CONTENT

[MySQL Spatial Data Types 3](#_Toc57705685)

[Features of MySQL Spatial Data Types 3](#_Toc57705686)

[MySQL supports a number of Spatial Data Types 3](#_Toc57705687)

[Geometry Type 4](#_Toc57705688)

[Point Type 5](#_Toc57705689)

[Curve Type 6](#_Toc57705690)

[LineString Type 6](#_Toc57705691)

[Surface Properties 7](#_Toc57705692)

[Polygon Type 7](#_Toc57705693)

[GeometryCollection Type 8](#_Toc57705694)

[MultiPoint Type 8](#_Toc57705695)

[MultiCurve Type 9](#_Toc57705696)

[MultiLineString Type 9](#_Toc57705697)

[MultiSurface Type 10](#_Toc57705698)

[MultiPolygon Type 10](#_Toc57705699)

[Creating a Spatially Enabled MySQL Database 11](#_Toc57705700)

[MySQL Spatial Data Types 11](#_Toc57705701)

[Creating Spatial Values 11](#_Toc57705702)

[Creating Geometry Values Using WKT Functions 12](#_Toc57705703)

[Creating Geometry Values Using WKB Functions 13](#_Toc57705704)

[Creating Geometry Values Using MySQL-Specific Functions 14](#_Toc57705705)

[Creating Spatial Columns 15](#_Toc57705706)

[Populating Spatial Columns 15](#_Toc57705707)

[Fetching Spatial Data 16](#_Toc57705708)

[Fetching Spatial Data in Internal Format 16](#_Toc57705709)

[Fetching Spatial Data in WKT Format 16](#_Toc57705710)

[Fetching Spatial Data in WKB Format 17](#_Toc57705711)

[Spatial Data Types 17](#_Toc57705712)

[The OpenGIS Geometry Model 19](#_Toc57705713)

[The Geometry Class Hierarchy 19](#_Toc57705714)

[Geometry Class 20](#_Toc57705715)

[Point Class 21](#_Toc57705716)

[Curve Class 21](#_Toc57705717)

[LineString Class 21](#_Toc57705718)

[Surface Class 22](#_Toc57705719)

[Polygon Class 22](#_Toc57705720)

[GeometryCollection Class 22](#_Toc57705721)

[MultiPoint Class 22](#_Toc57705722)

[MultiCurve Class 23](#_Toc57705723)

[MultiLineString Class 23](#_Toc57705724)

[MultiSurface Class 23](#_Toc57705725)

[MultiPolygon Class 23](#_Toc57705726)

[Supported Spatial Data Formats 24](#_Toc57705727)

[Well-Known Text (WKT) Format 24](#_Toc57705728)

[Well-Known Binary (WKB) Format 25](#_Toc57705729)

[Internal Geometry Storage Format 25](#_Toc57705730)

[Geometry Well-Formedness and Validity 26](#_Toc57705731)

[Spatial Reference System Support 27](#_Toc57705732)

[Creating Spatial Columns 28](#_Toc57705733)

[Populating Spatial Columns 28](#_Toc57705734)

[Fetching Spatial Data 29](#_Toc57705735)

[Optimizing Spatial Analysis 30](#_Toc57705736)

[Creating Spatial Indexes 30](#_Toc57705737)

[Using Spatial Indexes 31](#_Toc57705738)

[Using Geometry/Spatial Data Types in MySQL 33](#_Toc57705739)

[Cartesian Coordinate System 39](#_Toc57705740)

[Geographic Coordinate System 55](#_Toc57705741)

# MySQL Spatial Data Types

The Open Geospatial Consortium (OGC) is is an international consortium of more than 250 companies, agencies, and universities participating in the development of publicly available conceptual solutions that can be useful with all kinds of applications that manage spatial data.

The specification published by [Open Geospatial Consortium publishes (OGC)](http://www.opengeospatial.org/standards/sfs) specifies that how MySQL implements spatial extensions as a subset of the SQL with Geometry Types environment. This term refers to an SQL environment that has been extended with a set of geometry types. A geometry-valued SQL column is implemented as a column that has a geometry type. The specification describes a set of SQL geometry types, as well as functions on those types to create and analyze geometry values.

## **Features of MySQL Spatial Data Types**

MySQL spatial extensions enable the generation, storage, and analysis of geographic features:

* Data types for representing spatial values
* Functions for manipulating spatial values
* Spatial indexing for improved access times to spatial columns

## ****MySQL supports a number of Spatial Data Types****

MySQL has data types that correspond to OpenGIS classes. Some of these types hold single geometry values:

* GEOMETRY
* POINT
* LINESTRING
* POLYGON

The other data types hold collections of values:

* MULTIPOINT
* MULTILINESTRING
* MULTIPOLYGON
* GEOMETRYCOLLECTION

### ****Geometry Type****

Geometry is a word that denotes a geographic feature. Originally the word geometry meant measurement of the earth. Another meaning comes from cartography, referring to the geometric features that cartographers use to map the world. It is a noninstantiable class but has a number of properties, given below are common to all geometry values created from any of the Geometry subclasses.

|  |  |
| --- | --- |
| **Name** | **Description** |
| type | Each geometry belongs to one of the instantiable classes in the hierarchy. |
| SRID | The full form of SRID is Spatial Reference Identifier. This system describes the coordinate space in which the geometry object is defined. In MySQL, the SRID value is just an integer associated with the geometry value. |
| coordinates | All nonempty geometries include at least one pair of (X,Y) coordinates. Empty geometries contain no coordinates. Coordinates are related to the SRID. |
| interior, boundary, exterior. | Every geometry occupies some position in space. The exterior of a geometry is all space not occupied by the geometry. The interior is the space occupied by the geometry. The boundary is the interface between the geometry's interior and exterior. |
| MBR | Its MBR (minimum bounding rectangle), or envelope. This is the bounding geometry, formed by the minimum and maximum (X,Y) coordinates: |
| simple or nonsimple. | Whether the value is simple or nonsimple. Geometry values of types (LineString, MultiPoint, MultiLineString) are either simple or nonsimple. Each type determines its own assertions for being simple or nonsimple. |
| closed or not closed | Whether the value is closed or not closed. Geometry values of types (LineString, MultiString) are either closed or not closed. Each type determines its own assertions for being closed or not closed. |
| empty or nonempty | Whether the value is empty or nonempty A geometry is empty if it does not have any points. Exterior, interior, and boundary of an empty geometry are not defined. An empty geometry is defined to be always simple and has an area of 0. |
| dimension | Its dimension. A geometry can have a dimension of –1, 0, 1, or 2: – 1 for an empty geometry. 0 for a geometry with no length and no area. 1 for a geometry with nonzero length and zero area. 2 for a geometry with nonzero area. |

**Example**

Use the CREATE TABLE statement to create a table with a spatial column:

CREATE TABLE geotest (code int(5),descrip varchar(50), g GEOMETRY);

Copy

Here is the structure of the table:

Sample Output:

MySQL> describe geotest;

+---------+-------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+---------+-------------+------+-----+---------+-------+

| code | int(5) | YES | | NULL | |

| descrip | varchar(50) | YES | | NULL | |

| g | geometry | YES | | NULL | |

+---------+-------------+------+-----+---------+-------+

3 rows in set (0.01 sec)

Use the ALTER TABLE statement to add or drop a spatial column to or from an existing table:

ALTER TABLE geotest ADD pt\_loca POINT;

ALTER TABLE geotest DROP pt\_loca ;

### ****Point Type****

A Point is a geometry which represents a single location in coordinate space.

**Usage of Point**

On a city map, a Point object could represent a rail station.

**Point Properties**

* X-coordinate value.
* Y-coordinate value.
* Point is defined as a zero-dimensional geometry.
* The boundary of a Point is the empty set.

**Example**

MySQL> SELECT X(POINT(18, 23));

+------------------+

| X(POINT(18, 23)) |

+------------------+

| 18 |

+------------------+

1 row in set (0.00 sec)

MySQL> SELECT X(GeomFromText('POINT(18 23)'));

+---------------------------------+

| X(GeomFromText('POINT(18 23)')) |

+---------------------------------+

| 18 |

+---------------------------------+

1 row in set (0.00 sec)

### ****Curve Type****

A Curve is a one-dimensional geometry, in general, it represented by a sequence of points. Particular subclasses of Curve define the type of interpolation between points. The curve is a noninstantiable class.

**Curve Properties**

* A Curve has the coordinates of its points.
* A Curve is defined as a one-dimensional geometry.
* A Curve is simple if it does not pass through the same point twice.
* A Curve is closed if its start point is equal to its endpoint.
* The boundary of a closed Curve is empty.
* The boundary of a nonclosed Curve consists of its two endpoints.
* A Curve that is simple and closed is a LinearRing.

### ****LineString Type****

A LineString is a Curve with linear interpolation between points.

**Usage of LineString**

LineString objects could represent a river within a country map.

**LineString Properties**

* A LineString has coordinates of segments, defined by each consecutive pair of points.
* A LineString is a Line if it consists of exactly two points.
* A LineString is a LinearRing if it is both closed and simple.

**Example**

MySQL> SET @g = 'LINESTRING(0 0,1 2,2 4)';

Query OK, 0 rows affected (0.00 sec)

MySQL> INSERT INTO geotest VALUES (123,"Test Data",GeomFromText(@g));

Query OK, 1 row affected (0.00 sec)

**Surface Type**

A Surface is a two-dimensional geometry. It is a noninstantiable class. Its only instantiable subclass is Polygon.

### ****Surface Properties****

* A Surface is defined as a two-dimensional geometry.
* The OpenGIS specification defines a simple Surface as a geometry that consists of a single “patch” that is associated with a single exterior boundary and zero or more interior boundaries
* The boundary of a simple Surface is the set of closed curves corresponding to its exterior and interior boundaries.

### **Polygon Type**

A Polygon is a planar Surface representing a multisided geometry. It is defined by a single exterior boundary and zero or more interior boundaries, where each interior boundary defines a hole in the Polygon.

**Usage of Polygon**

The Polygon objects could represent districts, blocks and so on from a state map.

**Polygon Assertions**

* The boundary of a Polygon consists of a set of LinearRing objects (that is, LineString objects that are both simple and closed) that make up its exterior and interior boundaries.
* A Polygon has no rings that cross. The rings in the boundary of a Polygon may intersect at a Point, but only as a tangent.
* A Polygon has no lines, spikes, or punctures.
* A Polygon has an interior that is a connected point set.
* A Polygon may have holes. The exterior of a Polygon with holes is not connected. Each hole defines a connected component of the exterior.

**Example**

MySQL> SET @g = 'POLYGON((0 0,8 0,12 9,0 9,0 0),(5 3,4 5,7 9,3 7, 2 5))';

Query OK, 0 rows affected (0.00 sec)

MySQL> INSERT INTO geotest VALUES (123,"Test Data",GeomFromText(@g));

Query OK, 1 row affected (0.03 sec)

### ****GeometryCollection Type****

A GeometryCollection is a geometry that is a collection of one or more geometries of any class.

All the elements in a GeometryCollection must be in the same Spatial Reference System. There are no other constraints on the elements of a GeometryCollection, although the subclasses of GeometryCollection described in the following sections may restrict membership. Restrictions may be based on:

* Element type (for example, a MultiPoint may contain only Point elements)
* Dimension
* Constraints on the degree of spatial overlap between elements

**Example**

MySQL> SET @g ='GEOMETRYCOLLECTION(POINT(3 2),LINESTRING(0 0,1 3,2 5,3 5,4 7))';

Query OK, 0 rows affected (0.00 sec)

MySQL> INSERT INTO geotest VALUES (123,"Test Data",GeomFromText(@g));

Query OK, 1 row affected (0.00 sec)

### ****MultiPoint Type****

A MultiPoint is a geometry collection composed of Point elements. The points are not connected or ordered in any way.

**Usage of MultiPoint**

On a world map, a MultiPoint could represent a chain of small islands.

**MultiPoint Properties**

* A MultiPoint is a zero-dimensional geometry.
* A MultiPoint is simple if no two of its Point values are equal (have identical coordinate values).
* The boundary of a MultiPoint is the empty set.

### ****MultiCurve Type****

A MultiCurve is a geometry collection composed of Curve elements. MultiCurve is a noninstantiable class.

**MultiCurve Properties**

* A MultiCurve is a one-dimensional geometry.
* A MultiCurve is simple if and only if all of its elements are simple; the only intersections between any two elements occur at points that are on the boundaries of both elements.
* A MultiCurve boundary is obtained by applying the “mod 2 union rule” (also known as the “odd-even rule”): A point is the boundary of a MultiCurve if it is within the boundaries of an odd number of MultiCurve elements.
* A MultiCurve is closed if all of its elements are closed.
* The boundary of a closed MultiCurve is always empty.

MySQL> SET @g ='MULTIPOINT(0 0, 15 25, 45 65)';

Query OK, 0 rows affected (0.00 sec)

MySQL> INSERT INTO geotest VALUES (123,"Multipoint",GeomFromText(@g));

Query OK, 1 row affected (0.00 sec)

### ****MultiLineString Type****

A MultiLineString is a MultiCurve geometry collection composed of LineString elements.

**Usage of MultiLineString**

* On a region map, a MultiLineString could represent a river system or a highway system.

**Example**

MySQL> SET @g ='MULTILINESTRING((12 12, 22 22), (19 19, 32 18))';

Query OK, 0 rows affected (0.00 sec)

MySQL> INSERT INTO geotest VALUES (123,"Multistring",GeomFromText(@g));

Query OK, 1 row affected (0.00 sec)

### ****MultiSurface Type****

A MultiSurface is a geometry collection composed of surface elements. MultiSurface is a noninstantiable class. Its only instantiable subclass is MultiPolygon.

**MultiSurface Assertions**

* Two MultiSurface surfaces have no interiors that intersect.
* Two MultiSurface elements have boundaries that intersect at most at a finite number of points.

### ****MultiPolygon Type****

MultiPolygon is a MultiSurface object composed of Polygon elements.

**Usage of MultiPolygon**

A MultiPolygon could represent a system of lakes on a region map.

**MultiPolygon Assertions**

* A MultiPolygon has no two Polygon elements with interiors that intersect.
* A MultiPolygon has no two Polygon elements that cross (crossing is also forbidden by the previous assertion) or that touch at an infinite number of points.
* A MultiPolygon may not have cut lines, spikes, or punctures. A MultiPolygon is a regular, closed point set.
* A MultiPolygon that has more than one Polygon has an interior that is not connected. The number of connected components of the interior of a MultiPolygon is equal to the number of Polygon values in the MultiPolygon

**MultiPolygon Properties**

* A MultiPolygon is a two-dimensional geometry.
* A MultiPolygon boundary is a set of closed curves (LineString values) corresponding to the boundaries of its Polygon elements.
* Each Curve in the boundary of the MultiPolygon is in the boundary of exactly one Polygon element.
* Every Curve in the boundary of a Polygon element is in the boundary of the MultiPolygon.

**Example**

MySQL> SET @g ='MULTIPOLYGON(((0 0,11 0,12 11,0 9,0 0)),((3 5,7 4,4 7,7 7,3 5)))';

Query OK, 0 rows affected (0.00 sec)

MySQL> INSERT INTO geotest VALUES (123,"Multipolygon",GeomFromText(@g));

Query OK, 1 row affected (0.00 sec)

# Creating a Spatially Enabled MySQL Database

This section describes the data types you can use for representing spatial data in MySQL, and the functions available for creating and retrieving spatial values.

## MySQL Spatial Data Types

MySQL has data types that correspond to OpenGIS classes. Some of these types hold single geometry values:

* GEOMETRY
* POINT
* LINESTRING
* POLYGON

GEOMETRY can store geometry values of any type. The other single-value types, POINT and LINESTRING and POLYGON, restrict their values to a particular geometry type.

The other data types hold collections of values:

* MULTIPOINT
* MULTILINESTRING
* MULTIPOLYGON
* GEOMETRYCOLLECTION

GEOMETRYCOLLECTION can store a collection of objects of any type. The other collection types, MULTIPOINT and MULTILINESTRING and MULTIPOLYGON and GEOMETRYCOLLECTION, restrict collection members to those having a particular geometry type.

## Creating Spatial Values

This section describes how to create spatial values using Well-Known Text and Well-Known Binary functions that are defined in the OpenGIS standard, and using MySQL-specific functions.

## Creating Geometry Values Using WKT Functions

MySQL provides a number of functions that take as input parameters a Well-Known Text representation and, optionally, a spatial reference system identifier (SRID). They return the corresponding geometry.

GeomFromText() accepts a WKT of any geometry type as its first argument. An implementation also provides type-specific construction functions for construction of geometry values of each geometry type.

GeomCollFromText(*wkt*[,*srid*]), GeometryCollectionFromText(*wkt*[,*srid*]),

Constructs a GEOMETRYCOLLECTION value using its WKT representation and SRID.

GeomFromText(*wkt*[,*srid*]), GeometryFromText(*wkt*[,*srid*]),

Constructs a geometry value of any type using its WKT representation and SRID.

LineFromText(*wkt*[,*srid*]), LineStringFromText(*wkt*[,*srid*]),

Constructs a LINESTRING value using its WKT representation and SRID.

MLineFromText(*wkt*[,*srid*]), MultiLineStringFromText(*wkt*[,*srid*]),

Constructs a MULTILINESTRING value using its WKT representation and SRID.

MPointFromText(*wkt*[,*srid*]), MultiPointFromText(*wkt*[,*srid*]),

Constructs a MULTIPOINT value using its WKT representation and SRID.

MPolyFromText(*wkt*[,*srid*]), MultiPolygonFromText(*wkt*[,*srid*]),

Constructs a MULTIPOLYGON value using its WKT representation and SRID.

PointFromText(*wkt*[,*srid*])

Constructs a POINT value using its WKT representation and SRID.

PolyFromText(*wkt*[,*srid*]), PolygonFromText(*wkt*[,*srid*]),

Constructs a POLYGON value using its WKT representation and SRID.

The OpenGIS specification also describes optional functions for constructing Polygon or MultiPolygon values based on the WKT representation of a collection of rings or closed LineString values. These values may intersect. MySQL does not implement these functions:

BdMPolyFromText(*wkt*,*srid*)

Constructs a MultiPolygon value from a MultiLineString value in WKT format containing an arbitrary collection of closed LineString values.

BdPolyFromText(*wkt*,*srid*)

Constructs a Polygon value from a MultiLineString value in WKT format containing an arbitrary collection of closed LineString values.

