

# Playing with Geometry/Spatial Data Types in MySQL

In this tutorial, we are going to understand how to store and use spatial data types like coordinates and objects. But, we will mainly focus on Points (2D Cartesian Coordinate) and Geographic Locations (Geodetic Coordinates)

[Uday Hiwarale](#) Feb 22



(courtesy of [pexels.com](#))

Imagine that you are trying to develop an application that helps people find restaurants, pubs, bars and other hangout places near them. In nutshell, this will be a location discovery platform.

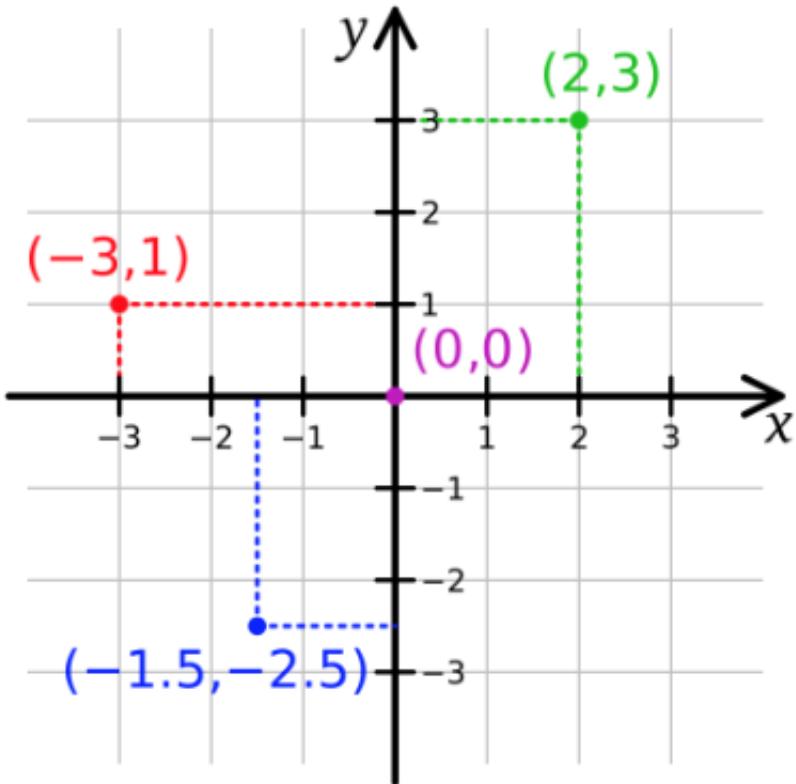
Looking from a backend perspective, we would be needing to store geographic data of these locations like **Latitude** and **Longitude**. Then we would need to write functions that calculate the distance between the user and the location (*to show how far the location is from him/her*). Using the same function, we can design an algorithm that finds closest places near to the user or within a given radius from him/her.

There are many tutorials and study material on the internet that helps you solve this problem by using simple data types like `float` for **latitude** and **longitude** and MySQL's capability to create internal **procedures/functions** to calculate & search locations. But in this tutorial, we are going to talk about MySQL's built-in **spatial data types**.

*This tutorial assumes that you have the latest MySQL/MariaDB version installed on your system and you are using InnoDB as your default database engine. All the tests in this tutorial are performed using MySQL Server Version 8.0.13 and InnoDB version 8.0.13. To Check MySQL version, use command `mysql --version` and same for InnoDB can be achieved by logging into the MySQL database and execute query `SHOW VARIABLES LIKE "innodb_version";`*

**OGC** or **Open Geospatial Consortium** is a non-profit organization consisting of many individuals, companies, and organizations that set the standard [OpenGIS](#) for open geospatial content and services. MySQL is one of the databases that follow a subset of this standard. You can read about OGC from [opengis.org](#).

Let's start by understanding some theory about the coordinate system. We are mainly focusing on a 2-dimensional coordinate system. Initially, we will focus on an **infinite flat cartesian coordinate system**.



(courtesy of Wikipedia)

On the left, you can see an infinite **2D** flat *unitless* surface where a point is represented like  $P(x, y)$  where  $x$  is the distance of a point  $P$  from the **origin** $(0,0)$  on the **x-axis** while  $y$  is the distance from the origin $(0,0)$  on the **y-axis**. Hence, if we know  $x$ ,  $y$  of a point, we can locate it on the coordinate surface.

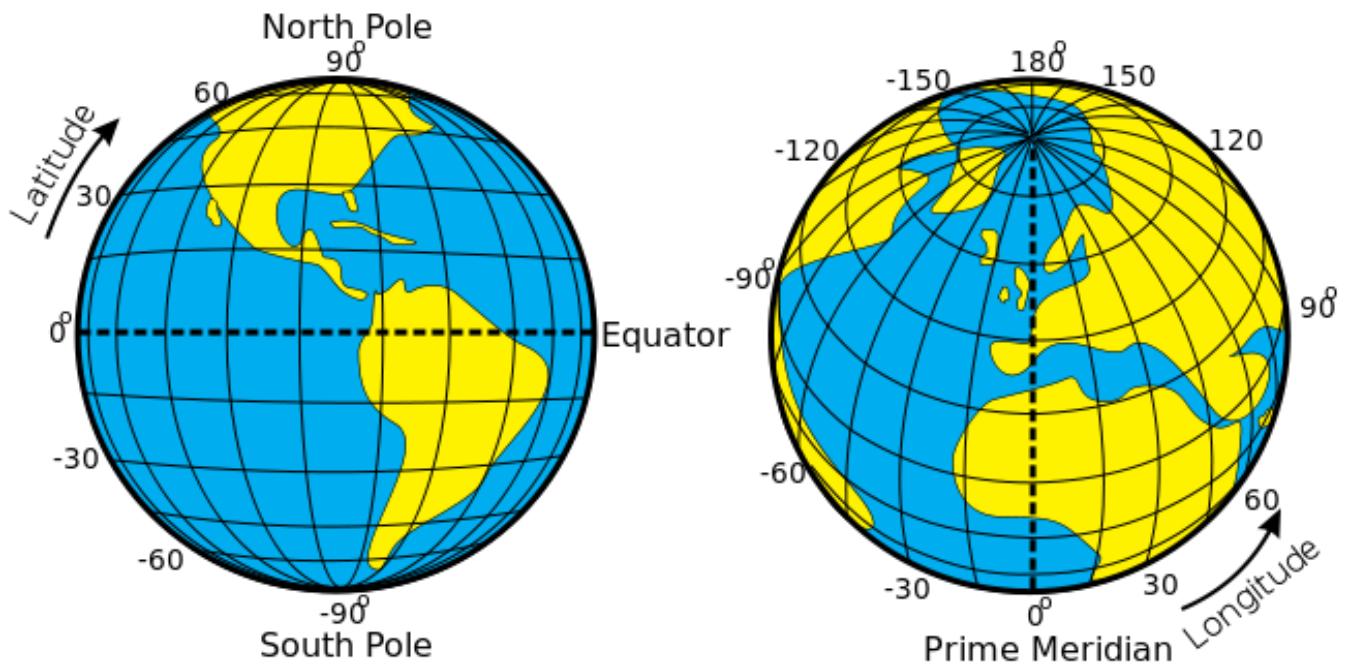
The distance of a point from the origin is calculated using **Pythagoras theorem**. If we consider the point in **green color**, using Pythagoras theorem, we can calculate its distance from the origin to be the **square root of the sum of the squares of the 2 and 3**.

If you want to calculate the **distance between two points**, you can shift the origin to any one point and measure the distance from it. If we had to find **points inside a circle** (*whose center is at the origin*), we have to calculate the distance of each point and check if it is less than or equal to equal to the radius of the circle

*I am assuming you understand basic math and geometry and I don't need to explain 2D cartesian coordinate system in formulae.*

For **Spherical Surface**, like **Planet Earth**, things are not as simple as a 2-dimensional surface. We follow the **geodetic reference system** for mapping locations on the surface of the planet. In a nutshell, the geodetic reference system

is based on **longitude** and **latitude** value.



