Fast nearest-location finder for SQL (MySQL, PostgreSQL, SQL Server)

Fri 31-Oct-2014 by Ollie

I’ve spent enough time goofing around with location-finder software that it’s worth writing up how to do it.   Of course, finding distances on the surface of the earth means using Great Circle distances, worked out with the Haversine formula, also called the Spherical Cosine Law formula. The problem is this:

> *Given a table of locations with latitudes and longitudes, which of those locations are nearest to a given location?*

Tables of locations

Where can I get a table of locations with latitudes and longitudes, you ask? Do an internet search for “free zipcode download” or “free postcode download.” Then get that loaded into a MySQL table. There are free downloads for plenty of different types of geographic data with latitude and longitude location attached.

Here’s a table of 5-digit zip codes for the United States. US Zip Code Data for MySQL. You should be able to load this into your MySQL server. If you’re going to go into production, please don’t rely on this data.

Here is the US Zip Code Data for SQL Server, if you happen to need that.

> *The logic in this article works for MySQL, MariaDB, PostgreSQL, and Microsoft SQL Server. Oracle works a little differently; here’s an article showing how to do this in Oracle.*

Please be careful when using ZIP codes or postcodes as a way to determine locations. ZIP and postcodes were designed for the single purpose of helping optimize postal delivery. Their value for other purposes is *limited*, and may give you *wrong* results. For example, here’s an article by a geographer about the water crisis in Flint, Michigan USA. For a long time it looked like kids in Flint weren’t getting lead poisoning because researchers only looked at their home ZIP codes to figure out where they lived. But they were getting lead poisoning. Don’t make the same mistake the MIchigan state government made.

Boring but necessary geography

Latitudes and longitudes are expressed in degrees. Latitude describes how far north or south of the equator a point is located. Points along the equator have latitudes of zero. The north pole has a positive (north) latitude of 90, and the south pole a negative (south) latitude of -90. Accordingly, northern-hemisphere locations have positive latitude, and southern-hemisphere locations have negative latitude.

Longitude describes how far east a point is, from the prime meridian: an arbitrary line on the earth surface running from pole to pole. The Empire State Building in New York City, USA, has a negative (west) longitude, specifically -73.9857. The Taj Mahal in Agra, India, has a positive (east) longitude, specifically 78.0422. The Greenwich Observatory near London, England, has, by definition, a longitude of zero.

Latitudes therefore are values in the range [-90, 90]. Longitudes are values in the range (-180, 180]. These values are sometimes expressed in degrees, minutes, and seconds, rather than degrees and decimals. If you’re intending to do calculations, convert the minutes and seconds to decimals first.

In the Napoleonic era, the meter was first defined so there were ten million of them in the distance from the equator to one of the poles. So, the original number of meters in a degree of latitude was 10,000,000 / 90 or 111.111 km. However the earth bulges a little, so **111.045 km per degree** is considered a better approximation.

For the purpose of the kind of calculation we’re doing here, we’ll assume the earth is a sphere. It isn’t really; it bulges a bit at the equator, but for a location-finder problem the spherical assumption is good enough.

Search …

Recent Posts

- Legacy MySQL VMs
- Porting from CEFSharp to WebView2
- WordPress multisite programming tricks
- Snippet to add Edit link to category archive page
- Narrow high-pressure road bike tires considered harmful

Categories

- Biking
- C#
- Cars
- Checklists
- Digital Signs
- Geography
- Green
- History
- Infosec
- Javascript
- Jetpack
- Kids
- MySQL
- Nonprofit
- Oracle
- Peace and Justice
- php
- PostgreSQL
- Programming
- Reference
- Rhetoric
- SAML
- Sermon

This formula (111.045 km per degree) works fine when you're moving north or south: if you are changing your latitude but not your longitude. It also works if you're moving east or west — changing your longitude — along the equator. But north or south of the equator, the lines of longitude get closer together, so if you move a degree to the east or west, you move less than 111.045 km. The distance you actually move when you go one degree east or west is actually this number of km.

```
111.045 * cos(latitude)
```

Those of us in some former British colonies use miles. A nautical mile is defined as a minute (1/60th of a degree) of latitude. So there are **69 statute miles per degree** or **60 nautical miles per degree**. If you are dealing with such applications as GPS control of ploughing with teams of oxen, you may find it helpful to know that there are **552 furlongs per degree**.

Some USA-centric applications mess up the longitudes, representing them as positive rather than negative for western-hemisphere locations. If you're debugging something be on the lookout for this.

The Great Circle Distance Formula

What's the distance along the surface of the (spherical) earth between two arbitrary points, in degrees, given by their latitude and longitude in degrees? That's determined by the **Spherical Cosine Law, or the Haversine Formula**, which is this in MySQL syntax.

```
DEGREES(ACOS(LEAST(1.0,COS(RADIANS(lat1)) * COS(RADIANS(lat2)) *
             COS(RADIANS(long1) - RADIANS(long2)) +
             SIN(RADIANS(lat1)) * SIN(RADIANS(lat2)))))
```

It's the distance along the surface of the spherical earth. It works equally well when the locations in question are your apartment and your local supermarket, or when they are the airports in Sydney, Australia and Reykjavik, Iceland. Notice that this result is in degrees. That means we have to multiply it by 111.045, our value for km per degree, if we want the distance in km.

