Where's Waldo? MySQL and Geodata

Mike Benshoof
March 18, 2015
Webinar

What is Geodata?

- Encompasses a wide array of topics
- Revolves around geo-positioning data (latitude/longitude)
  - Point data (single lat/lon)
  - Bounded areas (think radius from point)
  - Defined area (think city limits outline on map)
- Often includes some function of distance
  - Distance between points
  - All points within x, y

Why do we care?

- Here are some commonly asked questions based around geodata:
  - What are the 5 closest BBQ restaurants to my hotel?
  - How far is it from here to the airport?
  - How many restaurants are there within 25 miles of my hotel?
- These are all fairly common questions – especially with the prevalence of geo-enabled devices (anyone here ever enable “Location Services”?)

Other Industries

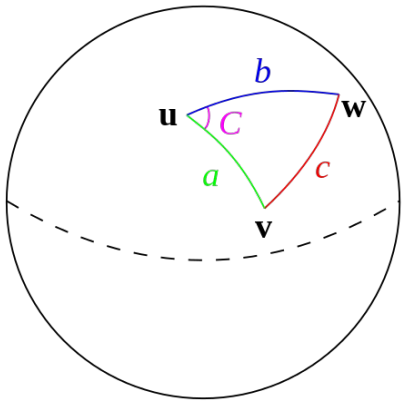
- Oil/gas exploration
- Meteorology
- Logistics companies
- < INSERT YOUR INDUSTRY HERE >

- The point: geodata is so readily available, you are likely already using it or will be soon!

High Level theory and formulas...

… everyone's favorite

- Distance between points on sphere (The Haversine Formula)



- Ok, everyone get out your calculators and slide rules...

$$\operatorname{haversin}\left(\frac{d}{r}\right) = \operatorname{haversin}(\phi_2 - \phi_1) + \cos(\phi_1)\cos(\phi_2)\operatorname{haversin}(\lambda_2 - \lambda_1)$$

$$\operatorname{haversin}(\theta) = \sin^2\left(\frac{\theta}{2}\right) = \frac{1-\cos(\theta)}{2}$$

en MySQL por favor...

- MySQL prior to 5.6
  - Get out your calculators
- MySQL 5.6
  - Introduced st_distance (built-in)

Enough with the theory already!

```
SET @src_lat = 37; SET @src_lng = -133;

SET @dest_lat = 38; SET @dest_lng = -133;

SELECT (3959 * acos(cos(radians(@src_lat)) *
cos(radians(@dest_lat)) * cos(radians(@dest_lng) -
radians(@src_lng)) + sin(radians(@src_lat)) *
sin( radians(@dest_lat)))) as GreatCircleDistance

+----------------------+
| GreatCircleDistance |
+----------------------+
|   69.09758508647379 |
+----------------------+
```

- Huzzah!! The distance between two latitude lines is ~69 miles!

… lets put it all together

Find me all zip codes within 25 miles of my current zip code...

- Table Structure (pre 5.6)

```
CREATE TABLE `zipcodes_indexed` (
  `ID` int(11) NOT NULL AUTO_INCREMENT,
  `ZIP` varchar(255) DEFAULT NULL,
  `Latitude` decimal(10,8) DEFAULT NULL,
  `Longitude` decimal(11,8) DEFAULT NULL,
  `City` varchar(255) DEFAULT NULL,
  `State` varchar(255) DEFAULT NULL,
  `County` varchar(255) DEFAULT NULL,
  `Type` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`ID`),
  KEY `lat` (`Latitude`,`Longitude`),
  KEY `city` (`City`,`State`),
  KEY `state` (`State`)
) ENGINE=InnoDB
```

… lets put it all together

- Helper Function (pre 5.6)

```
DELIMITER $$

DROP FUNCTION IF EXISTS DistanceInMiles$$
CREATE FUNCTION DistanceInMiles (src_lat decimal(10,8), src_lng
decimal(11,8), dest_lat decimal(10,8), dest_lng decimal(11,8)) RETURNS
decimal(15,8) DETERMINISTIC
BEGIN
RETURN CAST((3959 * acos(cos(radians(src_lat)) * cos(radians(dest_lat)) *
cos(radians(dest_lng) - radians(src_lng)) + sin( radians(src_lat)) * sin(
radians(dest_lat)))) as decimal(15,8));
END $$

DELIMITER ;
```

… and results!