(Courtesy of [Wikimedia Commons](#))

**Latitude** of a location (*point*) on the globe (*sphere*) is the number of degrees it is from the equator (*when measured along the axis of rotation of the Earth*).

**Longitude** of a location on the globe is the number of degrees it is from the **prime meridian** (*when measured along the equator*). Both Latitude and Longitude are measured from the center of the Earth. You can read more about the [\*\*Geographic Coordinate System\*\*](#) on Wikipedia.

The maximum value of the Latitude is  $90^\circ$  while the minimum is  $-90^\circ$ . Similarly, the maximum value of longitude is  $180^\circ$  while the minimum is  $-180^\circ$ .

*As we know, Earth isn't exactly a sphere but it is an ellipsoid. But for the sake of simplicity, we are going to assume it is a simple sphere of constant uniform radius. Later, towards the ending of this tutorial, we are going to talk about some of the issues with MySQL regarding this same exact problem.*

Moving on, the  **$0^\circ$  Latitude** line goes through the equator of the planet dividing the globe into northern and southern hemispheres. The  **$0^\circ$  Longitude line** AKA **international prime meridian**, however, goes through **British Royal Observatory** in **Greenwich**, England.

Finding the distance between two locations (points) on the globe (sphere) is not as easy as it was in the 2D coordinate system. In this case, we need to use [\*\*Haversine formula\*\*](#) which is used to find the distance between two points on the sphere using **latitude, longitude of the points** and **radius of the sphere**. We can use the same formula to find the shortest distance between two locations on the globe.

**geeksforgeeks.com** has provided some simple code snippets in different languages to calculate the distance between two places on the Earth based on Haversine formula, you can check it out from [here](#).

 *But the important fact to remember here is that Haversine formula is for a sphere and as we know, that's not the case for the Earth. Hence, this can lead to some error which might not be acceptable for precision measurements.*

So far, we have seen two coordinate systems, **flat** and **spherical**. There can be many coordinate systems based on the shape, location of the origin or [how distance is measured from an origin](#). Hence a point on a surface can have a different meaning based on in which coordinate system it lies. Hence saying P(x,y) is just not enough.

Therefore, we need an extra identifier associated with point signature. This is where **SRID** comes into the picture. **SRID** or [\*\*Coordinate Reference System Identifier\*\*](#) is the unique **integer** associated with a coordinate system proposed by OGC. MySQL database comes with more than **5000** of such coordinate systems which you can check using below query

```
SELECT `srs_name`, `srs_id`  
FROM INFORMATION_SCHEMA.ST_SPATIAL_REFERENCE_SYSTEMS
```

OGC has provided two formats for creating a spatial object like Point, Line, Polygon etc. They are **WKT** (*well-known text*) and **WKB** (well-known binary). **WKT** for a point is 2D cartesian coordinates as simple as **string “POINT(X, Y)”** where X and Y are axial distances from the origin. While in spherical coordinate system X and Y of **POINT(X, Y)** is **latitude** and **longitude** value respectively. WKB is the binary representation of WKT, which will cover in upcoming topics in this tutorial. Read more about WKT, WKB specifications for

different geometries from [here](#).

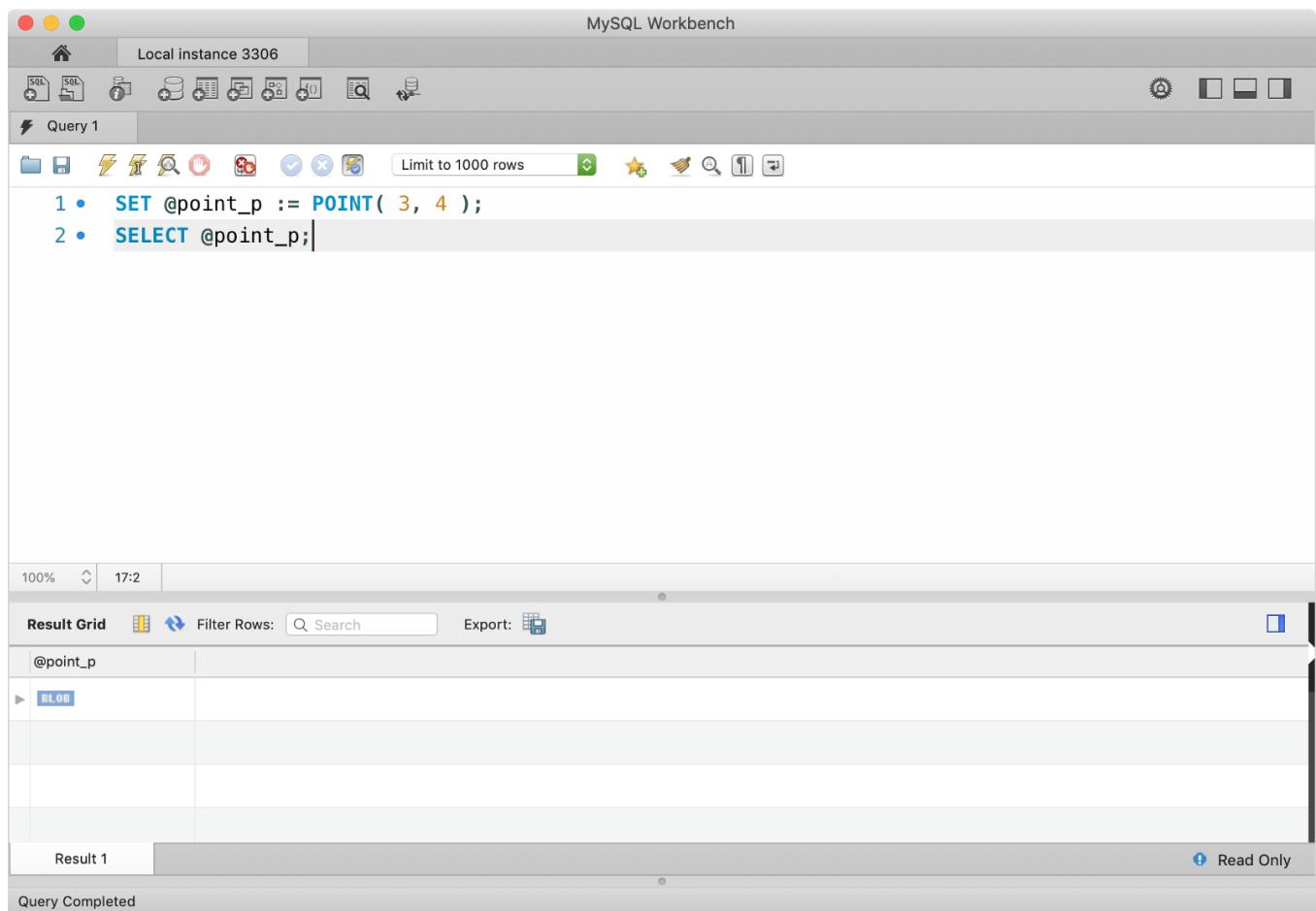
## Cartesian Coordinate System

Alright, enough with the theory. Let's now dive into the MySQL world and execute some queries to see things in real life. Initially, we will focus on the **2D cartesian coordinate system**.

SRID associated with the 2D cartesian coordinate system is **0**. This is default SRID associated with the `POINT` built-in function in MySQL.

*MySQL provides many built-in functions like `POINT`, `LINESTRING`, `POLYGON` etc. They are documented [here](#). Since, we are going to talk about `POINT` only, I thought I should mention this.*

Let's create a point with  $x = 3$  and  $y = 4$  on a coordinate system with **SRID 0** (*flat 2D cartesian coordinate system*). We will use `POINT` constructor function since its default **SRID** is **0**.



The screenshot shows the MySQL Workbench interface. The title bar says "MySQL Workbench" and "Local instance 3306". The toolbar has various icons for database management. The main window has a "Query 1" tab. The SQL editor contains the following code:

```
1 • SET @point_p := POINT( 3, 4 );
2 • SELECT @point_p;
```

The Result Grid below shows the output of the query:

@point_p
BLOB

At the bottom, it says "Result 1" and "Query Completed". There is a "Read Only" button at the bottom right.