> *Notice that MS SQL Server requires a float or double parameter to RADIANS. RADIANS(30) returns an incorrect value, but RADIANS(30.0) works correctly. In general, MS SQL Server doesn't coerce integer values to float or double reliably, so be careful not to try to use an integer where you should use a float. Also, please keep in mind that US zip codes, even though they look like numbers, are character strings. Where I live we have zip codes like '01950'. This is *not* the same thing as 1950.*

Querying the Nearest Locations

Cool. So to find the nearest points in a database to a given point, we can write a query like this. Let's use the point with latitude 42.81 and longitude -70.81. This MySQL query finds the fifteen nearest points to the given point in order of distance.

```
SELECT zip, primary_city, latitude, longitude,
      111.045* DEGREES(ACOS(LEAST(1.0, COS(RADIANS(latpoint))
                 * COS(RADIANS(latitude))
                 * COS(RADIANS(longpoint) - RADIANS(longitude))
                 + SIN(RADIANS(latpoint))
                 * SIN(RADIANS(latitude))))) AS distance_in_km
 FROM zip
 JOIN (
     SELECT  42.81  AS latpoint,  -70.81 AS longpoint
   ) AS p ON 1=1
 ORDER BY distance_in_km
 LIMIT 15
```

Notice the use of the join to put latpoint and longpoint into the query. It's convenient to write the query that way because latpoint and longpoint are referred to multiple times in the formula. (MySQL doesn't need the "ON 1=1" but PostgreSQL does.)

(In SQL Server, use "SELECT TOP(15) zip …" in place of "LIMIT 15.")

Great. We're done, right? Not so fast! This query is accurate, but it is very slow.

Optimization

The query is slow because It must compute the haversine formula for every possible pair of points. So it makes your MySQL server do a lot of math, and it forces it to scan through your whole location table. How can we optimize this? It would be nice if we could use indexes on the latitude and longitude columns in the table. To do this, let's introduce a constraint. Let's say we only care about points in the zip code table that are within 50 km of the (latpoint, longpoint). Let's figure out how to use an index to eliminate points that are further away.

Remember, from our background information earlier in this article, that a degree of latitude is 111.045 km. So, if we have an index on our latitude column, we can use a SQL clause like this to eliminate the points that are too far north or too far south to possibly be within 50 km.

```
latitude BETWEEN latpoint - (50.0 / 111.045)
             AND latpoint + (50.0 / 111.045)
```

This WHERE clause lets MySQL use an index to omit lots of latitude points before computing the haversine distance formula. It allows MySQL to perform a range scan on the latitude index.

Something I learned
SQL Time Processing
topten
Type
Uncategorized
WordPress

Archive

2021
2020
2019
2018
2017
2016
2015
2014
2013
2012
2009

Finally, we can use a similar but more complex SQL clause to eliminate points that are too far east or west. This clause is more complex because degrees of longitude are smaller distances the further away from the equator we move. This is the formula.

```
longitude BETWEEN longpoint - (50.0 / (111.045 * COS(RADIANS(latpoint))))
              AND longpoint + (50.0 / (111.045 * COS(RADIANS(latpoint))))
```

So, putting it all together, this query finds the neareast 15 points that are within a bounding box of 50km of the (latpoint,longpoint).

```
 SELECT z.zip,
        z.primary_city,
        z.latitude, z.longitude,
        p.distance_unit
                 * DEGREES(ACOS(LEAST(1.0, COS(RADIANS(p.latpoint))
                 * COS(RADIANS(z.latitude))
                 * COS(RADIANS(p.longpoint) - RADIANS(z.longitude))
                 + SIN(RADIANS(p.latpoint))
                 * SIN(RADIANS(z.latitude))))) AS distance_in_km
  FROM zip AS z
  JOIN (   /* these are the query parameters */
        SELECT  42.81  AS latpoint,  -70.81 AS longpoint,
                50.0 AS radius,      111.045 AS distance_unit
    ) AS p ON 1=1
  WHERE z.latitude
     BETWEEN p.latpoint  - (p.radius / p.distance_unit)
         AND p.latpoint  + (p.radius / p.distance_unit)
    AND z.longitude
     BETWEEN p.longpoint - (p.radius / (p.distance_unit * COS(RADIANS(p.latpoint))))
         AND p.longpoint + (p.radius / (p.distance_unit * COS(RADIANS(p.latpoint))))
  ORDER BY distance_in_km
  LIMIT 15
```

partialquery hosted with ❤ by **GitHub** **view raw**

This query, even though it's a bit complicated, takes advantage of your latitude and longitude indexes and works efficiently.

Notice as part of the overall query that we JOIN this little subquery.

```
SELECT  42.81  AS latpoint,  -70.81 AS longpoint,
        50.0 AS radius,      111.045 AS distance_unit
```

The purpose of this is to make it easier for application software to provide the parameters needed for the query. **latpoint** and **longpoint** are the specific location for which you need nearby places. **radius** specifies how far away the search should go. Finally **distance_unit** should be 111.045 if you want to give your distances in kilometers and 69.0 if you want them in statute miles.

Limiting diagonal distance

But, this bounding-box query has the potential of returning some points that lie more than 50km diagonally from your (latpoint, longpoint): it's only checking a bounding rectangle, not the diagonal distance. Let's enhance the query to eliminate points more than 50km as the crow flies.

```
SELECT zip, primary_city,
       latitude, longitude, distance
  FROM (
 SELECT z.zip,
        z.primary_city,
        z.latitude, z.longitude,
        p.radius,
        p.distance_unit
                 * DEGREES(ACOS(LEAST(1.0, COS(RADIANS(p.latpoint))
                 * COS(RADIANS(z.latitude))
                 * COS(RADIANS(p.longpoint - z.longitude))
                 + SIN(RADIANS(p.latpoint))
                 * SIN(RADIANS(z.latitude))))) AS distance
  FROM zip AS z
  JOIN (   /* these are the query parameters */
        SELECT  42.81  AS latpoint,  -70.81 AS longpoint,
                50.0 AS radius,      111.045 AS distance_unit
    ) AS p ON 1=1
  WHERE z.latitude
     BETWEEN p.latpoint  - (p.radius / p.distance_unit)
         AND p.latpoint  + (p.radius / p.distance_unit)
    AND z.longitude
     BETWEEN p.longpoint - (p.radius / (p.distance_unit * COS(RADIANS(p.latpoint))))
         AND p.longpoint + (p.radius / (p.distance_unit * COS(RADIANS(p.latpoint))))
 ) AS d
 WHERE distance <= radius
 ORDER BY distance
 LIMIT 15
```

fullquery hosted with ❤ by **GitHub** view raw

Using Miles Instead of Km

Finally, many people need to use miles instead of km for their distances. This is straightforward. Just change the value of the distance_unit to 69.0.

