : 3.508 | IC Value : 69.48

VIII. MultiLineString is a collection of zero or more geometry or geographyLineString instances.
The following example creates a simple `geometry``MultiLineString` instance containing two `LineString` elements with the SRID 0.

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

To instantiate this instance with a different SRID, use `STGeomFromText()` or `STMLineStringFromText()`. You can also use `Parse()` and then modify the SRID, as shown in the following example.

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

SET @g.STSrid = 13;

IX. MultiPolygon instance is a collection of zero or more **Polygon** instances.

The following example shows the creation of a `geometry``MultiPolygon` instance and returns the Well-Known Text (WKT) of the second component.

DECLARE @g geometry;
SET @g = geometry::Parse('MULTIPOLYGON(((0 0, 0 3, 3 3, 3 0, 0 0), (1 1, 1 2, 2 1, 1 1)), ((9 9, 9 10, 10 9, 9 9)))');
SELECT @g.STGeometryN(2).STAsText();
This example instantiates an empty `MultiPolygon` instance.

DECLARE @g geometry;
SET @g = geometry::Parse('MULTIPOLYGON EMPTY');

X. GeometryCollection is a collection of zero or more **geometry** or **geography** instances. A **GeometryCollection** can be empty.

The following example instantiates a `geometry``GeometryCollection` with Z values in SRID 1 containing a `Point` instance and a `Polygon` instance.

DECLARE @g geometry;
SET @g = geometry::STGeomCollFromText('GEOMETRYCOLLECTION(POINT(3 3 1), POLYGON((0 0 2, 1 10 3, 1 0 4, 0 0 2)))', 1);

IMPLEMENTATION PROCESS
Let's start by creating a test table that includes a Geography field and also a pair of Long/Lat fields that demonstrate how you can work with the geography functions even if you don't have geography/geometry fields in the database.

Here the CREATE command:
```
Create Database SpatialDB;
Use SpatialDB;
CREATE TABLE [dbo].[Geo](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [Location] [geography] NULL,
    [Long] [float] NOT NULL,
    [Lat] [float] NOT NULL
    )
```

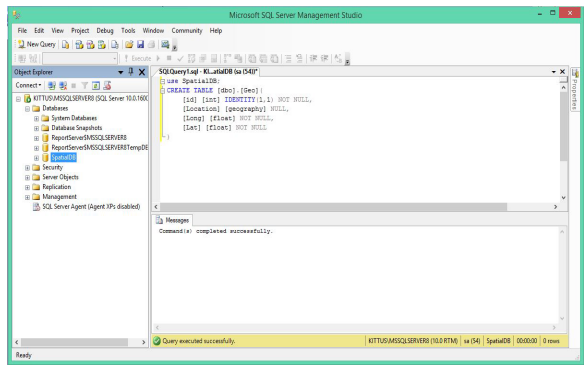


Figure-5: SpatialDB Creation

Now using plain SQL you can insert data into the table using geography::STGeoFromText SQL CLR function:

insert into Geo(Location , long, lat) values (geography::STGeomFromText ('POINT(-121.527200 45.712113)', 4326), -121.527200, 45.712113)

insert into Geo(Location , long, lat) values (geography::STGeomFromText ('POINT(-121.517265 45.714240)', 4326), -121.517265, 45.714240)

insert into Geo(Location , long, lat) values (geography::STGeomFromText ('POINT(-121.511536 45.714825)', 4326), -121.511536, 45.714825)

The STGeomFromText function [4] accepts a string that points to a geometric item (a point here but can also be a line or path or polygon and many others). You also need to provide an SRID (Spatial Reference System Identifier) which is an integer value that determines the rules for how geography/geometry values are calculated and returned. For mapping/distance functionality you typically want to use 4326 as this is the format used by most mapping software and geo-location libraries like Google.