## Creating Geometry Values Using WKB Functions

MySQL provides a number of functions that take as input parameters a BLOB containing a Well-Known Binary representation and, optionally, a spatial reference system identifier (SRID). They return the corresponding geometry.

GeomFromWKT() accepts a WKB of any geometry type as its first argument. An implementation also provides type-specific construction functions for construction of geometry values of each geometry type.

GeomCollFromWKB(*wkb*[,*srid*]), GeometryCollectionFromWKB(*wkt*[,*srid*]),

Constructs a GEOMETRYCOLLECTION value using its WKB representation and SRID.

GeomFromWKB(*wkb*[,*srid*]), GeometryFromWKB(*wkt*[,*srid*]),

Constructs a geometry value of any type using its WKB representation and SRID.

LineFromWKB(*wkb*[,*srid*]), LineStringFromWKB(*wkb*[,*srid*]),

Constructs a LINESTRING value using its WKB representation and SRID.

MLineFromWKB(*wkb*[,*srid*]), MultiLineStringFromWKB(*wkb*[,*srid*]),

Constructs a MULTILINESTRING value using its WKB representation and SRID.

MPointFromWKB(*wkb*[,*srid*]), MultiPointFromWKB(*wkb*[,*srid*]),

Constructs a MULTIPOINT value using its WKB representation and SRID.

MPolyFromWKB(*wkb*[,*srid*]), MultiPolygonFromWKB(*wkb*[,*srid*]),

Constructs a MULTIPOLYGON value using its WKB representation and SRID.

PointFromWKB(*wkb*[,*srid*])

Constructs a POINT value using its WKB representation and SRID.

PolyFromWKB(*wkb*[,*srid*]), PolygonFromWKB(*wkb*[,*srid*]),

Constructs a POLYGON value using its WKB representation and SRID.

The OpenGIS specification also describes optional functions for constructing Polygon or MultiPolygon values based on the WKB representation of a collection of rings or closed LineString values. These values may intersect. MySQL does not implement these functions:

BdMPolyFromWKB(*wkb*,*srid*)

Constructs a MultiPolygon value from a MultiLineString value in WKB format containing an arbitrary collection of closed LineString values.

BdPolyFromWKB(*wkb*,*srid*)

Constructs a Polygon value from a MultiLineString value in WKB format containing an arbitrary collection of closed LineString values.

## Creating Geometry Values Using MySQL-Specific Functions

**Note**: MySQL does not implement the functions listed in this section.

MySQL provides a set of useful functions for creating geometry WKB representations. The functions described in this section are MySQL extensions to the OpenGIS specifications. The results of these functions are BLOB values containing WKB representations of geometry values with no SRID. The results of these functions can be substituted as the first argument for any function in the GeomFromWKB() function family.

GeometryCollection(*g1*,*g2*,...)

Constructs a WKB GeometryCollection. If any argument is not a well-formed WKB representation of a geometry, the return value is NULL.

LineString(*pt1*,*pt2*,...)

Constructs a WKB LineString value from a number of WKB Point arguments. If any argument is not a WKB Point, the return value is NULL. If the number of Point arguments is less than two, the return value is NULL.

MultiLineString(*ls1*,*ls2*,...)

Constructs a WKB MultiLineString value using using WKB LineString arguments. If any argument is not a WKB LineString, the return value is NULL.

MultiPoint(*pt1*,*pt2*,...)

Constructs a WKB MultiPoint value using WKB Point arguments. If any argument is not a WKB Point, the return value is NULL.

MultiPolygon(*poly1*,*poly2*,...)

Constructs a WKB MultiPolygon value from a set of WKB Polygon arguments. If any argument is not a WKB Polygon, the rerurn value is NULL.

Point(*x*,*y*)

Constructs a WKB Point using its coordinates.

Polygon(*ls1*,*ls2*,...)

Constructs a WKB Polygon value from a number of WKB LineString arguments. If any argument does not represent the WKB of a LinearRing (that is, not a closed and simple LineString) the return value is NULL.

## Creating Spatial Columns

MySQL provides a standard way of creating spatial columns for geometry types, for example, with CREATE TABLE or ALTER TABLE. Currently, spatial columns are supported only for MyISAM tables.

* Use the CREATE TABLE statement to create a table with a spatial column:
* mysql> CREATE TABLE geom (g GEOMETRY);
* Query OK, 0 rows affected (0.02 sec)
* Use the ALTER TABLE statement to add or drop a spatial column to or from an existing table:
* mysql> ALTER TABLE geom ADD pt POINT;
* Query OK, 0 rows affected (0.00 sec)
* Records: 0 Duplicates: 0 Warnings: 0
* mysql> ALTER TABLE geom DROP pt;
* Query OK, 0 rows affected (0.00 sec)
* Records: 0 Duplicates: 0 Warnings: 0

## Populating Spatial Columns

After you have created spatial columns, you can populate them with spatial data.

Values should be stored in internal geometry format, but you can convert them to that format from either Well-Known Text (WKT) or Well-Known Binary (WKB) format. The following examples demonstrate how to insert geometry values into a table by converting WKT values into internal geometry format.

You can perform the conversion directly in the INSERT statement:

INSERT INTO geom VALUES (GeomFromText('POINT(1 1)'));

SET @g = 'POINT(1 1)';

INSERT INTO geom VALUES (GeomFromText(@g));

Or you can perform the conversion prior to the INSERT:

SET @g = GeomFromText('POINT(1 1)');

INSERT INTO geom VALUES (@g);

The following examples insert more complex geometries into the table:

SET @g = 'LINESTRING(0 0,1 1,2 2)';

INSERT INTO geom VALUES (GeomFromText(@g));

SET @g = 'POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))';

INSERT INTO geom VALUES (GeomFromText(@g));

SET @g =

'GEOMETRYCOLLECTION(POINT(1 1),LINESTRING(0 0,1 1,2 2,3 3,4 4))';

INSERT INTO geom VALUES (GeomFromText(@g));

The preceding examples all use GeomFromText() to create geometry values. You can also use type-specific functions:

SET @g = 'POINT(1 1)';

INSERT INTO geom VALUES (PointFromText(@g));

SET @g = 'LINESTRING(0 0,1 1,2 2)';

INSERT INTO geom VALUES (LineStringFromText(@g));

SET @g = 'POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))';

INSERT INTO geom VALUES (PolygonFromText(@g));

SET @g =

'GEOMETRYCOLLECTION(POINT(1 1),LINESTRING(0 0,1 1,2 2,3 3,4 4))';

INSERT INTO geom VALUES (GeomCollFromText(@g));

Note that if a client application program wants to use WKB representations of geometry values, it is responsible for sending correctly formed WKB in queries to the server. However, there are several ways of satisfying this requirement. For example:

* Inserting a POINT(1 1) value with hex literal syntax:
* mysql> INSERT INTO geom VALUES
* -> (GeomFromWKB(0x0101000000000000000000F03F000000000000F03F));
* An ODBC application can send a WKB representation, binding it to a placeholder using an argument of BLOB type:
* INSERT INTO geom VALUES (GeomFromWKB(?))

Other programming interfaces may support a similar placeholder mechanism.

* In a C program, you can escape a binary value using mysql\_real\_escape\_string() and include the result in a query string that is sent to the server. See [mysql\_real\_escape\_string()](http://dev.cs.ovgu.de/db/mysql/C.html#mysql-real-escape-string).

## Fetching Spatial Data

Geometry values stored in a table can be fetched in internal format. You can also convert them into WKT or WKB format.

### Fetching Spatial Data in Internal Format

Fetching geometry values using internal format can be useful in table-to-table transfers:

CREATE TABLE geom2 (g GEOMETRY) SELECT g FROM geom;

### Fetching Spatial Data in WKT Format

The AsText() function converts a geometry from internal format into a WKT string.

mysql> SELECT AsText(g) FROM geom;

+-------------------------+

| AsText(p1) |

+-------------------------+

| POINT(1 1) |

| LINESTRING(0 0,1 1,2 2) |

+-------------------------+

### Fetching Spatial Data in WKB Format

The AsBinary() function converts a geometry from internal format into a BLOB containing the WKB value.

SELECT AsBinary(g) FROM geom;

# Spatial Data Types

The [Open Geospatial Consortium](http://www.opengeospatial.org/) (OGC) is an international consortium of more than 250 companies, agencies, and universities participating in the development of publicly available conceptual solutions that can be useful with all kinds of applications that manage spatial data.

The Open Geospatial Consortium publishes the *OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 2: SQL option*, a document that proposes several conceptual ways for extending an SQL RDBMS to support spatial data. This specification is available from the OGC website at <http://www.opengeospatial.org/standards/sfs>.

Following the OGC specification, MySQL implements spatial extensions as a subset of the ***SQL with Geometry Types*** environment. This term refers to an SQL environment that has been extended with a set of geometry types. A geometry-valued SQL column is implemented as a column that has a geometry type. The specification describes a set of SQL geometry types, as well as functions on those types to create and analyze geometry values.

MySQL spatial extensions enable the generation, storage, and analysis of geographic features:

* Data types for representing spatial values
* Functions for manipulating spatial values
* Spatial indexing for improved access times to spatial columns

The spatial data types and functions are available for [MyISAM](https://dev.mysql.com/doc/refman/8.0/en/myisam-storage-engine.html), [InnoDB](https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html), [NDB](https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html), and [ARCHIVE](https://dev.mysql.com/doc/refman/8.0/en/archive-storage-engine.html) tables. For indexing spatial columns, MyISAM and InnoDB support both SPATIAL and non-SPATIAL indexes. The other storage engines support non-SPATIAL indexes, as described in [Section 13.1.15, “CREATE INDEX Statement”](https://dev.mysql.com/doc/refman/8.0/en/create-index.html).

A ***geographic feature*** is anything in the world that has a location. A feature can be:

* An entity. For example, a mountain, a pond, a city.
* A space. For example, town district, the tropics.
* A definable location. For example, a crossroad, as a particular place where two streets intersect.

Some documents use the term ***geospatial feature*** to refer to geographic features.

***Geometry*** is another word that denotes a geographic feature. Originally the word ***geometry*** meant measurement of the earth. Another meaning comes from cartography, referring to the geometric features that cartographers use to map the world.

The discussion here considers these terms synonymous: ***geographic feature***, ***geospatial feature***, ***feature***, or ***geometry***. The term most commonly used is ***geometry***, defined as *a point or an aggregate of points representing anything in the world that has a location*.

The following material covers these topics:

* The spatial data types implemented in MySQL model
* The basis of the spatial extensions in the OpenGIS geometry model
* Data formats for representing spatial data
* How to use spatial data in MySQL
* Use of indexing for spatial data
* MySQL differences from the OpenGIS specification

For information about functions that operate on spatial data, see [Section 12.17, “Spatial Analysis Functions”](https://dev.mysql.com/doc/refman/8.0/en/spatial-analysis-functions.html).

These standards are important for the MySQL implementation of spatial operations:

* The [Open Geospatial Consortium](http://www.opengeospatial.org/) publishes the *OpenGIS® Implementation Standard for Geographic information*, a document that proposes several conceptual ways for extending an SQL RDBMS to support spatial data. See in particular Simple Feature Access - Part 1: Common Architecture, and Simple Feature Access - Part 2: SQL Option. The Open Geospatial Consortium (OGC) maintains a website at <http://www.opengeospatial.org/>. The specification is available there at <http://www.opengeospatial.org/standards/sfs>. It contains additional information relevant to the material here.
* The grammar for [spatial reference system](https://dev.mysql.com/doc/refman/8.0/en/spatial-reference-systems.html) (SRS) definitions is based on the grammar defined in *OpenGIS Implementation Specification: Coordinate Transformation Services*, Revision 1.00, OGC 01-009, January 12, 2001, Section 7.2. This specification is available at <http://www.opengeospatial.org/standards/ct>. For differences from that specification in SRS definitions as implemented in MySQL, see [Section 13.1.19, “CREATE SPATIAL REFERENCE SYSTEM Statement”](https://dev.mysql.com/doc/refman/8.0/en/create-spatial-reference-system.html).

If you have questions or concerns about the use of the spatial extensions to MySQL, you can discuss them in the GIS forum: <https://forums.mysql.com/list.php?23>.

MySQL has spatial data types that correspond to OpenGIS classes. The basis for these types is described in [Section 11.4.2, “The OpenGIS Geometry Model”](https://dev.mysql.com/doc/refman/8.0/en/opengis-geometry-model.html).

Some spatial data types hold single geometry values:

* GEOMETRY
* POINT
* LINESTRING
* POLYGON

GEOMETRY can store geometry values of any type. The other single-value types (POINT, LINESTRING, and POLYGON) restrict their values to a particular geometry type.

The other spatial data types hold collections of values:

* MULTIPOINT
* MULTILINESTRING
* MULTIPOLYGON
* GEOMETRYCOLLECTION

GEOMETRYCOLLECTION can store a collection of objects of any type. The other collection types (MULTIPOINT, MULTILINESTRING, and MULTIPOLYGON) restrict collection members to those having a particular geometry type.

Example: To create a table named geom that has a column named g that can store values of any geometry type, use this statement:

CREATE TABLE geom (g GEOMETRY);

Columns with a spatial data type can have an SRID attribute, to explicitly indicate the spatial reference system (SRS) for values stored in the column. For example:

CREATE TABLE geom (

p POINT SRID 0,

g GEOMETRY NOT NULL SRID 4326

);

SPATIAL indexes can be created on spatial columns if they are NOT NULL and have a specific SRID, so if you plan to index the column, declare it with the NOT NULL and SRID attributes:

CREATE TABLE geom (g GEOMETRY NOT NULL SRID 4326);

InnoDB tables permit SRID values for Cartesian and geographic SRSs. MyISAM tables permit SRID values for Cartesian SRSs.

The SRID attribute makes a spatial column SRID-restricted, which has these implications:

* The column can contain only values with the given SRID. Attempts to insert values with a different SRID produce an error.
* The optimizer can use SPATIAL indexes on the column. See [Section 8.3.3, “SPATIAL Index Optimization”](https://dev.mysql.com/doc/refman/8.0/en/spatial-index-optimization.html).

Spatial columns with no SRID attribute are not SRID-restricted and accept values with any SRID. However, the optimizer cannot use SPATIAL indexes on them until the column definition is modified to include an SRID attribute, which may require that the column contents first be modified so that all values have the same SRID.

For other examples showing how to use spatial data types in MySQL, see [Section 11.4.6, “Creating Spatial Columns”](https://dev.mysql.com/doc/refman/8.0/en/creating-spatial-columns.html). For information about spatial reference systems, see [Section 11.4.5, “Spatial Reference System Support”](https://dev.mysql.com/doc/refman/8.0/en/spatial-reference-systems.html).

## The OpenGIS Geometry Model

The set of geometry types proposed by OGC's ***SQL with Geometry Types*** environment is based on the ***OpenGIS Geometry Model***. In this model, each geometric object has the following general properties:

* It is associated with a spatial reference system, which describes the coordinate space in which the object is defined.
* It belongs to some geometry class.

### The Geometry Class Hierarchy

The geometry classes define a hierarchy as follows:

* Geometry (noninstantiable)
  + Point (instantiable)
  + Curve (noninstantiable)
    - LineString (instantiable)
      * Line
      * LinearRing
  + Surface (noninstantiable)
    - Polygon (instantiable)
  + GeometryCollection (instantiable)
    - MultiPoint (instantiable)
    - MultiCurve (noninstantiable)
      * MultiLineString (instantiable)
    - MultiSurface (noninstantiable)
      * MultiPolygon (instantiable)

It is not possible to create objects in noninstantiable classes. It is possible to create objects in instantiable classes. All classes have properties, and instantiable classes may also have assertions (rules that define valid class instances).

Geometry is the base class. It is an abstract class. The instantiable subclasses of Geometry are restricted to zero-, one-, and two-dimensional geometric objects that exist in two-dimensional coordinate space. All instantiable geometry classes are defined so that valid instances of a geometry class are topologically closed (that is, all defined geometries include their boundary).

The base Geometry class has subclasses for Point, Curve, Surface, and GeometryCollection:

* Point represents zero-dimensional objects.
* Curve represents one-dimensional objects, and has subclass LineString, with sub-subclasses Line and LinearRing.
* Surface is designed for two-dimensional objects and has subclass Polygon.
* GeometryCollection has specialized zero-, one-, and two-dimensional collection classes named MultiPoint, MultiLineString, and MultiPolygon for modeling geometries corresponding to collections of Points, LineStrings, and Polygons, respectively. MultiCurve and MultiSurface are introduced as abstract superclasses that generalize the collection interfaces to handle Curves and Surfaces.

Geometry, Curve, Surface, MultiCurve, and MultiSurface are defined as noninstantiable classes. They define a common set of methods for their subclasses and are included for extensibility.

Point, LineString, Polygon, GeometryCollection, MultiPoint, MultiLineString, and MultiPolygon are instantiable classes.

## Geometry Class

Geometry is the root class of the hierarchy. It is a noninstantiable class but has a number of properties, described in the following list, that are common to all geometry values created from any of the Geometry subclasses. Particular subclasses have their own specific properties, described later.

***Geometry Properties***

A geometry value has the following properties:

* Its ***type***. Each geometry belongs to one of the instantiable classes in the hierarchy.
* Its ***SRID***, or spatial reference identifier. This value identifies the geometry's associated spatial reference system that describes the coordinate space in which the geometry object is defined.

In MySQL, the SRID value is an integer associated with the geometry value. The maximum usable SRID value is 2**32**−1. If a larger value is given, only the lower 32 bits are used.

SRID 0 represents an infinite flat Cartesian plane with no units assigned to its axes. To ensure SRID 0 behavior, create geometry values using SRID 0. SRID 0 is the default for new geometry values if no SRID is specified.

For computations on multiple geometry values, all values must have the same SRID or an error occurs.

* Its ***coordinates*** in its spatial reference system, represented as double-precision (8-byte) numbers. All nonempty geometries include at least one pair of (X,Y) coordinates. Empty geometries contain no coordinates.

Coordinates are related to the SRID. For example, in different coordinate systems, the distance between two objects may differ even when objects have the same coordinates, because the distance on the ***planar*** coordinate system and the distance on the ***geodetic*** system (coordinates on the Earth's surface) are different things.

* Its ***interior***, ***boundary***, and ***exterior***.

Every geometry occupies some position in space. The exterior of a geometry is all space not occupied by the geometry. The interior is the space occupied by the geometry. The boundary is the interface between the geometry's interior and exterior.