That's the query for a typical *store-finder* or *location-finder* application based on latitude and longitude. You should be able to adapt it to your use without too much trouble.

Adapting this query to other location-table definitions

This query is, of course, written to run with a particular table definition for **zip**, a US zip code table. That **zip** table has columns named **zip**, **primary_city**, **latitude**, and **longitude**, among others. Notice that the table is referred to by **FROM zip AS z** in the query. That makes it have the alias **z**.

Your location table very likely has different columns. It should be straightforward to rewrite this query to use yours. Look for the columns referred to as ***z.something*** in the query, and replace those with the column names from your table. So, for example, if your table is called **SHOP** and has **shopname**, **shoplat**, and **shoplong** columns, you'll put **z.shopname** in place of **z.primary_city** and so forth. You'll refer to your table by mentioning **FROM SHOP as z** in the query.

Geography, MySQL, Reference

‹ Transcoding JPEG optimizer from Mozilla
› Fast nearest-location finder for Oracle

84 thoughts on "Fast nearest-location finder for SQL (MySQL, PostgreSQL, SQL Server)"

Ryan
Wed 29-Jul-2020 at 2:49 am

Wow thanks for this Ollie this is a fantastic post. Works so efficiently there's no reason to use spatial indexes!

Reply

Ollie
Mon 31-Aug-2020 at 4:27 pm

That has been my experience as well. Less complexity -> better software.

Reply

tbo
Wed 4-Mar-2020 at 6:09 pm

This is the kind of post that you find online after searching the internet for hours trying to figure out why some particular feature of MySQL doesn't work. (In my case, why can't I make an ST_Buffer in a geographic SRS?) Though this might be a better solution anyway.

In any case, I think your latest fix has a bug! I'm pretty sure it should be LEAST(1.0, … not GREATEST. I spent a few minutes racking my brain trying to figure out what part of the query I copied incorrectly, then it suddenly occurred to me that I was always taking the ACOS of 1.

Reply

Ollie
Thu 5-Mar-2020 at 8:01 am

(smacks forehead). You're right, it should be LEAST. Sorry for the trouble and thanks.

Reply

Rohan
Mon 24-Feb-2020 at 5:24 pm

Hello Ollie,

Thank You for the article.
It is of great help.

I am writing the same query/logic in MSSQL.
And calculation of distance_units throwing me ,

An invalid floating point operation occurred.

Reply

Ollie
Wed 26-Feb-2020 at 3:35 pm

Hi, I believe I recently fixed that problem. Check the post again and make sure you're using `...ACOS(GREATEST(1.0...`

Thanks for your patience.

Reply

Louis
Tue 24-Dec-2019 at 3:35 pm

Hi Olie,

Thanks for the detailed tutorial. However, I need hep with actually outputting the results of the query. I have searched almost everywhere but no one is referring to how to retrieve the returned results. Below is my query:

$sql = "SELECT uid, inCallPriceOneTime, lat, lng
FROM (
SELECT z.uid,
z.inCallPriceOneTime,
z.lat, z.lng,
p.radius,
p.distance_unit
* DEGREES(ACOS(COS(RADIANS(p.latpoint))
* COS(RADIANS(z.lat))
* COS(RADIANS(p.longpoint – z.lng))
+ SIN(RADIANS(p.latpoint))
* SIN(RADIANS(z.lat)))) AS distance
FROM selectedservice AS z
JOIN (/* these are the query parameters */
SELECT $lat AS latpoint, $lng AS longpoint,
50.0 AS radius, 111.045 AS distance_unit
) AS p
WHERE z.lat
BETWEEN p.latpoint – (p.radius / p.distance_unit)
AND p.latpoint + (p.radius / p.distance_unit)
AND z.lng
BETWEEN p.longpoint – (p.radius / (p.distance_unit * COS(RADIANS(p.latpoint))))
AND p.longpoint + (p.radius / (p.distance_unit * COS(RADIANS(p.latpoint))))
) AS d
WHERE distance <= radius AND is_approve='1'
ORDER BY distance
LIMIT 15";

How do I display results of the query to user using php?

Reply

Ollie
Fri 14-Feb-2020 at 11:12 am

Hi, sorry for my delay answering your question. You can find all sorts of tutorials on the intertoobz about how to use SQL queries from php. Here's just one.
https://www.tutorialrepublic.com/php-tutorial/php-mysql-select-query.php

Happy hacking!

Reply

Alex
Tue 22-Oct-2019 at 10:42 am

Hey, nice script! How can I echo the distance for every city in the results? I'm using a mysql query call from php.

Thanks in advance.

Reply

Ollie
Fri 1-Nov-2019 at 8:41 am

My sample query's result set includes the city name and distance. Display the distance when you examine the result set.

Reply

Benjamin Wöster
Wed 17-May-2017 at 7:50 am

Found this one really helpful, so I wanted to give back. What's missing in the provided solution is the edge case of two locations being near each other, but at the opposite ends of the longitude range. For example one point at longitude -179.999 and a second point at +179.999.

