Finding Points Within a Distance of a Latitude/Longitude Using Bounding Coordinates

by Jan Philip Matuschek

Abstract

This article describes how to efficiently query a database for places that are within a certain distance from a point given in spherical coordinates (latitude and longitude). The method computes bounding coordinates that can be used for a database index scan – just like we would use minimum bounding rectangles to speed up queries in Cartesian space. It is accurate for short distances as well as for long distances, and works for any geographic location, even if it is near a pole or the 180th meridian.

Contents

1 Distance Between Two Given Points

The shortest distance (the geodesic) between two given points $P_1=(lat_1, lon_1)$ and $P_2=(lat_2, lon_2)$ on the surface of a sphere with radius R is the great circle distance. It can be calculated using the formula:

$$dist = \arccos(\sin(lat_1) \cdot \sin(lat_2) + \cos(lat_1) \cdot \cos(lat_2) \cdot \cos(lon_1 - lon_2)) \cdot R \qquad (1)$$

For example, the distance between the Statue of Liberty at (40.6892°, -74.0444°)=(0.7102 rad, -1.2923 rad) and the Eiffel Tower at (48.8583°, 2.2945°)=(0.8527 rad, 0.0400 rad) – assuming a spherical approximation[a] of the figure of the Earth with radius R=6371 km – is:

$$\begin{aligned} dist &= \arccos(\sin(0.7102) \cdot \sin(0.8527) + \cos(0.7102) \cdot \cos(0.8527) \cdot \cos(-1.2923 - 0.0400)) \cdot 6371\text{ km} \\ &= 5837\text{ km} \end{aligned} \qquad (2)$$

2 Finding Places Within a Distance Without Using an Index

Suppose we want to find places within a distance d=1000 km from $M=(lat, lon)$=(1.3963 rad, -0.6981 rad) in a database. Given that we have a table named Places with columns Lat and Lon that hold the coordinates in radians (the trigonometric functions of SQL expect radians), then we could use this SQL query:

```
SELECT * FROM Places WHERE acos(sin(1.3963) * sin(Lat) + cos(1.3963) * cos(Lat) * cos(Lon - (-0.6981))) * 6371 <= 1000;
```

The problem with this query is that our database system will not be able to utilize indices on the Lat and Lon columns because of the complexity of the formula in the WHERE clause.

3 Finding Places Within a Distance Using an Index

In order to utilize indices on the Lat and Lon columns we can use a method similar to the bounding rectangle method in Cartesian space. Let’s call the circle that is formed by all points which have distance d from M the query circle. The following subsections explain how to compute bounding coordinates (lat_{min}, lon_{min}) and (lat_{max}, lon_{max}) which are opposite corners of a bounding rectangle (on the sphere) that completely contains the query circle. Points outside this rectangle are therefore definitely not within distance d from M, so in order to find points that are within the query circle, we need to consider the points within the bounding rectangle only. Those points (called candidate places) can be found quickly using an index (see section 3.5 for SQL queries).

Let’s define $r = d/R$ = (1000 km)/(6371 km) = 0.1570 as the angular radius of the query circle.

3.1 Computing the Minimum and Maximum Latitude

Moving on a circle with radius R=6371 km from a point A to a point B such that there is an angle r=0.1570 between A and B means covering a distance of d=1000 km. Meridians are on such great circles with radius

R. Hence we can move along a meridian, i.e. keep the longitude fixed, and simply substract/add r from/to lat in order to obtain the minimum/maximum latitude of all points within the query circle with center M=(lat, lon)=(1.3963 rad, -0.6981 rad):

$$lat_{min} = lat - r = 1.2393 \text{ rad} \quad (3)$$

$$lat_{max} = lat + r = 1.5532 \text{ rad} \quad (4)$$

Note that special care must be taken if a pole is within the query circle. See section 3.4 for details.

3.2 Computing the Minimum and Maximum Longitude – the Incorrect Way

One approach seen on several web pages for computing the minimum and maximum longitudes is keeping the latitude fixed and changing the longitude, i.e. moving along a circle of latitude. This section will show that this approach gives inaccurate results.