* Its ***MBR*** (minimum bounding rectangle), or envelope. This is the bounding geometry, formed by the minimum and maximum (X,Y) coordinates:

((MINX MINY, MAXX MINY, MAXX MAXY, MINX MAXY, MINX MINY))

* Whether the value is ***simple*** or ***nonsimple***. Geometry values of types (LineString, MultiPoint, MultiLineString) are either simple or nonsimple. Each type determines its own assertions for being simple or nonsimple.
* Whether the value is ***closed*** or ***not closed***. Geometry values of types (LineString, MultiString) are either closed or not closed. Each type determines its own assertions for being closed or not closed.
* Whether the value is ***empty*** or ***nonempty*** A geometry is empty if it does not have any points. Exterior, interior, and boundary of an empty geometry are not defined (that is, they are represented by a NULL value). An empty geometry is defined to be always simple and has an area of 0.
* Its ***dimension***. A geometry can have a dimension of −1, 0, 1, or 2:
  + −1 for an empty geometry.
  + 0 for a geometry with no length and no area.
  + 1 for a geometry with nonzero length and zero area.
  + 2 for a geometry with nonzero area.

Point objects have a dimension of zero. LineString objects have a dimension of 1. Polygon objects have a dimension of 2. The dimensions of MultiPoint, MultiLineString, and MultiPolygon objects are the same as the dimensions of the elements they consist of.

## Point Class

A Point is a geometry that represents a single location in coordinate space.

*Point****Examples***

* Imagine a large-scale map of the world with many cities. A Point object could represent each city.
* On a city map, a Point object could represent a bus stop.

*Point****Properties***

* X-coordinate value.
* Y-coordinate value.
* Point is defined as a zero-dimensional geometry.
* The boundary of a Point is the empty set.

## Curve Class

A Curve is a one-dimensional geometry, usually represented by a sequence of points. Particular subclasses of Curve define the type of interpolation between points. Curve is a noninstantiable class.

*Curve****Properties***

* A Curve has the coordinates of its points.
* A Curve is defined as a one-dimensional geometry.
* A Curve is simple if it does not pass through the same point twice, with the exception that a curve can still be simple if the start and end points are the same.
* A Curve is closed if its start point is equal to its endpoint.
* The boundary of a closed Curve is empty.
* The boundary of a nonclosed Curve consists of its two endpoints.
* A Curve that is simple and closed is a LinearRing.

## LineString Class

A LineString is a Curve with linear interpolation between points.

*LineString****Examples***

* On a world map, LineString objects could represent rivers.
* In a city map, LineString objects could represent streets.

*LineString****Properties***

* A LineString has coordinates of segments, defined by each consecutive pair of points.
* A LineString is a Line if it consists of exactly two points.
* A LineString is a LinearRing if it is both closed and simple.

## Surface Class

A Surface is a two-dimensional geometry. It is a noninstantiable class. Its only instantiable subclass is Polygon.

*Surface****Properties***

* A Surface is defined as a two-dimensional geometry.
* The OpenGIS specification defines a simple Surface as a geometry that consists of a single “patch” that is associated with a single exterior boundary and zero or more interior boundaries.
* The boundary of a simple Surface is the set of closed curves corresponding to its exterior and interior boundaries.

## Polygon Class

A Polygon is a planar Surface representing a multisided geometry. It is defined by a single exterior boundary and zero or more interior boundaries, where each interior boundary defines a hole in the Polygon.

*Polygon****Examples***

* On a region map, Polygon objects could represent forests, districts, and so on.

*Polygon****Assertions***

* The boundary of a Polygon consists of a set of LinearRing objects (that is, LineString objects that are both simple and closed) that make up its exterior and interior boundaries.
* A Polygon has no rings that cross. The rings in the boundary of a Polygon may intersect at a Point, but only as a tangent.
* A Polygon has no lines, spikes, or punctures.
* A Polygon has an interior that is a connected point set.
* A Polygon may have holes. The exterior of a Polygon with holes is not connected. Each hole defines a connected component of the exterior.

The preceding assertions make a Polygon a simple geometry.

## GeometryCollection Class

A GeomCollection is a geometry that is a collection of zero or more geometries of any class.

GeomCollection and GeometryCollection are synonymous, with GeomCollection the preferred type name.

All the elements in a geometry collection must be in the same spatial reference system (that is, in the same coordinate system). There are no other constraints on the elements of a geometry collection, although the subclasses of GeomCollection described in the following sections may restrict membership. Restrictions may be based on:

* Element type (for example, a MultiPoint may contain only Point elements)
* Dimension
* Constraints on the degree of spatial overlap between elements

## MultiPoint Class

A MultiPoint is a geometry collection composed of Point elements. The points are not connected or ordered in any way.

*MultiPoint****Examples***

* On a world map, a MultiPoint could represent a chain of small islands.
* On a city map, a MultiPoint could represent the outlets for a ticket office.

*MultiPoint****Properties***

* A MultiPoint is a zero-dimensional geometry.
* A MultiPoint is simple if no two of its Point values are equal (have identical coordinate values).
* The boundary of a MultiPoint is the empty set.

## MultiCurve Class

A MultiCurve is a geometry collection composed of Curve elements. MultiCurve is a noninstantiable class.

*MultiCurve****Properties***

* A MultiCurve is a one-dimensional geometry.
* A MultiCurve is simple if and only if all of its elements are simple; the only intersections between any two elements occur at points that are on the boundaries of both elements.
* A MultiCurve boundary is obtained by applying the “mod 2 union rule” (also known as the “odd-even rule”): A point is in the boundary of a MultiCurve if it is in the boundaries of an odd number of Curve elements.
* A MultiCurve is closed if all of its elements are closed.
* The boundary of a closed MultiCurve is always empty.

## MultiLineString Class

A MultiLineString is a MultiCurve geometry collection composed of LineString elements.

*MultiLineString****Examples***

* On a region map, a MultiLineString could represent a river system or a highway system.

## MultiSurface Class

A MultiSurface is a geometry collection composed of surface elements. MultiSurface is a noninstantiable class. Its only instantiable subclass is MultiPolygon.

*MultiSurface****Assertions***

* Surfaces within a MultiSurface have no interiors that intersect.
* Surfaces within a MultiSurface have boundaries that intersect at most at a finite number of points.

## MultiPolygon Class

A MultiPolygon is a MultiSurface object composed of Polygon elements.

*MultiPolygon****Examples***

* On a region map, a MultiPolygon could represent a system of lakes.

*MultiPolygon****Assertions***

* A MultiPolygon has no two Polygon elements with interiors that intersect.
* A MultiPolygon has no two Polygon elements that cross (crossing is also forbidden by the previous assertion), or that touch at an infinite number of points.
* A MultiPolygon may not have cut lines, spikes, or punctures. A MultiPolygon is a regular, closed point set.
* A MultiPolygon that has more than one Polygon has an interior that is not connected. The number of connected components of the interior of a MultiPolygon is equal to the number of Polygon values in the MultiPolygon.

*MultiPolygon****Properties***

* A MultiPolygon is a two-dimensional geometry.
* A MultiPolygon boundary is a set of closed curves (LineString values) corresponding to the boundaries of its Polygon elements.
* Each Curve in the boundary of the MultiPolygon is in the boundary of exactly one Polygon element.
* Every Curve in the boundary of an Polygon element is in the boundary of the MultiPolygon.

## Supported Spatial Data Formats

Two standard spatial data formats are used to represent geometry objects in queries:

* Well-Known Text (WKT) format
* Well-Known Binary (WKB) format

Internally, MySQL stores geometry values in a format that is not identical to either WKT or WKB format. (Internal format is like WKB but with an initial 4 bytes to indicate the SRID.)

There are functions available to convert between different data formats; see [Section 12.17.6, “Geometry Format Conversion Functions”](https://dev.mysql.com/doc/refman/8.0/en/gis-format-conversion-functions.html).

The following sections describe the spatial data formats MySQL uses:

### Well-Known Text (WKT) Format

The Well-Known Text (WKT) representation of geometry values is designed for exchanging geometry data in ASCII form. The OpenGIS specification provides a Backus-Naur grammar that specifies the formal production rules for writing WKT values (see [Section 11.4, “Spatial Data Types”](https://dev.mysql.com/doc/refman/8.0/en/spatial-types.html)).

Examples of WKT representations of geometry objects:

* A Point:

POINT(15 20)

The point coordinates are specified with no separating comma. This differs from the syntax for the SQL [Point()](https://dev.mysql.com/doc/refman/8.0/en/gis-mysql-specific-functions.html#function_point) function, which requires a comma between the coordinates. Take care to use the syntax appropriate to the context of a given spatial operation. For example, the following statements both use [ST\_X()](https://dev.mysql.com/doc/refman/8.0/en/gis-point-property-functions.html#function_st-x) to extract the X-coordinate from a Point object. The first produces the object directly using the [Point()](https://dev.mysql.com/doc/refman/8.0/en/gis-mysql-specific-functions.html#function_point) function. The second uses a WKT representation converted to a Point with [ST\_GeomFromText()](https://dev.mysql.com/doc/refman/8.0/en/gis-wkt-functions.html#function_st-geomfromtext).

mysql> SELECT ST\_X(Point(15, 20));

+---------------------+

| ST\_X(POINT(15, 20)) |

+---------------------+

| 15 |

+---------------------+

mysql> SELECT ST\_X(ST\_GeomFromText('POINT(15 20)'));

+---------------------------------------+

| ST\_X(ST\_GeomFromText('POINT(15 20)')) |

+---------------------------------------+

| 15 |

+---------------------------------------+

* A LineString with four points:

LINESTRING(0 0, 10 10, 20 25, 50 60)

The point coordinate pairs are separated by commas.

* A Polygon with one exterior ring and one interior ring:

POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))

* A MultiPoint with three Point values:

MULTIPOINT(0 0, 20 20, 60 60)

Spatial functions such as [ST\_MPointFromText()](https://dev.mysql.com/doc/refman/8.0/en/gis-wkt-functions.html#function_st-mpointfromtext) and [ST\_GeomFromText()](https://dev.mysql.com/doc/refman/8.0/en/gis-wkt-functions.html#function_st-geomfromtext) that accept WKT-format representations of MultiPoint values permit individual points within values to be surrounded by parentheses. For example, both of the following function calls are valid:

ST\_MPointFromText('MULTIPOINT (1 1, 2 2, 3 3)')

ST\_MPointFromText('MULTIPOINT ((1 1), (2 2), (3 3))')

* A MultiLineString with two LineString values:

MULTILINESTRING((10 10, 20 20), (15 15, 30 15))

* A MultiPolygon with two Polygon values:

MULTIPOLYGON(((0 0,10 0,10 10,0 10,0 0)),((5 5,7 5,7 7,5 7, 5 5)))

* A GeometryCollection consisting of two Point values and one LineString:

GEOMETRYCOLLECTION(POINT(10 10), POINT(30 30), LINESTRING(15 15, 20 20))

### Well-Known Binary (WKB) Format

The Well-Known Binary (WKB) representation of geometric values is used for exchanging geometry data as binary streams represented by [BLOB](https://dev.mysql.com/doc/refman/8.0/en/blob.html) values containing geometric WKB information. This format is defined by the OpenGIS specification (see [Section 11.4, “Spatial Data Types”](https://dev.mysql.com/doc/refman/8.0/en/spatial-types.html)). It is also defined in the ISO SQL/MM Part 3: Spatial standard.

WKB uses 1-byte unsigned integers, 4-byte unsigned integers, and 8-byte double-precision numbers (IEEE 754 format). A byte is eight bits.

For example, a WKB value that corresponds to POINT(1 -1) consists of this sequence of 21 bytes, each represented by two hexadecimal digits:

0101000000000000000000F03F000000000000F0BF

The sequence consists of the components shown in the following table.

**Table  WKB Components Example**

| **Component** | **Size** | **Value** |
| --- | --- | --- |
| **Byte order** | 1 byte | 01 |
| **WKB type** | 4 bytes | 01000000 |
| **X coordinate** | 8 bytes | 000000000000F03F |
| **Y coordinate** | 8 bytes | 000000000000F0BF |

Component representation is as follows:

* The byte order indicator is either 1 or 0 to signify little-endian or big-endian storage. The little-endian and big-endian byte orders are also known as Network Data Representation (NDR) and External Data Representation (XDR), respectively.
* The WKB type is a code that indicates the geometry type. MySQL uses values from 1 through 7 to indicate Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection.
* A Point value has X and Y coordinates, each represented as a double-precision value.

WKB values for more complex geometry values have more complex data structures, as detailed in the OpenGIS specification.

### Internal Geometry Storage Format

MySQL stores geometry values using 4 bytes to indicate the SRID followed by the WKB representation of the value. For a description of WKB format, see [Well-Known Binary (WKB) Format](https://dev.mysql.com/doc/refman/8.0/en/gis-data-formats.html#gis-wkb-format).

For the WKB part, these MySQL-specific considerations apply:

* The byte-order indicator byte is 1 because MySQL stores geometries as little-endian values.
* MySQL supports geometry types of Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection. Other geometry types are not supported.
* Only GeometryCollection can be empty. Such a value is stored with 0 elements.
* Polygon rings can be specified both clockwise and counterclockwise. MySQL flips the rings automatically when reading data.

Cartesian coordinates are stored in the length unit of the spatial reference system, with X values in the X coordinates and Y values in the Y coordinates. Axis directions are those specified by the spatial reference system.

Geographic coordinates are stored in the angle unit of the spatial reference system, with longitudes in the X coordinates and latitudes in the Y coordinates. Axis directions and the meridian are those specified by the spatial reference system.

The [LENGTH()](https://dev.mysql.com/doc/refman/8.0/en/string-functions.html#function_length) function returns the space in bytes required for value storage. Example:

mysql> SET @g = ST\_GeomFromText('POINT(1 -1)');

mysql> SELECT LENGTH(@g);

+------------+

| LENGTH(@g) |

+------------+

| 25 |

+------------+

mysql> SELECT HEX(@g);

+----------------------------------------------------+

| HEX(@g) |

+----------------------------------------------------+

| 000000000101000000000000000000F03F000000000000F0BF |

+----------------------------------------------------+

The value length is 25 bytes, made up of these components (as can be seen from the hexadecimal value):

* 4 bytes for integer SRID (0)
* 1 byte for integer byte order (1 = little-endian)
* 4 bytes for integer type information (1 = Point)
* 8 bytes for double-precision X coordinate (1)
* 8 bytes for double-precision Y coordinate (−1)

### Geometry Well-Formedness and Validity

For geometry values, MySQL distinguishes between the concepts of syntactically well-formed and geometrically valid.

A geometry is syntactically well-formed if it satisfies conditions such as those in this (nonexhaustive) list:

* Linestrings have at least two points
* Polygons have at least one ring
* Polygon rings are closed (first and last points the same)
* Polygon rings have at least 4 points (minimum polygon is a triangle with first and last points the same)
* Collections are not empty (except GeometryCollection)

A geometry is geometrically valid if it is syntactically well-formed and satisfies conditions such as those in this (nonexhaustive) list:

* Polygons are not self-intersecting
* Polygon interior rings are inside the exterior ring
* Multipolygons do not have overlapping polygons

Spatial functions fail if a geometry is not syntactically well-formed. Spatial import functions that parse WKT or WKB values raise an error for attempts to create a geometry that is not syntactically well-formed. Syntactic well-formedness is also checked for attempts to store geometries into tables.

It is permitted to insert, select, and update geometrically invalid geometries, but they must be syntactically well-formed. Due to the computational expense, MySQL does not check explicitly for geometric validity. Spatial computations may detect some cases of invalid geometries and raise an error, but they may also return an undefined result without detecting the invalidity. Applications that require geometically valid geometries should check them using the [ST\_IsValid()](https://dev.mysql.com/doc/refman/8.0/en/spatial-convenience-functions.html#function_st-isvalid) function.

## Spatial Reference System Support

A spatial reference system (SRS) for spatial data is a coordinate-based system for geographic locations.

There are different types of spatial reference systems:

* A projected SRS is a projection of a globe onto a flat surface; that is, a flat map. For example, a light bulb inside a globe that shines on a paper cylinder surrounding the globe projects a map onto the paper. The result is georeferenced: Each point maps to a place on the globe. The coordinate system on that plane is Cartesian using a length unit (meters, feet, and so forth), rather than degrees of longitude and latitude.

The globes in this case are ellipsoids; that is, flattened spheres. Earth is a bit shorter in its North-South axis than its East-West axis, so a slightly flattened sphere is more correct, but perfect spheres permit faster calculations.

* A geographic SRS is a nonprojected SRS representing longitude-latitude (or latitude-longitude) coordinates on an ellipsoid, in any angular unit.
* The SRS denoted in MySQL by SRID 0 represents an infinite flat Cartesian plane with no units assigned to its axes. Unlike projected SRSs, it is not georeferenced and it does not necessarily represent Earth. It is an abstract plane that can be used for anything. SRID 0 is the default SRID for spatial data in MySQL.

MySQL maintains information about available spatial reference systems for spatial data in the data dictionary mysql.st\_spatial\_reference\_systems table, which can store entries for projected and geographic SRSs. This data dictionary table is invisible, but SRS entry contents are available through the INFORMATION\_SCHEMA [ST\_SPATIAL\_REFERENCE\_SYSTEMS](https://dev.mysql.com/doc/refman/8.0/en/information-schema-st-spatial-reference-systems-table.html) table, implemented as a view on mysql.st\_spatial\_reference\_systems (see [Section 26.36, “The INFORMATION\_SCHEMA ST\_SPATIAL\_REFERENCE\_SYSTEMS Table”](https://dev.mysql.com/doc/refman/8.0/en/information-schema-st-spatial-reference-systems-table.html)).

The following example shows what an SRS entry looks like:

mysql> SELECT \*

FROM INFORMATION\_SCHEMA.ST\_SPATIAL\_REFERENCE\_SYSTEMS

WHERE SRS\_ID = 4326\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SRS\_NAME: WGS 84

SRS\_ID: 4326

ORGANIZATION: EPSG

ORGANIZATION\_COORDSYS\_ID: 4326

DEFINITION: GEOGCS["WGS 84",DATUM["World Geodetic System 1984",

SPHEROID["WGS 84",6378137,298.257223563,

AUTHORITY["EPSG","7030"]],AUTHORITY["EPSG","6326"]],

PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],

UNIT["degree",0.017453292519943278,

AUTHORITY["EPSG","9122"]],

AXIS["Lat",NORTH],AXIS["Long",EAST],

AUTHORITY["EPSG","4326"]]

DESCRIPTION:

This entry describes the SRS used for GPS systems. It has a name (SRS\_NAME) of WGS 84 and an ID (SRS\_ID) of 4326, which is the ID used by the [European Petroleum Survey Group](http://epsg.org/) (EPSG).

