**Chapter 9 Databases**

*Last updated: 2021-08-24 14:54:59*

**9.1 Introduction**

In Chapters [7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#geojson-1)–[8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#symbology-and-interactivity), the foreground layers we displayed on Leaflet web maps came from GeoJSON files stored on the server. This is a viable approach when our data are relatively small and not constantly updated. When this is not the case, however, using GeoJSON files can become limiting.

For example, loading layers from GeoJSON files becomes prohibitive when files get too large, because the entire file needs to be transferred through the network, even if we only want to display *some* of the content, for example by subsetting the layer in the JavaScript code after it has been received. It is obviously unreasonable to have the user wait until tens or hundreds of megabytes are being received, in the meanwhile seeing an empty map. Processing very large amounts of data can also make the browser unresponsive. A natural solution is to use a **database**. Unlike a file, a database can be queried to request just the minimal required portion of information each time, thus making sure that we are transferring and processing manageable amounts of data.

Another limitation of using GeoJSON files becomes apparent when the data are constantly updated and/or used for different purposes rather than just being displayed on a particular web map. For example, we may wish to build a web map displaying real-time municipal events, which means the data are constantly **updated** or edited (e.g., by the municipality staff) and/or used in various contexts (e.g., examined in GIS software by other professionals). Again, a natural solution is to use a database, shared between numerous concurrent connections for viewing and editing the data, through many types of different interfaces. Our web map, making use of one such concurrent connection, will therefore be synchronized with the database so that the displayed information is always up-to-date.

This chapter (Chapter [9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#databases)) and the next two (Chapters [10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#non-spatial-queries-1)–[11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#spatial-queries-1)) introduce the idea of loading data from a **spatial database** to display them on an interactive map, while dynamically filtering the data to transfer just the portion that we need. That way, we are freed from the limitation regarding the amount of data “behind” the web map. In other words, the database that stands behind our web map can be very large in size, yet the web map will stay responsive, thanks to the fact that we load subsets of the data each time, based on what the user chooses to see. In this chapter (Chapter [9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#databases)), we introduce the concepts and technologies that enable a Leaflet map to load data from a spatial database. In the next two chapters, we go through examples of using non-spatial (Chapter [10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#non-spatial-queries-1)) and spatial (Chapter [11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#spatial-queries-1)) database queries for loading subsets of data from a database.

It should be mentioned that **Web Map Services (WMS)** (Section [6.6.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#other-layer-types)) comprise an alternative solution for displaying large, up-to-date amounts of data on a web map, however this solution is beyond the scope of this book. In short, with a WMS we are using a GIS database to build on-demand raster tiles. The server generates custom tiles based on the parameters it is given, so that the user has control of the displayed content, such as choosing which layers to display. This is unlike pre-compiled tile datasets, such as those introduced in Section [6.5.10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#what-are-tile-layers) and used as base layers in the examples in Chapters [6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#leaflet)–[8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#symbology-and-interactivity), since pre-compiled tiles are fixed and cannot be dynamically modified based on user input.

There are valid use cases for both the database and WMS approaches. Basically, the database approach works better when loading vector layers that the user interacts with, which is made possible by the fact that the server can send raw data (such as GeoJSON), and we can control the way that data are displayed on the client, using JavaScript code. The WMS approach works better when our data are very complex and have elaborate symbology. In such cases, it makes sense to have a dedicated map server with specialized software to build raster images with the displayed content, and send them to the client to be displayed as-is[87](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fn87).

**9.2 What is CARTO?**

**9.2.1 The CARTO platform.**A problem that immediately arises regarding retrieval of spatial data from a database onto a web map is that client-side scripts cannot directly connect to a database. A dynamic server, which we mentioned in Section [5.4.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\web-servers-1.html#dynamic-servers), is the solution to this problem. On the dynamic server, server-side scripts, which indeed *can* connect to the database, are used to query the database and send the data back to the client. In fact, the need to send information from a database to the browser is one of the main motives for setting up a dynamic server.

In this book, we focus on client-side solutions, so we will not be dealing with setting up our own dynamic server coupled with a database. Instead, we will use a cloud-based service by a company called [**CARTO**](https://carto.com/). The CARTO platform provides several cloud computing GIS and web-mapping services. One of the most notable services, and the one we are going to use in this book, is the **SQL API** (Section [9.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#the-carto-sql-api)). CARTO allows you to upload your own data into a managed spatial database, while CARTO’s SQL API allows you to interact with that database. In other words, CARTO takes care of setting up and maintaining a spatial database, as well as setting up server-side components to make that database reachable through HTTP.



In this chapter, we will introduce the CARTO platform and the technologies it is based on: databases (Section [9.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#databases-1)) and spatial databases (Section [9.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-databases)), PostGIS (Section [9.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#what-is-postgis)), SQL (Section [9.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#what-is-sql)), and the SQL API (Section [9.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#the-carto-sql-api)). Towards the end of the chapter, we will see how CARTO can be used for querying and displaying data from a database on a Leaflet map (Section [9.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#carto-and-leaflet)). In the next two chapters, we will dig a little deeper into different types of queries and their utilization in web mapping. In Chapter [10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#non-spatial-queries-1), we will see an example of non-spatial, attribute-based filtering of data, based on user input from a dropdown menu. In Chapter [11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#spatial-queries-1), we will see an example of using spatial queries to retrieve data based on proximity to a clicked location.

**9.2.2 Alternatives to CARTO.**CARTO is a commercial service that comes at a [price](https://carto.com/pricing/), though at the time of writing there is a free plan for trying CARTO for 12 months. It is worth mentioning that the CARTO platform is [**open-source**](https://github.com/CartoDB/cartodb). In principle, it can be installed on any computer to replicate [almost](https://gis.stackexchange.com/questions/269822/cartodb-opensource-vs-paid-one) the entire functionality of CARTO for free. The installation and maintenance are quite complicated though.

A minimal alternative to CARTO, comprising a simple dynamic server, a database, and an SQL API can be set up (relatively) more easily. The online version of the book (Section [0.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\index.html#online-version)) includes an additional supplement with [instructions](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\setting-up-an-sql-api.html) for one way to do that, using **Node.js** and **PostgreSQL/PostGIS**. This is beyond the scope of the main text of the book, but it is important to be aware that no matter how convenient CARTO is, if some day we need to cut costs and manage our own server for the tasks covered in the following chapters—it can be done. Accordingly, all examples where a CARTO database is used are also given in alternative versions, using the custom SQL API setup instead of CARTO. The latter are marked with -s, as in example-09-01.html and example-09-01-s.html (Appendix [B](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\list-of-examples.html#list-of-examples)).

**9.3 Databases**

The term [**database**](https://en.wikipedia.org/wiki/Database) describes an organized collection of data. A database stores data, but also facilitates indexing, searching, and querying the data, as well as modifying and adding new data. The most established and commonly used databases follow the [**relational model**](https://en.wikipedia.org/wiki/Relational_database), where the records are organized in tables, and the tables are usually associated with one another via common columns.

For example, Figure [9.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:relational-database-example) shows a small hypothetical database with two tables named airports and flights. The airports table gives the name and location (lon, lat) of seven different airports. The flights table lists the departure time (year, month, day, dep\_time), the origin (origin), and the destination (dest) of five different flights that took place between the airports listed in the airports table on a particular day. Importantly, the airports and flights tables are related through the *airport code* column faa in the airports table matching the origin and dest columns in the flights table.