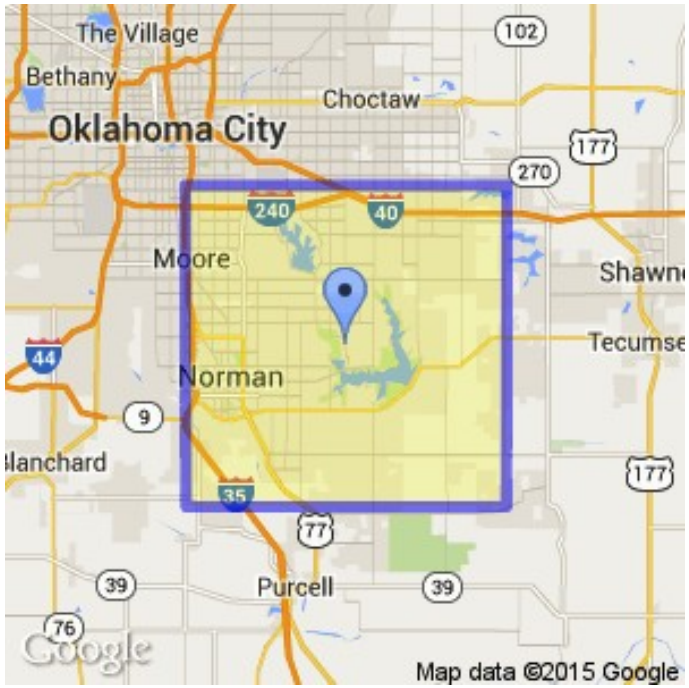
```
mysql> # Norman, OK Post Office (73071)
mysql> SET @srcLat = 35.254049;
Query OK, 0 rows affected (0.00 sec)

mysql> SET @srcLng = -97.300313;
Query OK, 0 rows affected (0.00 sec)

mysql> SET @dist = 25;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT z.ZIP, z.City, z.State,
    -> DistanceInMiles(@srcLat, @srcLng, z.Latitude, z.Longitude) as distance
    -> FROM zipcodes_indexed z
    -> HAVING distance < @dist
    -> ORDER BY distance
    -> LIMIT 10;
+-------+---------------+-------+-------------+
| ZIP   | City          | State | distance    |
+-------+---------------+-------+-------------+
| 73071 | NORMAN        | OK    |  0.00000000 |
| 73026 | NORMAN        | OK    |  1.45586169 |
| 73072 | NORMAN        | OK    |  4.30645795 |
| 73165 | OKLAHOMA CITY | OK    |  5.84183161 |
| 73070 | NORMAN        | OK    |  7.15379176 |
| 73068 | NOBLE         | OK    |  7.15998471 |
| 73160 | OKLAHOMA CITY | OK    |  7.83641939 |
| 73069 | NORMAN        | OK    |  7.91904816 |
| 73019 | NORMAN        | OK    |  8.72433716 |
| 73150 | OKLAHOMA CITY | OK    | 10.62626848 |
+-------+---------------+-------+-------------+
10 rows in set (0.79 sec)
```

Not so great...

```
mysql> EXPLAIN SELECT z.ZIP, z.City, z.State,
    -> DistanceInMiles(@srcLat, @srcLng, z.Latitude, z.Longitude) as distance
    -> FROM zipcodes_indexed z
    -> HAVING distance < @dist
    -> ORDER BY distance
    -> LIMIT 10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: z
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 42894
     filtered: 100.00
        Extra: Using temporary; Using filesort
1 row in set, 1 warning (0.00 sec)
```

More math to the rescue!

- Rather than scan the whole table, lets just look at a small rectangle of data (i.e. a bounding box):



1° Latitude ~= 69 miles
1° Longitude ~= cos(lat) * 69

$lat_range = radius / 69
$lng_range = abs(radius / (cos(lat) * 69))

$lon1 = $mylng + $lng_range
$lon2 = $mylng - $lng_range
$lat1 = $mylat +$lat_range
$lat2 = $mylat - $lat_range

… and respectable!

```
mysql> SELECT z.ZIP, z.City, z.State,
    -> DistanceInMiles(@srcLat, @srcLng, z.Latitude, z.Longitude) as distance
    -> FROM zipcodes_indexed z
    -> WHERE z.Longitude BETWEEN -97.744004 AND -96.856621
    -> AND z.Latitude BETWEEN 34.891730 AND 35.616367
    -> HAVING distance < @dist
    -> ORDER BY distance
    -> LIMIT 10;
+-------+---------------+-------+-------------+
| ZIP   | City          | State | distance    |
+-------+---------------+-------+-------------+
| 73071 | NORMAN        | OK    |  0.00000000 |
| 73026 | NORMAN        | OK    |  1.45586169 |
| 73072 | NORMAN        | OK    |  4.30645795 |
| 73165 | OKLAHOMA CITY | OK    |  5.84183161 |
| 73070 | NORMAN        | OK    |  7.15379176 |
| 73068 | NOBLE         | OK    |  7.15998471 |
| 73160 | OKLAHOMA CITY | OK    |  7.83641939 |
| 73069 | NORMAN        | OK    |  7.91904816 |
| 73019 | NORMAN        | OK    |  8.72433716 |
| 73150 | OKLAHOMA CITY | OK    | 10.62626848 |
+-------+---------------+-------+-------------+
10 rows in set (0.00 sec)
```