In the above query, we created a local variable `point_p` which is set to point `(3, 4)` on the 2D coordinate plane (*SRID 0*). As we can see from the result, MySQL successfully created the point and saved in `BLOB` format. `BLOB` is the default storage format of MySQL for all the geometries (*do not get confused with WKB format, that is different, as we will see*).

Let's calculate the distance between the origin (*point 0,0*) and above point.

The screenshot shows the MySQL Workbench interface. The top menu bar says "MySQL Workbench" and "Local instance 3306". The toolbar has various icons for database management. Below the toolbar is a "Query 1" tab. The SQL editor contains the following code:

```
1 • SET @point_o := POINT( 0, 0 ); #origin
2 • SET @point_p := POINT( 3, 4 );
3 • SELECT ST_Distance( @point_o, @point_p );
```

The results grid below shows one row of data:

ST_Distance( @point_o, @point_p )
5

At the bottom of the results grid, it says "Result 3" and "Query Completed". There is also a "Read Only" button.

In the above query, we created another point `point_o` which is set to the origin point `(0,0)`. Then we used MySQL's built-in function `ST_Distance` to calculate the distance between two points. As expected, we got result **5**. Hence, our theory is working just fine until now.

*MySQL provides many built-in functions to test the relationship between two geometries (above, points). They are documented [here](#), but we will cover most of them in this tutorial. `ST_` prefix in these functions stands for **Spatial Type**. You can find list of all **Spatial Type** functions from [here](#).*

**!** *In newer version of MySQL (8.0+), old spatial functions like*

*GeomFromText, AsText has been deprecated and not longer works. Make sure to follow [\*\*this list\*\*](#) from here onwards if you are facing issues with undefined function names.*

In this tutorial, we are not going to use MySQL's built-in constructor functions to create geometries (like points) **as it is non-standard and restrict to SRID 0**, but instead, we will be using MySQL's built-in functions to create geometries from **WKT** specification, they are documented [here](#).

**ST\_GeomFromText** is a built-in function to create a geometry in given SRID from WKT specification. **ST\_PointFromText** can also be used to specifically state that we are creating **POINT** geometry but it's just an alias.

The syntax of **ST\_GeomFromText** is as below

```
ST_GeomFromText(geom_wkt, [srid], [options])
```

**geom\_wkt** is **WKT** representation of a geometry, **srid** is optional and defaults to 0 while **options** (*comma separated key=value*) argument is also optional.

*options is additional information provided to ST\_GeomFromText function. As of now, it supports only axis-order key with the value being either long-lat, lat-long or srid-defined(default). This will come up later when SRID will be different than 0.*

MySQL Workbench

Local instance 3306

Query 1

```
1 •  SET @point_o := ST_GeomFromText('POINT(0 0)'); #WKT specification
2 •  SET @point_p := ST_GeomFromText('POINT(3 4)'); #WKT specification
3 •  SELECT
4      ST_DISTANCE(@point_o, @point_p) AS distance,
5      ST_SRID(@point_o) AS _srid;
```

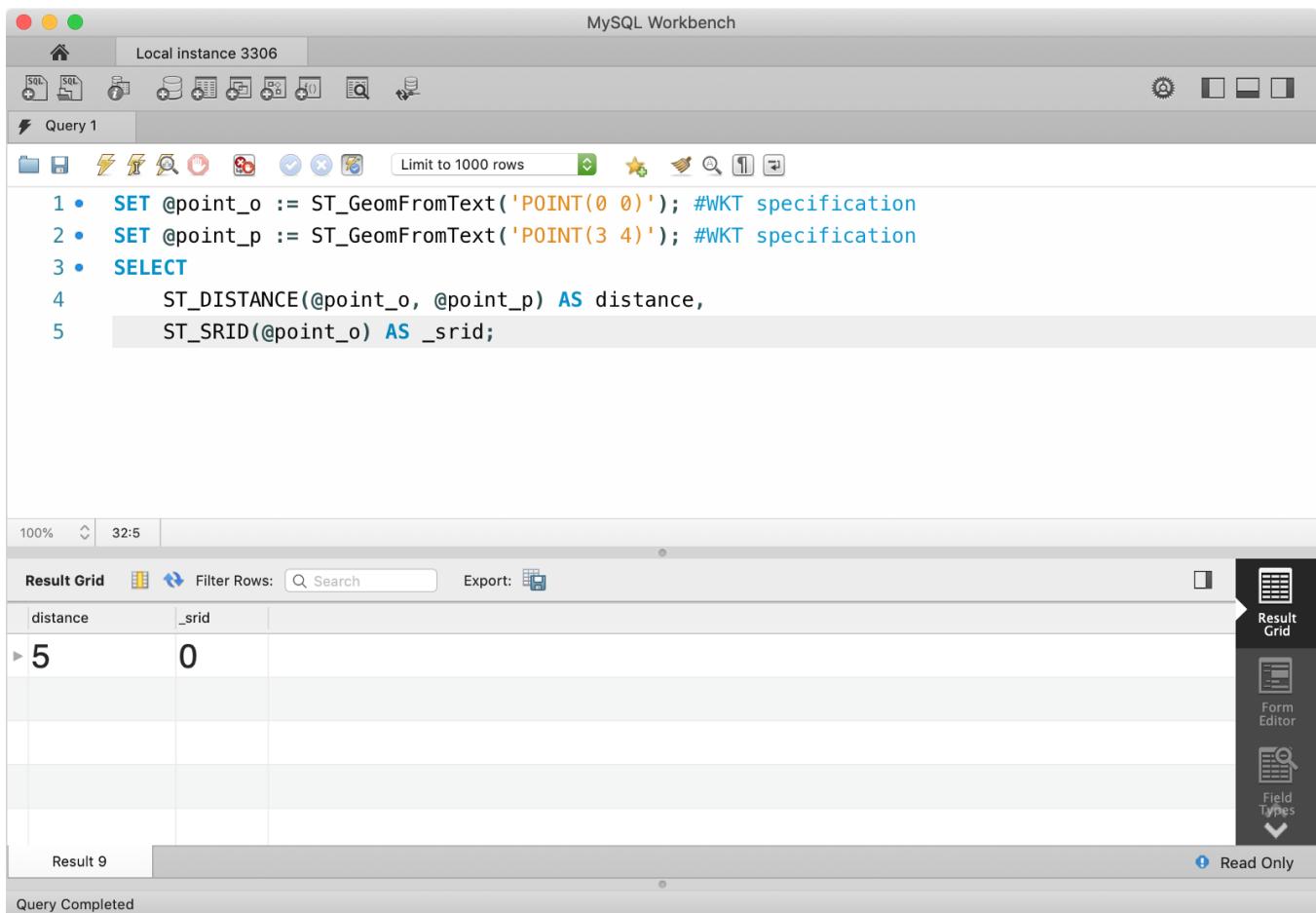
100% 32:5

Result Grid Filter Rows: Search Export:

distance	_srid
5	0

Result 9 Read Only

Query Completed



In the above query, we simply replaced the generation of `POINT` geometries from using MySQL's constructor function to creating geometries from WKT specifications. We can also verify that default SRID of `ST_GeomFromText` function is **0** using `ST_SRID` built-in function which expects a geometry and returns SRID associated with that geometry.

As we know, MySQL stores a geometry data type in `BLOB` format, but how can we make sure that the given point is correct if we can't read it. This can be solved using MySQL's built-in conversion functions, they are documented [here](#). Using these functions, we can convert geometries into WKT, WKB and JSON formats (*and vice-versa*).

`ST_AsText(g, [options])` function expects a geometry and optional `options` argument which is used to set axis-order (as described in `ST_GeomFromText`) and returns WKT format of a geometry. `ST_AsBinary` expects the same arguments as `ST_AsText` and returns WKB format of a geometry. `ST_AsGeoJson(g)` expects a geometry and returns coordinates in JSON.