SRS definitions in the DEFINITION column are WKT values, represented as specified in the [Open Geospatial Consortium](http://www.opengeospatial.org/) document [OGC 12-063r5](http://docs.opengeospatial.org/is/12-063r5/12-063r5.html).

SRS\_ID values represent the same kind of values as the SRID of geometry values or passed as the SRID argument to spatial functions. SRID 0 (the unitless Cartesian plane) is special. It is always a legal spatial reference system ID and can be used in any computations on spatial data that depend on SRID values.

For computations on multiple geometry values, all values must have the same SRID or an error occurs.

SRS definition parsing occurs on demand when definitions are needed by GIS functions. Parsed definitions are stored in the data dictionary cache to enable reuse and avoid incurring parsing overhead for every statement that needs SRS information.

To enable manipulation of SRS entries stored in the data dictionary, MySQL provides these SQL statements:

* [CREATE SPATIAL REFERENCE SYSTEM](https://dev.mysql.com/doc/refman/8.0/en/create-spatial-reference-system.html): See [Section 13.1.19, “CREATE SPATIAL REFERENCE SYSTEM Statement”](https://dev.mysql.com/doc/refman/8.0/en/create-spatial-reference-system.html). The description for this statement includes additional information about SRS components.
* [DROP SPATIAL REFERENCE SYSTEM](https://dev.mysql.com/doc/refman/8.0/en/drop-spatial-reference-system.html): See [Section 13.1.31, “DROP SPATIAL REFERENCE SYSTEM Statement”](https://dev.mysql.com/doc/refman/8.0/en/drop-spatial-reference-system.html).

## Creating Spatial Columns

MySQL provides a standard way of creating spatial columns for geometry types, for example, with [CREATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html) or [ALTER TABLE](https://dev.mysql.com/doc/refman/8.0/en/alter-table.html). Spatial columns are supported for [MyISAM](https://dev.mysql.com/doc/refman/8.0/en/myisam-storage-engine.html), [InnoDB](https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html), [NDB](https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html), and [ARCHIVE](https://dev.mysql.com/doc/refman/8.0/en/archive-storage-engine.html) tables. See also the notes about spatial indexes under [Section 11.4.10, “Creating Spatial Indexes”](https://dev.mysql.com/doc/refman/8.0/en/creating-spatial-indexes.html).

Columns with a spatial data type can have an SRID attribute, to explicitly indicate the spatial reference system (SRS) for values stored in the column. For implications of an SRID-restricted column, see [Section 11.4.1, “Spatial Data Types”](https://dev.mysql.com/doc/refman/8.0/en/spatial-type-overview.html).

* Use the [CREATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html) statement to create a table with a spatial column:

CREATE TABLE geom (g GEOMETRY);

* Use the [ALTER TABLE](https://dev.mysql.com/doc/refman/8.0/en/alter-table.html) statement to add or drop a spatial column to or from an existing table:
* ALTER TABLE geom ADD pt POINT;

ALTER TABLE geom DROP pt;

## Populating Spatial Columns

After you have created spatial columns, you can populate them with spatial data.

Values should be stored in internal geometry format, but you can convert them to that format from either Well-Known Text (WKT) or Well-Known Binary (WKB) format. The following examples demonstrate how to insert geometry values into a table by converting WKT values to internal geometry format:

* Perform the conversion directly in the [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement:
* INSERT INTO geom VALUES (ST\_GeomFromText('POINT(1 1)'));
* SET @g = 'POINT(1 1)';

INSERT INTO geom VALUES (ST\_GeomFromText(@g));

* Perform the conversion prior to the [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html):
* SET @g = ST\_GeomFromText('POINT(1 1)');

INSERT INTO geom VALUES (@g);

The following examples insert more complex geometries into the table:

SET @g = 'LINESTRING(0 0,1 1,2 2)';

INSERT INTO geom VALUES (ST\_GeomFromText(@g));

SET @g = 'POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))';

INSERT INTO geom VALUES (ST\_GeomFromText(@g));

SET @g =

'GEOMETRYCOLLECTION(POINT(1 1),LINESTRING(0 0,1 1,2 2,3 3,4 4))';

INSERT INTO geom VALUES (ST\_GeomFromText(@g));

The preceding examples use [ST\_GeomFromText()](https://dev.mysql.com/doc/refman/8.0/en/gis-wkt-functions.html#function_st-geomfromtext) to create geometry values. You can also use type-specific functions:

SET @g = 'POINT(1 1)';

INSERT INTO geom VALUES (ST\_PointFromText(@g));

SET @g = 'LINESTRING(0 0,1 1,2 2)';

INSERT INTO geom VALUES (ST\_LineStringFromText(@g));

SET @g = 'POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))';

INSERT INTO geom VALUES (ST\_PolygonFromText(@g));

SET @g =

'GEOMETRYCOLLECTION(POINT(1 1),LINESTRING(0 0,1 1,2 2,3 3,4 4))';

INSERT INTO geom VALUES (ST\_GeomCollFromText(@g));

A client application program that wants to use WKB representations of geometry values is responsible for sending correctly formed WKB in queries to the server. There are several ways to satisfy this requirement. For example:

* Inserting a POINT(1 1) value with hex literal syntax:
* INSERT INTO geom VALUES

(ST\_GeomFromWKB(X'0101000000000000000000F03F000000000000F03F'));

* An ODBC application can send a WKB representation, binding it to a placeholder using an argument of [BLOB](https://dev.mysql.com/doc/refman/8.0/en/blob.html) type:

INSERT INTO geom VALUES (ST\_GeomFromWKB(?))

Other programming interfaces may support a similar placeholder mechanism.

* In a C program, you can escape a binary value using [mysql\_real\_escape\_string\_quote()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-real-escape-string-quote.html) and include the result in a query string that is sent to the server. See [mysql\_real\_escape\_string\_quote()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-real-escape-string-quote.html).

## Fetching Spatial Data

Geometry values stored in a table can be fetched in internal format. You can also convert them to WKT or WKB format.

* Fetching spatial data in internal format:

Fetching geometry values using internal format can be useful in table-to-table transfers:

CREATE TABLE geom2 (g GEOMETRY) SELECT g FROM geom;

* Fetching spatial data in WKT format:

The [ST\_AsText()](https://dev.mysql.com/doc/refman/8.0/en/gis-format-conversion-functions.html#function_st-astext) function converts a geometry from internal format to a WKT string.

SELECT ST\_AsText(g) FROM geom;

* Fetching spatial data in WKB format:

The [ST\_AsBinary()](https://dev.mysql.com/doc/refman/8.0/en/gis-format-conversion-functions.html#function_st-asbinary) function converts a geometry from internal format to a [BLOB](https://dev.mysql.com/doc/refman/8.0/en/blob.html) containing the WKB value.

SELECT ST\_AsBinary(g) FROM geom;

## Optimizing Spatial Analysis

For [MyISAM](https://dev.mysql.com/doc/refman/8.0/en/myisam-storage-engine.html) and InnoDB tables, search operations in columns containing spatial data can be optimized using SPATIAL indexes. The most typical operations are:

* Point queries that search for all objects that contain a given point
* Region queries that search for all objects that overlap a given region

MySQL uses ***R-Trees with quadratic splitting*** for SPATIAL indexes on spatial columns. A SPATIAL index is built using the minimum bounding rectangle (MBR) of a geometry. For most geometries, the MBR is a minimum rectangle that surrounds the geometries. For a horizontal or a vertical linestring, the MBR is a rectangle degenerated into the linestring. For a point, the MBR is a rectangle degenerated into the point.

It is also possible to create normal indexes on spatial columns. In a non-SPATIAL index, you must declare a prefix for any spatial column except for POINT columns.

MyISAM and InnoDB support both SPATIAL and non-SPATIAL indexes. Other storage engines support non-SPATIAL indexes, as described in [Section 13.1.15, “CREATE INDEX Statement”](https://dev.mysql.com/doc/refman/8.0/en/create-index.html).

## Creating Spatial Indexes

For InnoDB and MyISAM tables, MySQL can create spatial indexes using syntax similar to that for creating regular indexes, but using the SPATIAL keyword. Columns in spatial indexes must be declared NOT NULL. The following examples demonstrate how to create spatial indexes:

* With [CREATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html):

CREATE TABLE geom (g GEOMETRY NOT NULL SRID 4326, SPATIAL INDEX(g));

* With [ALTER TABLE](https://dev.mysql.com/doc/refman/8.0/en/alter-table.html):
* CREATE TABLE geom (g GEOMETRY NOT NULL SRID 4326);

ALTER TABLE geom ADD SPATIAL INDEX(g);

* With [CREATE INDEX](https://dev.mysql.com/doc/refman/8.0/en/create-index.html):
* CREATE TABLE geom (g GEOMETRY NOT NULL SRID 4326);

CREATE SPATIAL INDEX g ON geom (g);

SPATIAL INDEX creates an R-tree index. For storage engines that support nonspatial indexing of spatial columns, the engine creates a B-tree index. A B-tree index on spatial values is useful for exact-value lookups, but not for range scans.

The optimizer can use spatial indexes defined on columns that are SRID-restricted. For more information, see [Section 11.4.1, “Spatial Data Types”](https://dev.mysql.com/doc/refman/8.0/en/spatial-type-overview.html), and [Section 8.3.3, “SPATIAL Index Optimization”](https://dev.mysql.com/doc/refman/8.0/en/spatial-index-optimization.html).

For more information on indexing spatial columns, see [Section 13.1.15, “CREATE INDEX Statement”](https://dev.mysql.com/doc/refman/8.0/en/create-index.html).

To drop spatial indexes, use [ALTER TABLE](https://dev.mysql.com/doc/refman/8.0/en/alter-table.html) or [DROP INDEX](https://dev.mysql.com/doc/refman/8.0/en/drop-index.html):

* With [ALTER TABLE](https://dev.mysql.com/doc/refman/8.0/en/alter-table.html):

ALTER TABLE geom DROP INDEX g;

* With [DROP INDEX](https://dev.mysql.com/doc/refman/8.0/en/drop-index.html):

DROP INDEX g ON geom;

Example: Suppose that a table geom contains more than 32,000 geometries, which are stored in the column g of type GEOMETRY. The table also has an AUTO\_INCREMENT column fid for storing object ID values.

mysql> DESCRIBE geom;

+-------+----------+------+-----+---------+----------------+

| Field | Type | Null | Key | Default | Extra |

+-------+----------+------+-----+---------+----------------+

| fid | int(11) | | PRI | NULL | auto\_increment |

| g | geometry | | | | |

+-------+----------+------+-----+---------+----------------+

2 rows in set (0.00 sec)

mysql> SELECT COUNT(\*) FROM geom;

+----------+

| count(\*) |

+----------+

| 32376 |

+----------+

1 row in set (0.00 sec)

To add a spatial index on the column g, use this statement:

mysql> ALTER TABLE geom ADD SPATIAL INDEX(g);

Query OK, 32376 rows affected (4.05 sec)

Records: 32376 Duplicates: 0 Warnings: 0

## Using Spatial Indexes

The optimizer investigates whether available spatial indexes can be involved in the search for queries that use a function such as [MBRContains()](https://dev.mysql.com/doc/refman/8.0/en/spatial-relation-functions-mbr.html#function_mbrcontains) or [MBRWithin()](https://dev.mysql.com/doc/refman/8.0/en/spatial-relation-functions-mbr.html#function_mbrwithin) in the WHERE clause. The following query finds all objects that are in the given rectangle:

mysql> SET @poly =

-> 'Polygon((30000 15000,

31000 15000,

31000 16000,

30000 16000,

30000 15000))';

mysql> SELECT fid,ST\_AsText(g) FROM geom WHERE

-> MBRContains(ST\_GeomFromText(@poly),g);

+-----+---------------------------------------------------------------+

| fid | ST\_AsText(g) |

+-----+---------------------------------------------------------------+

| 21 | LINESTRING(30350.4 15828.8,30350.6 15845,30333.8 15845,30 ... |

| 22 | LINESTRING(30350.6 15871.4,30350.6 15887.8,30334 15887.8, ... |

| 23 | LINESTRING(30350.6 15914.2,30350.6 15930.4,30334 15930.4, ... |

| 24 | LINESTRING(30290.2 15823,30290.2 15839.4,30273.4 15839.4, ... |

| 25 | LINESTRING(30291.4 15866.2,30291.6 15882.4,30274.8 15882. ... |

| 26 | LINESTRING(30291.6 15918.2,30291.6 15934.4,30275 15934.4, ... |

| 249 | LINESTRING(30337.8 15938.6,30337.8 15946.8,30320.4 15946. ... |

| 1 | LINESTRING(30250.4 15129.2,30248.8 15138.4,30238.2 15136. ... |

| 2 | LINESTRING(30220.2 15122.8,30217.2 15137.8,30207.6 15136, ... |

| 3 | LINESTRING(30179 15114.4,30176.6 15129.4,30167 15128,3016 ... |

| 4 | LINESTRING(30155.2 15121.4,30140.4 15118.6,30142 15109,30 ... |

| 5 | LINESTRING(30192.4 15085,30177.6 15082.2,30179.2 15072.4, ... |

| 6 | LINESTRING(30244 15087,30229 15086.2,30229.4 15076.4,3024 ... |

| 7 | LINESTRING(30200.6 15059.4,30185.6 15058.6,30186 15048.8, ... |

| 10 | LINESTRING(30179.6 15017.8,30181 15002.8,30190.8 15003.6, ... |

| 11 | LINESTRING(30154.2 15000.4,30168.6 15004.8,30166 15014.2, ... |

| 13 | LINESTRING(30105 15065.8,30108.4 15050.8,30118 15053,3011 ... |

| 154 | LINESTRING(30276.2 15143.8,30261.4 15141,30263 15131.4,30 ... |

| 155 | LINESTRING(30269.8 15084,30269.4 15093.4,30258.6 15093,30 ... |

| 157 | LINESTRING(30128.2 15011,30113.2 15010.2,30113.6 15000.4, ... |

+-----+---------------------------------------------------------------+

20 rows in set (0.00 sec)

Use [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html) to check the way this query is executed:

mysql> SET @poly =

-> 'Polygon((30000 15000,

31000 15000,

31000 16000,

30000 16000,

30000 15000))';

mysql> EXPLAIN SELECT fid,ST\_AsText(g) FROM geom WHERE

-> MBRContains(ST\_GeomFromText(@poly),g)\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: geom

type: range

possible\_keys: g

key: g

key\_len: 32

ref: NULL

rows: 50

Extra: Using where

1 row in set (0.00 sec)

Check what would happen without a spatial index:

mysql> SET @poly =

-> 'Polygon((30000 15000,

31000 15000,

31000 16000,

30000 16000,

30000 15000))';

mysql> EXPLAIN SELECT fid,ST\_AsText(g) FROM g IGNORE INDEX (g) WHERE

-> MBRContains(ST\_GeomFromText(@poly),g)\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: geom

type: ALL

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 32376

Extra: Using where

1 row in set (0.00 sec)

Executing the [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statement without the spatial index yields the same result but causes the execution time to rise from 0.00 seconds to 0.46 seconds:

mysql> SET @poly =

-> 'Polygon((30000 15000,

31000 15000,

31000 16000,

30000 16000,

30000 15000))';

mysql> SELECT fid,ST\_AsText(g) FROM geom IGNORE INDEX (g) WHERE

-> MBRContains(ST\_GeomFromText(@poly),g);

+-----+---------------------------------------------------------------+

| fid | ST\_AsText(g) |

+-----+---------------------------------------------------------------+

| 1 | LINESTRING(30250.4 15129.2,30248.8 15138.4,30238.2 15136. ... |

| 2 | LINESTRING(30220.2 15122.8,30217.2 15137.8,30207.6 15136, ... |

| 3 | LINESTRING(30179 15114.4,30176.6 15129.4,30167 15128,3016 ... |

| 4 | LINESTRING(30155.2 15121.4,30140.4 15118.6,30142 15109,30 ... |

| 5 | LINESTRING(30192.4 15085,30177.6 15082.2,30179.2 15072.4, ... |

| 6 | LINESTRING(30244 15087,30229 15086.2,30229.4 15076.4,3024 ... |

| 7 | LINESTRING(30200.6 15059.4,30185.6 15058.6,30186 15048.8, ... |

| 10 | LINESTRING(30179.6 15017.8,30181 15002.8,30190.8 15003.6, ... |

| 11 | LINESTRING(30154.2 15000.4,30168.6 15004.8,30166 15014.2, ... |

| 13 | LINESTRING(30105 15065.8,30108.4 15050.8,30118 15053,3011 ... |

| 21 | LINESTRING(30350.4 15828.8,30350.6 15845,30333.8 15845,30 ... |

| 22 | LINESTRING(30350.6 15871.4,30350.6 15887.8,30334 15887.8, ... |

| 23 | LINESTRING(30350.6 15914.2,30350.6 15930.4,30334 15930.4, ... |

| 24 | LINESTRING(30290.2 15823,30290.2 15839.4,30273.4 15839.4, ... |

| 25 | LINESTRING(30291.4 15866.2,30291.6 15882.4,30274.8 15882. ... |

| 26 | LINESTRING(30291.6 15918.2,30291.6 15934.4,30275 15934.4, ... |

| 154 | LINESTRING(30276.2 15143.8,30261.4 15141,30263 15131.4,30 ... |

| 155 | LINESTRING(30269.8 15084,30269.4 15093.4,30258.6 15093,30 ... |

| 157 | LINESTRING(30128.2 15011,30113.2 15010.2,30113.6 15000.4, ... |

| 249 | LINESTRING(30337.8 15938.6,30337.8 15946.8,30320.4 15946. ... |

+-----+---------------------------------------------------------------+

20 rows in set (0.46 sec)

# Using 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)

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.