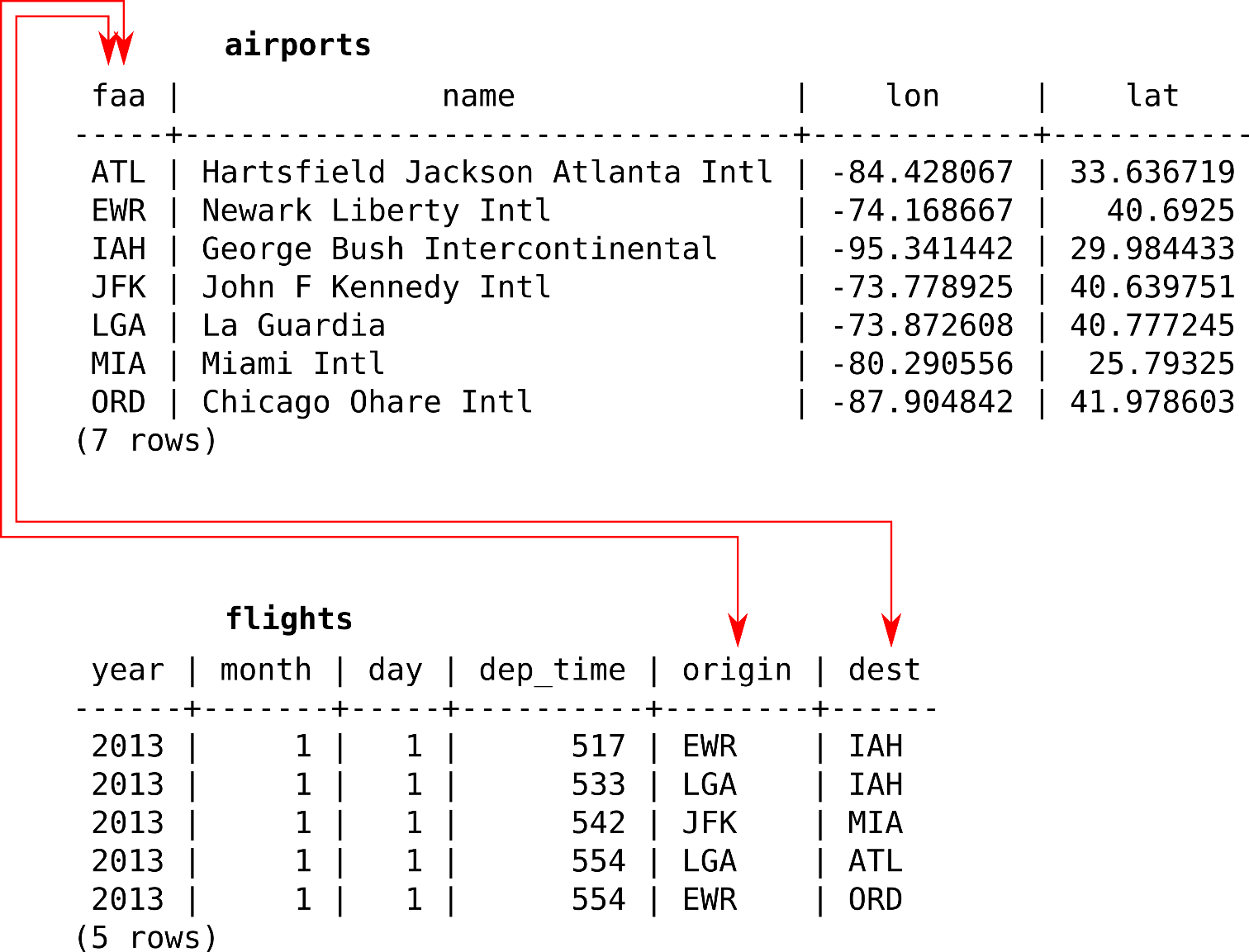


FIGURE 9.1: An example of a relational database with two tables

Relational database queries, including both “ordinary” (Section [9.6.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#non-spatial-queries)) and spatial (Sections [9.6.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#the-geometry-column)–[9.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-queries)) queries, are expressed in a language called SQL (Section [9.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#what-is-sql)).

**9.4 Spatial databases**

A [**spatial database**](https://en.wikipedia.org/wiki/Spatial_database) is a database that is optimized to store and query data that represents objects defined in a geometric space. Regarding the *storage* part, plainly speaking, the tables in a spatial database have a special type of **geometry column**, which holds the geometric component of that specific record, i.e., the geometry type and the coordinates. This may sound familiar—recall that geometry GeoJSON types (Section [7.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#geometries)), which represent just the geometric part of a feature, consist of two properties: "type" and "coordinates". The similarity between the geometry column and the GeoJSON geometry types in not incidental, but due to the fact that both are based on the Simple Features standard, which we mentioned in Section [7.3.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#overview-geojson). The difference is that in a spatial database, the geometries are usually encoded in a format called **Well-Known Binary (WKB)**, a binary version of the Well-Known Text (WKT) format (which we mentioned in Section [7.3.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#overview-geojson)), rather than in the GeoJSON format.

In addition to geometry storage, spatial databases define special functions that allow for queries based on *geometry*. This means we can use the database to make spatial numeric calculations (e.g., geographical distance; 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)), retrieve data based on location (e.g., K-nearest neighbors; Section [11.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#geographical-distance)), or create new geometries (e.g., calculating the centroid of a geometry). These are called **spatial queries** (Section [9.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-queries)), since they involve the spatial component of the database, i.e., the geometry column of at least one table. The concept is very similar to spatial operators and functions used in GIS software, such as the [*Select by Location*](http://desktop.arcgis.com/en/arcmap/10.3/map/working-with-layers/using-select-by-location.htm) tool in ArcGIS.

Commonly used open-source spatial databases include **PostgreSQL/PostGIS** (used by CARTO; see Section [9.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#what-is-postgis) below), [**SQLite/SpatiaLite**](https://en.wikipedia.org/wiki/SpatiaLite), and [**MySQL**](https://dev.mysql.com/doc/refman/8.0/en/spatial-types.html). There are also proprietary databases that support spatial data, such as [**Oracle Spatial**](https://en.wikipedia.org/wiki/Oracle_Spatial_and_Graph) and [**Microsoft SQL server**](https://docs.microsoft.com/en-us/sql/relational-databases/spatial/spatial-data-sql-server).

**9.5 What is PostGIS?**

[**PostGIS**](https://en.wikipedia.org/wiki/PostGIS) is a popular extension for the [**PostgreSQL**](https://en.wikipedia.org/wiki/PostgreSQL) database, making the PostgreSQL/PostGIS combination a spatial database ([Obe and Hsu 2020](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#ref-obe2020postgis)). In other words, a PostgreSQL database with the PostGIS extension enabled allows for storage of spatial data and execution of spatial SQL queries (Section [9.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-queries)). At the moment, the PostgreSQL/PostGIS combination[88](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fn88) makes the most powerful open-source spatial database available.

Since both PostgreSQL and PostGIS are free and open-source, you can install PostGIS on your computer and set up your own database. However, running a database requires some advanced setup and maintenance, which is outside the scope of this book[89](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fn89). As mentioned previously, the CARTO platform provides a cloud-based PostGIS database as a service, which we are going to use in this book.