MySQL Workbench

Local instance 3306

Query 1

```
1 •  SET @point_o := ST_GeomFromText('POINT(0 0)');  
2 •  
3 •  SELECT  
4      ST_AsText(@point_o) AS `wkt_value`,  
5      ST_AsBinary(@point_o) AS `wkb_value`,  
6      ST_AsGeoJson(@point_o) AS `geo_json_value`;  
7 •  
8
```

100% 47:1

Result Grid Filter Rows: Search Export:

wkt_value	wkb_value	geo_json_value
▶ POINT(0 0)	BLOB	{"type": "Point", "coordinates": [0.0, 0.0]}

Result 2 Read Only

Query Completed

The screenshot shows the MySQL Workbench interface. In the top-left, there's a toolbar with various icons. Below it is a menu bar with 'Query 1' selected. The main area contains a SQL editor with the following code:

```
1 •  SET @point_o := ST_GeomFromText('POINT(0 0)');  
2 •  
3 •  SELECT  
4      ST_AsText(@point_o) AS `wkt_value`,  
5      ST_AsBinary(@point_o) AS `wkb_value`,  
6      ST_AsGeoJson(@point_o) AS `geo_json_value`;  
7 •  
8
```

Below the code, the status bar shows '100%' and '47:1'. The results pane is titled 'Result Grid' and displays a single row of data:

wkt_value	wkb_value	geo_json_value
▶ POINT(0 0)	BLOB	{"type": "Point", "coordinates": [0.0, 0.0]}

The 'geo\_json\_value' column contains a JSON object representing a point at coordinates (0, 0). The 'wkb\_value' column is labeled 'BLOB'. The bottom right corner of the results pane has a 'Read Only' button.

In the above example, we have converted a geometric shape of the type **POINT** into readable **WKT** value, binary **WKB** value and **JSON** format.

If you have a **GeoJSON** of a geometry, you can convert it back a geometry object using **ST\_GeomFromGeoJSON(geo\_json\_string)** built-in function. You can also create geometry from WKB using **ST\_GeomFromWKB / ST\_PointFromWKB** function. Arguments for **ST\_GeomFromWKB** is the same as **ST\_GeomFromText** except geometry must be in **WKB** format.

MySQL Workbench

Local instance 3306

Query 1

```
1 • SET @point_o := ST_GeomFromText('POINT(0 0)'); # geometry from WKT
2 • SET @point_o_wkb := ST_AsBinary( @point_o ); # wkb of a geometry
3 • SET @point_o_from_wkb := ST_GeomFromWKB( @point_o_wkb ); # geometry from WKB
4 • SET @point_o_from_geoJSON := ST_GeomFromGeoJSON( '{"type": "Point", "coordinates": [0.0, 0.0]}' ); # geometry from geoJSON
5
6 • SELECT
7     ST_AsText(@point_o_from_wkb) AS `point_o_from_wkb`,
8     ST_AsText(@point_o_from_geoJSON) AS `point_o_from_geoJSON`;
9
10
```

Result Grid

point_o_from_wkb	point_o_from_geoJSON
POINT(0 0)	POINT(0 0)

Result 4

Query Completed

Read Only

If you want to get X and Y coordinates of a POINT geometry, you can use `ST_X(g)` and `ST_Y(g)` built-in functions.

MySQL Workbench

Local instance 3306

Query 1

```
1 • SET @point_o := ST_GeomFromText('POINT(3 4)'); # geometry from WKT
2
3 • SELECT
4     ST_X(@point_o) AS `point_o_x`,
5     ST_Y(@point_o) AS `point_o_y`;
6
7
```

Result Grid

point_o_x	point_o_y
3	4

Result 5

Query Completed

Read Only

Let's create a `locations_flat` database table and populate it with some sample geometry POINT values.

```

CREATE TABLE IF NOT EXISTS `locations_flat`(
  `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `name` VARCHAR(100),
  `position` POINT NOT NULL SRID 0
);

```

```
SHOW COLUMNS FROM `locations_flat`;
```

The screenshot shows the MySQL Workbench interface with the following details:

- Title Bar:** MySQL Workbench - Local instance 3306
- Toolbar:** Standard MySQL Workbench toolbar with various icons for database management.
- Query Editor:**
  - Query 1 tab is selected.
  - SQL tab is active.
  - Code:
 

```

1 CREATE TABLE IF NOT EXISTS `locations_flat`(
2   `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
3   `name` VARCHAR(100),
4   `position` POINT NOT NULL SRID 0
5 );
6
7 SHOW COLUMNS FROM `locations_flat`;

```
- Result Grid:**

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
name	varchar(100)	YES		NULL	
position	point	NO		NULL	
- Right Panel:** Shows tabs for Result Grid, Form Editor, and Field Types.
- Status Bar:** 100% zoom, 36:7 ratio, Read Only mode.

You can use POINT, POLYGON, LINESTRING and other geometrical types (*mentioned [here](#)*) to explicitly state the data type. MySQL also provides GEOMETRY data type if your column contains variable geometry types. SRID is optional, if omitted, the column can contain geometries of multiple SRIDs. But this is not recommended as this will not utilize the INDEX and query performance on large data set will be slower, as we will see later.

To insert some sample data, use below query

```

INSERT INTO `locations_flat`(`name`, `position`)
VALUES
( 'point_1', ST_GeomFromText( 'POINT( 1 1 )', 0 ) ),

```

```
( 'point_2', ST_GeomFromText( 'POINT( 2 2 )', 0 ) ),
( 'point_3', ST_GeomFromText( 'POINT( 3 3 )', 0 ) );
```

But since we need a large data set, let's take help of JavaScript. Inside your JavaScript console, execute below code which will print above insert statement but with 1000 entry points.

```
var fn = `ST_GeomFromText`;
var values = new Array( 1000 ).fill(null).map( ( val, index ) => {
  var id = index + 1;
  return `('point_${ id }', ${fn}('POINT(${ id } ${ id + 1 })'))`;
} ).join(',\n\t');
```

```
console.log(`  
  INSERT INTO `locations_flat`(`name`, `position`)  
  VALUES  
  ${ values };  
`);
```

Let's see a few data points in our table with a quick SELECT query.

```
SELECT
  *, ST_ASTEXT(`position`) AS `pos_wkt`
FROM
  `locations_flat`
LIMIT 10;
```

```

1 •  SELECT
2      *, ST_AsText(`position`) AS `pos_wkt`
3  FROM
4      `locations_flat`
5  LIMIT 10;

```

id	name	position	pos_wkt
1	point_1	BLOB	POINT(1 2)
2	point_2	BLOB	POINT(2 3)
3	point_3	BLOB	POINT(3 4)
4	point_4	BLOB	POINT(4 5)
5	point_5	BLOB	POINT(5 6)
6	point_6	BLOB	POINT(6 7)
7	point_7	BLOB	POINT(7 8)
8	point_8	BLOB	POINT(8 9)
9	point_9	BLOB	POINT(9 10)
10	point_10	BLOB	POINT(10 11)
11	point_11	BLOB	POINT(11 12)
12	point_12	BLOB	POINT(12 13)

Result 12      Read Only

Query Completed

*It is perfectly legal to pass a column name to `ST_` built-in function as long as its input parameters are legal. Also, all `ST_` function names are case-insensitive. Hence, you can also use `ST_AsText` as `ST_ASTEXT` or `st_asText`.*

Let's consider these points as places and our user is located at the `origin(0,0)`. If we want to search for places which are within 100 unit distance from him (*let's say the user is male*), we need to calculate the distance of each place and compare if it is less than or equal to 100. We have used distance function before, which is `ST_Distance`. We are going to use this function in `WHERE` clause in our MySQL query.

```

SET @user_location = ST_GeomFromText( 'POINT(0 0)' );

```

---

```

SELECT
    *,
    ST_AsText(`position`) AS `pos_wkt`,
    ST_Distance(`position`, @user_location) AS `distance`
FROM
    `locations_flat`
WHERE ST_Distance(`position`, @user_location) <= 100;

```

From the above query, we created a sample origin geometry POINT at (0,0) and used a reference point to calculate the distance of each point. WHERE clause filtered the results based on distance and we got around 70 results.

