**Chapter 11 Spatial Queries**

**11.1 Introduction**

In this chapter, we are going to integrate our plant observations web map with spatial SQL queries to selectively load data from the CARTO database, based on geographical proximity. Specifically, we will build a web map where the user can click on any location, and as a result the nearest *n* plants observations to the clicked location will be loaded from the database and displayed, along with straight-line paths towards those plants. You can see the final result in Figure [11.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:example-11-04).

We will build the web map going through several steps, each time adding more functionality:

* Adding a marker at the clicked location on the map (Section [11.2.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#getting-click-coordinates))
* Adding a *custom* marker at clicked location, to distinguish it from the observation markers (Section [11.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#adding-custom-marker))
* Finding *n* nearest features, using a spatial SQL query, and adding them on the map (Section [11.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#adding-nearest-points-to-map))
* Drawing line segments from clicked location to the nearest features (Section [11.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#drawing-line-connectors))

**11.2 Adding markers on click**

**11.2.1 Getting click coordinates.**Our first step towards a web map that queries the plants table based on spatial proximity is to mark the clicked location on the map, while capturing its coordinates. The coordinates will be stored, to pass them on with a spatial SQL query (Section [9.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-queries)). We start from the basic map example-06-02.html from Section [6.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#creating-a-basic-web-map) and make changes on top of it. First, we modify the initially displayed map extent to a larger one:

**let** map = L.map("map").setView([32, 35], 8);

Next, we add a layer group named myLocation. This layer group will contain the clicked location marker, created on map click. As we have seen in Sections [7.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#adding-an-event-listener) and [10.5.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#automatic-updating), using a layer group will make it easy to remove the old marker before adding a new one, in case the user changed his/her mind and subsequently clicked on a different location:

**let** myLocation = L.layerGroup().addTo(map);

We then define an event listener, which will execute a function named mapClick each time the user clicks on the map. Remember the map click event and its .latlng property introduced in the example-06-08.html (Section [6.9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#introducing-map-events))? We use exactly the same principle here, only instead of adding a popup with the clicked coordinates in the clicked location, we are adding a *marker*. To do that, we first set the event listener for "click" events on the map object, referencing the mapClick function that we have yet to define:

map.addEventListener("click", mapClick);

Second, we define the mapClick function itself. The mapClick function needs to do two things:

* **Clear** the myLocation layer of any previous markers, from a previously clicked location (if any)
* **Add** a new marker to the myLocation layer, at the clicked coordinates, using the .latlng property of the event object

Here is the definition of the mapClick function:

**function** mapClick(e) {

myLocation.clearLayers();

L.marker(e.latlng).addTo(myLocation);

}

As a result, we now have a map (example-11-01.html) where the user can click, with the last clicked location displayed on the map (Figure [11.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:example-11-01)).

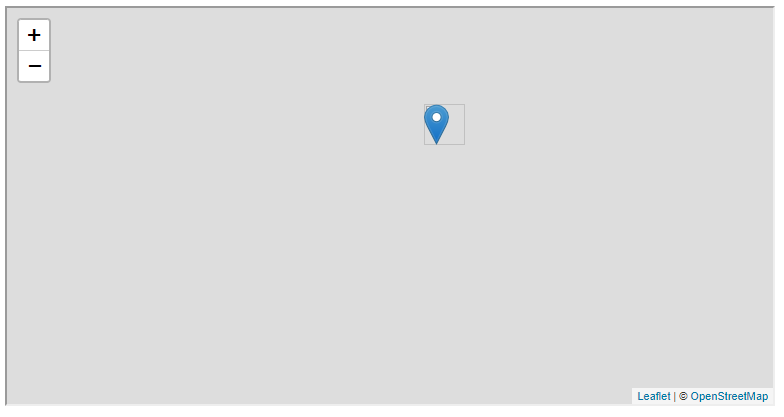


FIGURE 11.1: example-11-01.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-11-01.html)

**11.2.2 Adding a custom marker.**Shortly, we will write code to load plant observations next to to our clicked location, and display them as markers too (Figure [11.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:example-11-03)). To distinguish the marker for the clicked location from the markers for the plant observations, we need to override the default blue marker settings in Leaflet and display a different type of marker for one of the two layers. For example, we can use a *red* marker for the clicked location and keep using the default *blue* markers for denoting the plant locations (Figure [11.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:example-11-03)). To change marker appearance, however, we first need to understand a little better how it is defined in the first place.

The markers we see on a Leaflet map, created with L.marker (Section [6.6.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#adding-markers)), are in fact PNG images displayed at the specified locations placed on top of the map background. If you look closely, you will see that the default blue marker also has a “shadow” effect behind it, to create an illusion of depth, as if the marker “sticks out” of the map (Figure [11.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:default-custom-icon)). The shadow is a PNG image too, displayed behind the PNG image of the marker itself. We can use the *Inspect Element* mode (Section [1.9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#inspecting-elements)) in the **developer tools** to figure out which PNG image is actually being loaded by Leaflet, and where it comes from (Figure [11.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:inspecting-leaflet-icon)). Doing so reveals the following values for the src attributes of the <img> elements for the marker and marker shadow images:

* [http://localhost:8000/examples/css/images/marker-icon-2x.png](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\localhost_8000\examples\css\images\marker-icon-2x.png)
* [http://localhost:8000/examples/css/images/marker-shadow.png](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\localhost_8000\examples\css\images\marker-shadow.png)

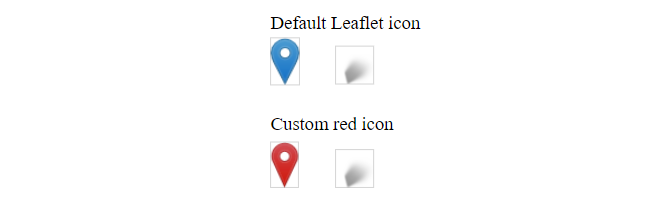


FIGURE 11.2: Default and custom images for drawing a Leaflet marker

Note that the prefix <http://localhost:8000/> is specific to a locally-served (Section [5.6.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\web-servers-1.html#local-with-python)) copy of the examples, and may be different when viewing them on a different server. The *local* Leaflet JavaScript library, which is included in example-11-01.html, looks for the marker PNG images in a sub-directory named images inside the directory where the leaflet.css file is placed (Section [A](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\associated-files.html#associated-files)). That is the reason for us placing the images in the sub-directory, which we mentioned in Section [6.5.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#including-leaflet-css-js). In case we use a *remote* copy of the Leaflet library (Section [6.5.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#including-leaflet-css-js)), the markers would have been loaded from remote PNG files, such as the following ones:

* [https://unpkg.com/leaflet@1.7.1/dist/images/marker-icon-2x.png](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\unpkg.com\leaflet%401.7.1\dist\images\marker-icon-2x.png)
* [https://unpkg.com/leaflet@1.7.1/dist/images/marker-shadow.png](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\unpkg.com\leaflet%401.7.1\dist\images\marker-shadow.png)

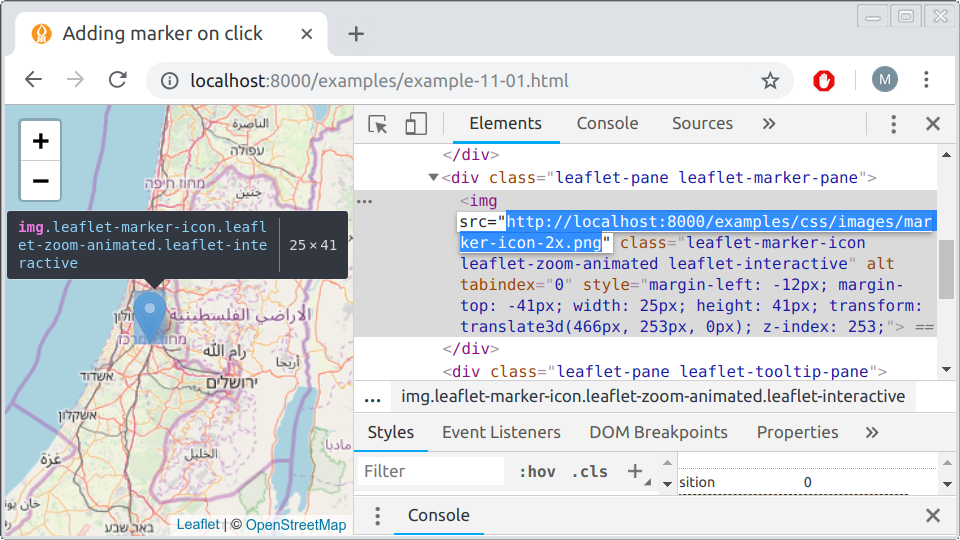


FIGURE 11.3: Inspecting Leaflet icon <img> element

You can follow these URLs to download and inspect the PNG images in any graphical viewer or editor, such as [**Microsoft Paint**](https://en.wikipedia.org/wiki/Microsoft_Paint).

To distinguish our clicked location from other markers on the map, we will use a different marker for the clicked location. In our example, we will use a marker PNG image, which is similar to the default one, only colored in red instead of blue (Figure [11.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:default-custom-icon)). The PNG images for the red icon and its shadow are also included in the online book materials, in the images sub-directory (Appendix [A](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\associated-files.html#associated-files)):

* images/redIcon.png
* images/marker-shadow.png

To set a custom Leaflet marker, based on the PNG images redIcon.png and marker-shadow.png, you need to place these files on your server. For example, the files can be placed in a folder named images within the root directory of the web page (see Section [5.5.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\web-servers-1.html#file-structure)), as given in the online materials. A custom marker using these PNG images, assuming they are in the images folder, is then defined with the L.icon function as follows:

**let** redIcon = L.icon({

iconUrl: "images/redIcon.png",

shadowUrl: "images/marker-shadow.png",

iconAnchor: [13, 41]

});

The L.icon function requires a path to the PNG images for the icon (iconUrl), and, optionally, a path to the PNG image for the shadow (shadowUrl). The other important parameter is iconAnchor, which sets the anchor point, i.e., the exact icon image pixel which corresponds to the point coordinates where the marker is initiated. The custom icon object is assigned to a variable, hereby named redIcon, which we can later use to draw custom markers on the map with L.marker.