To make that work, I extended the condition specifying the longitude part of the bounding box from

z.longitude BETWEEN
p.longpoint – MAX_DIST
AND p.longpoint + MAX_DIST

to

(
— (-180 to 0)
z.longitude BETWEEN
GREATEST(-180, LEAST(0, p.longpoint – MAX_DIST))
AND
GREATEST(-180, LEAST(0, p.longpoint + MAX_DIST))
OR
— (0 to 180)
z.longitude BETWEEN
GREATEST(0, LEAST(180, p.longpoint – MAX_DIST))
AND
GREATEST(0, LEAST(180, p.longpoint + MAX_DIST))
)

Reply

Ollie
Fri 14-Feb-2020 at 11:14 am

You are exactly right. I've updated the post. Thanks!

Reply

Will
Tue 14-Mar-2017 at 11:16 am

Hi Ollie,

Really great work, thanks for this! How would I use this to find only the nearest zip to the query parameter? I have a table of UK postcodes, equivalent to your 'zip' table in the example. I also have a table of store locations, and I would like to match each UK postcode to its nearest store.

I have been getting close and have been trying to use a solution I've used before that returns the minimum value in column B for each entry in column A (https://dev.mysql.com/doc/refman/5.6/en/example-maximum-column-group-row.html). However, because distance is a calculated column, this is not appropriate here.

Any help appreciated! Thanks!

Reply

Ollie
Tue 14-Mar-2017 at 12:31 pm

Easy (if I understand your question; I might not.)

Change "ORDER BY distance LIMIT 15" to "ORDER BY distance LIMIT 1" in the last line of the query I showed.

Reply

Will
Tue 14-Mar-2017 at 12:56 pm

Correct me if I'm wrong but the LIMIT condition limits the result to 1 line. What I want is, for every line in my postcode table, I want to know which is the nearest store in my store table.

So in your final chunk of code from the blog post, I have swapped out your zip table for my postcode table, and where you have

```
JOIN ( /* these are the query parameters */
SELECT 42.81 AS latpoint, -70.81 AS longpoint,
50.0 AS radius, 111.045 AS distance_unit
) AS p ON 1=1
```

I have instead JOINed onto my list of stores.

However, this is trying to calculate the distances between each postcode and all the stores that fall within a specific radius, whereas I want to keep just the one closest store for each postcode (otherwise the result is in the trillions!). Does that make more sense? Hope this isn't too much of a derailment from the original post!

HARISH BARSIWAL
Mon 21-Oct-2019 at 3:25 pm

Hi, I have list of Lat, long records in the table, this records i need to show on a map. in another table i have Lat,long , i need to see if this lat , long falls inside the shape that 1st table created a shape. Can we do this in sql query?

Ollie
Fri 1-Nov-2019 at 8:46 am

This requires using your database server's geospatial capability. MySQL offers the ST_Contains() function to determine whether one item (a lat/long point for example) is contained within another item (a polygon for example).

Jarrod
Sat 25-Feb-2017 at 6:38 pm

Hi thanks the for the code. Is there anyway to SELECT columns latitude and longitude to enable the distance calculation, but only echo the distance in the result and not the latitude and longitude (considering user security/sensitive information).

Reply

Ollie
Tue 28-Feb-2017 at 7:30 pm

Yes, of course. Just leave the latitude and longitude columns out of the top-level SELECT in the query.

Reply

Ashif Shereef
Fri 2-Dec-2016 at 3:58 am

This is awesome. This is exactly what I was looking for. However, I have a concern.

1. Does the join operation helps with the execution time?

2. Does it help in execution time if I pre-calculate the (p.distance_unit * COS(RADIANS(p.latpoint) similar values using a PHP function and then eliminate the JOIN? I don't know whether it would help or not.

3. Where can I get the distance value to each (lat,long) point? In the "distance" variable? I kind of access the result set through a PHP loop that selects the values like row["distance"]. How can I get the distance value?

Reply

Ollie
Fri 2-Dec-2016 at 5:21 pm

The `JOIN` operation in my query has a tiny effect, if any, on execution time. It's there to allow the parameters necessary for the query to be supplied simply. That's all.

If you precalculate the rectangular ranges in your client php program, you'll save several nanoseconds of CPU time on the MySQL server each time you run a query. Those ranges are only calculated once per query. I don't think it's worth your trouble.

My query's result set includes a `distance` column. A php program might fetch that with `row['distance']` or a similar associative array lookup. For each row, it contains the distance to the starting point.

Reply

Ashif Shereef
Sat 3-Dec-2016 at 1:17 am

Thanks for your reply man.
What are my chances of optimization if my tables consists of over one million places (As latitude,longitude,altitude) values and based on user's current location (userlatitude,userlongitude), I want to fetch places around him for a 300 meter radius? I have already indexed the latitude and longitude columns of my table. Does eliminating the join and writing a client side script to precalculate values help in optimization if my table has more than a million rows?

Ollie
Wed 16-Aug-2017 at 10:33 am

Yes, eliminating the JOIN is an optimization that will save you several nanoseconds per query. That's true whether your table has ten rows or a million rows.

Gustavo
Tue 16-Aug-2016 at 2:33 pm

Hi, this is a great formula and script, but it wasn't working for me, nor this formula and all of the other that I was trying so I assume there is something about my country location and the formulas.

I'm living in the south, and all the values are negatives (i.e.: latitude -25.282495 and long -57.635013), so I receiving wrong results from the database, for example a place which is supossed to be at least 32 kilometers but I get 18 kilometers, another place is near the site, located at 8,55 kilometers but the database returns 30,73.

Any idea about what changes I need to make on the script? thanks in advance

Reply

Ollie
Sat 3-Sep-2016 at 8:27 am

Gustavo, someone from Australia told me they succeeded. If you post or email a few of those coordinates, I'll take a look.