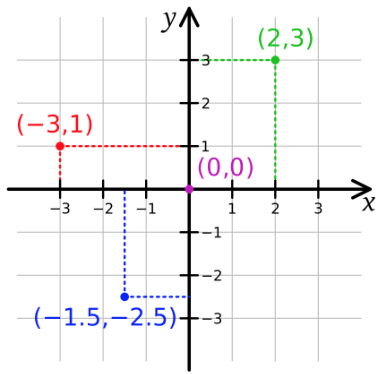
Looking from a backend perspective, we would be needing to store geographic data of these locations like **Latitude** and **Longitude**. Then we would need to write functions that calculate the distance between the user and the location (to show how far the location is from him/her). Using the same function, 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 achived by loging 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**](http://docs.geotools.org/latest/userguide/library/opengis/index.html) 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 [**opengeospatial.org**](http://opengeospatial.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**.



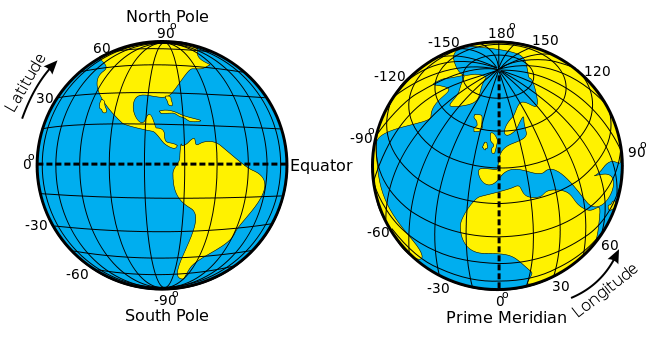
On the left, you can see an infinite **2D** flat unitless surface where a point is represented like P(x,y) where x is the distance of a point **P** from the **origin**(0,0) on the **x-axis** while y is the distance from the origin(0,0) on the **y-axis**. Hence, if we know x, y of a point, we can locate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



**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**](https://en.wikipedia.org/wiki/Geographic_coordinate_system) on Wikipedia.

The maximum value of the Latitude is 90° while the minimum is -90°. Similarly, the maximum value of longitude is 180° while the minimum is -180°.

*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° Latitude** line goes through the equator of the planet dividing the globe into northern and southern hemispheres. The **0° 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**](https://en.wikipedia.org/wiki/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**](https://www.geeksforgeeks.org/haversine-formula-to-find-distance-between-two-points-on-a-sphere/)**.**

*⚠️ 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](https://en.wikipedia.org/wiki/Polar_coordinate_system). 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**](https://en.wikipedia.org/wiki/Spatial_reference_system#Identifier) is the unique **integer** associated with a coordinate system proposed by OGC. MySQL databased 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 is 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 WKB, which will cover in upcoming topics in this tutorial. Read more about WKT, WKB specifications for different geometries from [**here**](http://www.skylinesoft.com/SkylineGlobe/TerraExplorer/v6.5.0/APIReferenceGuide/Well_Known_Text_and_Well_Known_Binary_WKT_and_WKB.htm).

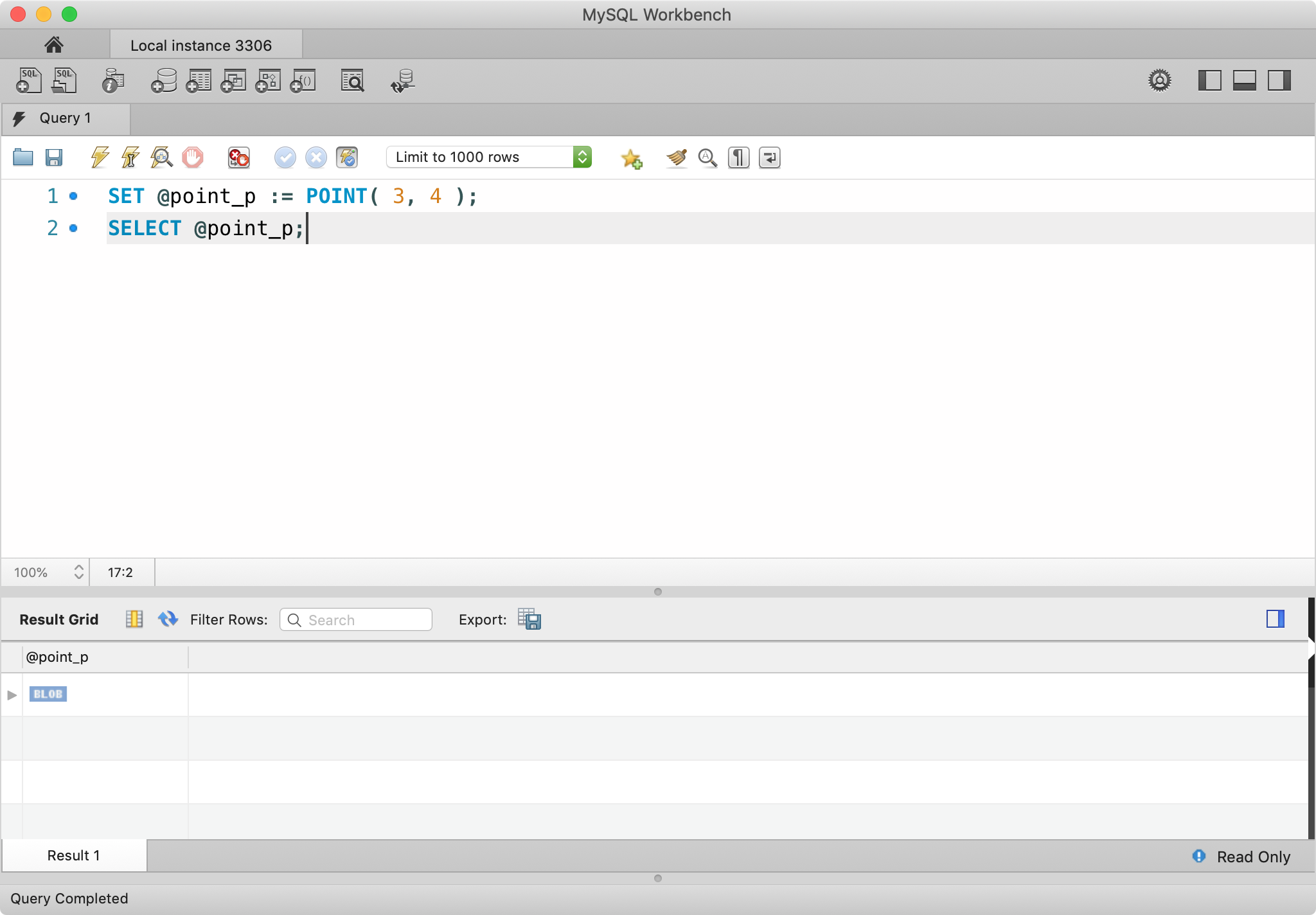
## ****Cartesian Coordinate System****

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.

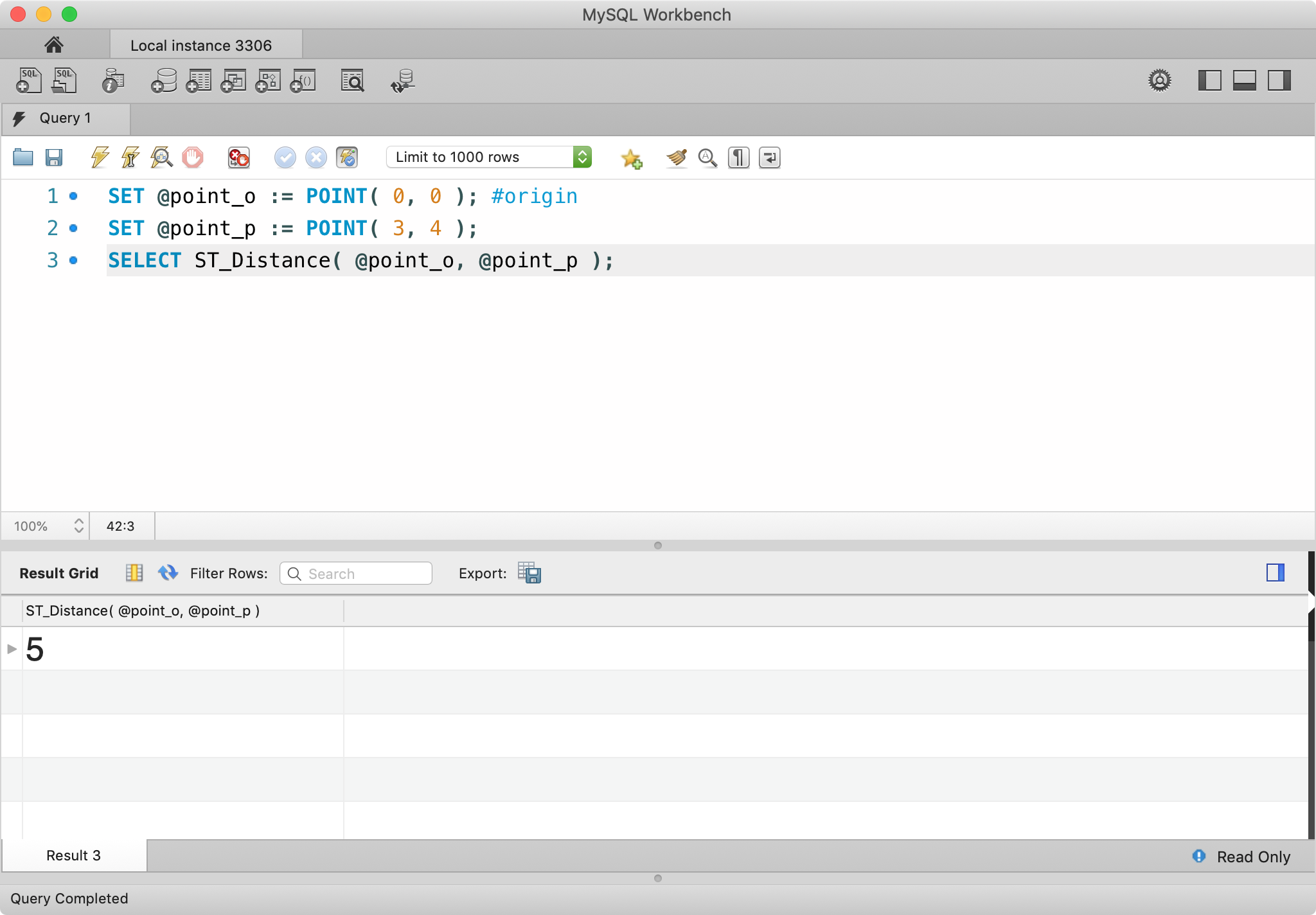
*MySQL provides many built-in functions like POINT, LINESTRING, POLYGON etc. They are documented*[***here***](https://dev.mysql.com/doc/refman/8.0/en/gis-mysql-specific-functions.html)*. Since, we are going to talk about POINT only, I though I should mention this.*

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**.



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.



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***](https://dev.mysql.com/doc/refman/8.0/en/spatial-relation-functions-object-shapes.html)*, 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***](https://dev.mysql.com/doc/refman/8.0/en/spatial-function-reference.html)*.*

*⚠️ 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***](https://mysqlserverteam.com/detecting-incompatible-use-of-spatial-functions-before-upgrading-to-mysql-8-0/)*from here onwords 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**](https://dev.mysql.com/doc/refman/8.0/en/gis-wkt-functions.html).

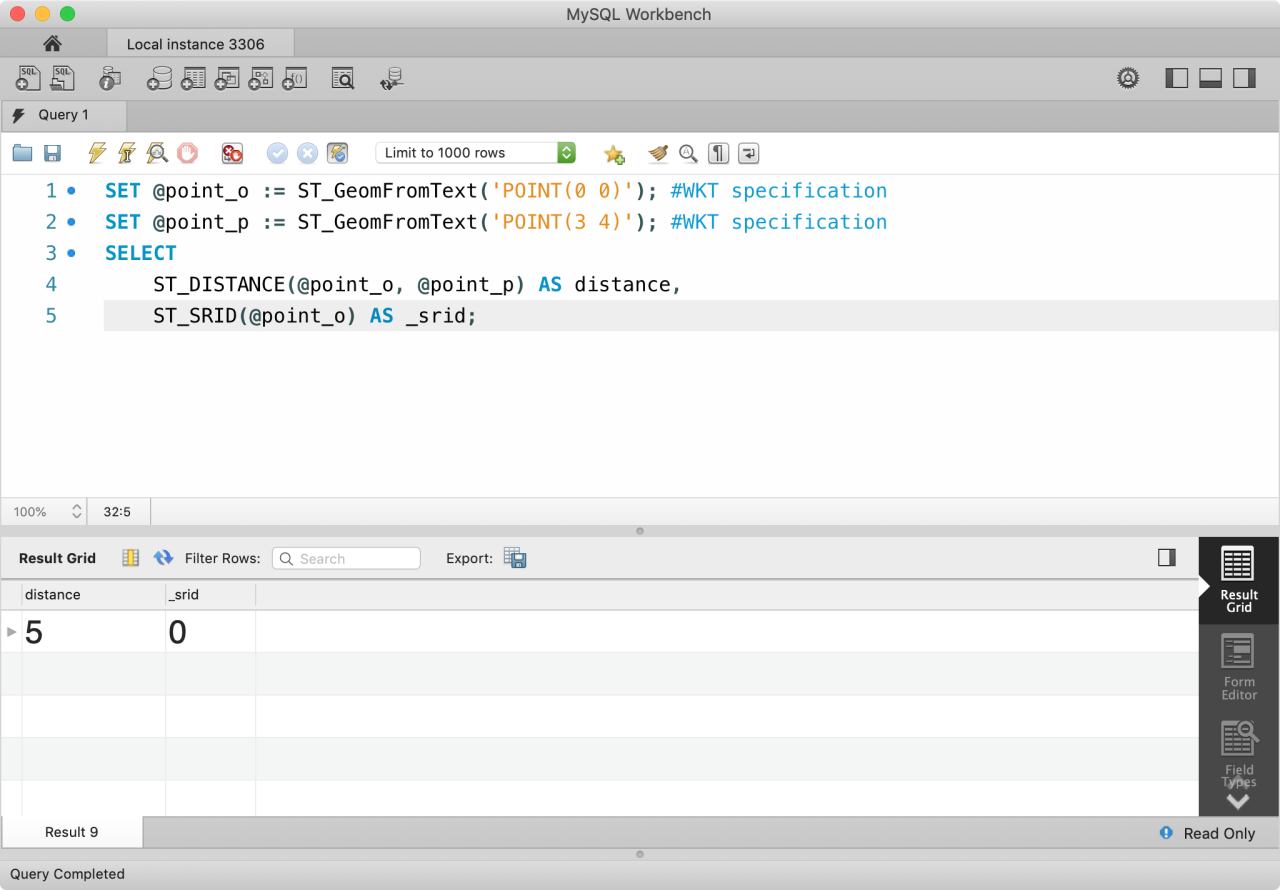
**ST\_GeomFromText** is a built-in function to create a geometry in given SRID from WKT specification. ST\_PointFromText can also be used to specifically state that we are creating POINT geometry but it’s just an alias.

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.

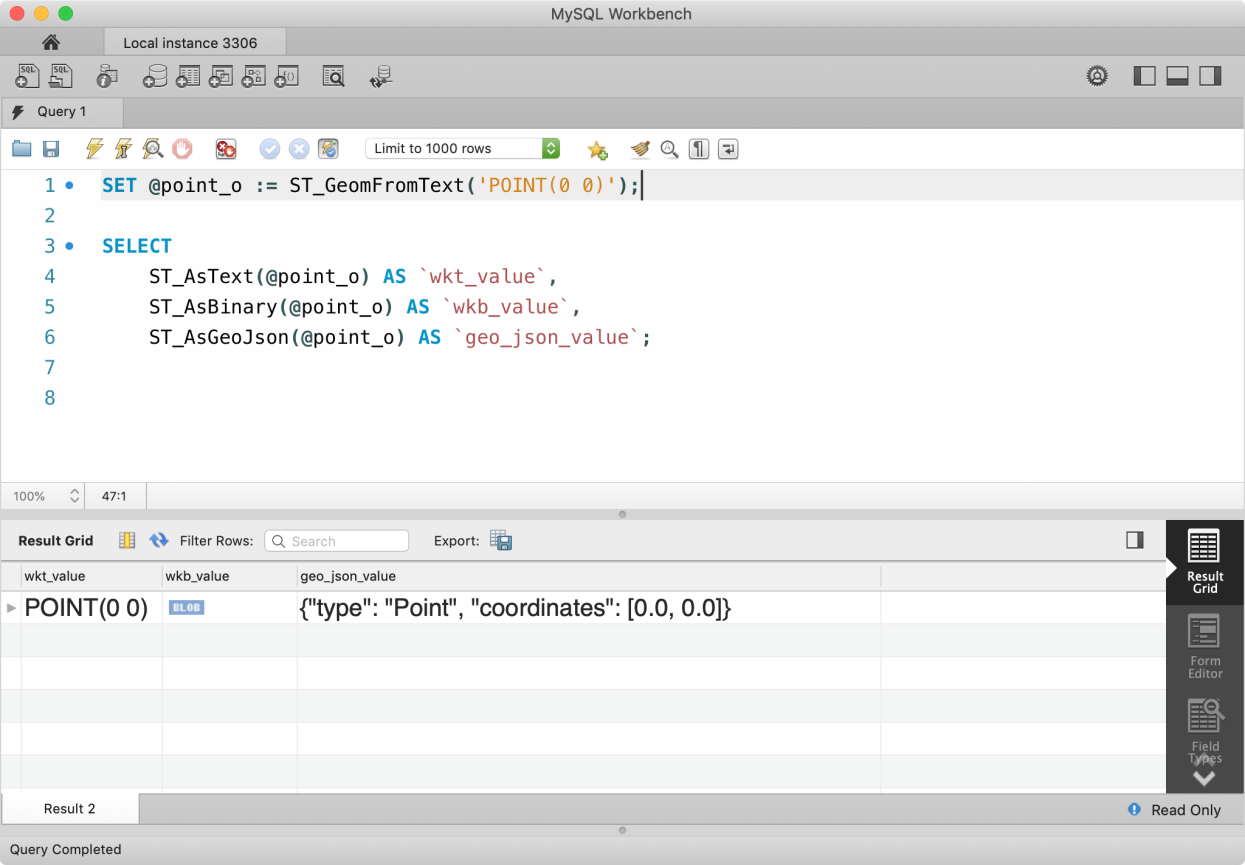
*options is additional information provided to ST\_GeomFromText function. As of now, it supports only axis-order key with the value being either long-lat, lat-long or srid-defined(*default*). This will come up later when SRID will be different than 0.*



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.