What is the meaning of iconAnchor, and why did we use the value of [13, 41]? The iconAnchor parameter specifies which pixel of the marker PNG image will be placed on the [lon, lat] point where the marker is initialized. To determine the right anchor point coordinates, we need to figure out the size (in pixels) of our particular marker, and the image region where we would like the anchor to be placed. The image size can be determined by viewing the PNG file properties, for example clicking on the file with the right mouse button, choosing **Properties** and then the **Details** tab. In our case, the size of the redIcon.png image is 25 by 41 pixels (Figure [11.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:red-icon-properties)).

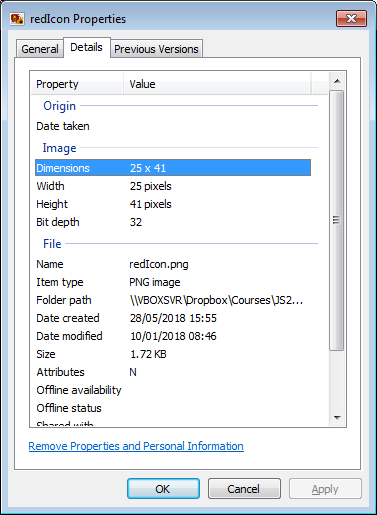


FIGURE 11.4: File properties for the red icon redIcon.png. The highlighted entry shows image dimensions in pixels (25 x 41).

Conventionally, image coordinates are set from a [0, 0] point at the top-left corner of the icon[103](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fn103). The anchor point for our particular icon should be at its tip, at the bottom-middle. Starting from the top-left corner [0, 0], this means going all the way *down* on the Y-axis, then half-way to the *right* on the X-axis. Since the PNG image size is [25, 41], this means we set the the pixel on the center of the X-axis ([13, ...]) and bottom of the Y-axis ([..., 41]), thus the anchor value of [13, 41] (Figure [11.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:icon-anchor)).

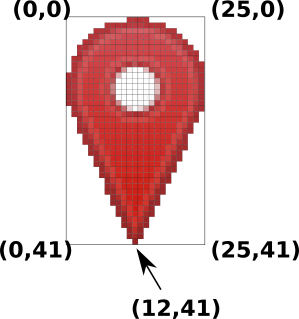


FIGURE 11.5: PNG image for the red marker redIcon.png, with coordinates of the four corners and the coordinate of the marker anchor point. Note that the coordinate system starts from the top-left corner, with reversed Y-axis direction.

Now that the redIcon object is ready and the PNG images are in place, we can replace the expression for adding a marker inside the map click event listener in example-11-01.html:

L.marker(e.latlng).addTo(myLocation);

with a new expression that loads our *custom* redIcon marker in example-11-02.html:

L.marker(e.latlng, {icon: redIcon}).addTo(myLocation);

The resulting map example-11-02.html is shown in Figure [11.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:example-11-02). Clicking on the map now adds the red marker icon instead of the default blue one.

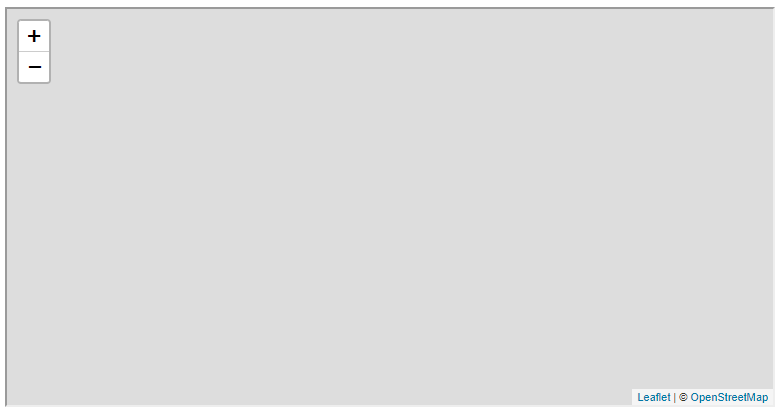


FIGURE 11.6: example-11-02.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-11-02.html)

You can use just about any PNG image as a marker icon this way, not just a differently-colored default marker. There are many places where you can find PNG images suitable for map markers, such as the *Maps and Navigation* [collection](https://www.flaticon.com/packs/maps-and-navigation-20) on [https://www.flaticon.com](https://www.flaticon.com/) website[104](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fn104).

* Download one of the PNG images for custom map icons from <https://www.flaticon.com/packs/maps-and-navigation-20>, or use any other small PNG image that you have.
* Adapt example-11-02.html to display your custom icon instead of the red icon.

**11.3 Spatial PostGIS operators**

**11.3.1 Overview.**Now that we know how to add a custom red marker on map click, we are going to add some code into our mapClick function incorporating a *spatial* SQL query (Section [9.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-queries)) to find the closest plants to the clicked location. As a result, each time the user clicks on a new point and mapClick adds a red marker, the function also queries for the nearest plants and displays them on the map (Figure [11.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:example-11-03)).

As we have already discussed in Section [9.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-queries), spatial databases are characterized by the fact that their tables may contain a geometry column. On CARTO, the geometry column is conventionally named the\_geom (Section [9.7.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#query-example)). The geometry column is composed of encoded geometric data in the WKB format, which can be used to transform returned table records to spatial formats, such as GeoJSON. For example, in Chapters [9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#databases)–[10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#non-spatial-queries-1) we used SELECT statements to return GeoJSON objects based on data from the plants table on CARTO. These queries utilized the the\_geom geometry column to create the "geometry" properties of the GeoJSON layer. However, the geometry column can also be used to make various spatial *calculations* on our data, using spatial SQL operators and functions. A very common example of a spatial calculation is the calculation of **geographical distance**.