Reply

Tom
Wed 8-Jun-2016 at 4:19 am

Can I find a nearest location in a direction like east, west, north, south. Like nearest store in east

Reply

Ollie
Wed 8-Jun-2016 at 10:17 am

Yes, of course. You'll need to filter both on the distance and the direction from the input point to the target point. To filter for target points to the east of the input point you could simply change this :

```
AND z.longitude
BETWEEN p.longpoint – (p.radius / (p.distance_unit * COS(RADIANS(p.latpoint))))
AND p.longpoint + (p.radius / (p.distance_unit * COS(RADIANS(p.latpoint))))
```

to this

```
AND z.longitude
BETWEEN p.longpoint
AND p.longpoint + (p.radius / (p.distance_unit * COS(RADIANS(p.latpoint))))
```

This sets the western edge of the bounding box around your point p to the same longitude as the point, and the eastern edge as described in my post. So no points to the west of p will be matched.

Reply

Ollie
Tue 19-Apr-2016 at 2:48 pm

Yes, there's another problem with this zip code table too. It isn't up to date with the latest USPS changes. My purpose with the cheap and nasty one I posted is to make it possible for people to experiment.

I'll be delighted to post a link to a better zip code table if you want to suggest one.

Reply

Johny
Mon 18-Apr-2016 at 7:59 am

Thank you very much Ollie.How can I calculate the distance in metres?

Reply

Ollie
Tue 19-Apr-2016 at 2:25 pm

If you want distance in metres, use 111045.0 for the distance_unit value. You can use any distance_unit value you choose. If you choose 1, you'll get your result in degrees.

Reply

Rick James
Sun 17-Apr-2016 at 5:00 pm

The Haversine formula is nice and fast, but it has a 'fatal' flaw of returning NULL sometimes when measuring the distance between a point and itself. About 3% of latitudes have this problem; example: 60.5544. Any longitude works, because it cancels out. About 6% compute that the distance from a point to itself is a few inches. The cause of the NULL is taking the ACOS of a number slightly larger than 1.

Reply

Ollie
Tue 19-Apr-2016 at 2:16 pm

It's true that the spherical cosine law formula is unstable in the cases you mention. A better formula is by Vincenty — http://www.plumislandmedia.net/mysql/vicenty-great-circle-distance-formula/ . It uses ATAN2 in place of ACOS and so dodges the 1 +/- epsilon issue you mention.

Reply

MD
Tue 5-Apr-2016 at 10:57 am

Although it's probably not relevant for most (if not all) uses of this query. I think there will be a problem with handling locations around the international date line.

Let's for example take two points one with longitude 179.9 and one with longitude -179.9, the distance between them is very small since they both located just to the east and west of the international date line. The bounding rectangle for one will not include the other, so a valid result is lost.

Reply

Ollie
Wed 21-Sep-2016 at 11:58 am

That's correct. Some special work is required for locations antipodal to the prime meridian.

Reply

Ollie
Sun 13-Dec-2015 at 5:23 pm

You wrote

> *You will find that INDEX(lat, lng) will not use both columns to play the bounding-box game without a range scan of part of the index.*

What you say is true. But still, for a postcode-size table, range-scanning part of the index is measurably faster than leaving out the longitude column from the index (at least on my MariaDB 10 server). For searching a table a couple of orders of magnitude larger than postcode size, your approach is good. So is using the MySQL geo extension.

Reply

John Manny
Fri 23-Oct-2015 at 8:50 pm

How do I limit the output when I have a boundary rectangle based on say State. For example, I want to extract all transactions with lat / longs within Washington State. I don't want to limit it by radius but by coordinates based on in this case – a state boundary.

Reply

Ollie
Sat 24-Oct-2015 at 9:44 am

This sort of computation is called point-in-polygon inclusion. It's used in an service called geofencing. The geo extension to the open source postgreSQL rdbms contains functions for doing this computation. The bounding-rectangle trick I used to make the query fast can also be used to optimize point-in-polygon operations. There are plenty of sources for client-side point-in-polygon computation code as well.

If the lat-long values in the boundary polygon are reasonably close together and/or boundaries lie on parallels or meridians, the lat/long values can be used directly for this computation. Big chunks of the northern, eastern, and southern borders of Washington lie on parallels and meridians.

Geographically, Washington State is an interesting example. Is the surface of Puget Sound / the strait of Juan de Fuca considered inside the territory? Does the boundary polygon you obtain from some source or other encompass those bodies of water?

Reply

Selva
Thu 20-Aug-2015 at 9:04 am

Thank you Ollie for the awesome explanation on intricacies of handling geographic coordinates in a database setting. I have incrementally extended some of the techniques and been using bit more approximate methods in the context of real big data environments. Taking your lead as an example, wanted to share that knowledge, so I wrote a brief post.
here it is
http://geospatialsql.blogspot.com/2015/08/handling-billions-of-geographical.html

Reply

Ollie
Thu 20-Aug-2015 at 10:17 am

Nice refinement! Thanks.

Reply

César
Thu 29-Jan-2015 at 10:15 pm

Thanks so much for your work! It helped me a lot.

By the way, Can I translate this article to portuguese? I'll keep origin source!

Just a suggest for improvement: store the cos and sin latitude.

Reply

Ollie
Fri 30-Jan-2015 at 9:30 am

I am delighted you will translate the article to Portuguese. Thanks! By the way, my web site uses the Creative Commons license shown here. The idea is to encourage you to use my material as you wish.
http://creativecommons.org/licenses/by/4.0/deed.pt_BR

As for your suggestion of storing the latitudes' cosines and sines, I am not sure I agree. The problem is this; In some cases — points near one another — the numerical precision of the cosine of the angle is not as good as the angle itself. (These values are very close to 1, and what counts is the difference between them.) This means that the cosines need to be stored in DOUBLE rather than FLOAT. That means a bit more data has to travel over the network and on hard drives.