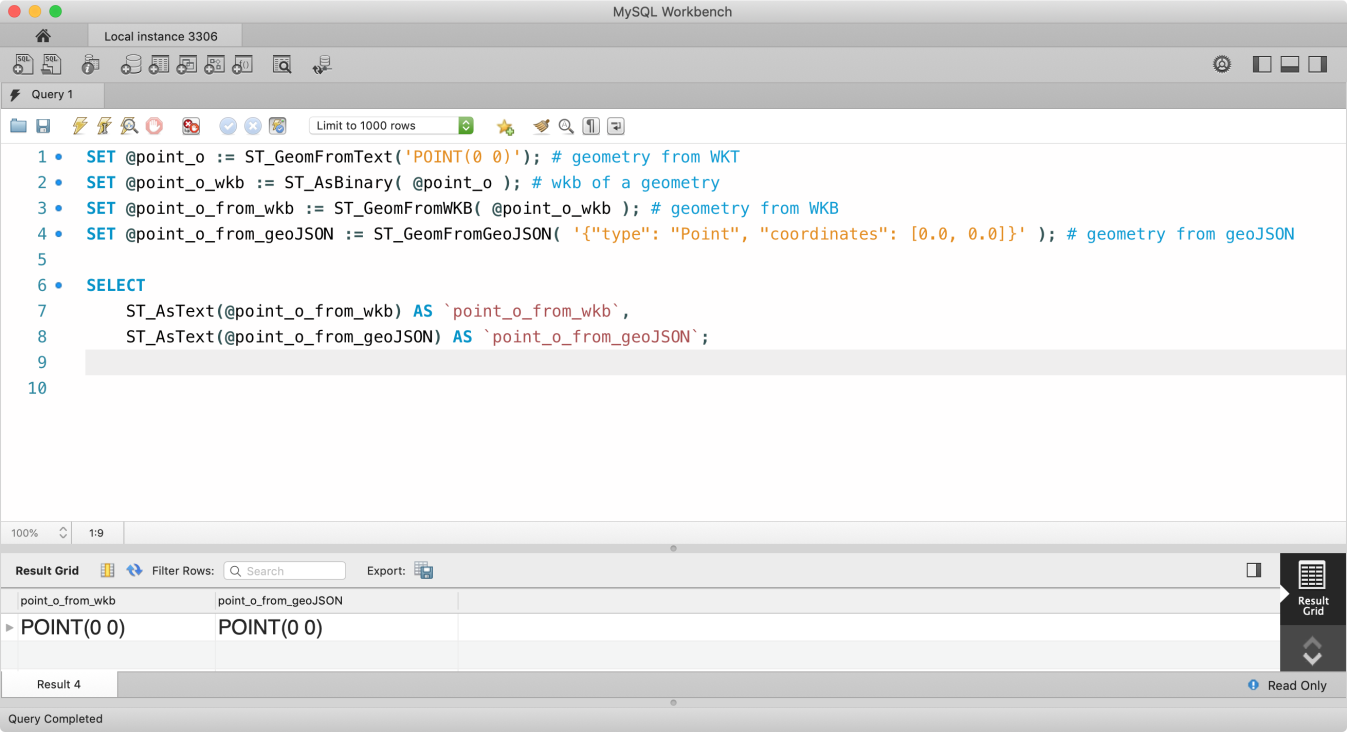
As we know, MySQL stores a geometry data type in BLOB format, but how can we make sure that the given point is correct if we can’t read it. This can be solved using MySQL’s built-in conversion functions, they are documented [**here**](https://dev.mysql.com/doc/refman/8.0/en/gis-format-conversion-functions.html). 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.

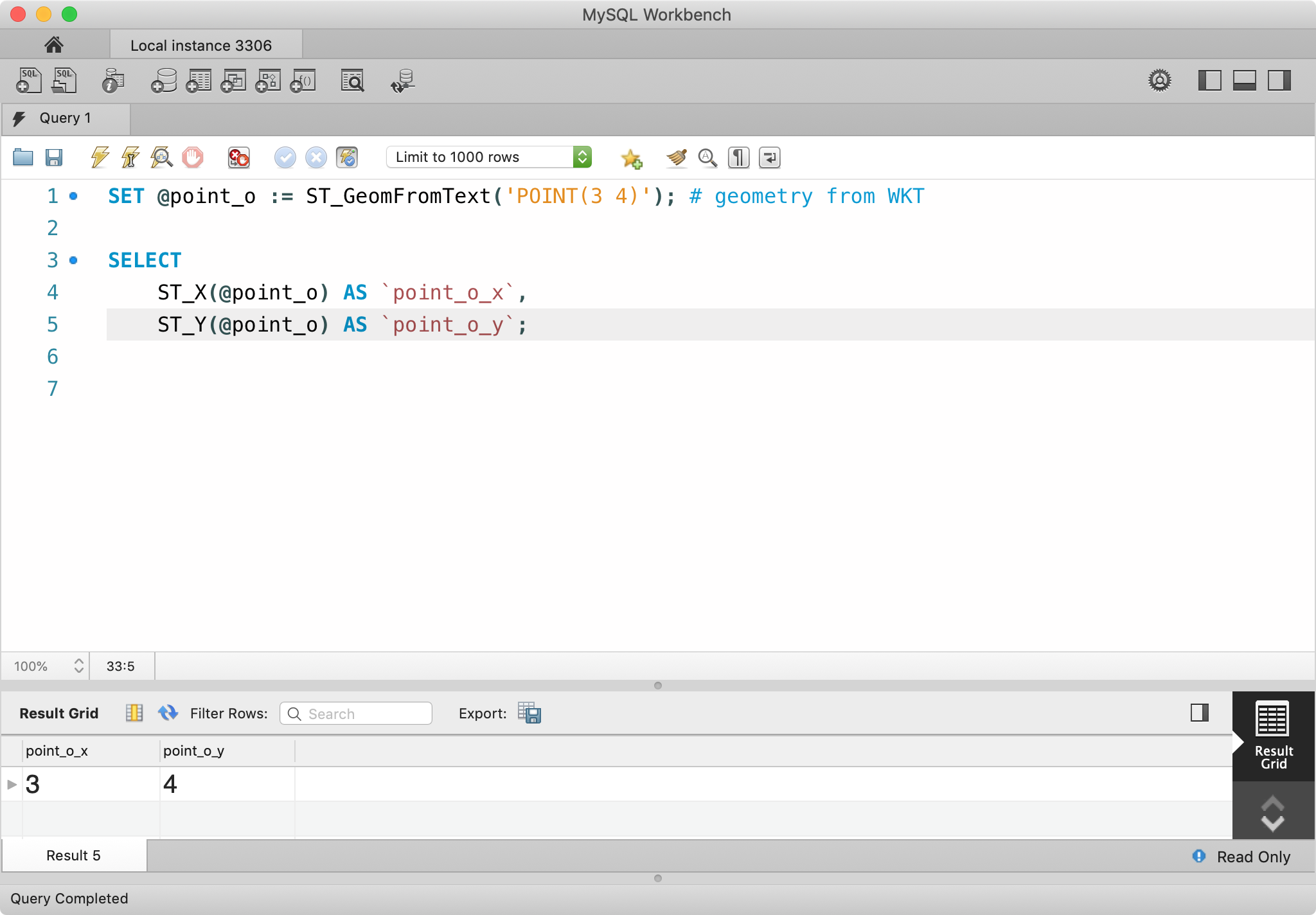


In the above example, we have converted a geometric shape of the type POINT into readable WKT value, binary WKB value and JSON format.

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.

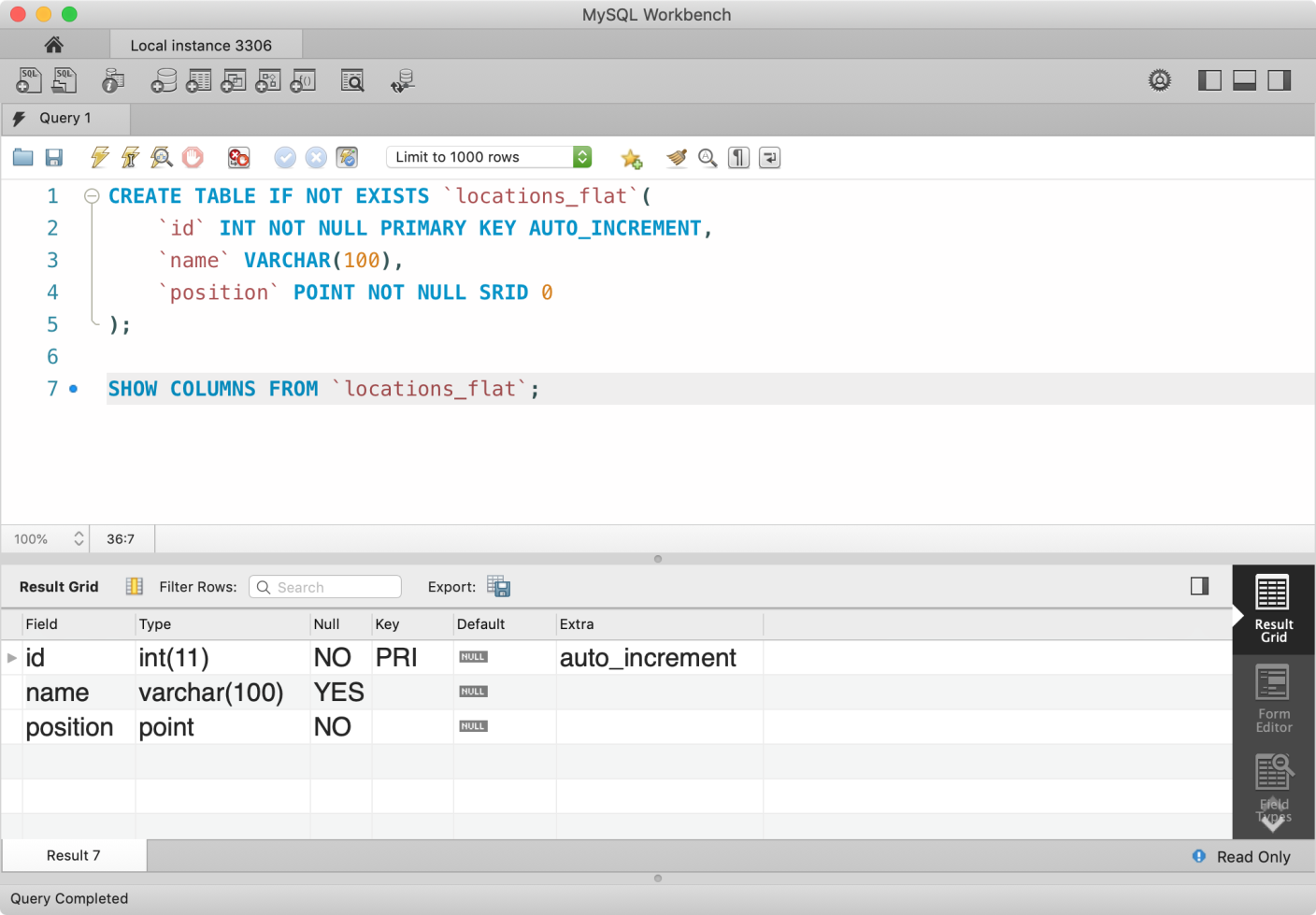


If you want to get X and Y coordinates of a POINT geometry, you can use ST\_X(g) and ST\_Y(g) built-in functions.



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`;



You can use POINT, POLYGON, LINESTRING and other geometrical types (mentioned [**here**](https://dev.mysql.com/doc/refman/5.7/en/spatial-type-overview.html)) 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, set 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;

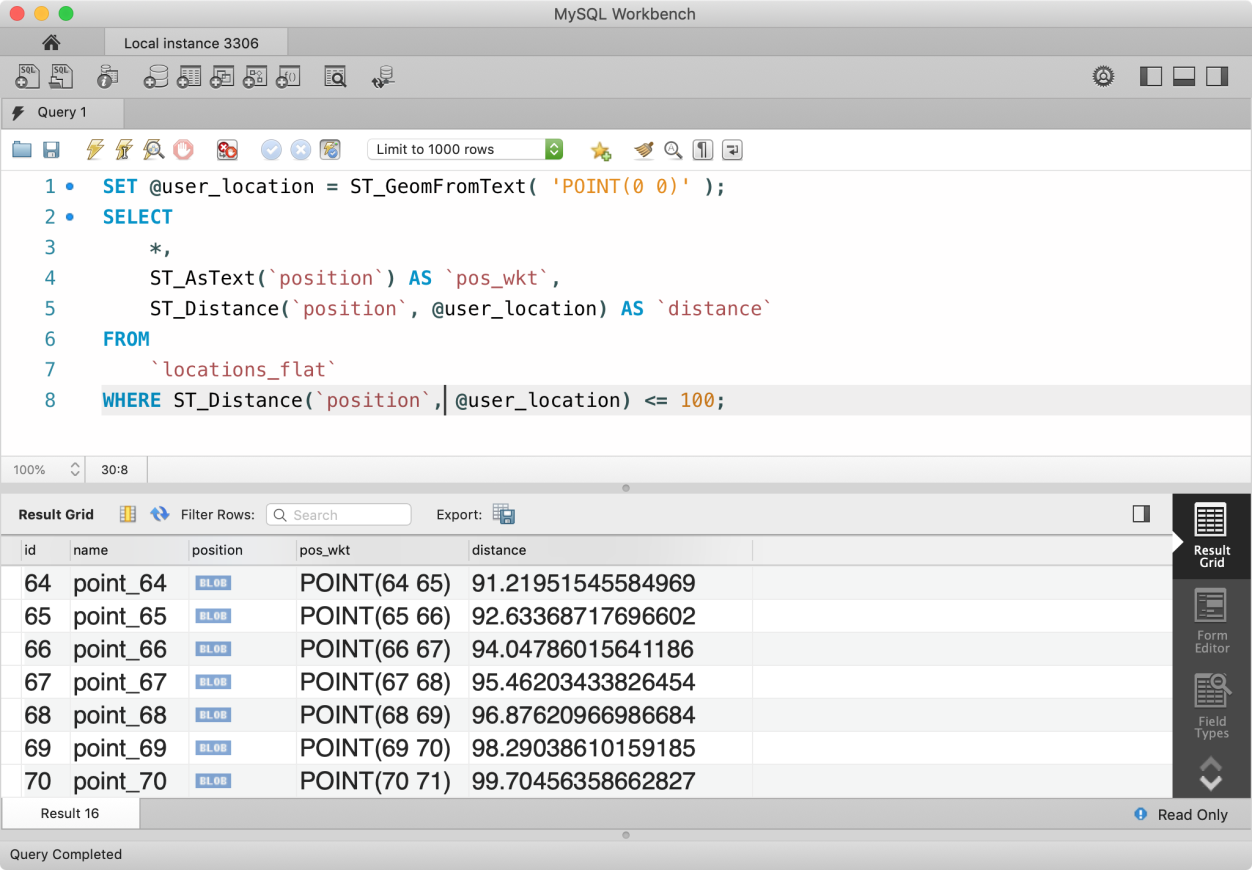


*It is perfectly legal to pass a column name to ST\_ built-in function as long as its input parameters are legal. Also, all ST\_ function names are case-insensitive. Hence, you can also use ST\_AsText as ST\_ASTEXT or st\_asText.*

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.

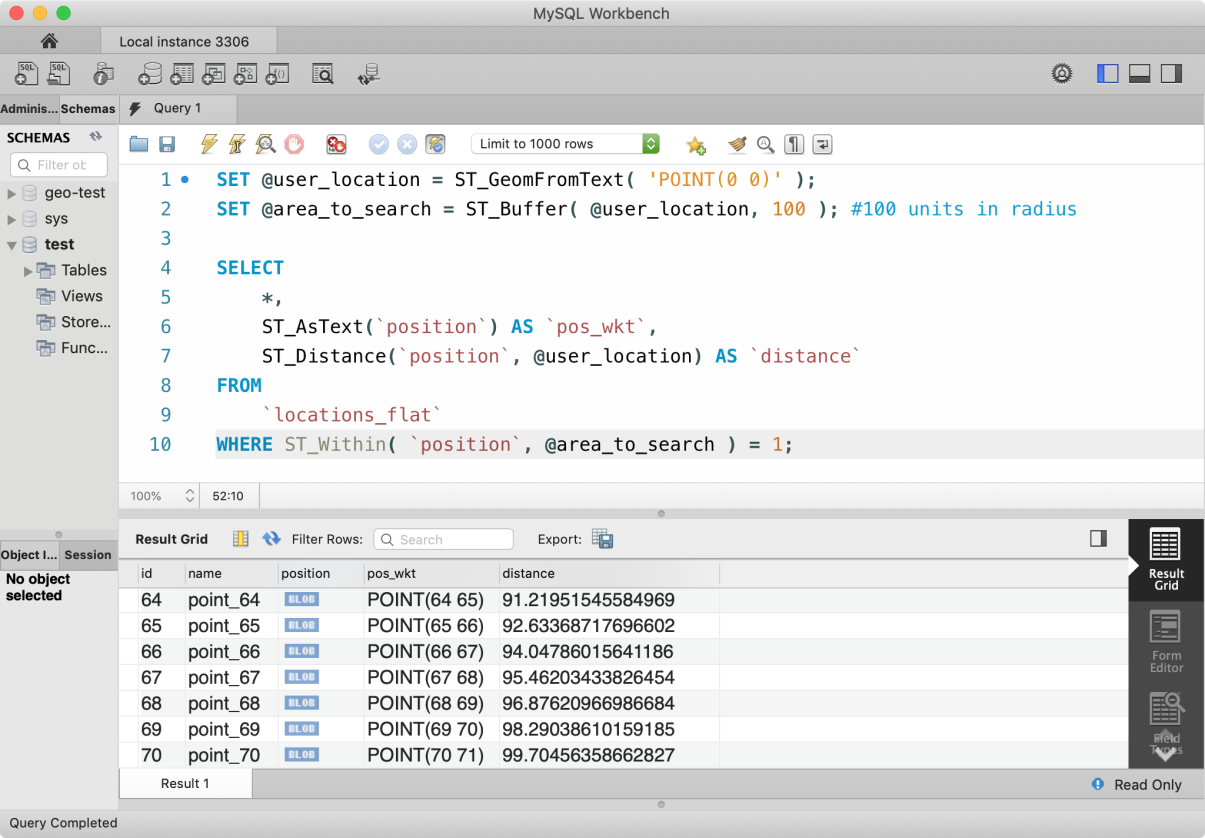


But we can also achieve this using ST\_Buffer(g, d) built-in function (documented [**here**](https://dev.mysql.com/doc/refman/8.0/en/spatial-operator-functions.html)) to create geometry (as stated in the documentation) that represents all points whose distance from the geometry value **g** is less than 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;



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**](https://en.wikipedia.org/wiki/B-tree). 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**](https://en.wikipedia.org/wiki/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**.