In the next two sections ([11.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#geographical-distance)–[11.3.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#sphere-vs-spheroid)), we will discuss the structure of the required spatial SQL query to return nearest features. Then, in Section [11.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#adding-nearest-points-to-map), we will see how to implement this type of SQL query in our Leaflet web map.

**11.3.2 Geographical distance.**To find the five plant observations *nearest* to our clicked location, we can use the following query, which was introduced as an example of spatial SQL syntax in Section [9.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-queries):

**SELECT** name\_lat, obsr\_date, ST\_AsText(the\_geom) **AS** geom

**FROM** plants

**ORDER** **BY**

the\_geom::geography <->

ST\_SetSRID(

ST\_MakePoint(34.810696, 31.895923), 4326

)::geography

**LIMIT** 5;

This query returns the nearest five observations from the plants table, based on distance to a specific point [34.810696, 31.895923]. In this section, we will explain the structure of the query more in depth.

First of all, as we already saw in Chapters [9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#databases)–[10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#non-spatial-queries-1), limiting the returned table to the first n records can be triggered using the term LIMIT n, as in LIMIT 5 or LIMIT 25. However, we need the five *nearest* plants, not just the five plants incidentally being first in terms of the table row ordering. This means the data need to be ordered before being returned.

As an example of *non-spatial* ordering, the table can be ordered with ORDER BY followed by column name(s) to select the five plant observations with the lowest (or highest) values for the given variable(s). We already saw one example of non-spatial ordering when producing the alphabetically ordered list of species in Section [10.4.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#finding-unique-values). As another example, consider the following query, which returns the five *earliest* plant observation records. The ORDER BY obsr\_date part means the table is ordered based on the values in the obsr\_date (observation date) column:

**SELECT** name\_lat, obsr\_date, ST\_AsText(the\_geom) **AS** geom

**FROM** plants

**ORDER** **BY** obsr\_date

**LIMIT** 5;

which gives the following result:

name\_lat | obsr\_date | geom

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

Iris haynei | 1900-01-01 | POINT(35.654005 32.741372)

Iris atrofusca | 1900-01-01 | POINT(35.193367 31.44711)

Iris atrofusca | 1900-01-01 | POINT(35.189142 31.514754)

Iris vartanii | 1900-01-01 | POINT(35.139696 31.474149)

Iris haynei | 1900-01-01 | POINT(35.679761 32.770133)

(5 rows)

Try the corresponding CARTO SQL API query, choosing the CSV output format, to examine the above result on your own:

https://michaeldorman.carto.com/api/v2/sql?format=CSV&q=

SELECT name\_lat, obsr\_date, ST\_AsText(the\_geom) AS geom

FROM plants ORDER BY obsr\_date LIMIT 5

Indeed, all returned records are from the 1900-01-01, which is the earliest date in the obsr\_date field. *Spatial* ordering is similar, only that the table records are ordered based on their spatial arrangement, namely based on their distances to another spatial feature, or set of features. In other words, with spatial ordering, instead of ordering by non-spatial column values, we are ordering by geographical distances, which are calculated using the geometry column. In the above spatial query example for getting the five nearest points from a given location [34.810696, 31.895923], the only part different from the non-spatial query example is basically just the ORDER BY term. Instead of the following ORDER BY term for non-spatial ordering, based on obsr\_date values:

**ORDER** **BY** obsr\_date

we use the following ORDER BY term, for spatial ordering, based on distance from a specific point [34.810696, 31.895923]:

**ORDER** **BY**

the\_geom::geography <->

ST\_SetSRID(ST\_MakePoint(34.810696, 31.895923), 4326)::geography

The result of the spatial query is as follows:

name\_lat | obsr\_date | geom

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

Lavandula stoechas | 1931-04-30 | POINT(34.808564 31.897377)

Bunium ferulaceum | | POINT(34.808504 31.897328)

Bunium ferulaceum | 1930-02-23 | POINT(34.808504 31.897328)

Silene modesta | 1900-01-01 | POINT(34.822295 31.900125)

Corrigiola litoralis | 2016-01-30 | POINT(34.825931 31.900792)

(5 rows)

These are the locations of the five nearest plants to the specified location. You can see that the longitude and latitude values are fairly similar to [34.810696, 31.895923], reflecting the fact that the points are proximate. Again, you can experiment with this query in the CARTO SQL API:

https://michaeldorman.carto.com/api/v2/sql?format=CSV&q=

SELECT name\_lat, obsr\_date, ST\_AsText(the\_geom) AS geom

FROM plants

ORDER BY the\_geom::geography <-> ST\_SetSRID(

ST\_MakePoint(34.810696, 31.895923), 4326)::geography LIMIT 5

As you probably noticed, the expression used for spatial ordering is more complicated than simply a column name such as obsr\_date:

the\_geom::geography <->

ST\_SetSRID(ST\_MakePoint(34.810696, 31.895923), 4326)::geography

In addition to the geometry column name (the\_geom), this particular ORDER BY term contains four spatial PostGIS functions and operators, which we will now explain:

* ST\_MakePoint—Creates a point geometry
* ST\_SetSRID—Sets the coordinate reference system (CRS)
* ::geography—Casts to the geography type
* <->—Calculates 2D distance

A two-dimensional point geometry is constructed using the ST\_MakePoint function, given two coordinates x and y, in this case 34.810696 and 31.895923, respectively. Thus, the expression ST\_MakePoint(34.810696, 31.895923) defines a single geometry of type "Point", which we can use in spatial calculations. The ST\_SetSRID function then sets the coordinate reference system (CRS) for the geometry. The 4326 argument is the EPSG code of the WGS84 geographical projection (i.e., lon/lat) (Figure [6.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#fig:EPSG-4326-3857)).

The ::geography part [*casts*](http://postgis.net/workshops/postgis-intro/geography.html) the geometry to a special type called geography, thus determining that what follows is a calculation with **spherical geometry**, which is the appropriate way to do distance-based calculations with lon/lat data. With ::geography, distance calculations give the spherical [**Great Circle**](https://en.wikipedia.org/wiki/Great_circle) distance, in meters (Figure [12.10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:solution-12)). Omitting the ::geography part is equivalent to using the default ::geometry type, implying that what follows is a **planar geometry** calculation. Planar geometry calculations are only appropriate for projected data, which we do not use in this book. Using ::geometry when calculating distances on lon/lat data gives straight-line euclidean distances, in degree units, which is almost always inappropriate.

The <-> operator returns the [2D distance](https://postgis.net/docs/geometry_distance_knn.html) between two sets of geometries. Since we set ::geography, the result represents spherical Great Circle distance in meters. In the present example, we are calculating the distances between the\_geom, which is the geometry column of the plants table, and an individual point. The result is a series of distances in meters, corresponding to all features of the plants table.

Finally, the series of distances is passed to the ORDER BY keyword, thus rearranging the table based on the calculated distances, from the smallest to largest, i.e., from nearest observation to furthest. The LIMIT 5 part then takes the top five records, which are the five nearest ones to [34.810696, 31.895923].

**11.3.3 Sphere vs. spheroid**

As another demonstration of the four spatial PostGIS functions discussed in Section [11.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#geographical-distance) (above), consider the following small query. This query calculates the distance between two points [0, 0] and [0, 1] in geographic coordinates (lon/lat), i.e., the length of one degree of longitude along the equator:

**SELECT**

(ST\_SetSRID(ST\_MakePoint(0, 0), 4326)::geography <->

ST\_SetSRID(ST\_MakePoint(1, 0), 4326)::geography) / 1000

**AS** dist\_km;

According to the result, the distance is 111.195 km:

dist\_km

------------------

111.195079734632

(1 row)

In this query, we are manually constructing two points in lon/lat, [0, 0] and [1, 0], using ST\_MakePoint, ST\_SetSRID and ::geography. The 2D distance operator <-> is the applied on the points to calculate the Great Circle distance between them, in meters. Dividing the result by 1000 transforms the distance from meter to kilometer units[105](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fn105).

The true distance between [0, 0] and [1, 0], however, is [111.320 km](https://en.wikipedia.org/wiki/Longitude#Length_of_a_degree_of_longitude) and not 111.195. What is the reason for the discrepancy? The reason is that the <-> operator, though using spherical geometry, relies on a **sphere** model of the earth, rather than the more accurate **spheroid** model. In PostGIS, the more accurate but somewhat slower distance calculation based on a spheroid can be obtained with ST\_Distance instead of <->, as in:

**SELECT** ST\_distance(

ST\_SetSRID(ST\_MakePoint(0, 0), 4326)::geography,

ST\_SetSRID(ST\_MakePoint(1, 0), 4326)::geography) / 1000

**AS** dist\_km;

This gives the more accurate result of 111.319 km:

dist\_km

-----------------

111.31949079327

(1 row)

Though ST\_distance gives the more accurate estimate, the calculation takes longer. For example, finding the 5 nearest neighbors from the plants table took 0.17 seconds using the <-> operator, compared to 0.37 seconds with ST\_Distance, on an average laptop computer. This is a more than twice longer calculation time. Although, in this particular example, exactly the same five plants are returned in both cases, theoretically the ordering may differ among the two methods in some cases, due to small differences in distance estimates between the sphere and spheroid models. In practice, the trade-off between speed and accuracy should always be considered when choosing the right distance-based calculation given the application requirements, dataset resolution and dataset size. With small amounts of data and/or high accuracy requirements ST\_Distance should be preferred; otherwise the accuracy given by <-> may be sufficient.

* What do you think will happen if we omit the ::geography keyword from both places where it appears in the above query?
* Check your answer using the CARTO SQL API.

**11.4 Adding nearest points to map**

We are now ready to take the SQL spatial query from Section [11.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#geographical-distance) and incorporate it into the mapClick function, so that the nearest 25 plants are displayed along with the red marker. The key here is that we are going to make the spatial query *dynamic*, each time replacing the proximity search according to the location clicked by the user on the web map. Unlike in Section [11.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#geographical-distance), where the longitude and latitude were hard-coded into the SQL query ([34.810696, 31.895923]), we are going to generate the SQL query by pasting user-clicked coordinates into the SQL query “template.” Specifically, we will use longitude and latitude returned from our map click event, e.latlng (Section [11.2.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#getting-click-coordinates)).