Moving along a circle of latitude means moving along a small circle. A circle of latitude at latitude lat=1.3963 rad has the radius $R_s = R \cdot \cos(lat)$ = 1106 km, so d=1000 km now corresponds to an angular radius of $r_s = d/R_s = d/(R \cdot \cos(lat))$ = 0.9039. Hence, covering d=1000 km on a circle of latitude gets you to longitude $lon_S = lon \pm d/(R \cdot \cos(lat))$ = -0.6981 rad ± 0.9039 rad. But this is **not** the minimum/maximum longitude you can get to by moving d=1000 km in any direction! This is because we moved the distance on a small circle, but small circle distances are greater than great circle distances. Although M=(lat, lon)=(1.3963 rad, -0.6981 rad) and P_S=(lat, lon_S)=(1.3963 rad, -1.6020 rad) have a distance of d=1000 km on the small circle, we could take a shortcut from M to P_S on a great circle. According to equation 1, that shortcut has a length of 967 km. So we could move another 33 km and would still be within the query circle. In particular, we could reach even smaller respectively greater longitudes. So using $lon \pm d/(R \cdot \cos(lat))$ as bounding values for longitude would miss some locations that are actually within distance d. For example, the point P_3=(1.4618 rad, -1.6021 rad) is outside the computed bounding “box”, although its distance to M is only 872 km. That is an error of more than 12 %!

3.3 Computing the Minimum and Maximum Longitude – the Correct Way

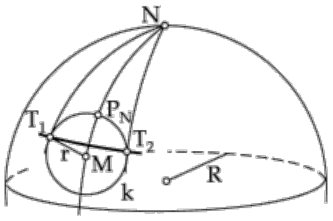

Figure 1: Tangent meridians to the query circle [1]

Moving along a circle of latitude in order to find the minimum and maximum longitude does not work at all as you can see in figure 1: The points on the query circle having the minimum/maximum longitude, T_1 and T_2, are not on the same circle of latitude as M but closer to the pole. The formulae for the coordinates of these points can be found in a good math handbook like [1]. They are:

$$lat_T = \arcsin(\sin(lat)/\cos(r)) = 1.4942 \text{ rad} \quad (5)$$

$$lon_{min} = lon_{T_1} = lon - \Delta lon = -1.8184 \text{ rad} \quad (6)$$

$$lon_{max} = lon_{T_2} = lon + \Delta lon = 0.4221 \text{ rad} \quad (7)$$

where

$$\Delta lon = \arccos(\ (\cos(r) - \sin(lat_T) \cdot \sin(lat)\ )\ /\ (\ \cos(lat_T) \cdot \cos(lat)\ )\ )$$
$$= \arcsin(\sin(r)/\cos(lat)) = 1.1202 \text{ rad} \quad (8)$$

Note that special care must be taken if the 180th meridian is within the query circle. See section 3.4 for details.

3.4 Dealing with Poles and the 180th Meridian

If lat_{max}, as defined in section 3.1, is greater than π/2, then the North Pole is within the query circle, which means that parts of all meridians are within the query circle as well. Thus, in that case the bounding coordinates are (lat_{min}, -π) and (π/2, π). If lat_{min} is smaller than -π/2, then the South Pole is within the query circle and the bounding coordinates are (-π/2, -π) and (lat_{max}, π).

If one of $lon_{min/max}$, as defined in section 3.3, is outside the range of valid longitude values [-π, π], the 180th meridian is within the query circle. In that case one could use (lat_{min}, -π) and (lat_{max}, π) as the bounding coordinates, i.e. something with a shape like a belt around the sphere. But that would include many candidate places that are not actually within the distance. Alternatively one could use two bounding “boxes” and it is sufficient for a candidate place to be in one of them. If lon_{min} < -π, the two sets of bounding coordinates are (lat_{min}, lon_{min} + 2π), (lat_{max}, π) and (lat_{min}, -π), (lat_{max}, lon_{max}). If lon_{max} > π, the two sets of bounding coordinates are (lat_{min}, lon_{min}), (lat_{max}, π) and (lat_{min}, -π), (lat_{max}, lon_{max} - 2π).