**9.6 What is SQL?**

**9.6.1 Overview.**[**SQL**](https://en.wikipedia.org/wiki/SQL) is a language for writing statements to query or to modify tables stored in a relational database, whether spatial or non-spatial. Using SQL, you can perform many types of tasks: filtering, joining, inserting new data, updating existing data, etc. SQL statements can be executed in many types of database **interfaces**, from command lines interfaces, through database administrator consoles in GIS software, and to APIs that connect to the database through HTTP—such as the CARTO SQL API, which we are going to use (Section [9.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#the-carto-sql-api)). You may already be familiar with SQL syntax from GIS software, such as [**ArcGIS**](http://pro.arcgis.com/en/pro-app/help/mapping/navigation/sql-reference-for-elements-used-in-query-expressions.htm) and [**QGIS**](https://docs.qgis.org/2.8/en/docs/user_manual/working_with_vector/query_builder.html), where SQL can be used to select features from a spatial layer.

Using CARTO, we will experiment with writing SQL queries to extract data from a cloud-based database (Section [9.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#the-carto-sql-api)) and to display these data on Leaflet map (Section [9.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#carto-and-leaflet)). That way, we will become familiar with the whole idea of querying spatial databases, from the web-mapping perspective. Hopefully, this introduction will be of use if, later on, you decide to go deeper into the subject and set up a spatial database on your own (Section [9.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#alternatives-to-carto)).

SQL, as you can imagine, is a very large topic ([Nield 2016](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#ref-nield2016getting); [DeBarros 2018](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#ref-debarros2018practical)). The syntax of SQL is not the focus of this book, so we will not go deeply into details nor will we cover the whole range of query types that can be used for various tasks. In the following Chapters [10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#non-spatial-queries-1)–[11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#spatial-queries-1), we will only encounter about ~5-6 relatively simple types of SQL queries, most of which are briefly introduced below (Sections [9.6.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#non-spatial-queries)–[9.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-queries)). This set of SQL queries will be enough for our purposes, and you will be able to modify the code to apply the same type of queries to different data, even if you have never used SQL before.

In the following examples (Sections [9.6.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#non-spatial-queries)–[9.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-queries)), we will demonstrate several types of SQL queries on a database that contains just one table, a table named plants. The plants table contains rare plant observations in Israel[90](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fn90). Each row in the plants table represents an individual observation of a rare plant species. The table has different columns describing each observation, such as:

* id—Observation ID
* name\_lat—Latin species name
* obsr\_date—Observation date
* geometry—The location; this column is the **geometry column** (Section [9.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-databases))

The query examples are just for illustration and are not meant to be replicated in a console or command line, since we are not setting up our own database. However, shortly you will be able to execute them through the CARTO platform (Section [9.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#the-carto-sql-api)).

**9.6.2 Non-spatial queries.**The most basic SQL statement is the SELECT statement. A SELECT statement pulls data from a table, possibly filtered on various criteria and supplemented with new columns resulting from table joins or transformations. For example, we can use the following SELECT query to get a subset of the plants table, with just three of its columns: id, name\_lat and obsr\_date. The table is also filtered, to include only those rows where the Latin species name is equal to the specific value 'Anticharis glandulosa':

**SELECT** **id**, name\_lat, obsr\_date

**FROM** plants

**WHERE** name\_lat = 'Anticharis glandulosa';

By convention, SQL keywords are written in uppercase, while specific values—such as column names—are written in lowercase. This is not strictly required, as SQL is not case-sensitive, unlike JavaScript, for instance, which *is* case-sensitive. Spaces and line breaks are ignored in SQL, like in JavaScript. The query ends with the ; symbol.

Note the way that the query is structured. The queried column names are listed after the SELECT keyword, the table name is specified after FROM, and the condition for filtering returned records is constructed after the WHERE keyword. In this example, the condition name\_lat='Anticharis glandulosa' means “return all records (rows) where the value of name\_lat is equal to 'Anticharis glandulosa'.”

If we had access to a PostGIS database with the plants table and could type the above SQL query through its command line interface (called [**psql**](http://postgresguide.com/utilities/psql.html)), the following textual printout with the query result would appear in the command line[91](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fn91):

id | name\_lat | obsr\_date

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

339632 | Anticharis glandulosa | 1988-03-18

359135 | Anticharis glandulosa | 2012-12-15

367327 | Anticharis glandulosa | 2012-12-15

(3 rows)

According to the result, we can tell that there are only three observations of 'Anticharis glandulosa' in the plants table. Note that the last line is not part of the result, but only specifies the number of returned rows.

**9.6.3 The geometry column.**As mentioned in Section [9.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#spatial-databases), the distinctive feature of a spatial database is that its tables may contain a geometry column. The values in the geometry column specify the spatial locations of the respective database records (i.e., the table rows). The geometry column usually contains binary code, which is an encoded version of the Well-Known Text (WKT) format, known as Well-Known Binary (WKB). The binary compression is conventionally used to reduce the required storage space for the database.

For example, the geometry column in our plants table is named geometry. The following query returns the contents of three columns from the plants table, the “ordinary” id and name\_lat columns, as well as the geometry column named geometry. The query is also limited to the first five records, with the LIMIT 5 part:

**SELECT** **id**, name\_lat, geometry

**FROM** plants

**LIMIT** 5;

Here is the printout we would see on the command line in this case:

id | name\_lat | geometry

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

321432 | Iris haynei | 0101000000520C906802D741400249D8B793624040

321433 | Iris haynei | 0101000000D235936FB6D34140C6151747E55E4040

321456 | Iris atrofusca | 01010000001590F63FC0984140EDB60BCD75723F40

321457 | Iris atrofusca | 0101000000672618CE35984140357C0BEBC6833F40

321459 | Iris vartanii | 0101000000E6B0FB8EE19141405D6E30D461793F40

(5 rows)

It is evident the WKB strings in the geometry column make no sense to the human eye. However, WKB can always be converted to its textual counterpart **WKT**, using the ST\_AsText operator, as demonstrated in the following, slightly modified, version of the above SQL query:

**SELECT** **id**, name\_lat, ST\_AsText(geometry) **AS** geom

**FROM** plants

**LIMIT** 5;

In the modified query, we replaced the geometry part with ST\_AsText(geometry), thus *transforming* the column from WKB to WKT. The AS geom part is used to rename the new column to geom (otherwise it would get a default name such as st\_astext). Here is the resulting table, with the geometry column transformed to its WKT representation and renamed to geom:

id | name\_lat | geom

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

321432 | Iris haynei | POINT(35.679761 32.770133)

321433 | Iris haynei | POINT(35.654005 32.741372)

321456 | Iris atrofusca | POINT(35.193367 31.44711)

321457 | Iris atrofusca | POINT(35.189142 31.514754)

321459 | Iris vartanii | POINT(35.139696 31.474149)

(5 rows)

Similarly, we can convert the WKB geometry column to **GeoJSON**, which we are familiar with from Chapter [7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#geojson-1). To do that, we simply use the ST\_AsGeoJSON function instead of the ST\_AsText function, as follows:

**SELECT** **id**, name\_lat, ST\_AsGeoJSON(geometry) **AS** geom

**FROM** plants

**LIMIT** 5;

Here is the result, with the geometry column now given in the GeoJSON format:

id | name\_lat | geom

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

321432 | Iris haynei | {"type":"Point","coordinates":[35.679761,32.770133]}

321433 | Iris haynei | {"type":"Point","coordinates":[35.654005,32.741372]}

321456 | Iris atrofusca | {"type":"Point","coordinates":[35.193367,31.44711]}

321457 | Iris atrofusca | {"type":"Point","coordinates":[35.189142,31.514754]}

321459 | Iris vartanii | {"type":"Point","coordinates":[35.139696,31.474149]}

(5 rows)

Examining either one of the last two query results, we can tell that the the plants table—or at least its first five records—contains geometries of type "Point" (Table [7.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#tab:geojson-single-part1)).