We proceed, modifying example-11-02.html. First, we add two more variable definitions: a layer group for the nearest plant markers (plantLocations), and the URL prefix for querying the CARTO SQL API (url). Along with the the layer group for the clicked location (myLocation), which we already defined in the previous example (Section [11.2.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#getting-click-coordinates)), the variable definition part in our <script> is now composed of the following three expressions:

**let** myLocation = L.layerGroup().addTo(map);

**let** plantLocations = L.layerGroup().addTo(map);

**let** url = "https://michaeldorman.carto.com/api/v2/sql?format=GeoJSON&q=";

The remaining code changes are all inside our mapClick function. In its new version, the function will not only display the clicked location, but also the locations of 25 nearest observations of rare plants. Recall that the previous version of the mapClick function in example-11-02.html (Section [11.2.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#getting-click-coordinates)) merely cleared the old marker and added a new one according to the e.latlng object:

**function** mapClick(e) {

myLocation.clearLayers();

L.marker(e.latlng, {icon: redIcon}).addTo(myLocation);

}

In the new version of the function, we add four new expressions. First, we define a new local variable clickCoords capturing the contents of e.latlng (Section [6.9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#introducing-map-events)) for that particular event. This is just a convenience, for typing clickCoords instead of e.latlng in the various places where we will use the clicked coordinates in the new function code body:

**let** clickCoords = e.latlng;

Second, we make sure that the nearest plant observations are cleared between consecutive clicks, just like the red marker is. The following expression takes care of clearing any previously loaded plantLocations contents:

plantLocations.clearLayers();

In the third expression, we dynamically compose the SQL query to get 25 nearest records from the plants table, considering the clicked location. We basically replace the hard-coded lon/lat coordinates from the specific SQL query discussed in Section [11.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#geographical-distance) with the lng and lat properties of the clickCoords object. The result, named sqlQueryClosest, is the specific SQL query string needed to obtain the 25 nearest plant observations from our currently clicked location. This is conceptually similar to the way that we dynamically constructed an SQL query to select the observations of a particular species (Section [10.5.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#automatic-updating)):

**let** sqlQueryClosest =

"SELECT name\_lat, the\_geom FROM plants" +

"ORDER BY the\_geom::geography <-> ST\_SetSRID(ST\_MakePoint(" +

clickCoords.lng + "," + clickCoords.lat +

"), 4326)::geography LIMIT 25";

Fourth, we use the sqlQueryClosest string to request the nearest 25 observations from CARTO, and add them to the plantLocations layer. The popup, in this case, contains the Latin species name (name\_lat) displayed in italics:

fetch(url + sqlQueryClosest)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

L.geoJSON(data, {

onEachFeature: **function**(feature, layer) {

layer.bindPopup("<i>" + feature.properties.name\_lat + "</i>");

}

}).addTo(plantLocations);

});

Here is the complete code for the new version of the mapClick function:

**function** mapClick(e) {

*// Get clicked coordinates*

**let** clickCoords = e.latlng;

*// Clear map*

myLocation.clearLayers();

plantLocations.clearLayers();

*// Add location marker*

L.marker(clickCoords, {icon: redIcon}).addTo(myLocation);

*// Set SQL query*

**let** sqlQueryClosest =

"SELECT name\_lat, the\_geom FROM plants" +

"ORDER BY the\_geom::geography <-> ST\_SetSRID(ST\_MakePoint(" +

clickCoords.lng + "," + clickCoords.lat +

"), 4326)::geography LIMIT 25";

*// Get GeoJSON & add to map*

fetch(url + sqlQueryClosest)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

L.geoJSON(data, {

onEachFeature: **function**(feature, layer) {

layer.bindPopup("<i>" + feature.properties.name\_lat + "</i>");

}

}).addTo(plantLocations);

});

}

The resulting map example-11-03.html is shown in Figure [11.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:example-11-03). The red marker denotes the clicked location, while the blue markers denote the 25 nearest plant observations loaded from the database.

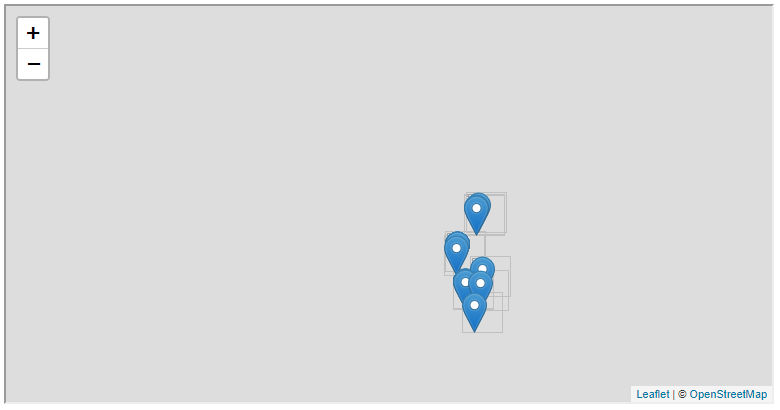


FIGURE 11.7: example-11-03.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-11-03.html)

**11.5 Drawing line connectors**

To highlight the direction and distance from the clicked location to each of the nearest plants, the final version example-11-04.html adds code for drawing **line segments** between the focal point and each observation (Figure [11.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:example-11-04)). This is a common visualization technique, also known as a “spider diagram” or “desire lines” (e.g., in [ArcGIS documentation](http://desktop.arcgis.com/en/arcmap/latest/extensions/business-analyst/create-spider-diagrams-desire-lines.htm)). Line segments are frequently used to highlight the association between a focal point or points, and their associated (usually nearest) surrounding points from a different layer. For example, the lines can be used to visualize the association between a set of business stores and the customers who are potentially affiliated with each store.