3.5 SQL Queries

Now that we have computed the bounding coordinates (lat_{min}, lon_{min})=(1.2393 rad, -1.8184 rad) and (lat_{max}, lon_{max})=(1.5532 rad, 0.4221 rad), we can use them in an SQL query. In the following, it is assumed that latitude and longitude – both in radians – are stored in two separate columns with an index that supports range queries (e.g. a B-tree) on each of them. If your database management system supports a point data type and a spatial index (e.g. an R-tree) on such columns, you can also use that.

There are different ways to combine the filter using the bounding coordinates with the formula for the great circle distance in an SQL statement:

- Simply combining the conditions with AND:

  ```
  SELECT * FROM Places WHERE
      (Lat => 1.2393 AND Lat <= 1.5532) AND (Lon >= -1.8184 AND Lon <= 0.4221)
  AND
      acos(sin(1.3963) * sin(Lat) + cos(1.3963) * cos(Lat) * cos(Lon - (-0.6981))) <= 0.1570;
  ```

  Most query optimizers are smart enough to perform an index scan to quickly find places satisfying `(Lat >= 1.2393 AND Lat <= 1.5532) AND (Lon >= -1.8184 AND Lon <= 0.4221)` and evaluate the formula for the great circle distance for each remaining candidate result only.
- The pre-filter in the WHERE clause and the more specific formula in a HAVING clause:

  ```
  SELECT * FROM Places WHERE
      (Lat >= 1.2393 AND Lat <= 1.5532) AND (Lon >= -1.8184 AND Lon <= 0.4221)
  HAVING
      acos(sin(1.3963) * sin(Lat) + cos(1.3963) * cos(Lat) * cos(Lon - (-0.6981))) <= 0.1570;
  ```
- The pre-filter in a sub-query:

  ```
  SELECT * FROM (
      SELECT * FROM Places WHERE
          (Lat >= 1.2393 AND Lat <= 1.5532) AND (Lon >= -1.8184 AND Lon <= 0.4221)
  ) WHERE
      acos(sin(1.3963) * sin(Lat) + cos(1.3963) * cos(Lat) * cos(Lon - (-0.6981))) <= 0.1570;
  ```

4 Java Source Code

4.1 The GeoLocation Class

Below you will find a ready-to-use Java GeoLocation class. The boundingCoordinates() method computes the bounding coordinates using the method explained in section 3. Section 4.2 shows an example usage of the GeoLocation class.