**9.6.4 Spatial queries.**The geometry column can be used to apply **spatial operators** on our table, just like in GIS software. Much like general SQL (shown previously), the syntax of spatial SQL queries is a very large topic ([Obe and Hsu 2020](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#ref-obe2020postgis)), and mostly beyond the scope of this book. In Chapter [11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#spatial-queries-1) we will experiment with just one type of a spatial query, which returns the *nearest* records from a given point.

For example, the following spatial query returns the nearest five observations from the plants table based on distance to the specific point [34.810696, 31.895923] (as in [lon, lat]). Plainly speaking, this SQL query *sorts* the entire plants table by decreasing proximity of all geometries to [34.810696, 31.895923], then the top five records are returned:

**SELECT** **id**, name\_lat, ST\_AsText(geometry) **AS** geom

**FROM** plants

**ORDER** **BY**

geometry::geography <->

ST\_SetSRID(

ST\_MakePoint(34.810696, 31.895923), 4326

)::geography

**LIMIT** 5;

The selection of top five records is done using the LIMIT 5 part. The spatial operators part comes after the ORDER BY keyword, where we calculate all distances from plants points to a specific point [34.810696, 31.895923], and use those distances to sort the table. We will elaborate on this part in Chapter [11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#spatial-queries-1). Here is the result, with the five nearest observations to [34.810696, 31.895923]:

id | name\_lat | geom

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

341210 | Lavandula stoechas | POINT(34.808564 31.897377)

368026 | Bunium ferulaceum | POINT(34.808504 31.897328)

332743 | Bunium ferulaceum | POINT(34.808504 31.897328)

328390 | Silene modesta | POINT(34.822295 31.900125)

360546 | Corrigiola litoralis | POINT(34.825931 31.900792)

(5 rows)

For more information, Chapter 7 in the *Introduction to Data Technologies* book ([Murrell 2009](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#ref-murrell2009introduction)) gives a good introduction to (non-spatial) SQL. The [*W3Schools SQL Tutorial*](https://www.w3schools.com/sql/) can also be useful for quick reference of commonly used SQL commands. An introduction to spatial operators and PostGIS can be found in the official [*Introduction to PostGIS*](https://postgis.net/workshops/postgis-intro/) tutorial and in the *PostGIS in Action* book ([Obe and Hsu 2020](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#ref-obe2020postgis)).

**9.7 The CARTO SQL API**

**9.7.1 API usage.**The [**CARTO SQL API**](https://carto.com/developers/sql-api/) is an API for communication between a program that understands HTTP, such as the browser, and a PostGIS database hosted on the CARTO platform. The CARTO SQL API allows for users to send SQL queries to their PostGIS database on the CARTO platform. The queries are sent via HTTP (Section [5.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\web-servers-1.html#communicating-through-http)), typically by making a GET request (Section [5.3.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\web-servers-1.html#http-methods-get)) using a URL which includes the CARTO user name and the SQL query. The CARTO server processes the request and prepares the returned data, according to the SQL query applied on the particular user’s database. The result is then sent back, in a format of choice, such as **CSV**, **JSON** or **GeoJSON**. Importantly, the fact that the requests are made through HTTP means that we can send requests to the database, and get the responses, from client-side JavaScript code using Ajax (Section [7.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#ajax)).

It is important to note that some types of queries other than SELECT, namely queries that modify our data, such as INSERT, require an **API key** as an additional parameter in our request. The API key serves as a password for making sensitive queries. Without this restriction, anyone who knows our username could make changes to our database, or even delete its entire contents. We will not be using this type of sensitive queries up until Section [13.6.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\collaborative-mapping.html#setting-database-permissions), where a method to hide the API key from the client and still be able to write to the database will be introduced.

The basic URL structure for sending a GET request to the CARTO SQL API looks like this:

https://CARTO\_USERNAME.carto.com/api/v2/sql?format=FORMAT&q=SQL\_STATEMENT

where:

* CARTO\_USERNAME should be replaced with your CARTO **user name**
* FORMAT should be replaced with the required **format**
* SQL\_STATEMENT should be replaced with the SQL **query**

Note that this is a special URL structure, which contains a [**query string**](https://en.wikipedia.org/wiki/Query_string). A query string is used to send parameters to the server as part of the URL. The query string comes at the end of the URL, after the ? symbol, with the parameters separated by & symbols. In this case, the query string contains two parameters, format and q.

For example, here is a specific query:

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

SELECT id, name\_lat, the\_geom FROM plants LIMIT 2

where:

* CARTO\_USERNAME was replaced with michaeldorman
* FORMAT was replaced with GeoJSON
* SQL\_STATEMENT was replaced with SELECT id, name\_lat, the\_geom FROM plants LIMIT 2

Possible values for the format parameter include JSON, GeoJSON, and CSV. The default returned format is JSON, so to get your result returned in JSON you can omit the format parameter to get a slightly simplified URL:

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

SELECT id, name\_lat, the\_geom FROM plants LIMIT 2

To get your results in a format other than JSON, such as GeoJSON or CSV, you need to explicitly specify the format, as in format=GeoJSON or format=CSV. Other possible formats include GPKG, SHP, SVG, KML, and SpatiaLite[92](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fn92).

**9.7.2 Query example.**The plants table used in the above SQL examples (Section [9.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#what-is-sql)) was already uploaded to a CARTO account named michaeldorman. We will query the database associated with this account to experiment with the CARTO SQL API.

Figure [9.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:carto-plants-table) shows how the plants table appears on the CARTO web interface. Note the table name—plants, in the top-left corner—and column names—such as id and name\_lat—which we referred to when constructing the SQL queries (Section [9.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#what-is-sql)). Importantly, note the geometry column—the one with the small **GEO** icon next to it—named the\_geom[93](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fn93). We will shortly go over the procedure of uploading data to your own CARTO account (Section [9.7.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#uploading-your-data)).