The screenshot shows the MySQL Workbench interface with a query editor and a results grid. The query is:

```

1 •  SET @user_location = ST_GeomFromText( 'POINT(0 0)' );
2 •  SELECT
3      *,
4      ST_AsText(`position`) AS `pos_wkt`,
5      ST_Distance(`position`, @user_location) AS `distance`
6  FROM
7      `locations_flat`
8  WHERE ST_Distance(`position`, | @user_location) <= 100;

```

The results grid displays the following data:

id	name	position	pos_wkt	distance
64	point_64	POINT	POINT(64 65)	91.21951545584969
65	point_65	POINT	POINT(65 66)	92.63368717696602
66	point_66	POINT	POINT(66 67)	94.04786015641186
67	point_67	POINT	POINT(67 68)	95.46203433826454
68	point_68	POINT	POINT(68 69)	96.87620966986684
69	point_69	POINT	POINT(69 70)	98.29038610159185
70	point_70	POINT	POINT(70 71)	99.70456358662827

Result 16      Read Only

Query Completed

But we can also achieve this using `ST_Buffer(g, d)` built-in function ([documented here](#)) to create geometry (as stated in the documentation) that represents all points whose distance from the geometry value `g` is less than or equal to a distance of `d`. In nutshell, it creates a **circular surface area**.

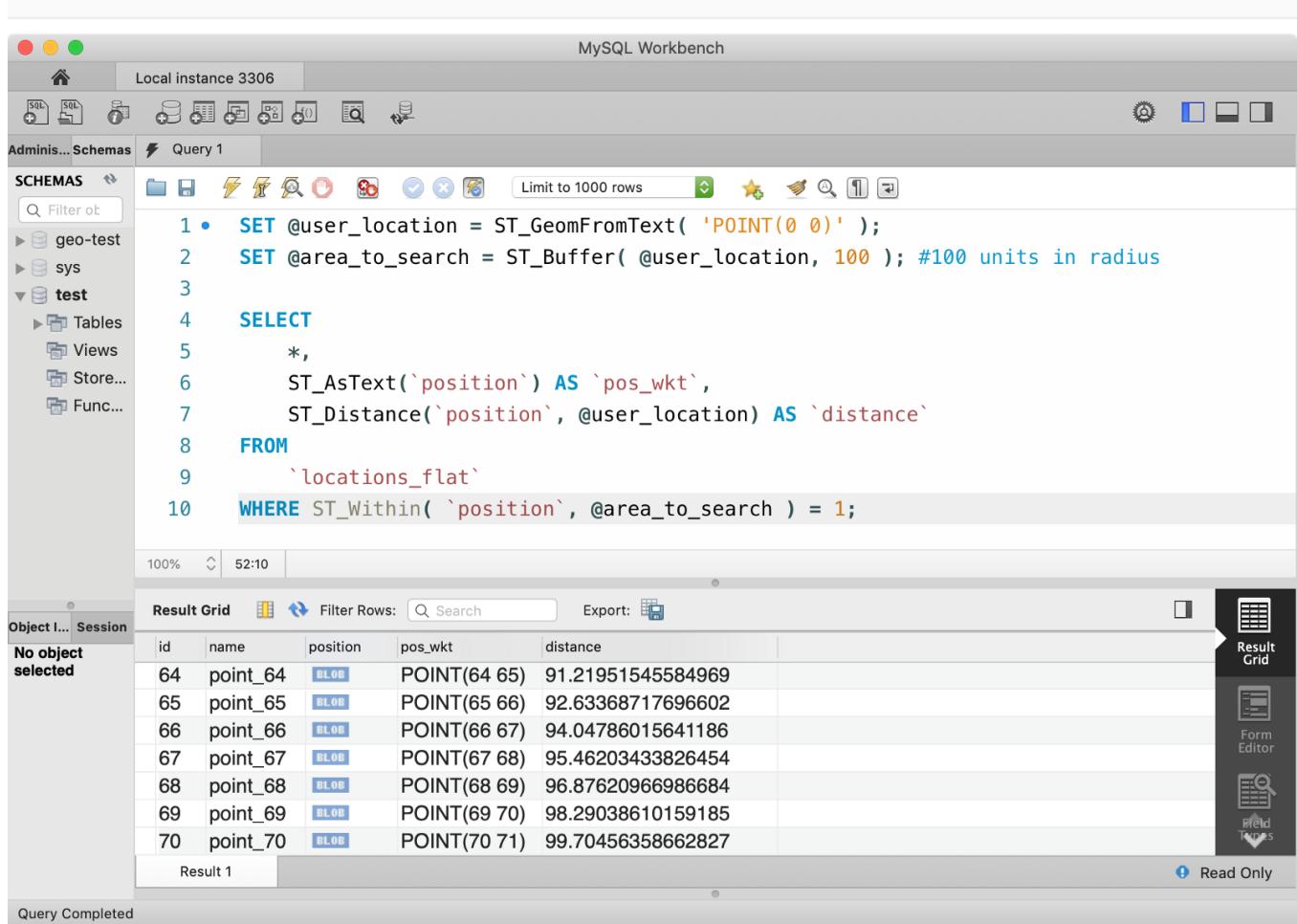
**!** It should be noted that `ST_Buffer` can only create geometry for **SRID 0** (plane surface). Since, `g` in our case has `SRID = 0`, hence this works for us. But a circular area on a spherical surface is rather complex, MySQL just gives up there.

Using this geometry, we can check whether a geometry (point) lies inside it or not. This can be done using `ST_Contains(g1, g2)` function which returns **1** if the geometry `g1` contains `g2`, else **0**. Otherwise, we can use `ST_Within(g1, g2)`

function which returns 1 if g1 is within g2, which is quite opposite of ST\_Contains function but works for us as well.

```
SET @user_location = ST_GeomFromText( 'POINT(0 0)' );
SET @area_to_search = ST_Buffer( @user_location, 100 );
```

```
SELECT
  *,
  ST_AsText(`position`) AS `pos_wkt`,
  ST_Distance(`position`, @user_location) AS `distance`
FROM
  `locations_flat`
WHERE ST_Within( `position`, @area_to_search ) = 1;
```



The screenshot shows the MySQL Workbench interface. The top bar displays "MySQL Workbench" and "Local instance 3306". The left sidebar shows the database structure with a schema named "test" selected. The main area contains the SQL query:

```
1 •  SET @user_location = ST_GeomFromText( 'POINT(0 0)' );
2  SET @area_to_search = ST_Buffer( @user_location, 100 ); #100 units in radius
3
4  SELECT
5    *,
6    ST_AsText(`position`) AS `pos_wkt`,
7    ST_Distance(`position`, @user_location) AS `distance`
8  FROM
9    `locations_flat`
10 WHERE ST_Within( `position`, @area_to_search ) = 1;
```

The results grid below shows the output of the query:

id	name	position	pos_wkt	distance
64	point_64	BLOB	POINT(64 65)	91.21951545584969
65	point_65	BLOB	POINT(65 66)	92.63368717696602
66	point_66	BLOB	POINT(66 67)	94.04786015641186
67	point_67	BLOB	POINT(67 68)	95.46203433826454
68	point_68	BLOB	POINT(68 69)	96.87620966986684
69	point_69	BLOB	POINT(69 70)	98.29038610159185
70	point_70	BLOB	POINT(70 71)	99.70456358662827

The status bar at the bottom indicates "Query Completed".

Let's talk about performance now. We all know about MySQL Indexes. If not then there is a lot of literature about it on the internet which you can read. Basically, MySQL maintains an index of a table with rows being ordered according to column values in the [B-Tree index](#). Whenever we try to search something from the table, MySQL looks into the index and uses nodes created by **B-Tree** to shortened the search path. An index of a table won't be created unless we tell it (*in*

some case, MySQL does that on its own, like **Primary Key**).