Download GeoLocation.java

```
/**
 * <p>Represents a point on the surface of a sphere. (The Earth is almost
 * spherical.)</p>
 *
 * <p>To create an instance, call one of the static methods fromDegrees() or
 * fromRadians().</p>
 *
 * <p>This code was originally published at
 * <a href="http://JanMatuschek.de/LatitudeLongitudeBoundingCoordinates#Java">
 * http://JanMatuschek.de/LatitudeLongitudeBoundingCoordinates#Java</a>.</p>
 *
 * @author Jan Philip Matuschek
 * @version 22 September 2010
 */
public class GeoLocation {

        private double radLat;  // latitude in radians
        private double radLon;  // longitude in radians

        private double degLat;  // latitude in degrees
        private double degLon;  // longitude in degrees

        private static final double MIN_LAT = Math.toRadians(-90d);  // -PI/2
        private static final double MAX_LAT = Math.toRadians(90d);   //  PI/2
        private static final double MIN_LON = Math.toRadians(-180d); // -PI
        private static final double MAX_LON = Math.toRadians(180d);  //  PI

        private GeoLocation () {
        }

        /**
         * @param latitude the latitude, in degrees.
         * @param longitude the longitude, in degrees.
         */
        public static GeoLocation fromDegrees(double latitude, double longitude) {
                GeoLocation result = new GeoLocation();
                result.radLat = Math.toRadians(latitude);
                result.radLon = Math.toRadians(longitude);
                result.degLat = latitude;
                result.degLon = longitude;
                result.checkBounds();
                return result;
        }

        /**
         * @param latitude the latitude, in radians.
         * @param longitude the longitude, in radians.
         */
```

```
public static GeoLocation fromRadians(double latitude, double longitude) {
        GeoLocation result = new GeoLocation();
        result.radLat = latitude;
        result.radLon = longitude;
        result.degLat = Math.toDegrees(latitude);
        result.degLon = Math.toDegrees(longitude);
        result.checkBounds();
        return result;
}

private void checkBounds() {
        if (radLat < MIN_LAT || radLat > MAX_LAT ||
                        radLon < MIN_LON || radLon > MAX_LON)
                throw new IllegalArgumentException();
}

/**
 * @return the latitude, in degrees.
 */
public double getLatitudeInDegrees() {
        return degLat;
}

/**
 * @return the longitude, in degrees.
 */
public double getLongitudeInDegrees() {
        return degLon;
}

/**
 * @return the latitude, in radians.
 */
public double getLatitudeInRadians() {
        return radLat;
}

/**
 * @return the longitude, in radians.
 */
public double getLongitudeInRadians() {
        return radLon;
}

@Override
public String toString() {
        return "(" + degLat + "\u00B0, " + degLon + "\u00B0) = (" +
                        radLat + " rad, " + radLon + " rad)";
}

/**
 * Computes the great circle distance between this GeoLocation instance
 * and the location argument.
 * @param radius the radius of the sphere, e.g. the average radius for a
 * spherical approximation of the figure of the Earth is approximately
 * 6371.01 kilometers.
 * @return the distance, measured in the same unit as the radius
 * argument.
 */
public double distanceTo(GeoLocation location, double radius) {
        return Math.acos(Math.sin(radLat) * Math.sin(location.radLat) +
                        Math.cos(radLat) * Math.cos(location.radLat) *
                        Math.cos(radLon - location.radLon)) * radius;
}

/**
 * <p>Computes the bounding coordinates of all points on the surface
 * of a sphere that have a great circle distance to the point represented
 * by this GeoLocation instance that is less or equal to the distance
 * argument.</p>
 * <p>For more information about the formulae used in this method visit
 * <a href="http://JanMatuschek.de/LatitudeLongitudeBoundingCoordinates">
 * http://JanMatuschek.de/LatitudeLongitudeBoundingCoordinates</a>.</p>
 * @param distance the distance from the point represented by this
 * GeoLocation instance. Must me measured in the same unit as the radius
 * argument.
 * @param radius the radius of the sphere, e.g. the average radius for a
 * spherical approximation of the figure of the Earth is approximately
 * 6371.01 kilometers.
 * @return an array of two GeoLocation objects such that:<ul>
 * <li>The latitude of any point within the specified distance is greater
 * or equal to the latitude of the first array element and smaller or
 * equal to the latitude of the second array element.</li>
 * <li>If the longitude of the first array element is smaller or equal to
 * the longitude of the second element, then
 * the longitude of any point within the specified distance is greater
 * or equal to the longitude of the first array element and smaller or
 * equal to the longitude of the second array element.</li>
 * <li>If the longitude of the first array element is greater than the
 * longitude of the second element (this is the case if the 180th
 * meridian is within the distance), then
 * the longitude of any point within the specified distance is greater
 * or equal to the longitude of the first array element
 * <strong>or</strong> smaller or equal to the longitude of the second
 * array element.</li>
 * </ul>
 */
public GeoLocation[] boundingCoordinates(double distance, double radius) {

        if (radius < 0d || distance < 0d)
```

```
                        throw new IllegalArgumentException();

                // angular distance in radians on a great circle
                double radDist = distance / radius;

                double minLat = radLat - radDist;
                double maxLat = radLat + radDist;

                double minLon, maxLon;
                if (minLat > MIN_LAT && maxLat < MAX_LAT) {
                        double deltaLon = Math.asin(Math.sin(radDist) /
                                Math.cos(radLat));
                        minLon = radLon - deltaLon;
                        if (minLon < MIN_LON) minLon += 2d * Math.PI;
                        maxLon = radLon + deltaLon;
                        if (maxLon > MAX_LON) maxLon -= 2d * Math.PI;
                } else {
                        // a pole is within the distance
                        minLat = Math.max(minLat, MIN_LAT);
                        maxLat = Math.min(maxLat, MAX_LAT);
                        minLon = MIN_LON;
                        maxLon = MAX_LON;
                }

                return new GeoLocation[]{fromRadians(minLat, minLon),
                                fromRadians(maxLat, maxLon)};
        }

}
```

Download GeoLocation.java

4.2 Using the GeoLocation Class

Here is some demo code that shows how to use the GeoLocation class from section 4.1 to query a database for places within a certain distance.