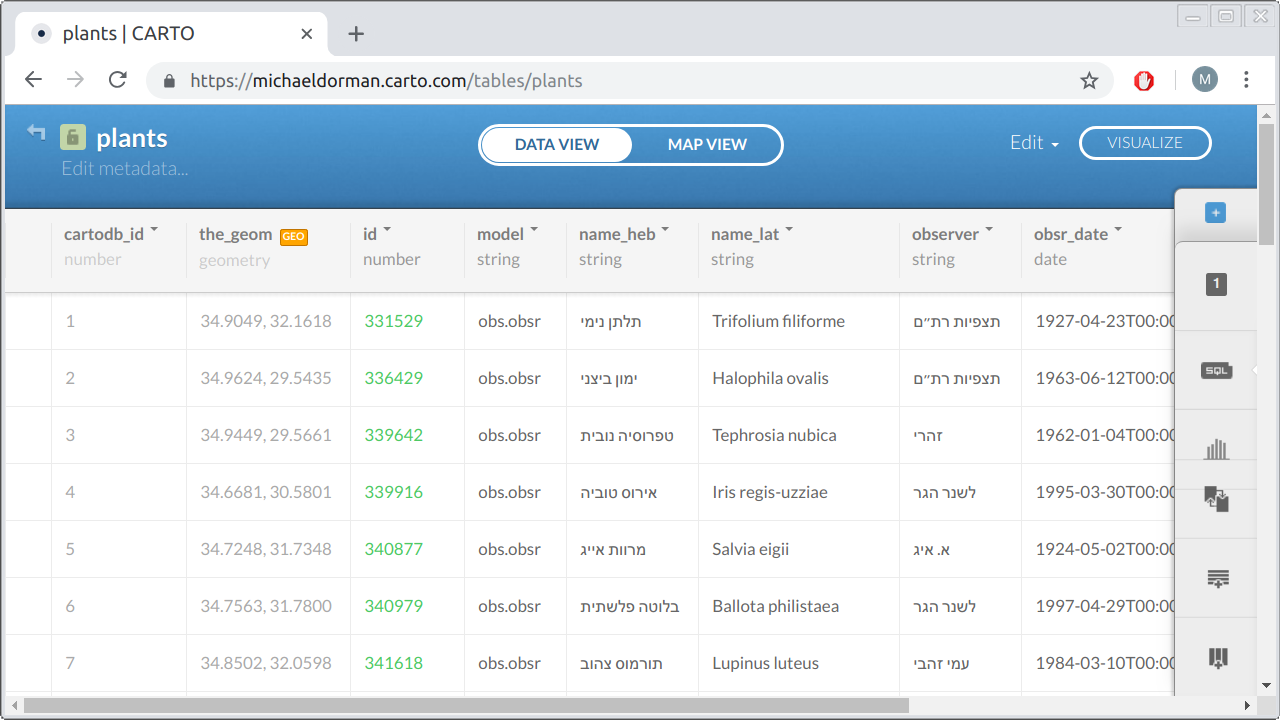


FIGURE 9.2: The plants table, as displayed in the CARTO web interface

Let’s try to send a query to the CARTO SQL API to get some data, in the GeoJSON format, from the plants table. Paste the following query into the browser’s address bar:

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

SELECT id, name\_lat, the\_geom FROM plants LIMIT 2

Since we specified format=GeoJSON, a GeoJSON file will be returned (Section [9.7.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#api-usage)). The query q was SELECT id, name\_lat, the\_geom FROM plants LIMIT 2, which means that we request the id, name\_lat and the\_geom columns from the plants table, limited to the first 2 records (Section [9.6.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#the-geometry-column)). As a result, the CARTO server takes the relevant information from the plants table and returns the following GeoJSON content[94](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fn94):

{

"type": "FeatureCollection",

"features": [

{

"type": "Feature",

"geometry": {

"type": "Point",

"coordinates": [35.032018, 32.800539]

},

"properties": {

"id": 345305,

"name\_lat": "Elymus elongatus"

}

},

{

"type": "Feature",

"geometry": {

"type": "Point",

"coordinates": [35.564703, 33.047197]

},

"properties": {

"id": 346805,

"name\_lat": "Galium chaetopodum"

}

}

]

}

This is a GeoJSON string of type "FeatureCollection" (Section [7.3.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#feature-collections)). It contains two features with "Point" geometries, each having two non-spatial attributes: id and name\_lat.

Remember that the geometry column the\_geom needs to appear in the query whenever we export the result in a spatial format, such as format=GeoJSON. Otherwise, the server cannot generate the geometric part of the layer and we get an error. For example, omitting the the\_geom column from the above query:

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

SELECT id, name\_lat FROM plants LIMIT 2

returns the following error message instead of the requested GeoJSON:

{"error":["column \"the\_geom\" does not exist"]}

By the way, while pasting these URL examples into the browser, you may have noticed how the browser automatically [**encodes**](https://en.wikipedia.org/wiki/Percent-encoding) the URL into a format that can be transmitted over the Internet. This is something that happens automatically, and we do not need to worry about. For example, as part of URL encoding, spaces are converted to %20, so that the URL we typed above:

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

SELECT id, name\_lat, the\_geom FROM plants LIMIT 2

becomes:

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

SELECT%20id,%20name\_lat,%20the\_geom%20FROM%20plants%20LIMIT%202

Since the returned file is in the GeoJSON format, we can immediately import it into various spatial applications. For example, the file can be displayed and inspected in GIS software such as **QGIS** (Figure [9.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:qgis-geojson-query)). If you are not using GIS software, you can still examine the GeoJSON file by importing it into the **geojson.io** web interface (Section [7.4.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#geojson-io)). More importantly for our cause, the GeoJSON content can be instantly loaded in a Leaflet web map, as will be demonstrated next in Section [9.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#carto-and-leaflet).