In the case of Spatial Data, MySQL uses **SPATIAL INDEX** which is for complex multi-dimensional data values and it maintains this index in [R-Tree](#) tree structure. Understanding how it works is not very important at this moment.

Let's first add **SPATIAL INDEX** on the table `locations_flat`. You can do this in three ways. You can add index while creation of the table.

```
CREATE TABLE IF NOT EXISTS `locations_flat`(
    `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    `name` VARCHAR(100),
    `position` POINT NOT NULL SRID 0,
    SPATIAL INDEX(`position`)
);
```

You can also alter the table and add **SPATIAL INDEX**.

```
ALTER TABLE `locations_flat` ADD SPATIAL INDEX(`position`);
```

Or you can add index manually using `CREATE SPATIAL INDEX` statement.

```
CREATE SPATIAL INDEX position_index ON locations_flat(`position`);
```

 A spatial index can only be created on a column with geometry type. It must be a NOT NULL column and should contain data of only one SRID.

Once, **SPATIAL INDEX** is created, you can verify indexes on the table using query `SHOW INDEXES FROM `locations_flat`;`

MySQL Workbench

Local instance 3306

Query 1

SHOW INDEXES FROM `locations\_flat`;

Result Grid

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
locations_flat	0	PRIMARY	1	id	A	1000	NULL	NULL	NULL	BTREE
locations_flat	1	position	1	position	A	1000	32	NULL	NULL	SPATIAL

Result 3      Read Only

Query Completed

Let's see how our earlier location search example is doing without using SPATIAL INDEX (ignore index).

```
SET @user_location = ST_GeomFromText( 'POINT(0 0)' );
SET @area_to_search = ST_Buffer( @user_location, 100 );
EXPLAIN SELECT
    *,
    ST_AsText(`position`) AS `pos_wkt`,
    ST_Distance(`position`, @user_location) AS `distance`
FROM
    `locations_flat`
IGNORE INDEX(`position`)
WHERE ST_Within( `position`, @area_to_search ) = 1;
```

MySQL Workbench

Local instance 3306

Query 1

```

1   SET @user_location = ST_GeomFromText( 'POINT(0 0)' );
2 • SET @area_to_search = ST_Buffer( @user_location, 100 );
3 • EXPLAIN SELECT
4
5     *,
6     ST_AsText(`position`) AS `pos_wkt`,
7     ST_Distance(`position`, @user_location) AS `distance`
8 FROM
9     `locations_flat`
10 IGNORE INDEX(`position`)
11 WHERE ST_Within( `position` , @area_to_search ) = 1;

```

Result Grid

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	locations_flat	NULL	ALL	NULL	NULL	NULL	NULL	1000	100.00	Using where

Result 8      Read Only

Query Completed

Seems like MySQL is using **full-table scan** and results show that 1000 rows are being searched for the match. Also, `possible_keys` are empty which shows that no index was used to perform the search.

Let's now use the index to find locations, we are going to use statement `USE INDEX` but that is optional as `position` column by-default uses the index.

```

SET @user_location = ST_GeomFromText( 'POINT(0 0)' );
SET @area_to_search = ST_Buffer( @user_location, 100 );
EXPLAIN SELECT
*
ST_AsText(`position`) AS `pos_wkt`,
ST_Distance(`position`, @user_location) AS `distance`
FROM
`locations_flat`
USE INDEX(`position`)
WHERE ST_Within( `position` , @area_to_search ) = 1;

```

MySQL Workbench

Local instance 3306

Query 1

```

1   SET @user_location = ST_GeomFromText( 'POINT(0 0)' );
2 • SET @area_to_search = ST_Buffer( @user_location, 100 );
3 • EXPLAIN SELECT
4     *,
5         ST_AsText(`position`) AS `pos_wkt`,
6         ST_Distance(`position`, @user_location) AS `distance`
7 FROM
8     `locations_flat`
9 USE INDEX(`position`)
10 WHERE ST_Within( `position` , @area_to_search ) = 1;

```

Result Grid

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	locations_flat	NULL	ALL	NULL	NULL	NULL	NULL	1000	100.00	Using where

Result 8      Read Only

Query Completed

Aha! We see no difference. Don't worry, this is a **⚠️ documented MySQL bug**. When we compare the return value of a spatial function in WHERE clause (*like we did with `ST_Within( `position` , @area_to_search ) = 1;`*, MySQL ignores the index. Since `ST_Within` returns **1** or **0**, it's perfectly safe to ignore `=1` in the statement and WHERE clause will only select value which returns **1**.

The screenshot shows the MySQL Workbench interface with a query editor and an explain grid.

```

1 SET @user_location = ST_GeomFromText( 'POINT(0 0)' );
2 • SET @area_to_search = ST_Buffer( @user_location, 100 );
3 • EXPLAIN SELECT
4     *,
5         ST_AsText(`position`) AS `pos_wkt`,
6         ST_Distance(`position`, @user_location) AS `distance`
7     FROM
8         `locations_flat`
9     USE INDEX(`position`)
10    WHERE ST_Within( `position` , @area_to_search );

```

The EXPLAIN output shows:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	locations_flat	NULL	range	position	position	34	NULL	53	100.00	Using where

Result Grid: Result 9 | Read Only

Query Completed

Yess! Now we can see that only 53 rows were scanned to return the result and `possible_keys` column shows that `position` index was used for the search.

**⚠️** *In my experience, I found that in case of a search on small data set, MySQL ignores the index. So, don't worry if your EXPLAIN statement doesn't seem ok.*

## Geographic Coordinate System

We talked a lot about 2D Cartesian Coordinate System, but I know, you must be thinking what's the use of that. Basically, it helped us understand how we can use MySQL's powerful spatial features and use spatial indexes to improve our query performance.

Now, it's time to come back to Earth and focus on Geographic data points. Before we begin, let me tell you a bitter truth. In MySQL, Earth has been approximated as a sphere with a mean radius equal to **6,370,986** meters. This may create some errors while calculating distances but works in the approximated world. Also, SRID for the geographic coordinate system is **4326**, hence any `POINT` we are

going to create must use 4326 SRID.

Let's create a table `locations_sphere` with **SPATIAL INDEX** and populate it with some sample data points.

```
CREATE TABLE IF NOT EXISTS `locations_earth`(
    `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    `name` VARCHAR(100),
    `position` POINT NOT NULL SRID 4326,
    SPATIAL INDEX(`position`)
);
SHOW COLUMNS FROM `locations_earth`;
```

The screenshot shows the MySQL Workbench interface. In the SQL tab, the following code is visible:

```
1 • CREATE TABLE IF NOT EXISTS `locations_earth`(
2     `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
3     `name` VARCHAR(100),
4     `position` POINT NOT NULL SRID 4326,
5     SPATIAL INDEX(`position`)
6 );
7 • SHOW COLUMNS FROM `locations_earth`;
```

Below the SQL tab, the Results tab displays the column information for the `locations_earth` table:

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
name	varchar(100)	YES		NULL	
position	point	NO	MUL	NULL	

The status bar at the bottom indicates "Query Completed".

Let's populate the `locations_earth` table with sample geometry points with SRID 4326. We are going to take help of JavaScript here.

```
var fn = `ST_GeomFromText`;
var values = new Array( 1000 ).fill(null).map( ( val, index ) => {
    var id = index + 1;
    var lat = (90/1000) * id; // 0-90 deg latitude
    var long = (180/1000) * id; // 0-180 deg longitude
```

```

    return `('point_${ id }', ${fn}('POINT(${ lat.toFixed(5) } ${ long.toFixed(5) })', 4326))`;
} ).join(',\n\t');
console.log(`

  INSERT INTO `locations_earth`(`name`, `position`)
  VALUES
  ${ values };
`);

```

In the above query generator, we made sure that each geometry point on the sphere should not exceed 90 degree latitude and 180 degrees longitude, though all our points are in the northern-eastern hemisphere. Above code will produce a query like below.

```

INSERT INTO `locations_earth`(`name`, `position`) VALUES
('point_1', ST_GeomFromText('POINT(0.09000 0.18000)', 4326)),
('point_2', ST_GeomFromText('POINT(0.18000 0.36000)', 4326)),
('point_3', ST_GeomFromText('POINT(0.27000 0.54000)', 4326));

```

As per the OpenGIS specification by OGC, WKT for `POINT` geometry is `POINT(lat, long)`. In our case, `ST_GeomFromText` function returns a geometry `POINT` with SRID 4326, hence our data points truly belong to spherical coordinates or geographic coordinate system.