Download GeoLocationDemo.java

```
/**
 * <p>See
 * <a href="http://JanMatuschek.de/LatitudeLongitudeBoundingCoordinates#Java">
 * http://JanMatuschek.de/LatitudeLongitudeBoundingCoordinates#Java</a>
 * for the GeoLocation class referenced from this code.</p>
 *
 * @author Jan Philip Matuschek
 * @version 26 May 2010
 */
public class GeoLocationDemo {

        /**
         * @param radius radius of the sphere.
         * @param location center of the query circle.
         * @param distance radius of the query circle.
         * @param connection an SQL connection.
         * @return places within the specified distance from location.
         */
        public static java.sql.ResultSet findPlacesWithinDistance(
                        double radius, GeoLocation location, double distance,
                        java.sql.Connection connection) throws java.sql.SQLException {

                GeoLocation[] boundingCoordinates =
                        location.boundingCoordinates(distance, radius);
                boolean meridian180WithinDistance =
                        boundingCoordinates[0].getLongitudeInRadians() >
                        boundingCoordinates[1].getLongitudeInRadians();

                java.sql.PreparedStatement statement = connection.prepareStatement(
                        "SELECT * FROM Places WHERE (Lat >= ? AND Lat <= ?) AND (Lon >= ? " +
                        (meridian180WithinDistance ? "OR" : "AND") + " Lon <= ?) AND " +
                        "acos(sin(?) * sin(Lat) + cos(?) * cos(Lat) * cos(Lon - ?)) <= ?");
                statement.setDouble(1, boundingCoordinates[0].getLatitudeInRadians());
                statement.setDouble(2, boundingCoordinates[1].getLatitudeInRadians());
                statement.setDouble(3, boundingCoordinates[0].getLongitudeInRadians());
                statement.setDouble(4, boundingCoordinates[1].getLongitudeInRadians());
                statement.setDouble(5, location.getLatitudeInRadians());
                statement.setDouble(6, location.getLatitudeInRadians());
                statement.setDouble(7, location.getLongitudeInRadians());
                statement.setDouble(8, distance / radius);
                return statement.executeQuery();
        }

        public static void main(String[] args) {

                double earthRadius = 6371.01;
                GeoLocation myLocation = GeoLocation.fromRadians(1.3963, -0.6981);
                double distance = 1000;
                java.sql.Connection connection = ...;

                java.sql.ResultSet resultSet = findPlacesWithinDistance(
                                earthRadius, myLocation, distance, connection);

                ...;
        }

}
```

Download GeoLocationDemo.java

5 Implementations in Other Programming Languages

Some people have translated the Java code from section 4 into other programming languages and published their results. There are implementations in C#, Clojure, JavaScript (Node.js), Objective-C, PHP, Python and Swift. Note that I have not checked these for correctness.

Footnotes

a. The maximum error that results from using a spherical Earth model for distance calculations is no more than 0.5 %. [2]

References

1.
   - **German edition:** Bronstein, Semendjajew, Musiol, Mühlig: Taschenbuch der Mathematik. Verlag Harri Deutsch, Frankfurt am Main. with CD at Amazon.de, without CD at Amazon.de
   - **English edition:** Bronshtein, Semendyayev, Musiol, Mühlig: Handbook of Mathematics. Springer, Berlin. at Amazon.com, at Amazon.co.uk
2. Clynch: Earth Models and Maps. http://www.oc.nps.edu/oc2902w/general/mapmodel.pdf

Additional keywords: geospatial index, geospatial bounding box, spherical bounding box, minimum bounding box, minimum bounding rectangle (MBR), location-based service, geolocation, geographical location, geographic position, geographical position, geographic point, geographical point, bounding circle, GPS, geographic information system (GIS), inclination, azimuth, fast lookup

First published 16 May 2010, last updated 5 June 2016

If you have comments or suggestions for improving this article, please send me e-mail.