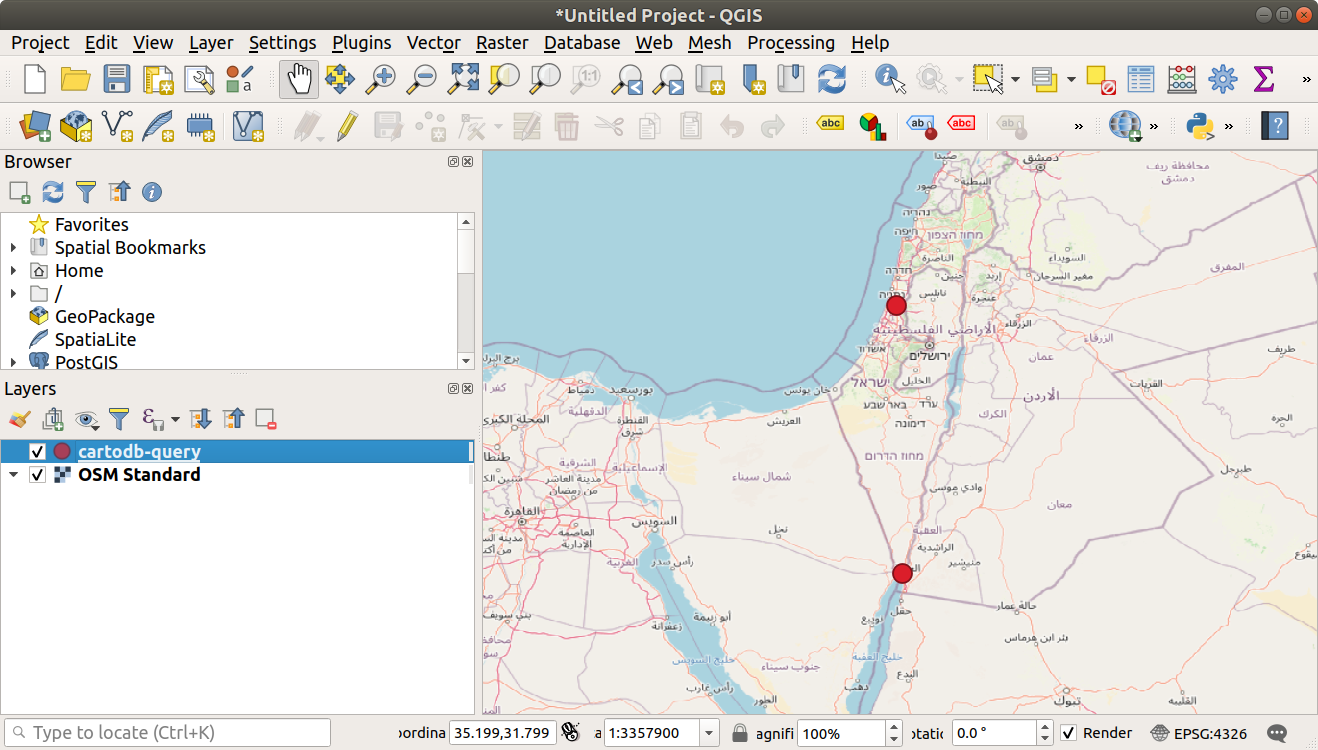


FIGURE 9.3: A GeoJSON file, obtained from the CARTO SQL API, displayed in QGIS

Exporting data in the JSON format is very similar to GeoJSON, but applicable for *non-spatial* queries that cannot be converted to GeoJSON. We will see a practical example of exporting JSON data from CARTO 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).

* Try executing the non-spatial SQL query example from Section [9.6.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#non-spatial-queries) with the CARTO SQL API, using **JSON** as the export format.

Lastly, as an example of the **CSV** export format, try the following request to the SQL API:

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

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

FROM plants WHERE name\_lat = 'Iris mariae' LIMIT 3

Note that we use the format=CSV parameter so that the result comes as a CSV file. CSV is a plain-text tabular format. Given the above query, the resulting CSV file contains the following text:

id, name\_lat, obsr\_date, geom

358449, Iris mariae, 2010-03-13, POINT(34.412502 31.019879)

359806, Iris mariae, 2015-03-08, POINT(34.713009 30.972615)

337260, Iris mariae, 2001-02-23, POINT(34.63678 30.92807)

The CSV file can also be opened in a spreadsheet software such as **Microsoft Excel** or **LibreOffice Calc** (Figure [9.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:carto-query-excel))[95](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fn95).

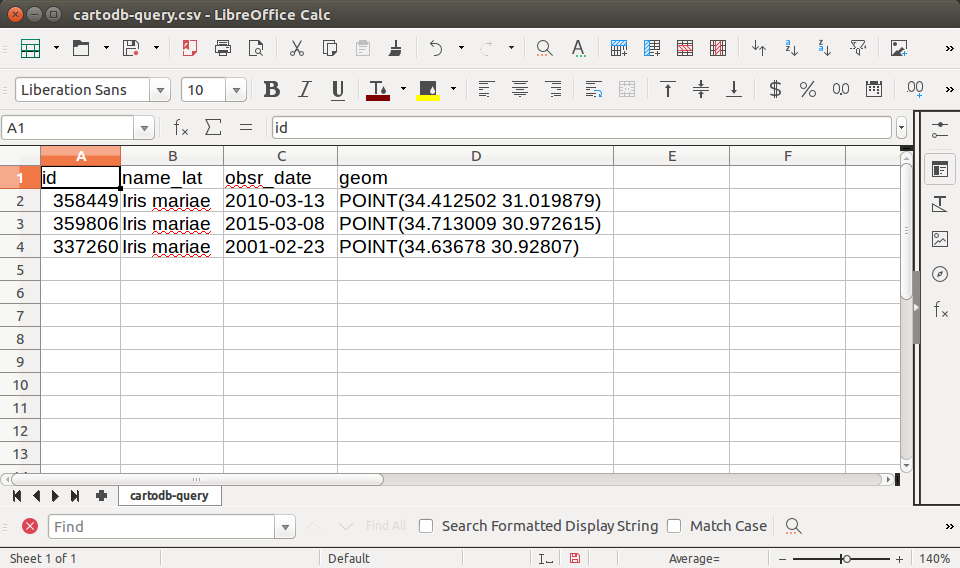


FIGURE 9.4: CSV file exported from the CARTO SQL API displayed in a spreadsheet software (LibreOffice Calc)

**9.7.3 Uploading your data.**Before we begin with connecting a Leaflet map with data from CARTO (Section [9.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#carto-and-leaflet)), you may want to experiment with your own account, possibly with different data instead of the plants table. Assuming you already signed up on <https://carto.com/> and have a CARTO account, the easiest way to upload data is the use the CARTO web interface, as follows:

* Go to <https://carto.com/> and log in to your account.
* Once you are in your user’s home page, click on **Data** in the upper-left corner. This will show the different tables in your database on CARTO. For example, Figure [9.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:carto-datasets-screen) shows the datasets page with three tables, named beer\_sheva, plants, and earthquake\_sql. This screen may be empty if you have just created a new CARTO account and have not uploaded any data yet.
* Click on the **New Dataset** button in the top-right corner of the screen.
* You will see different buttons for various methods of importing data. The simplest option is to upload a GeoJSON file. Choose the **UPLOAD** option in the upper ribbon, then click on the **BROWSE** button and navigate to your GeoJSON file. Finally, click on the **CONNECT DATASET** button (Figure [9.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:carto-upload-screen)). You can upload the [plants.geojson](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\data\plants.geojson) file from the book materials (Section [A](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\associated-files.html#associated-files)) into your own account to experiment with the same dataset as shown in the examples.
* Once the file is uploaded, you should be able to see it as a new table in your list of datasets on CARTO. You can view the table in the CARTO web interface (Figure [9.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:carto-plants-table)), and even edit its contents. For example, you can change the table name, rename any of the columns, edit cell contents, add new rows, etc.