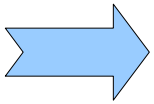
Onwards and upwards to Spatial Data types!

Converting our table to geospatial...

- Non-Spatial way:
  - Lat / Lon as individual **decimal** columns
  - Composite B-Tree index covering each column
- Spatial way:
  - Use the **geometry** type with a POINT object (single lat/lon)
  - Use the **geometry** type with a POLYGON object (defined border)
  - Use a SPATIAL index on the geometry column (MyISAM only through 5.6, added to InnoDB in 5.7)

And voila!

```
CREATE TABLE `zipcodes` (
  `ID` int(11) NOT NULL AUTO_INCREMENT,
  `ZIP` varchar(255) DEFAULT NULL,
  `Latitude` decimal(10,8) DEFAULT NULL,
  `Longitude` decimal(11,8) DEFAULT NULL,
  `City` varchar(255) DEFAULT NULL,
  `State` varchar(255) DEFAULT NULL,
  `County` varchar(255) DEFAULT NULL,
  `Type` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`ID`),
  KEY `lat` (`Latitude`,`Longitude`),
  KEY `city` (`City`,`State`),
  KEY `state` (`State`),
  KEY `ZIP` (`ZIP`),
  KEY `County` (`County`)
) ENGINE=InnoDB
```



```
CREATE TABLE `zipcodes` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `zip` varchar(255) DEFAULT NULL,
  `geo` geometry NOT NULL,
  `city` varchar(255) DEFAULT NULL,
  `state` varchar(255) DEFAULT NULL,
  `county` varchar(255) DEFAULT NULL,
  `type` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`),
  SPATIAL KEY `geo` (`geo`),
  KEY `city` (`city`,`state`),
  KEY `state` (`state`)
) ENGINE=InnoDB
```

Now, we can use GIS notation to find our zip code:

```
SELECT zip, city, state, county, type, astext(geo) lon_lat
FROM zipcodes
WHERE ST_CONTAINS(geo, POINT(-97.300313, 35.254049));
```

So what?

- We had already written a query do just that
- And this one is still doing the same amount of handler operations!

- This opens up the opportunity to run other GIS based calculations as we'll see now...

Not just points!

- In earlier slides, we based everything on a single lat/lon coordinate
- Several “zip code” databases follow this model
- In recent years, the prevalence of full Polygon region databases has increased

- Enter the US Census provided **tl_2014_us_zcta510** schema...

Not just points...

- This table defines full **regions** for zip codes as opposed to a centered lat/lon for the post office

```
CREATE TABLE `tl_2014_us_zcta510` (
  `OGR_FID` int(11) NOT NULL AUTO_INCREMENT,
  `SHAPE` geometry NOT NULL,
  `zcta5ce10` varchar(5) DEFAULT NULL,
  `geoid10` varchar(5) DEFAULT NULL,
  `classfp10` varchar(2) DEFAULT NULL,
  `mtfcc10` varchar(5) DEFAULT NULL,
  `funcstat10` varchar(1) DEFAULT NULL,
  `aland10` double DEFAULT NULL,
  `awater10` double DEFAULT NULL,
  `intptlat10` varchar(11) DEFAULT NULL,
  `intptlon10` varchar(12) DEFAULT NULL,
  UNIQUE KEY `OGR_FID` (`OGR_FID`),
  SPATIAL KEY `SHAPE` (`SHAPE`),
  KEY `zcta5ce10` (`zcta5ce10`)
) ENGINE=InnoDB
```

Difference in storage

- Old version for Norman zip code 73071:

```
astext(geo): POINT(-97.300313 35.254049)
```

- New version in region based version:

```
astext(shape): POLYGON((-97.44109 35.228675,-97.442413
35.228673,-97.442713 35.228669,-97.442722 35.229179,-
97.442729 35.229632,-97.442727 35.230083,-97.442726
35.230542,-97.442725 35.230998,-97.441071 35.23099,-
97.441072 35.231275,-97.441072 35.231437,-97.442724
35.231464,-97.442722 35.231912,-97.442647 35.231898,-
97.441073 35.231884,-97.441073 35.232346,...
```

- Note that now, we have a full region as opposed to a point...