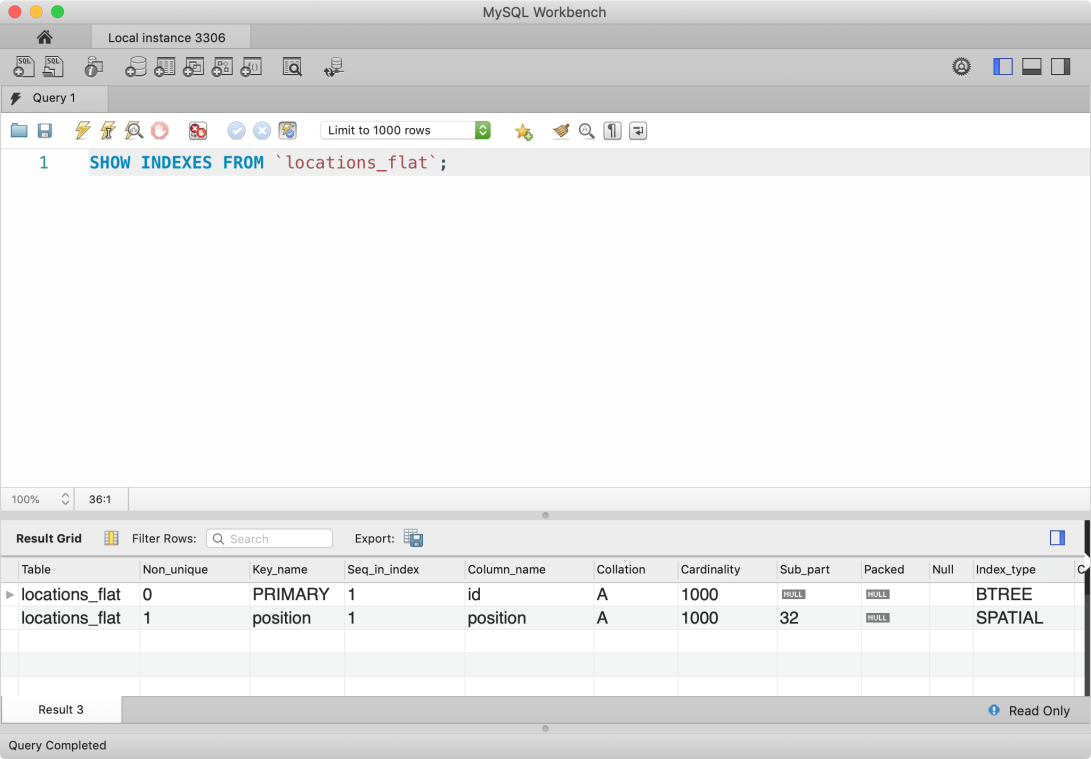
ALTER TABLE `locations\_flat` ADD SPATIAL INDEX(`position`);

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

CREATE SPATIAL INDEX position\_index ON locations\_flat(`position`);

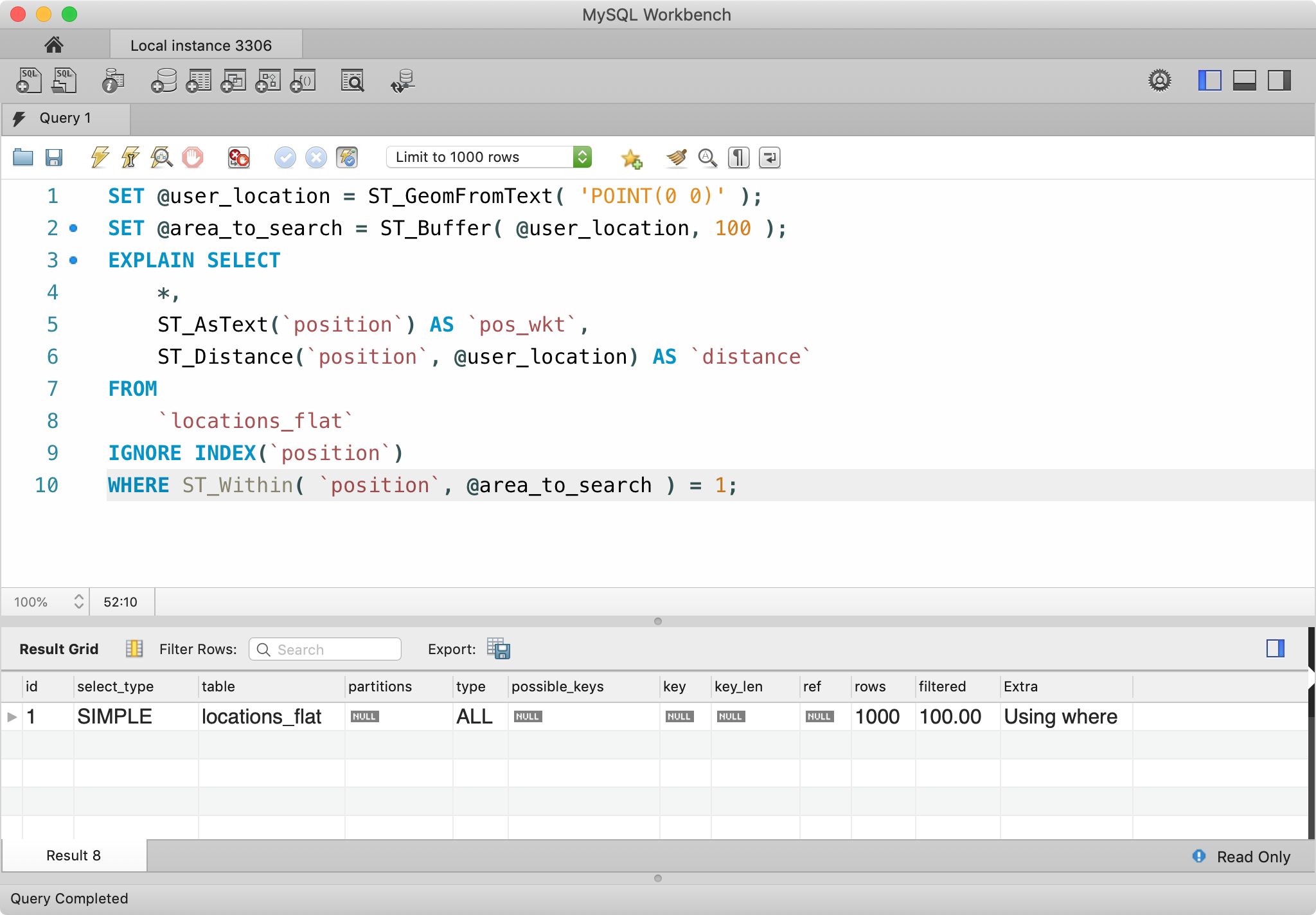
*⚠️ A spatial index can only be created on a column with geometry type. It must be a NOT NULL column and should contain data of only one SRID.*

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



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

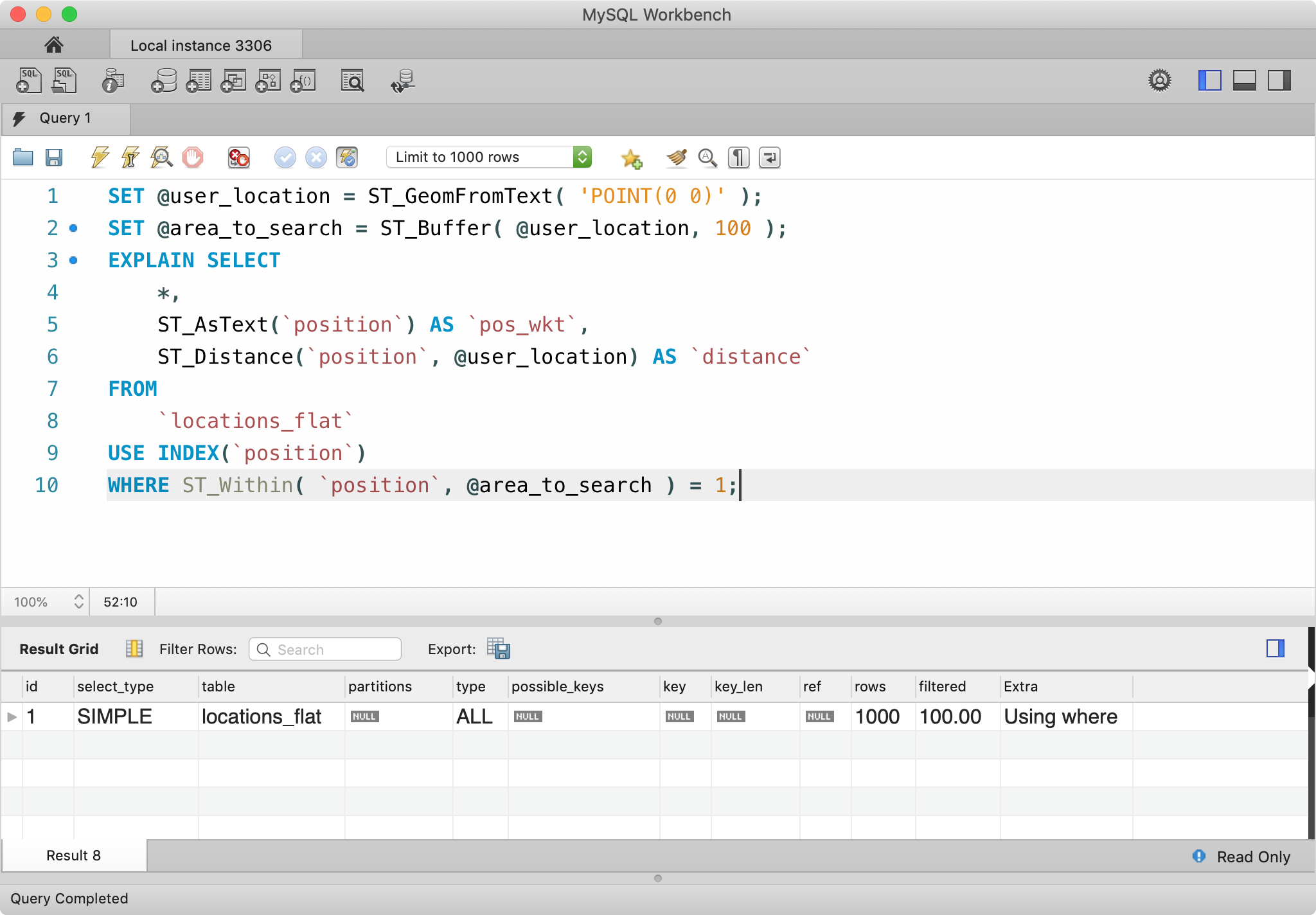
SET @user\_location = ST\_GeomFromText( 'POINT(0 0)' );  
SET @area\_to\_search = ST\_Buffer( @user\_location, 100 );  
EXPLAIN SELECT   
 \*,  
 ST\_AsText(`position`) AS `pos\_wkt`,  
 ST\_Distance(`position`, @user\_location) AS `distance`  
FROM  
 `locations\_flat`  
IGNORE INDEX(`position`)  
WHERE ST\_Within( `position`, @area\_to\_search ) = 1;



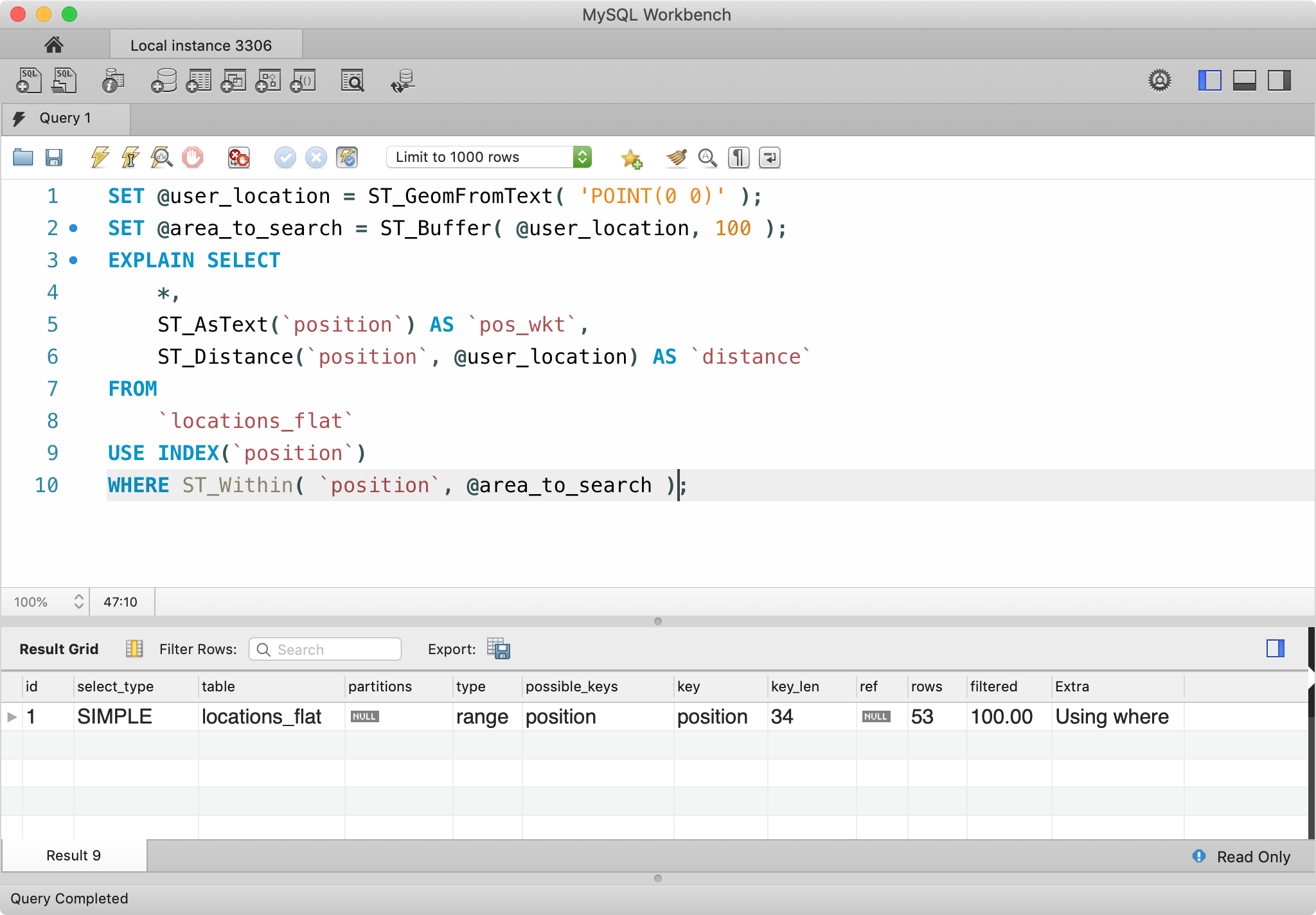
Seems like MySQL is using **full-table scan** and results show that 1000 rows are being searched for the match. Also, possible\_keys are empty which shows that no index was used to perform the search.

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;



Aha! We see no difference. Don’t worry, this is a ⚠️ [**documented MySQL bug**](https://bugs.mysql.com/bug.php?id=76384). 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**.



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.*

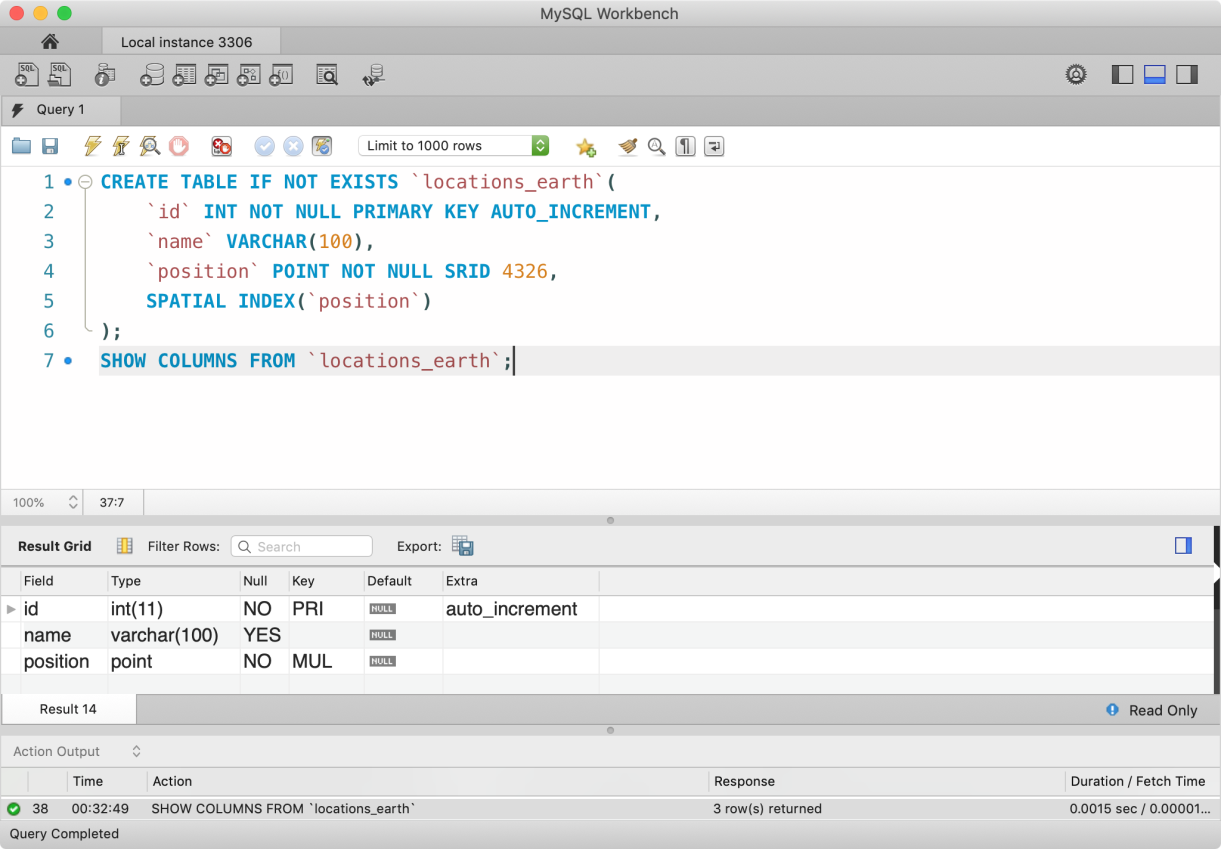
## Geographic Coordinate System

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`;



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.