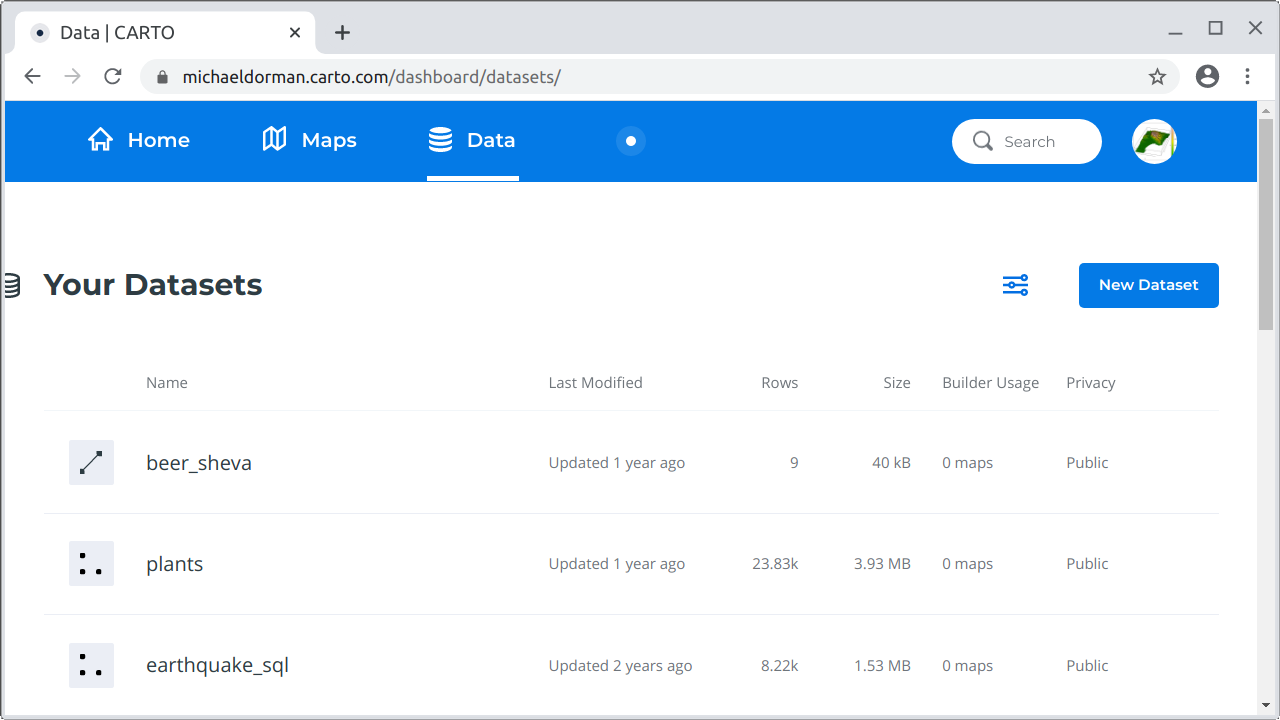


FIGURE 9.5: Datasets screen on CARTO

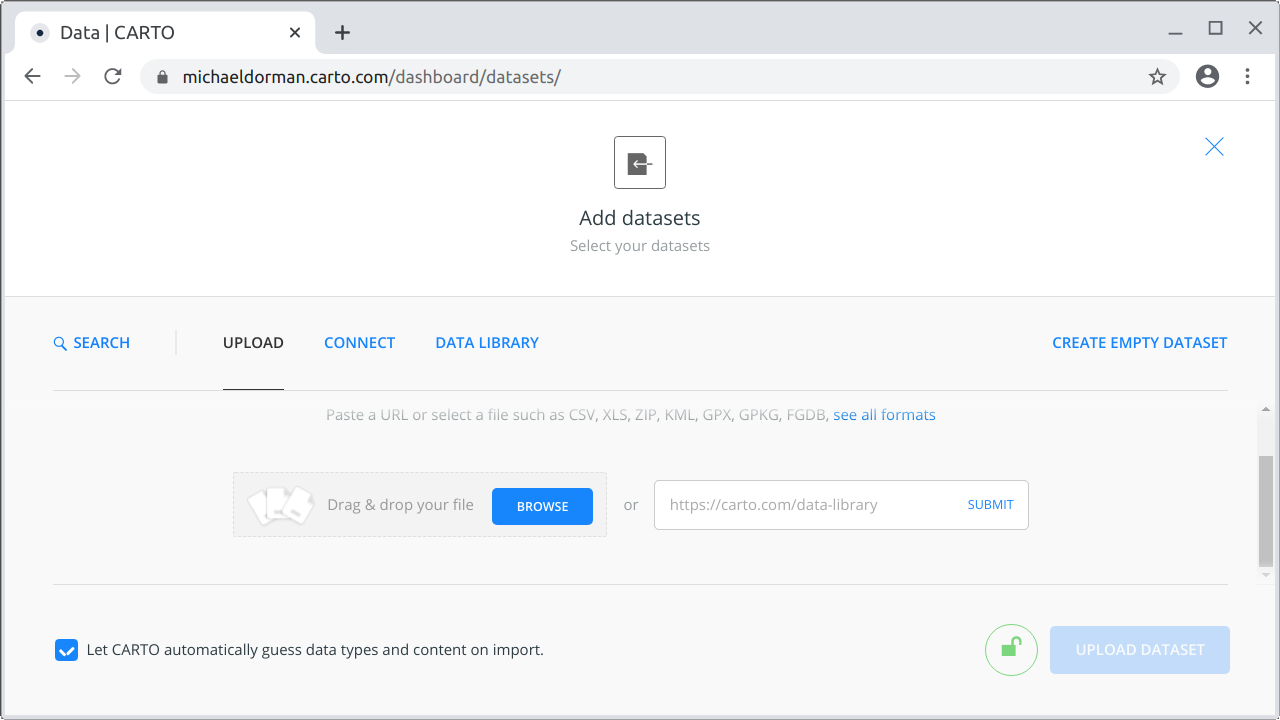


FIGURE 9.6: The file upload screen in the CARTO web interface

* Upload any GeoJSON file other than plants.gejson to CARTO, then try to adapt the above SQL API queries to your own username, table name, and column names.
* Test the new queries by pasting the respective URLs into the browser address bar and examining the returned results.

**9.8 CARTO and Leaflet**

We have just learned how to use the CARTO SQL API to send SQL queries to a CARTO database (Section [9.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#the-carto-sql-api)). Importantly, since we are working with a *spatial* database, one of the formats in which we can choose to get the returned results is GeoJSON (Section [9.7.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#query-example)). In this section, we will load a CARTO query result in a web page and display it on a Leaflet map. The method we are going to use for loading the query result is the fetch function, which we introduced in Section [7.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#loading-geojson-files) and used in many of the examples in Chapters [7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#geojson-1)–[8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#symbology-and-interactivity) when loading GeoJSON layers from files.

Our starting point is the basic map example-06-02.html from Section [6.5.11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#adding-a-tile-layer), with a modified initial map extent, as follows, so that the plants observations will be visible on page load:

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

Now, in order to load data from CARTO on the Leaflet map, we need to go through the following steps:

* Construct the URL to query the CARTO SQL API
* Get the data from CARTO and add it on the map

As the first step, we will construct the query URL. For convenience, the URL will be split in two parts: the fixed **base URL** prefix and the varying **SQL query** suffix. Combining both parts gives the complete URL, which we will use to retrieve data from the CARTO database. The fixed base URL, specific to a particular CARTO user, can be kept in a separate variable hereby named url. That way, we do not need to repeat it in each and every query we make in our script:

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

Note that to make the code even more manageable you can split the base URL into two parts too, keeping the user name in a separate variable. That way, if we need to switch to a different CARTO account it is more clear which part of the code needs to be modified:

**let** carto\_user = "michaeldorman";

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

Either way, our next step is to define the varying SQL query part, used to retrieve data from the database according to a specific query. For example, we can use the following query, which returns the name\_lat and the\_geom columns for the first 25 records from the plants table:

**let** sqlQuery = "SELECT name\_lat, the\_geom FROM plants LIMIT 25";

Remember that you need to include the geometry column in your query whenever the requested format is GeoJSON. Otherwise, the layer cannot be generated and we get an error (Section [9.7.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#query-example)). When the base URL and the SQL query are combined, using url+sqlQuery, we get the complete URL:

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

SELECT name\_lat, the\_geom FROM plants LIMIT 25

The complete URL can be passed to fetch to load the resulting GeoJSON from CARTO on the Leaflet map:

fetch(url + sqlQuery)

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

**return** response.json();

})

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

L.geoJSON(data, {

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

layer.bindPopup(feature.properties.name\_lat);

}

}).addTo(map);

});

This code should be familiar from Chapters [7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#geojson-1)–[8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#symbology-and-interactivity). The primary function is fetch, which we use to make an Ajax GET request from another location on the internet (CARTO). Since the returned data are in the GeoJSON format (as specified with format=GeoJSON), the callback function of .then extracts its contents, as an object, using the .json method. Finally, in the second .then, we can use the L.geoJSON function to convert the GeoJSON object to a Leaflet GeoJSON layer. Using the onEachFeature option we are also binding specific popups (Section [8.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#constructing-popups-from-data)) for each feature to display the Latin name of the observed plant species. Finally, the layer is added on the map with the .addTo method.