The time it takes a modern server to compute a COS or SIN is trivial — dozens of nanoseconds — compared to the time it takes to move data from hard drives and across networks — microseconds. So it's better to stay with angles, I believe.

In an earlier age, when computers used software floating point computation, your method would have been very much superior.

Reply

Michael
Mon 17-Nov-2014 at 8:33 pm

How would this work if I have a fixed location I want to search within a dynamic distance range of locations in the database? For example, I have a customer that wants to find a plumber to hire, the database has a list of those plumbers but each plumber has a set number of miles they are willing to travel (one might only be willing to go 10 miles, another 50 miles). Any help appreciated!

Reply

Ollie
Sun 23-Nov-2014 at 7:07 am

Two things to worry about here. First, you still need a bounding rectangle to make the search indexable. I suggest you choose a bounding rectangle that's the largest distance one of your putative plumbers will travel. (If you're serving Alaska, and those guys are willing to go 600 miles, you might keep track of the maximum distance by state.)

Then, simply use `WHERE distance <= plumber.service_radius` instead of `WHERE distance <= constant`

It should work well.

Reply

Michael
Mon 24-Nov-2014 at 12:38 pm

thank you! I'll give that a try

Alejandro
Wed 29-Oct-2014 at 3:26 am

Bravo!!! Thanks Ollie, amazing query!!!

Reply

Ollie
Wed 29-Oct-2014 at 10:13 pm

You're welcome. Glad you found it helpful.

Reply

Paul
Mon 6-Oct-2014 at 5:30 pm

Just a note regarding distance-sorting with the haversine distance:

I was tasked to do a proof-of-concept for an algorithm that required distance calculations on census block data. Unfortunately, our database system didn't support trig functions, and was a bit on the slow side– and I had a table with >100k rows to sift through.

Turns out, if you're willing to put off the final distance calculations for application code, you can actually dispense with the arcsine computations. Arcsine is an increasing function over its entire domain. So if asin(x)>asin(y), then x>y.

You need arcsine to COMPUTE distance, but not to SORT BY distance– the inputs to arcsine are basically proxies for the actual distances! As an added bonus, because we're avoiding the flattening behavior of arcsine near 0, points really close to one another still behave relatively well.

Plus, if you directly store the sine and cosine of the latitude and longitude in the table, a little work with trig identities (most of which you already seem to have integrated into your queries above) can get each pairwise distance-proxy calculation down to no more than 5 or 6 multiplications and a similar number of addition operations.

Reply

Ollie
Wed 8-Oct-2014 at 6:55 pm

This is true, and a good optimization. It's also true that you can use the square of Cartesian distance as a proxy for the actual distance.

But here's the thing: in a modern DBMS avoiding a full table scan (or full index scan) is ordinarily much more important to performance than avoiding computation on some rows.

Reply

David B
Fri 25-Jul-2014 at 12:52 am

Thanks for sharing this valuable info, Ollie! Really appreciate it!

I am currently using the non-optimized version of the Haversine and every so often I've had a few queries show up in mysql_slow_queries log – some taking as long as 18 seconds. So I've been looking for ways to optimize the query and I found your post.
# Query_time: 18.721642 Lock_time: 0.000216 Rows_sent: 17 Rows_examined: 2936

Do you know if this would work better with the InnoDB table engine? My table currently used MyISAM because its 95+% used for read only operations?

Reply

Ollie
Fri 25-Jul-2014 at 9:06 am

David, by "optimized version" I suppose you mean using the bounding-box distance to narrow down the distance search. I've tried this with a 44K row zip code table with both InnoDB and MyISAM, and it makes very little difference.

Reply

David B
Fri 25-Jul-2014 at 10:00 am

Thanks for your response Ollie! Yes, I meant your solution – the bounding-box optimization for the distance search.

Actually, I downloaded your Zip table and tried it on my MYSQL server (installed on my PC) with both
– Basic Haversine Query &
– Haversine with bounding box optimizatin Query

In both cases for a 100 mile radius, I was getting ~250 rows returned back within ~25 milliseconds. In other words, the execution time was pretty much identical.

Then I tried a similar test with my own table on my Webhost MySQL server. For returning roughly 60 rows in a 25 mile radius, the server was taking roughly ~50 ms.

Are the real gains from this optimization observed when the Server resources (RAM, processor cycles) are constrained (I'm using shared hosting) ?

Thanks again Ollie!

happy
Mon 16-Jun-2014 at 8:17 am

hi Mr.Ollie really it is a great article, i was looking for like.

But I have one problem with my table and I was stuck in spatial query for it.

description of my table is like this
field – type
——————-
id – int (10)
property – geometry

in property field we stored latitude and longitudes as POINT type.
(select astext(property) from mytable; will return POINT(12.000 77.0000))

how to apply haversine formula as you mentioned on mytable?

do i need to extract lat lon values first and apply your query after?
(i tried select X(coordinates),Y(coordinates) from mytable; but ended up with unknown column coordinates error)
or
is there any direct way to apply haversine formula on mytable?

thank you

Reply

Ollie
Tue 17-Jun-2014 at 8:42 am

Take a look at this post: http://www.plumislandmedia.net/mysql/using-mysqls-geospatial-extension-location-finder/. It explains how to use a table structure such as yours for this purpose.

It's possible there's something corrupt in your property column. You might try this query. If you get an error, you may need to rebuild this table:

```
SELECT AVG(X(property)) AS avgX, AVG(Y(property)) AS avgY
FROM table
```

This query will try to use the X() and Y() functions on every item in your geometric column. If something's wrong it may fail. It's a diagnostic.