The spatial data in the Location field is stored in binary format which looks something like this:

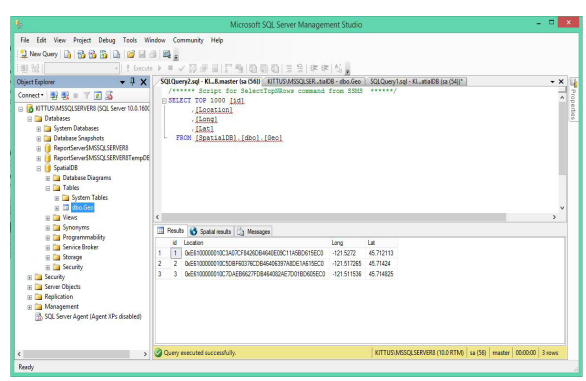


Figure-6: SpatialData

Once the location data is in the database you can query the data and do simple distance computations very easily. For example to calculate the distance of each of the values in the database to another spatial point is very easy to calculate.

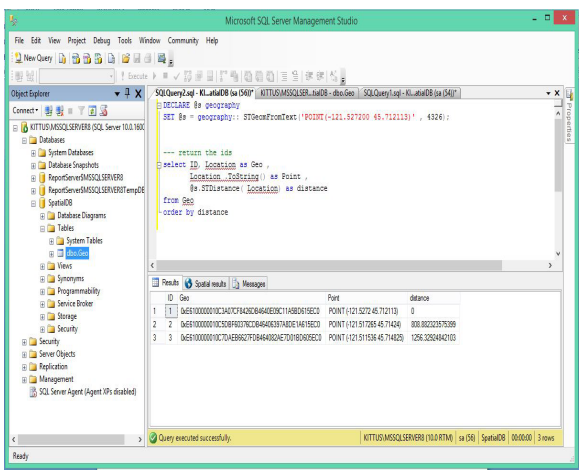
Distance calculations compare two points in space using a direct line calculation. For our example I'll compare a new point to all the points in the database.

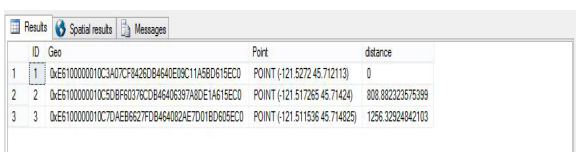
Using the Location field the SQL looks like this:

```
DECLARE @s geography
SET @s = geography:: STGeomFromText('POI
NT(-121.527200 45.712113)' , 4326);
select ID, Location as Geo ,
    Location .ToString() as Point ,
    @s.STDistance( Location) as distance
from Geo
order by distance
```

The code defines a new point which is the base point to compare each of the values to. You can also compare values from the database directly, but typically you'll want to match a location to another location and determine the difference for which you can use the geography::STDistance function.

This query produces the following output:





	ID	Geo	Point	distance
1	1	0xE6100000010C3A07CF8426D8464E09C11A5BD615EC0	POINT (-121.5272 45.712113)	0
2	2	0xE6100000010C5DBF60376CD8464063978ADE1A615EC0	POINT (-121.517265 45.71424)	808.882323575399
3	3	0xE6100000010C7DAEB627FD8464082AE7D01BD605EC0	POINT (-121.511536 45.714825)	1256.32924842103

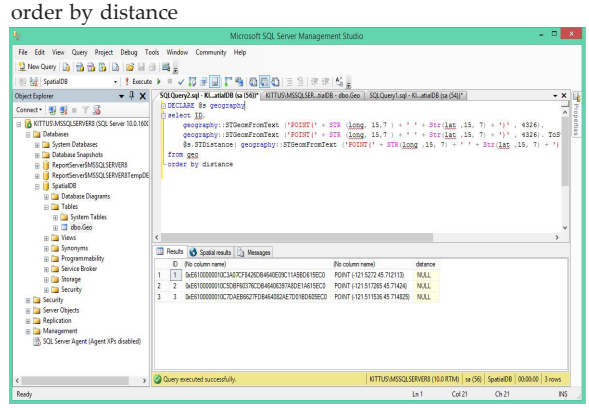
Figure-7: Spatial Type Distance

The STDistance function returns the straight line distance between the passed in point and the point in the database field. The result for SRID 4326 is always in meters. Notice that the first value passed was the same point so the difference is 0. The other two points are two points here in town in Hood River a little ways away - 808 and 1256 meters respectively.

Notice also that you can order the result by the resulting distance, which effectively gives you results that are ordered radially out from closer to further away. This is great for searches of points of interest near a central location (YOU typically!).