To draw each of the line segments connecting the red marker (clicked location) with one of the blue ones (plant observation), we can take the two pairs of coordinates and apply the L.polyline function (Section [6.6.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#adding-lines)) on them. The principle is similar to the function we wrote in the exercise for Chapter [3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#javascript-basics) (Section [3.12](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#exercise-2)). Suppose that the clicked location coordinates are [lon0, lat0] and the coordinates of the nearest plant observations are [lon1, lat1], [lon2, lat2], [lon3, lat3], etc. The coordinate arrays of the corresponding line segments, connecting each pair of points, are then:

* [[lon0, lat0], [lon1, lat1]] for the first segment
* [[lon0, lat0], [lon2, lat2]] for the second segment
* [[lon0, lat0], [lon3, lat3]] for the third segment
* etc.

Each of these coordinate arrays can be passed to the L.polyline function to draw the corresponding segment connecting a pair of points, much like the line segment connecting Building 72 with the Library (Figure [6.9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#fig:example-06-04)). Note that we will actually be using reversed coordinate pairs, such as [lat0, lon0], as expected by all Leaflet functions for drawing shapes, including L.polyline (Section [6.5.9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#creating-map-object)).

To implement the above procedure of drawing line segments, we first create yet another layer group named lines at the beginning of our script, right below the expressions where we already defined the myLocation and plantLocations layer groups:

**let** lines = L.layerGroup().addTo(map);

Accordingly, inside the mapClick function we clear all previous line segments before new ones are drawn, with the following expression clearing the lines layer group, right below the expressions where we clear myLocation and plantLocations:

lines.clearLayers();

Moving on, we are ready to actually draw the line segments. Inside the L.geoJSON function call, in the previous example-11-03.html (Section [11.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#adding-nearest-points-to-map)) we only binded the Latin species name popup like so:

L.geoJSON(data, {

onEachFeature: **function**(feature, layer) {

*// Bind popup*

layer.bindPopup("<i>" + feature.properties.name\_lat + "<i>");

}

}).addTo(plantLocations);

Now, in example-11-04.html, we add three more expressions inside the onEachFeature option. These expressions are used to draw a line segment between the “central” point where the red marker is, which is stored in clickCoords, and the currently-added GeoJSON point, which is given in feature as part of the onEachFeature iteration (Section [8.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#constructing-popups-from-data)):

L.geoJSON(data, {

onEachFeature: **function**(feature, layer) {

*// Bind popup*

layer.bindPopup("<i>" + feature.properties.name\_lat + "<i>");

*// Draw line segment*

**let** layerCoords = feature.geometry.coordinates;

**let** lineCoords = [

[clickCoords.lat, clickCoords.lng],

[layerCoords[1], layerCoords[0]]

];

L.polyline(lineCoords, {

color: "red", weight: 3, opacity: 0.75

}).addTo(lines);

}

}).addTo(plantLocations);

The new code section, right below the //Draw line segment comment, is composed of three expressions, which we now discuss step by step.

In the first expression, we extract the coordinates of the current plant observation, and assign them to a variable named layerCoords. As discussed in Section [8.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#constructing-popups-from-data), as part of the onEachFeature iteration, the currently processed GeoJSON feature is accessible through the feature parameter. Just like we are accessing the current species name with feature.properties.name\_lat to include it in the popup, we can also extract the point coordinates with feature.geometry.coordinates. The feature.geometry.coordinates property of a GeoJSON "Point" geometry is an array of the form [lon, lat] (Section [7.3.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#single-part-geometries)), which we store in a variable named layerCoords:

**let** layerCoords = feature.geometry.coordinates;

In the second expression, we build the segment coordinates array by combining the coordinates of the focal point stored in clickCoords with the coordinates of the current plant observation stored in layerCoords. Again, while layerCoords are stored as [lon, lat] according to the GeoJSON specification, L.polyline expects [lat, lon] (Section [6.6.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#adding-lines)). This is why the layerCoords array is reversed with [layerCoords[1], layerCoords[0]]. The complete segment coordinates array is assigned into a variable named lineCoords:

**let** lineCoords = [

[clickCoords.lat, clickCoords.lng],

[layerCoords[1], layerCoords[0]]

];

In the third expressions, lineCoords is passed to the L.polyline function to create a line layer object. The .addTo method is then applied, in order to add the segment to the lines layer group and thus actually draw it on the map:

L.polyline(lineCoords, {color: "red", weight: 3, opacity: 0.75})

.addTo(lines);

The resulting map example-11-04.html is shown in Figure [11.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:example-11-04). Clicking on the map now displays both nearest plant observations and the connecting line segments.