Reply

Alex
Mon 2-Jun-2014 at 9:20 pm

Great piece of code! Do you have any advice in terms of how to specify WHERE clauses on the subquery, when adapting to different tables? I also have a PITA issue of storing lat/lon from IP, and sibling fields for real geocoding of address (lat_geocoded, lon_geocoded), requiring an IF statement.

Performance is WAY better with this.

Reply

Ollie
Tue 3-Jun-2014 at 8:29 am

Alex, thanks for the kind feedback. I've updated the article a bit to describe how to adapt the query to other table definitions. Notice that the performance of this query depends critically on MySQL's ability to access indexed columns of **latitude** and **longitude** values using an index range scan. So, your query needs to make that possible.

WHERE a.lat BETWEEN this AND that

will work fine, but for example,

WHERE IFNULL(a.lat,b.lat) BETWEEN this AND that

will not exploit the index, and as we say here in New England, your query will be wicked pissah slow.

Reply

Alex
Tue 3-Jun-2014 at 11:28 am

Thanks so much for taking the time to do that. I've implemented the Haversine a good few times now but always lacked a way to optimize it properly: this is the best one i've seen. Spent a good few hours hacking away at it last night with moderate success, as i've never used a subquery in the FROM clause before, although i've got the indexes in. It's also slightly complicated as i'm attempting to adapt it to PDO-style functionality in Laravel (PHP). I may have to leave out the IF NULL and perform a separate query, with some app caching.

In case you're curious as to the implementation, here's the thread with the original question:
http://laravel.io/forum/06-03-2014-eager-loading-from-a-list-of-user-ids-haversine

Aah, New England. The wife lives in Rhode Island, and i miss it. Toronto's not the same!

Martin Kovachev
Tue 27-May-2014 at 12:11 am

Thank you,

Really neat and fast indeed!

Reply

Luca
Sat 24-May-2014 at 8:41 am

Hello, thanks for your article, as mentioned by Mark Roland, there is another article that uses the Haversine theorem.

I have compared results from your method to the Haversine formula method and I have found a small difference.

The haversine formula is explained in this web site:
http://en.wikipedia.org/wiki/Haversine_formula
and is this:

1.609344 * (3956 * 2 * ASIN(SQRT(POWER(SIN((lat1 – ABS(lat2)) * pi()/180 /2), 2) + COS(lat1 * pi()/180) * COS(ABS(lat2) * pi()/180) * POWER(SIN((long1 – long2) * pi()/180 / 2), 2))))

From this point(37.8644738, 13.4621642) to point(38.1156879, 13.3612671),

your method: 29.26024725139351 Km,
the other: 29.27935591444092 Km.

Ok, the difference is 19 m, but the precision is important!

What method is the best?

Thank you very much!

Reply

Ollie
Sat 24-May-2014 at 10:19 am

Luca, the difference between your result and mine is solely due to small differences in the way my formula and yours handle the radius of the earth. I am using a value of 111.045 * 180 / π for that, and you are using 111.1175 (that is, 3956 * 1.609344 * π / 180). That accounts for the difference in results.

My value of 111.045 km per degree on the surface of the earth, and your value of 111.1175, are both within the bounds of the approximation that the earth is a perfect sphere. It isn't; it bulges at the equator so there are slightly more km per degree near the equator.

If you are greatly concerned about numerical stability in the computation, please see the Vincenty formula described here. It's more stable for small distances.
http://www.plumislandmedia.net/mysql/vicenty-great-circle-distance-formula/

Reply

Luca
Sat 24-May-2014 at 10:48 am

Sorry, I meant Haversine formula:

((3956 * 2 * ASIN(SQRT(POWER(SIN((lat1 – ABS(lat2)) * pi()/180 /2), 2) + COS(lat1 * pi()/180) * COS(ABS(lat2) * pi()/180) * POWER(SIN((long1 – long2) * pi()/180 / 2), 2)))) * 1.609344)

The final * 1.609344 is used to convert miles in km.

Thank you!

Lei Ho
Sun 4-May-2014 at 12:17 pm

Great article ! Thanks !

Reply

Mark Roland
Wed 16-Apr-2014 at 4:29 pm

This is amazing and has just helped me greatly improve my geo-searching. I was able to adapt your query to find results for a 5,000 mile search area across a list of 190,000 records in 70ms!

For others that may be interested, here is another similar article, http://www.scribd.com/doc/2569355/Geo-Distance-Search-with-MySQL, although I found this to be much more easily digestible.

Reply

Ollie
Sun 12-Apr-2015 at 12:08 pm

Indeed, that slide show from Alexander Rubin of the late great MySQL AB company is one of my inspirations for writing this one.

His “3956” magic number always bugged me, so I figured out the units for the computation.

Reply

Marty
Sat 29-Mar-2014 at 12:14 am

Excellent article Ollie!! If you’re familiar with SQL Server, what would the queries look like?

Reply

Cai
Fri 28-Feb-2014 at 1:29 am

Thanks a lot. Great helpful for me.

Reply

Garrett
Sun 23-Feb-2014 at 7:43 pm

Perfect! I was pulling my hair out trying to figure out a good solution. Thanks so much

-G

Reply

Ollie
Sun 23-Feb-2014 at 8:37 pm

Garrett, glad to be of help. It looks from your website like you might have some experience down under (that is, with south latitude and east longitude). Please let me know if you find any oddities in the location finder code there.

Reply

Tim Forsythe
Thu 20-Feb-2014 at 12:38 pm

Ollie, great improvement in speed, but I’m not getting any results for extremely close matches, where the distance in miles is essentially 0.00000. For example the database has 7.85 and 49.57 and geonames returns 7.84999906 and 49.56999901. I get 0 results which is intended to mean “not a match”. Is this an issue you’ve seen before?