GIS Functions

- st_contains
  - Check if an object is entirely within another object
- st_within
  - Check if an object is spatially within another object
- st_*
  - Detailed list here: http://dev.mysql.com/doc/refman/5.7/en/spatial-relation-functions-object-shapes.html

Sample GIS Queries

- # Find me all the zipcodes (polygons) for my points of interest

```
SELECT raw.zcta5ce10 AS zipcode, pts.name
FROM my_points pts
JOIN tl_2014_us_zcta510 raw ON st_within(pts.loc, raw.SHAPE);
```

- # Find me all points of interest within a zip code (polygon)

```
SELECT raw.zcta5ce10 AS zipcode, pts.name
FROM my_points pts
JOIN tl_2014_us_zcta510 raw ON st_contains(raw.SHAPE, pts.loc)
WHERE raw.zcta5ce10 = "73071";
```

Sample GIS Queries

- # Find me all the points of interest within a county

```
SELECT raw.zcta5ce10 AS zipcode, pts.name
FROM my_points pts
JOIN tl_2014_us_zcta510 raw ON st_contains(raw.SHAPE, pts.loc)
WHERE raw.zcta5ce10 IN (SELECT ZIP from legacy.zipcodes_indexed
where County = "Cleveland");
```

Impact of SPATIAL Key

- I mentioned earlier that SPATIAL keys will be available within InnoDB starting in 5.7
- Here is just quick sample of the same query, with and without a SPATIAL key around a polygon
- Lets grab one of the earlier queries:

```
SELECT raw.zcta5ce10 AS zipcode, pts.name
FROM my_points pts
JOIN tl_2014_us_zcta510 raw ON st_within(pts.loc, raw.SHAPE);
```

Impact of SPATIAL Key

- With Key

```
+---------+-------+
| zipcode | name  |
+---------+-------+
| 73071   | POI 2 |
| 73069   | POI 3 |
+---------+-------+
2 rows in set (0.00 sec)
```

```
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Handler_read_first         | 1     |
| Handler_read_key           | 4     |
| Handler_read_last          | 0     |
| Handler_read_next          | 4     |
| Handler_read_prev          | 0     |
| Handler_read_rnd           | 0     |
| Handler_read_rnd_next      | 4     |
| Handler_write              | 0     |
+----------------------------+-------+
```

- Without Key

```
+---------+-------+
| zipcode | name  |
+---------+-------+
| 73069   | POI 3 |
| 73071   | POI 2 |
+---------+-------+
2 rows in set (1 min 4.36 sec)
```

```
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Handler_read_first         | 2     |
| Handler_read_key           | 2     |
| Handler_read_last          | 0     |
| Handler_read_next          | 0     |
| Handler_read_prev          | 0     |
| Handler_read_rnd           | 0     |
| Handler_read_rnd_next      | 33149 |
| Handler_write              | 0     |
+----------------------------+-------+
```

Now, on to some (useful) samples!

Sample Usage

- Living in Oklahoma, the natural first choice for a quick database generally revolves around... weather!
- This sample uses data from the readily available “current conditions” flat files available from http://mesonet.org

Mesonet Collector

- Every five minutes (when new observations are posted), grab the current CSV of all sites
- Load the raw CSV data into a “staging” table
- Clean up the data and move it to the main “observations” table

- Also, an intermediate step that was only run once extracted all of the site information (lat/lon, name, etc) from the raw data to populate a base site table (mesonet.sites)

Some Fun Queries

```
# Get all the mesonet sites in a county (based on GIS coords)
SELECT raw.zcta5ce10 AS zipcode, sites.name
FROM mesonet.sites
JOIN geopoly.tl_2014_us_zcta510 raw ON st_contains(raw.SHAPE, sites.pt)
WHERE raw.zcta5ce10 IN (
    SELECT ZIP from legacy.zipcodes_indexed where County IN ("Oklahoma")
);
```

```
+---------+----------------------+
| zipcode | name                 |
+---------+----------------------+
| 73084   | Spencer              |
| 73107   | Oklahoma City West   |
| 73114   | Oklahoma City North  |
| 73117   | Oklahoma City East   |
+---------+----------------------+
4 rows in set (0.00 sec)
```