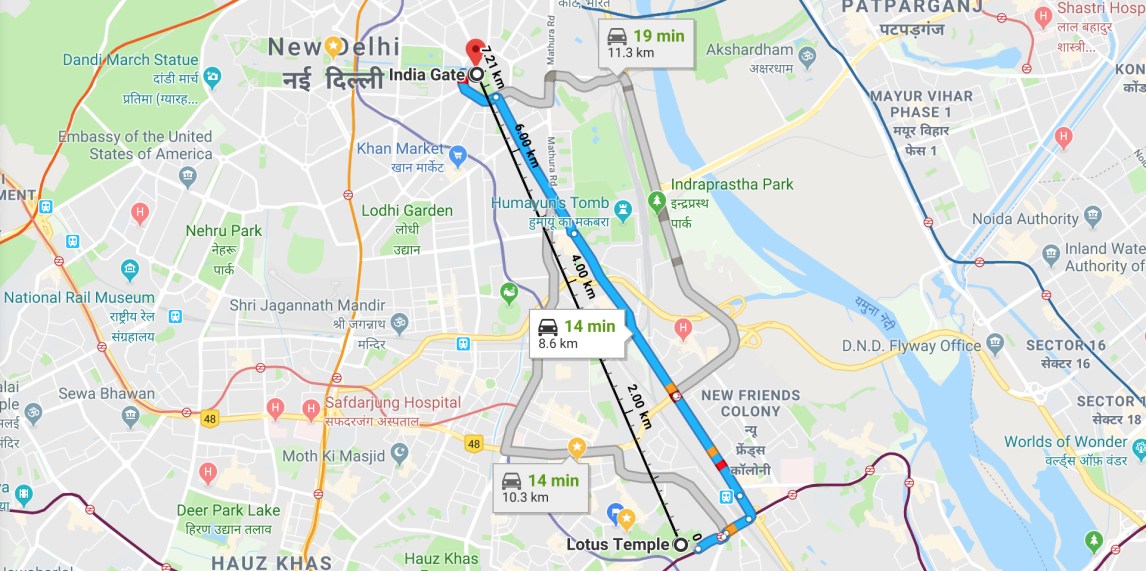
Let’s calculate the distance between two points on the globe. As we seen in the case of the 2D surface, we used ST\_Distance function to calculate the distance between two points. But in the case of a spherical coordinate system, we are going to use ST\_Distance\_Sphere function. The signature of ST\_Distance\_Sphere 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.

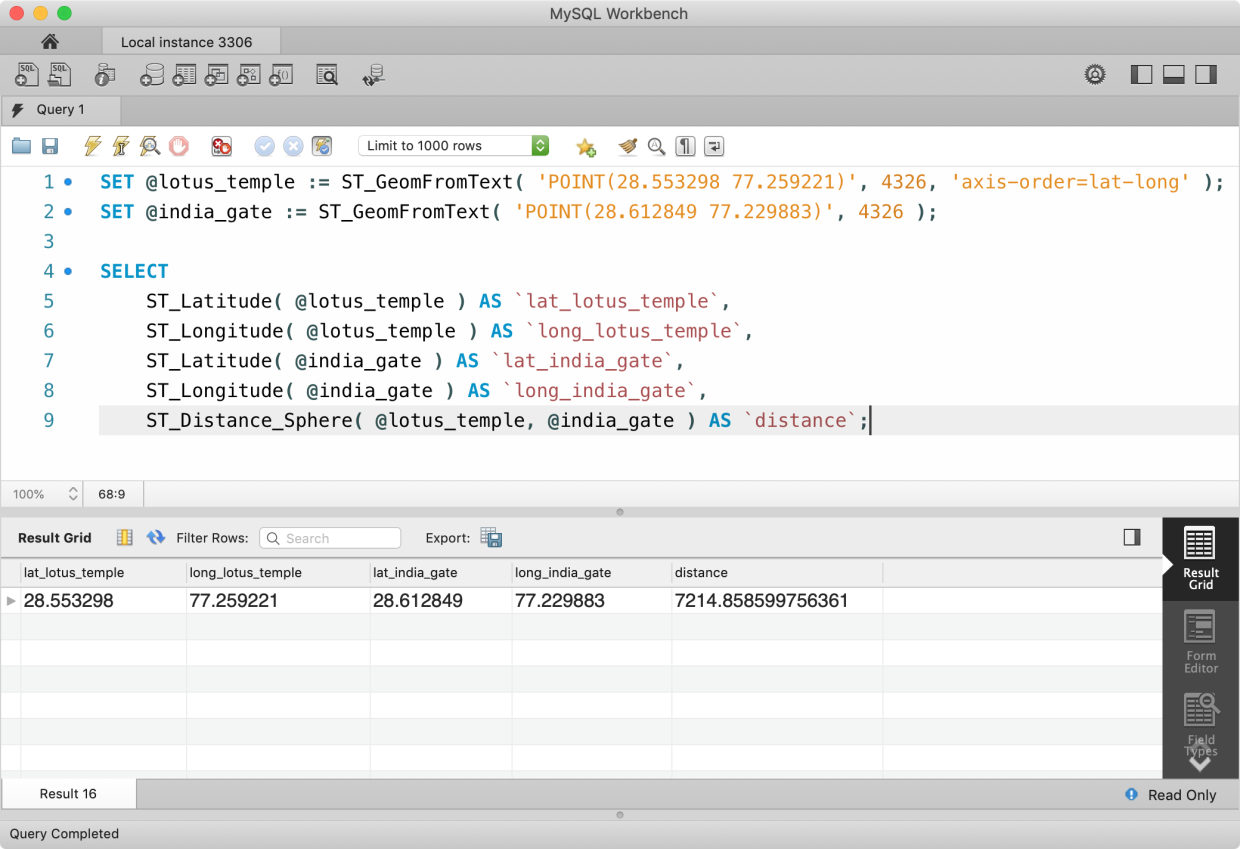
For the test and accuracy test of this function, let’s pick two random locations on the google map. Since I live in Delhi - India, I am going to pick **Lotus Temple**(28.553298, 77.259221) and **India Gate**(28.612849, 77.229883).

*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`;



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.

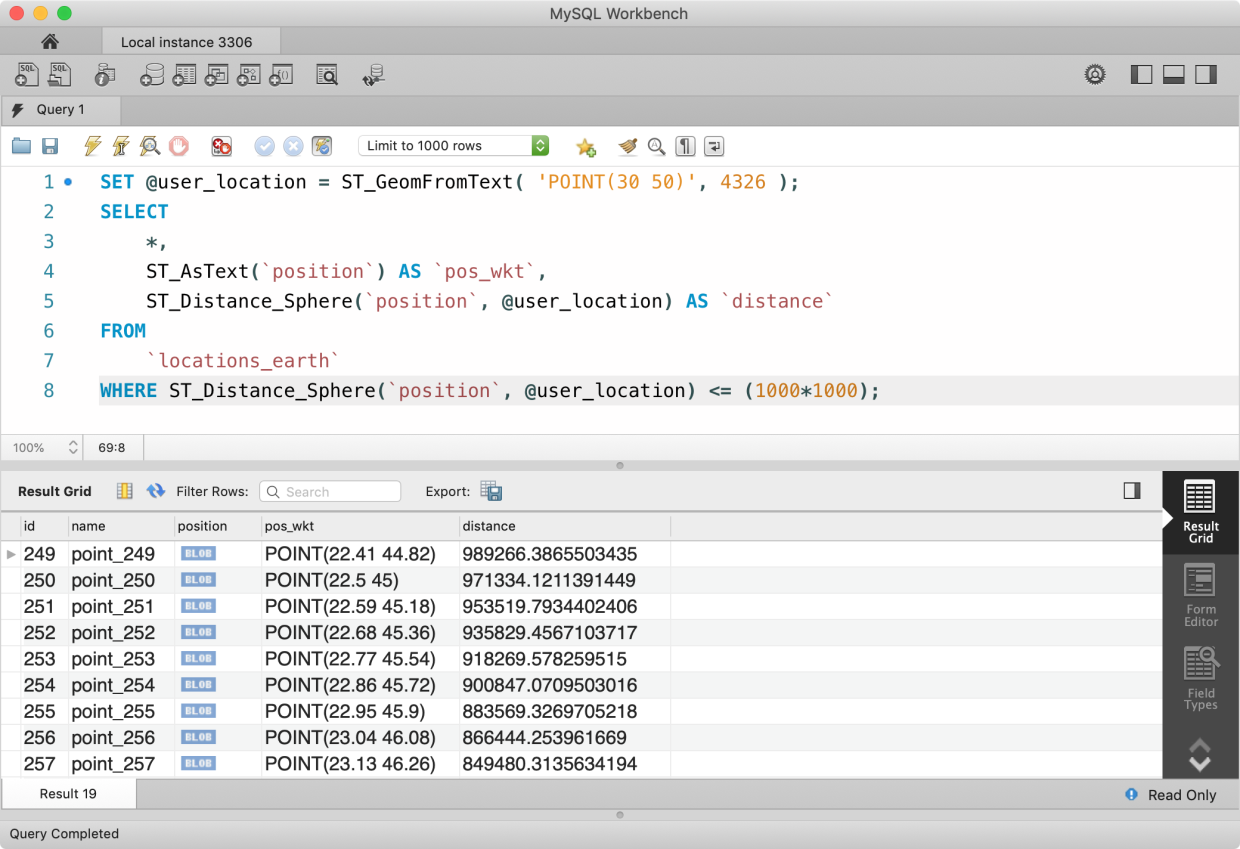
As you can see in the above query, we have used options argument of ST\_GeomFromText function to explicitly provide information about the axis order of POINT geometry in **WKT**. Also, we can use ST\_Latitude and ST\_Longitude to get **latitude** and **longitude** value geometry types of 4326 SRID.

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**](https://dev.mysql.com/doc/refman/8.0/en/spatial-operator-functions.html#function_st-transform) function. Seems like our luck is running out.

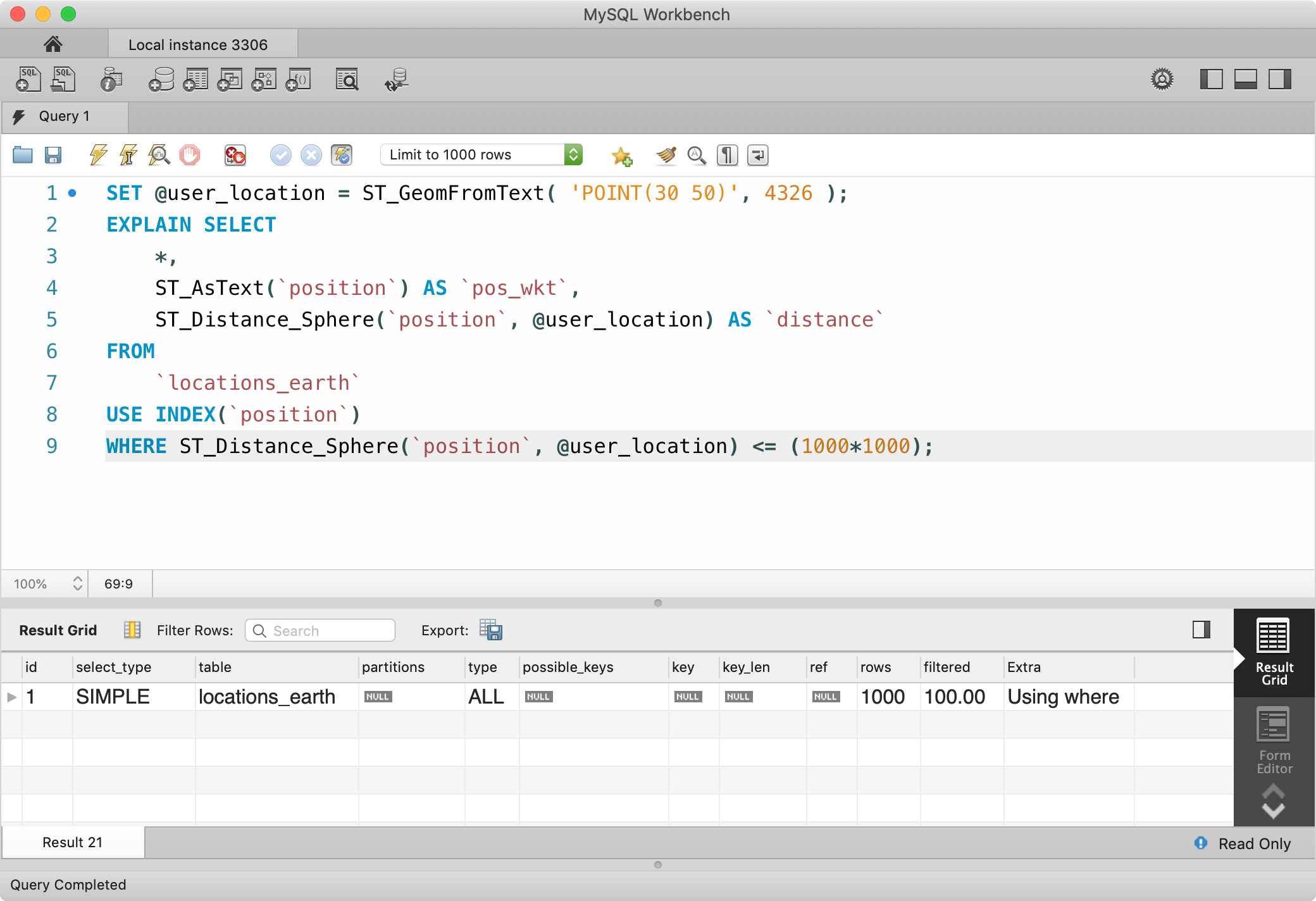
But we can use a simple but expensive approach to calculate distances of each point on the sphere from the reference point and check if they are within a given radius, like we did earlier.

Let’s say that we are looking for places in 1000km (1000 \* 1000m) 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);



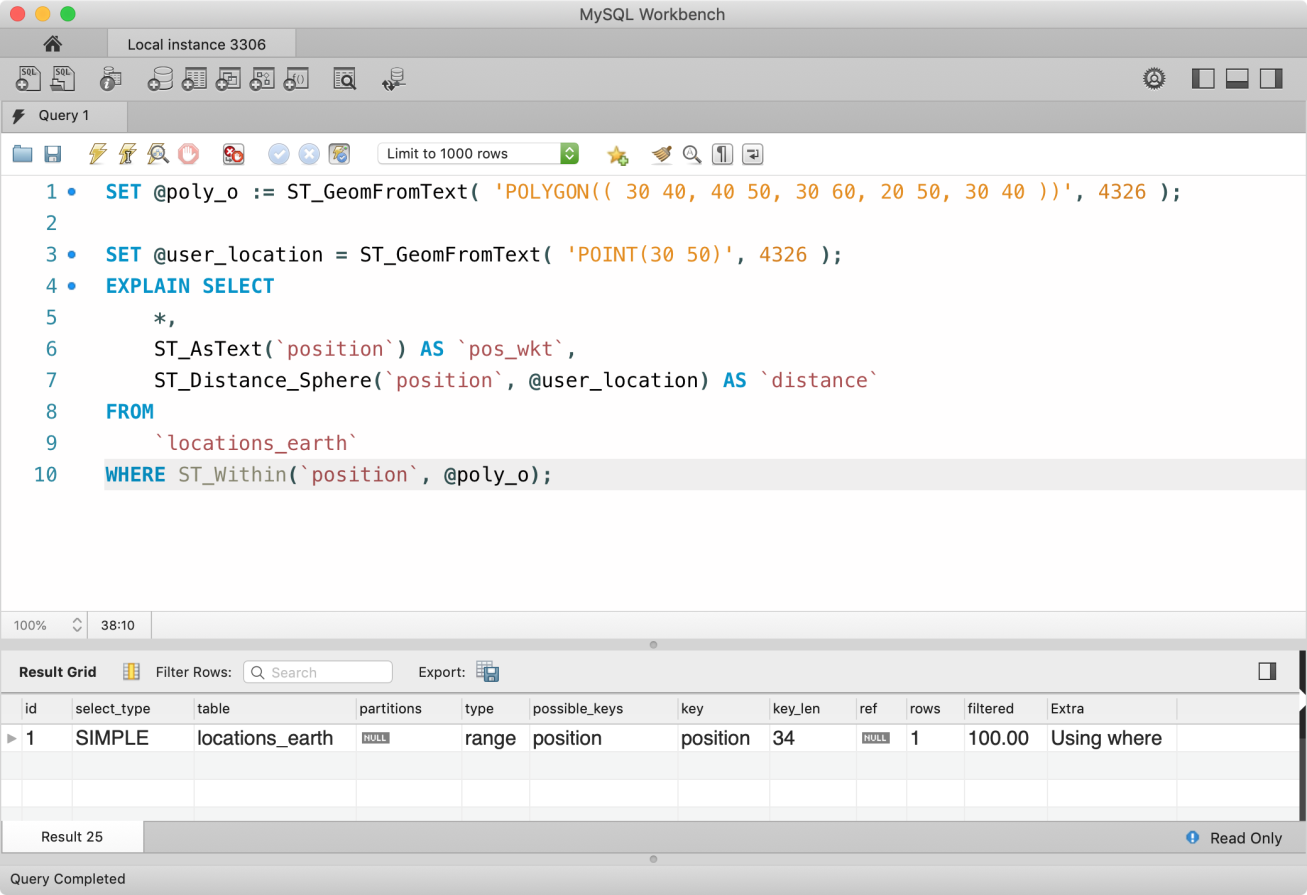
We have found around 87 results but let’s see how our query is doing.



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.



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**](https://www.postgresql.org/) database has far better support for spacial geographic data calculation using [**PostGIS extension**](http://www.postgis.net/). 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.