Reply

Ollie
Fri 21-Feb-2014 at 8:19 am

TL;DR Don't use a comparison radius of zero. Use a small fraction of a mile or kilometer.

Tim, this difficulty in comparing pairs of numbers like 49.57 and 49.56999901 goes to the heart of using floating point numbers and digital computers for physical measurements. The two numbers you mention are only a fraction of a millimeter apart on the ground. Yet when you look for them to be precisely equal they aren't so you get strange results.

I suggest you try using a comparison radius of a tenth of a mile (or maybe 100 meters) when you're trying to find very-nearby This should yield reliable results for store-finder or GPS-style location search applications.

The sources of the tiny mismatches in floating point values are two. One is errors in measurement: GPS and surveyors' equipment is not completely perfect. The other is a characteristic of computer arithmetic called machine epsilon. You can read a good Wikipedia article about it here.

Reply

Tim Forsythe
Fri 21-Feb-2014 at 3:59 pm

Ollie, I forgot to mention I was using 0.01 mile as a comparison radius which is 52.8 ft. Any larger than that and I could be putting markers on the wrong house.

Currently, if I don't get a match I fallback to my slower check (the traditional haversine formula), which does the comparison on the distance in miles returned in the result array. When done like this, I've found, for "exact" matching coords that the distance returned by MySQL can be anywhere for 0.00….01 to 0.3 or even larger (which makes no sense). It is almost as though the algorithm, or rounding errors in MySQL, can cause strange behavior when the distance should be near zero. In the case of my example, the distance returned from the fallback version was indeed 0.000, so I don't understand why your version returns no hits using 0.01. I guess the next step is to try some other larger radii to see if there is minimum that works. It may be necessary to round comparison coords to the closest thousandth to eliminate creeping radii.

Ollie
Sat 22-Feb-2014 at 7:43 am

It may be the difference between float and double arithmetic. At any rate, if you ever find two computed float or double items to be exactly equal, it's a fluke. You really do need to take machine epsilon into account.

Oli
Sat 15-Feb-2014 at 9:20 pm

Amazing stuff! Querying my 2.5 million row table in 0.004s (YES! 0.004!!!!!) compared to 2.5s. Really really really great article thanks so much.

Reply

Ollie
Sun 16-Feb-2014 at 8:10 pm

Yes, MySQL is pretty good when you figure out the indexes. Glad your system is working so efficiently.

Reply

MarkL
Thu 2-Jan-2014 at 4:57 pm

Ollie, the type is FLOAT in my DB. The reason why I had to get rid of join is because of this: http://forums.laravel.io/viewtopic.php?pid=66739 I can't parametarize column names and JOIN complicates fluent query building.

Reply

MarkL
Thu 2-Jan-2014 at 11:23 am

Thanks for the great write-up. How would one get rid of the JOIN clause to simplify the statement?

Reply

Ollie
Thu 2-Jan-2014 at 12:38 pm

Mark, if you must omit the JOIN clause, you simply substitute your input values for latpoint, longpoint, and r in each place they occur in the rest of the SQL statement.

latpoint shows up in six places in the statement, longpoint in three, and r in five. You may make the statement a little shorter by eliminating the join clause, but you won't make it any faster.

Reply

MarkL
Thu 2-Jan-2014 at 2:47 pm

Thanks. I used:

```
DECLARE @latpoint DECIMAL (6, 2)
DECLARE @longpoint DECIMAL (6, 2)
DECLARE @r DECIMAL (6, 2)

SET @latpoint=42
SET @longpoint=-70
SET @r=50
```

and then replaced the values with variable bindings.

Ollie
Thu 2-Jan-2014 at 4:49 pm

You probably want FLOAT rather than DECIMAL (6,2) for these values. You probably also want three or four decimal places of precision when you're specifying positions on the ground. 42,-70 is in the ocean just east of Cape Cod, whereas 41.81, -70.81 is on land in my neighborhood near the Gulf of Maine. The point you gave, without the decimal places, is some seventy-odd statute miles south-southeast of here.

And, I think you're actually making the query more complex by eliminating the join and using bound variables, especially when they need to be type-converted.

mak
Fri 27-Dec-2013 at 5:39 am

Thank you so much for a wonderful article, this has saved tons of time and precisely what I was looking for..I am using the database from geonames.org

Reply

Ollie
Fri 27-Dec-2013 at 8:48 am

Hey, thanks for the compliment, and for the pointer to geonames.org!

Reply

Levi Lewis
Fri 6-Dec-2013 at 10:27 pm

Really well written with good examples. Math just blows my mind. Nice work.

Reply

Nick
Fri 29-Nov-2013 at 6:05 pm

Holy cow, this is awesome and exactly what I've been looking for. Thank you for sharing! It looks like the data sample you give here stores the lat/lons as double type. Is this generally what you use? It sounds like there are also some built in data types in MySQL for storing coordinates. But these are no good?

Thanks!

Reply

Ollie
Fri 29-Nov-2013 at 6:27 pm

Thanks for the kind words. Either FLOAT or DOUBLE data is sufficient for storing lat/long. As a matter of fact DOUBLE is overkill if you're using the haversine formula, for complex reasons relating to the inaccuracy of the assumption that the earth is a sphere. If you know the difference between a Lambert and a UTM projection, you may know what I'm talking about. But if you don't, it doesn't matter.

You could use the MySQL spatial extension for this purpose. If you had millions of lat/long points to process it would help performance. But you'd have to deal with the fact that it's oriented around Euclidean planar geometry rather than spherical.

Reply

Cameron Macfarlane
Fri 29-Nov-2013 at 12:38 pm

Nice work! Thank you very much, this helped me improve my location finder app.

Reply

Leave a Comment

Name *

Email *

Website

Post Comment