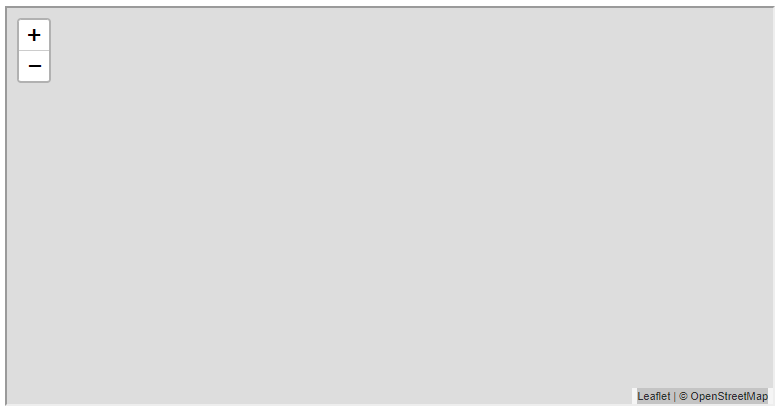


FIGURE 11.8: example-11-04.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-11-04.html)

**11.6 Exercise**

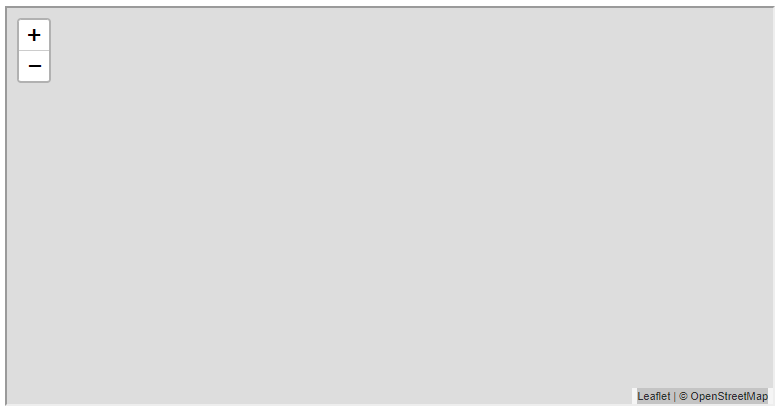
* Start with example-11-04.html and modify the code so that the popup for each of the nearest plants also specifies its *distance* to the queried point (Figure [11.9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:solution-11)).
* 

FIGURE 11.9: solution-11.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\solution-11.html)

* To get the distances, you can use the following SQL query example which returns the five nearest plants along with the distance in *kilometers* to the specific point [34.810696, 31.895923]. The distances are given in an additional column named dist\_km.

**SELECT** name\_lat,

(

the\_geom::geography <->

ST\_SetSRID(ST\_MakePoint(34.810696, 31.895923), 4326)::geography

) / 1000 **AS** dist\_km,

ST\_AsText(the\_geom) **AS** geom

**FROM** plants

**ORDER** **BY** dist\_km

**LIMIT** 5;

* The query gives the following result. Note the dist\_km column, which contains the distances in kilometers:

name\_lat | dist\_km | geom

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

Lavandula stoechas | 0.258166195505697 | POINT(34.808564 31.897377)

Bunium ferulaceum | 0.25928724896688 | POINT(34.808504 31.897328)

Bunium ferulaceum | 0.25928724896688 | POINT(34.808504 31.897328)

Silene modesta | 1.19050813583538 | POINT(34.822295 31.900125)

Corrigiola litoralis | 1.53676126266166 | POINT(34.825931 31.900792)

(5 rows)

* Here is the corresponding CARTO SQL API query you can experiment with:

https://michaeldorman.carto.com/api/v2/sql?format=CSV&q=

SELECT name\_lat, (the\_geom::geography <->

ST\_SetSRID(ST\_MakePoint(34.810696, 31.895923), 4326

)::geography) / 1000 AS dist\_km, ST\_AsText(the\_geom) as geom

FROM plants ORDER BY dist\_km LIMIT 5

* Remember that in the actual code you need to use format=GeoJSON instead of format=CSV, and the\_geom instead of ST\_AsText(the\_geom) to get GeoJSON rather than CSV. It is also useful to round the distance values before putting them into the popup (e.g., to two decimal places, as shown in Figure [11.9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:solution-11)). This can be done with JavaScript, applying the .toFixed method with the required number of digits on the numeric variable:

x = 2.321342;

x.toFixed(2); *// Returns "2.32"*

1. Measuring the coordinates from top-left and using an inverted Y-axis (values increase when moving *down*) is an old convention in computer graphics, emerging in many different situations. Going back to example-04-07.html (Section [4.10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#the-event-object)), you will see that mouse coordinates in the browser window are also measured from the top-left corner.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fnref103)
2. For other examples of using custom icons in Leaflet, check out the *Leaflet Custom Markers* tutorial (<https://leafletjs.com/examples/custom-icons/>) and the *DUSPviz Map Design* tutorial (<http://duspviz.mit.edu/web-map-workshop/map-symbolization/>).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fnref104)
3. Note that this query does not use data from any table, because it calculates distance between two points created as part of the query itself.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fnref105)