These geolocation functions are also available to you if you don't use the Geography/Geometry types, but plain float values. It's a little more work, as each point has to be created in the query using the string syntax, but the following code doesn't use a geography field but produces the same result as the previous query.

```
select ID,
geography::STGeomFromText ('POINT(' + STR (long, 15,7 )
+ ' ' + Str(lat ,15, 7) + ')' , 4326),
    geography::STGeomFromText ('POINT(' + STR (long,
15,7 ) + ' ' + Str(lat ,15, 7) + ')' , 4326). ToString(),
    @s.STDistance( geography::STGeomFromText ('POINT('
+ STR(long ,15, 7) + ' ' + Str(lat ,15, 7) + ')' , 4326)) as dis-
tance
from geo
order by distance
```



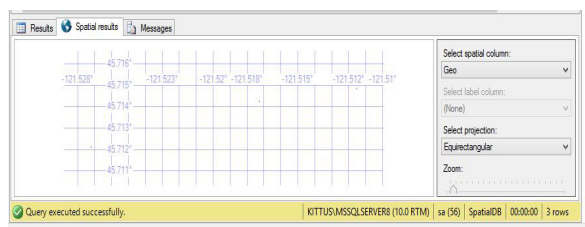


Figure: Spatial Map as Spatial Results

Note that DbGeography and DbGeometry[5] are exclusive to Entity Framework 5.0 (not 4.4 which ships in the same NuGet package or installer) and **requires** .NET 4.5. That's because the new DbGeometry and DbGeography (and related) types are defined in the 4.5 version of System.Data. Entity which is a CLR assembly and is only updated by major versions of .NET. Why this decision was made to add these types to System.Data.Entity rather than to the frequently updated EntityFramework assembly that would have possibly made this work in .NET 4.0 is beyond me, especially given that there are no native .NET framework spatial types to begin with.

I find it also odd that there is no native CLR spatial type[6]. The DbGeography and DbGeometry types are specific to Entity Framework and live on those assemblies. They will also work for general purpose, non-database spatial data manipulation, but then you are forced into having a dependency on System.Data.Entity, which seems a bit silly. There's also a System.Spatial assembly that's apparently part of WCF Data Services which in turn don't work with Entity framework. Another example of multiple teams at Microsoft not communicating and implementing the same functionality (differently) in several different places. Perplexed as a I may be, for EF specific code the Entity framework specific types are easy to use and work well.

CONCLUSIONS

Therefore, from this present experimental study, it was concluded that using minimum functions of SQL Server 2008 to create and implementation of Spatial Data. These queries we can embed with any object originated programming languages like java, C# we can do miracles with less code. With this knowledge we can develop mobile based apps as well as website with all the available dynamic data. In any project, for the developing countries

Volume : 5 | Issue : 10 | October 2016 • ISSN No 2277 - 8179 | IF : 3.508 | IC Value : 69.48

like India, cost is the primary consideration for the adaptability of spatial technology. Under these circumstances, free and open-source geospatial tools provide all the data storage, analysis and information visualization for free. This paper provides a foundation in web-based mapping using open-source geospatial tools and web technology. It highlights building a prototype mapping application by providing step-by-step instructions which can encourage the eager developers and interested readers to publish their maps on the web with no prior technical experience in map servers.

REFERENCES:

[1] W. M. Lee, "C#. NET Web Developers Guide," Syngress, Rockland, 2002.

[2] P. Corti, "Thinking in GIS," 2006. http://www.paolocorti.net/2006/09/20/mapserver-tutorialfor-c-mapscript-asp-net/.

[3] https://blogs.msdn.microsoft.com.

[4] https://weblog.west-wind.com.

[5] A. Mansourian, A. Fasihi and M. Taleai, "A Web-Based Spatial Decision Support System to Enhance Public Par- ticipation in Urban Planning Processes," Journal of Spa- tial Science, Vol. 56, No. 2, 2001, pp. 269-282. http://dx.doi.org/10.1080/14498596.2011.623347

[6] R. Mari, L. Bottai, C. Busillo, F. Calastrini, B. Gozzini and G. Gualtieri, "A GIS-Based Interactive Web Decision Support System for Planning Wind Farms in Tuscany (It- aly)," Renewable Energy, Vol. 36, No. 2, 2011, pp. 754- 763. http://dx.doi.org/10.1016/j.renene.2010.07.005