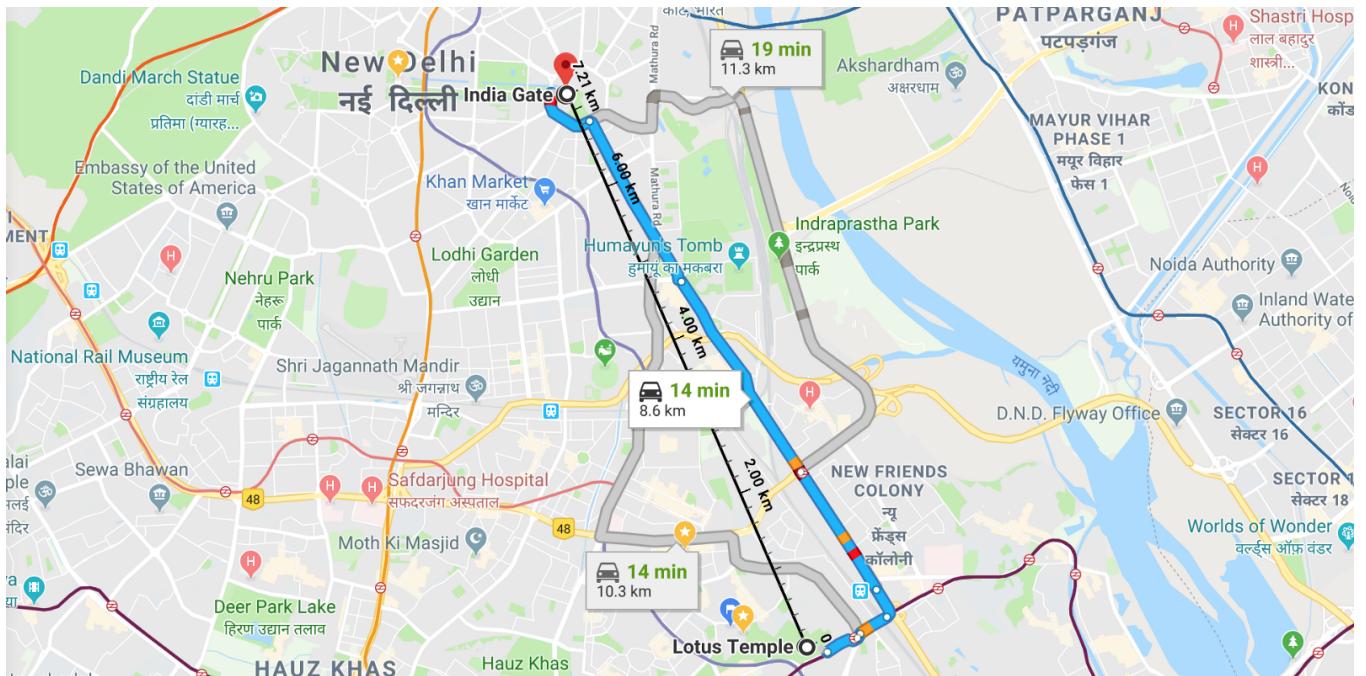
Let's calculate the distance between two points on the globe. As we seen in the case of the 2D surface, we used `ST_Distance` function to calculate the distance between two points. But in the case of a spherical coordinate system, we are going to use `ST_Distance_Sphere` function. The signature of `ST_Distance_Sphere` function is as below.

```
ST_Distance_Sphere(g1, g2 [, radius])
```

Here, `g1` and `g2` are geometries on the sphere, like `POINTS` in our case. `radius` argument is optional and defaults to the radius of the earth. This function returns the shortest distance on the sphere (earth) in meters.

For the test and accuracy test of this function, let's pick two random locations on the google map. Since I live in Delhi - India, I am going to pick **Lotus Temple** (28.553298, 77.259221) and **India Gate** (28.612849, 77.229883).

You can find (*latitude, longitude*) combination of a location on google map by right-clicking on a place and selecting what's here?



The distance between these two places is 7.21km or 7210 meters.

```
SET @lotus_temple := ST_GeomFromText( 'POINT(28.553298 77.259221)', 4326,  
'axis-order=lat-long' );  
SET @india_gate := ST_GeomFromText( 'POINT(28.612849 77.229883)', 4326 );
```

```
SELECT  
    ST_Latitude( @lotus_temple ) AS `lat_lotus_temple`,  
    ST_Longitude( @lotus_temple ) AS `long_lotus_temple`,  
    ST_Latitude( @india_gate ) AS `lat_india_gate`,  
    ST_Longitude( @india_gate ) AS `long_india_gate`,  
    ST_Distance_Sphere( @lotus_temple, @india_gate ) AS `distance`;
```

The screenshot shows the MySQL Workbench interface with a query editor and a results grid. The query is:

```
1 • SET @lotus_temple := ST_GeomFromText( 'POINT(28.553298 77.259221)', 4326, 'axis-order=lat-long' );
2 • SET @india_gate := ST_GeomFromText( 'POINT(28.612849 77.229883)', 4326 );
3
4 • SELECT
5     ST_Latitude( @lotus_temple ) AS `lat_lotus_temple`,
6     ST_Longitude( @lotus_temple ) AS `long_lotus_temple`,
7     ST_Latitude( @india_gate ) AS `lat_india_gate`,
8     ST_Longitude( @india_gate ) AS `long_india_gate`,
9     ST_Distance_Sphere( @lotus_temple, @india_gate ) AS `distance`;
```

The results grid shows one row of data:

lat_lotus_temple	long_lotus_temple	lat_india_gate	long_india_gate	distance
28.553298	77.259221	28.612849	77.229883	7214.858599756361

Query Completed

From the above result, we can see that Google and we are not that far off. Google measured distance **7210** meters while we measured it to be **7214** meters (*difference of 4 meters*). As we know, both of these values certainly can't be used in sensitive studies but it's pretty damn close.

As you can see in the above query, we have used `options` argument of `ST_GeomFromText` function to explicitly provide information about the axis order of `POINT` geometry in **WKT**. Also, we can use `ST_Latitude` and `ST_Longitude` to get **latitude** and **longitude** value geometry types of 4326 SRID.