The resulting map example-09-01.html is shown in Figure [9.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:example-09-01). Our data from CARTO, i.e., the first 25 plant observations, are loaded on the map!

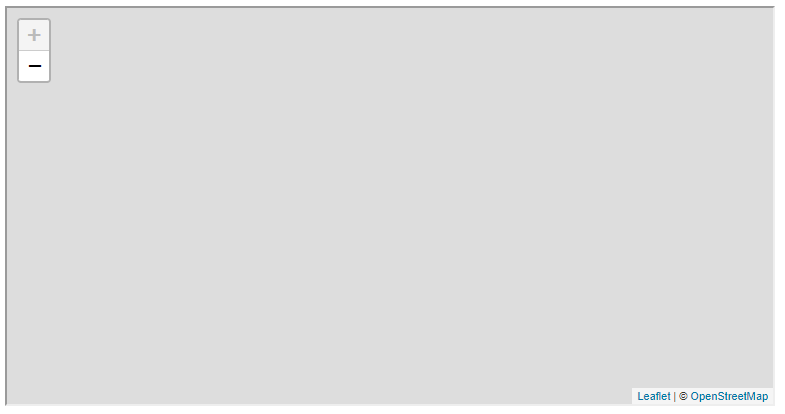


FIGURE 9.7: example-09-01.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-09-01.html)

* Paste the above code section into the console of example-09-01.html.
* Modify the SQL query (sqlQuery) to experiment with adding different observations on the map, according to the SQL examples shown in Section [9.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#what-is-sql).
* For example, you can replace the LIMIT 25 part with a condition of the form WHERE name\_lat = '...' to load all observations of a particular species (Section [9.6.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#non-spatial-queries)).

We have now covered the general principles of using the CARTO SQL API to display layers coming from a database on a Leaflet map. So far, however, what we did was not very different from loading a GeoJSON *file* on a map, like we did in Chapters [7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#geojson-1)–[8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#symbology-and-interactivity). The only difference is that the path to the GeoJSON file was a URL addressing the CARTO SQL API, rather than a local (Section [7.8.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#loading-local-files)) or remote (Section [7.8.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#loading-remote-files)) GeoJSON file. Still, the query was fixed, in the sense that exactly the same layer with 25 observations (Figure [9.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:example-09-01)) will be displayed each time the page is loaded (unless the database itself is modified).

In the beginning of this chapter, we mentioned that one of the main reasons of using a database in web mapping is that we can display subsets of the data, filtered according to user input (Section [9.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#introduction-databases)). That way, we can have large amounts of data “behind” the web map, while maintaining responsiveness thanks to the fact that small portions of the data are transferred to the client each time. To fully exploit the advantages of connecting a database to a web map, in the next two Chapters [10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#non-spatial-queries-1)–[11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#spatial-queries-1) we will see examples where the SQL query is generated *dynamically*, in response to user input:

* In Chapter [10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#non-spatial-queries-1), we will load data according to an *attribute* value the user selects in a dropdown menu.
* In Chapter [11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#spatial-queries-1), we will load data according to *spatial* proximity to a clicked location on the map.

**9.9 Exercise**

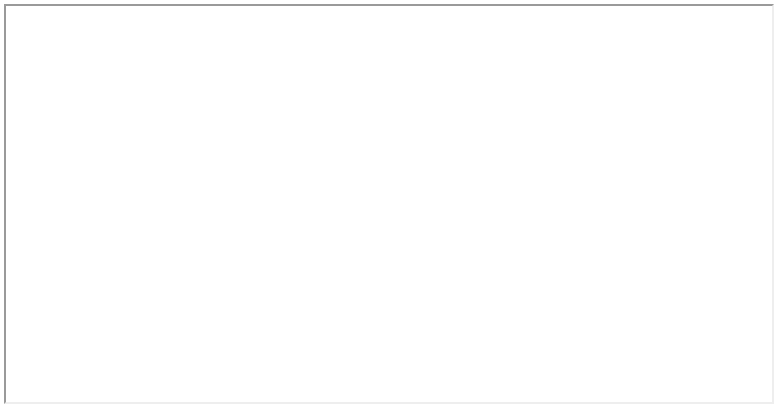
* The following non-spatial SQL query returns the (sorted) species list from the plants table: SELECT DISTINCT name\_lat FROM plants ORDER BY name\_lat. When sending the latter query with format=JSON, the CARTO SQL API returns a JSON string with a property named "rows", which is an array of obejcts of the form {"name\_lat":"Abutilon indicum"}—one for each distinct species (see Section [10.4.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#finding-unique-values)).
* Load the result of the query inside a web page, and use the .rows array to dynamically generate an unordered list (<ul>) of all unique plant species names in the plants table (Figure [9.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:solution-09)).
* Hint: use example-04-06.html from Section [4.11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#modifying-page-based-on-data) (Figure [4.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#fig:example-04-06)), where we generated an unordered list based on an array, as a starting point for this exercise.
* 

FIGURE 9.8: solution-09.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\solution-09.html)

**References**

DeBarros, Anthony. 2018. *Practical SQL: A Beginner’s Guide to Storytelling with Data*. San Francisco, CA, USA: No Starch Press.

Murrell, Paul. 2009. *Introduction to Data Technologies*. Boca Raton, FL, USA: Chapman; Hall/CRC.

Nield, Thomas. 2016. *Getting Started with SQL: A Hands-on Approach for Beginners*. Sebastopol, CA, USA: O’Reilly Media, Inc.

Obe, R, and L Hsu. 2020. *PostGIS in Action*. 3rd ed. Shelter Island, NY, USA.: Manning Publications Co.

1. There is an official tutorial on using *WMS with Leaflet* (<https://leafletjs.com/examples/wms/wms.html>), where you can see a practical example.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fnref87)
2. PostgreSQL with the PostGIS extension will be referred to as *PostGIS* from now on, for simplicity.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fnref88)
3. The online supplement includes instructions for installing PostGIS, as part of the custom SQL API setup (Section [9.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#alternatives-to-carto)).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fnref89)
4. The data source of this table is the *Endangered Plants of Israel* (<https://redlist.parks.org.il/>) website by the Israel Nature and Parks Authority.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fnref90)
5. Instructions to set up a PostGIS database and import the plants table are given in the online supplement, as part of setting up an alternative SQL API (Section [9.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#alternatives-to-carto)).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fnref91)
6. For the full list, refer to the *CARTO SQL API documentation* (<https://carto.com/developers/sql-api/reference/>).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fnref92)
7. The CARTO platform conventionally uses the name the\_geom for the geometry column. In principle, the geometry column can be named any other way. For example, in the SQL examples in Section [9.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#what-is-sql) the geometry column was named geometry, which is another commonly used convention.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fnref93)
8. Note that the GeoJSON response was formatted for convenience. The actual response is returned as an unformatted string, without any spaces and line breaks.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fnref94)
9. For a more detailed description of the CARTO SQL API, see the *documentation* (<https://carto.com/developers/sql-api/>).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fnref95)