Some Fun Queries

```
# Get the AVG air temp for all sites in a county (based on GIS)
SELECT raw.zcta5ce10 AS zipcode, sts.name, AVG(TAIR)
FROM mesonet.sites sts
JOIN geopoly.tl_2014_us_zcta510 raw ON st_contains(raw.SHAPE, sts.pt)
JOIN mesonet.observations obs ON obs.stid = sts.stid
WHERE raw.zcta5ce10 IN (
    SELECT ZIP from legacy.zipcodes_indexed where County IN ("Oklahoma")
)
GROUP BY sts.name;
```

```
+---------+----------------------+-----------+
| zipcode | name                 | AVG(TAIR) |
+---------+----------------------+-----------+
| 73117   | Oklahoma City East   |   44.7393 |
| 73114   | Oklahoma City North  |   44.3037 |
| 73107   | Oklahoma City West   |   44.6729 |
| 73084   | Spencer              |   44.0250 |
+---------+----------------------+-----------+
4 rows in set (0.01 sec)
```

GPX Parsing

- As I wanted another sample, I looked through my phone to find an app I use with GPS frequently: MotionX-GPS (mainly for hiking/hunting)
- This app lets you:
  - Record waypoints
  - Record full tracks
  - Export tracks in GPX format (bonus)

GPX Parsing App

- Now to come up with something useful :)
- For no particular reason, I thought it would be cool to record a drive, then map those points out and see how many miles I had driven in each zip code
- Exciting stuff, I know!
- Note some possible other uses would be real logistics tracking (think fleet vehicles), tornado tracks, pipeline, etc

GPX File

- Sample snippet from a GPX file (collection of points in a track)

```
<?xml version="1.0" encoding="UTF-8"?>
<gpx xmlns="http://www.topografix.com/GPX/1/1" version="1.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.topografix.com/GPX/1/1
http://www.topografix.com/GPX/1/1/gpx.xsd"
creator="MotionXGPSFull 22.2 Build 4844R">
<trk>
<name><![CDATA[Track 001]]></name>
<desc><![CDATA[Jan 30, 2015  5:15 pm]]></desc>
<trkseg>
<trkpt lat="35.2397784" lon="-97.4411065">
<ele>364.148</ele>
<time>2015-01-30T23:15:02.060Z</time>
</trkpt>
<trkpt lat="35.2399270" lon="-97.4411044">
<ele>364.057</ele>
<time>2015-01-30T23:15:03.182Z</time>
</trkpt>
...
...
```

Core Code

- Parse the input file, then insert each point

```
gpx = gpxpy.parse(gpx_file)

for track in gpx.tracks:
     for segment in track.segments:
          for point in segment.points:

               sqlPointList.append(
                    (
                         tripID,
                         "POINT(%0.7f %0.7f)" % (point.longitude, point.latitude),
                         point.time.strftime('%Y-%m-%d %H:%M:%S')
                    )
               )

               rawPointList.append((point.latitude, point.longitude))

insertPointsSQL = """INSERT INTO `raw_trip_points` (`trip_id`, `pt`, `date_created`)
                         VALUES (%s, GeomFromText(%s, 1), %s) """

c.executemany(insertPointsSQL, sqlPointList)
db.commit()
```

Core Code

- Join the points table to the zip code table to get the postal code of each point

```
appendZipSql = """ SELECT raw.zcta5ce10 AS zipcode, astext(pts.pt)
FROM trip_tracker.raw_trip_points pts
JOIN geopoly.tl_2014_us_zcta510 raw ON st_contains(raw.SHAPE, pts.pt)
WHERE pts.trip_id = %s""";


c.execute(appendZipSql, (tripID, ))
rows = c.fetchall()

# Some code omitted for formatting

for newZip, newPoint in rows:

     if newZip not in pointsByZip:
          pointsByZip[newZip] = 0

     if newZip == curZip:
          # Just get the distance
          pointsByZip[newZip] += vincenty(mysqlPointToTuple(curPoint),
                                          mysqlPointToTuple(newPoint)).miles
     # Omitted for formatting
```

And some output...

```
Distance Breakdown (by zip)
73102 : 0.7812 miles
73109 : 0.5431 miles
73117 : 0.6803 miles
73129 : 4.4827 miles
73149 : 1.9740 miles
73160 : 2.2176 miles
trans:73109-73102 : 0.0127 miles
trans:73117-73129 : 0.0493 miles
trans:73129-73109 : 0.0437 miles
trans:73129-73117 : 0.0535 miles
trans:73149-73129 : 0.0347 miles
trans:73160-73149 : 0.0476 miles
Total Distance: 10.92 miles
Total Transition Distance: 0.24 miles
```

Thanks for joining!

michael.benshoof@percona.com