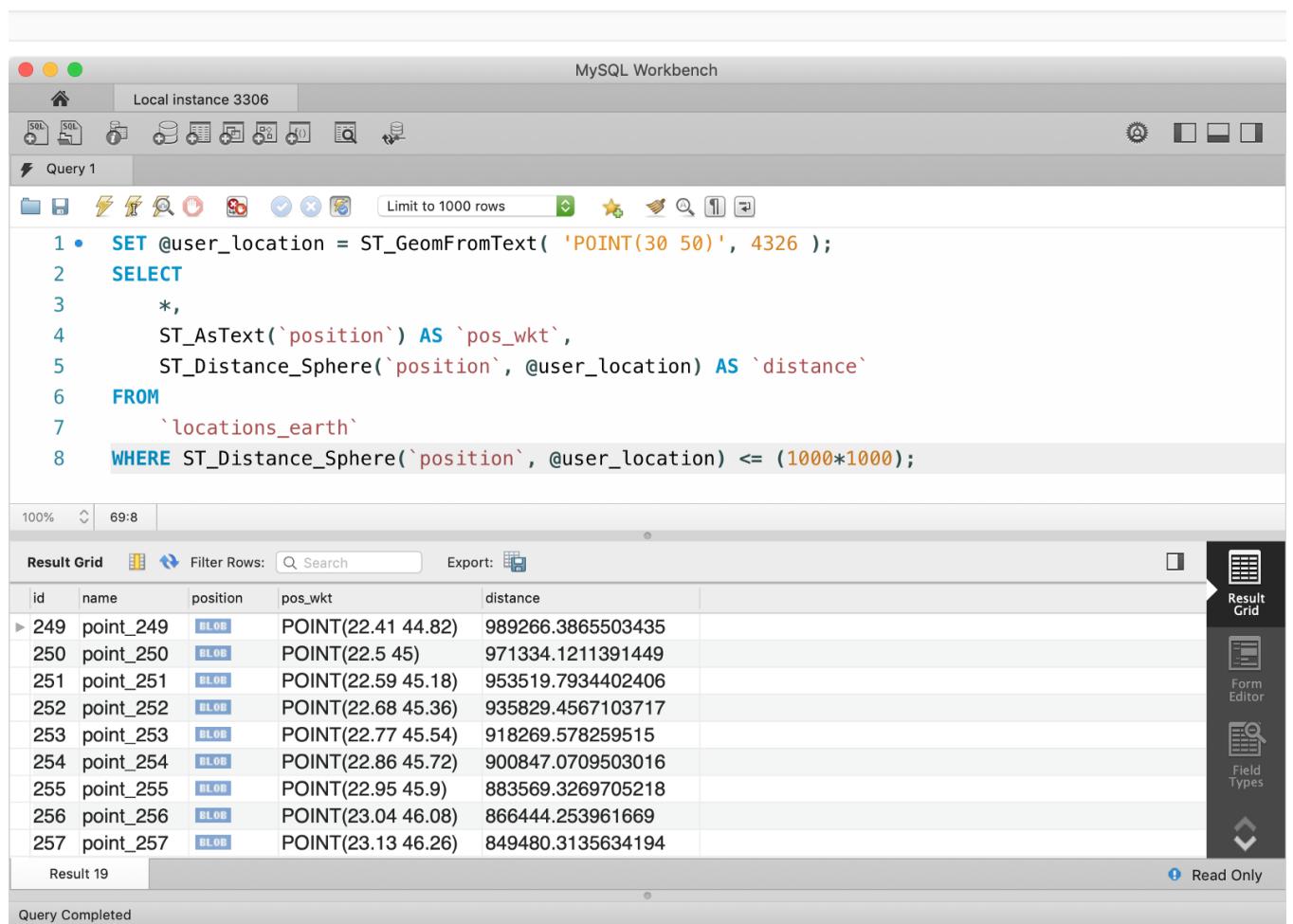
Now, let's focus on the application side. As in earlier examples of 2D flat surface, we calculated points inside a given radius using `ST_Buffer` circular area geometry. **But unfortunately, MySQL does not support creating circular area geometry in any other SRID besides 0**. Also, we can not transform any geometry from SRID 0 to other coordinate systems using the [`ST\_Transform`](#) function. Seems like our luck is running out.

But we can use a simple but expensive approach to calculate distances of each point on the sphere from the reference point and check if they are within a given

radius, like we did earlier.

Let's say that we are looking for places in 1000km ( $1000 * 1000$ m) radius from `location(30, 50)`. Here 30 is latitude and 50 is longitude. Using same old SQL query, we get below result.

```
SET @user_location = ST_GeomFromText( 'POINT(30 50)', 4326 );
SELECT
  *,
  ST_AsText(`position`) AS `pos_wkt`,
  ST_Distance_Sphere(`position`, @user_location) AS `distance`
FROM
  `locations_earth`
WHERE ST_Distance_Sphere(`position`, @user_location) <= (1000*1000);
```



The screenshot shows the MySQL Workbench interface with the following details:

- Toolbar:** Includes icons for Home, Local instance 3306, and various database management functions.
- Query Editor:** Labeled "Query 1". It contains the SQL query provided above, with syntax highlighting for keywords and comments.
- Result Grid:** A table showing the results of the query. The columns are: id, name, position, pos\_wkt, and distance. The data includes approximately 87 rows of location data.
- Result Grid Headers:** The columns are labeled id, name, position, pos\_wkt, and distance.
- Result Grid Data:** The first few rows of data are as follows:

id	name	position	pos_wkt	distance
249	point_249	blob	POINT(22.41 44.82)	989266.3865503435
250	point_250	blob	POINT(22.5 45)	971334.1211391449
251	point_251	blob	POINT(22.59 45.18)	953519.7934402406
252	point_252	blob	POINT(22.68 45.36)	935829.4567103717
253	point_253	blob	POINT(22.77 45.54)	918269.578259515
254	point_254	blob	POINT(22.86 45.72)	900847.0709503016
255	point_255	blob	POINT(22.95 45.9)	883569.3269705218
256	point_256	blob	POINT(23.04 46.08)	866444.253961669
257	point_257	blob	POINT(23.13 46.26)	849480.3135634194

- Bottom Status Bar:** Shows "Result 19" and "Read Only".
- Message Bar:** Shows "Query Completed".

We have found around 87 results but let's see how our query is doing.

The screenshot shows the MySQL Workbench interface. In the top-left pane, there is a query editor window titled "Query 1" containing the following SQL code:

```

1 • SET @user_location = ST_GeomFromText( 'POINT(30 50)', 4326 );
2 EXPLAIN SELECT
3     *,
4     ST_AsText(`position`) AS `pos_wkt`,
5     ST_Distance_Sphere(`position`, @user_location) AS `distance`
6 FROM
7     `locations_earth`
8 USE INDEX(`position`)
9 WHERE ST_Distance_Sphere(`position`, @user_location) <= (1000*1000);

```

In the bottom-right pane, the "Result Grid" shows the EXPLAIN output for the query. The results are as follows:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	locations_earth	NULL	ALL	NULL	NULL	NULL	NULL	1000	100.00	Using where

Below the grid, the status bar indicates "Result 21" and "Query Completed".

So even with the SPATIAL INDEX on, the query is performing the **full-table scan**. Since we are comparing value returned by `ST_Distance_Sphere` function inside the `WHERE` clause, MySQL is ignoring the index.

The only solution is to ignore comparison in `WHERE` clause but since we can not create a circular geometry in a spherical coordinate system, the closest thing we have is `POLYGON` geometry type. To approximate a circle but not wasting too much memory, we can safely use 6 sided polygon.

Let's see if this works by enclosing a 4 sided polygon around user location. I know this is very inaccurate, but it will prove that we can do it.

MySQL Workbench

Local instance 3306

Query 1

```

1 • SET @poly_o := ST_GeomFromText( 'POLYGON(( 30 40, 40 50, 30 60, 20 50, 30 40 ))', 4326 );
2
3 • SET @user_location = ST_GeomFromText( 'POINT(30 50)', 4326 );
4 • EXPLAIN SELECT
5     *,
6     ST_AsText(`position`) AS `pos_wkt`,
7     ST_Distance_Sphere(`position`, @user_location) AS `distance`
8 FROM
9     `locations_earth`
10 WHERE ST_Within(`position`, @poly_o);

```

Result Grid

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	locations_earth	NULL	range	position	position	34	NULL	1	100.00	Using where

Result 25

Query Completed

Hurray! So this works. So far I haven't found a formula that can take proximity radius from a point on the globe and draw a polygon with n sides. I guess this will be a TODO and as soon as I find some solution, I will make sure to update this tutorial. But if you have any suggestions or know any resource material on the internet, please drop a comment.

**!** By the way, [Postgres](#) database has far better support for spacial geographic data calculation using [PostGIS extension](#). Just to mention one, unlike MySQL, PostGIS provides a built-in function `ST_DWithin` to check if two geometries are within the specified distance of one another, which can be used in `WHERE` clause and increase performance by utilizing the index.



- [ITNEXT](#)

ITNEXT is a platform for IT developers & software

engineers to share knowledge, connect, collaborate, learn and experience next-gen technologies.