```
Shapefile import completed.
```

Switch to pgAdmin, and in the object browser, expand the `analysis` node and continue expanding by selecting **Schemas▶public▶Tables**. Refresh your tables by right-clicking **Tables** and selecting **Refresh** from the pop-up menu. You should see `us_counties_2019_shp` listed. Congrats! You've loaded your shapefile into a table. As part of the import, the shapefile loader also indexed the `geom` column. You can move ahead to the section "Exploring the Census 2019 Counties Shapefile."

Importing Shapefiles using shp2pgsql

The Shapefile Import/Export Manager isn't available on all PostGIS distributions for macOS and Linux. For that reason, I'll show you how to import shapefiles using the PostGIS command line tool `shp2pgsql`, which lets you accomplish the same thing using a single text command.

On macOS and Linux, you execute command line tools in the Terminal application. If you're not familiar with working on the command line, you may want to pause here and read Chapter 18, "Using PostgreSQL from the Command Line," to get set up. Otherwise, on macOS launch Terminal from your Applications folder (under Utilities); on Linux, open your distribution's terminal.

At the command line, you use the following syntax to import a shapefile into a new table; the italicized code here are placeholders:

```
shp2pgsql -I -s SRID -W encoding shapefile_name table_name |
psql -d database -U user
```

A lot's happening here. Let's look at each argument following the command:

`-I` adds an index on the new table's geometry column using GiST.

`-s` lets you specify an SRID for the geometric data.

`-W` lets you specify file encoding, if necessary.

`shapefile_name` is the name (including full path) of the file ending with the *.shp* extension.

`table_name` indicates the new table you want the shapefile imported to.

Following these arguments, you place a pipe symbol (`|`) to direct the output of `shp2pgsql` to `psql`, the PostgreSQL command line utility. That's followed by arguments for naming the database and user. For example, to load the *tl_2019_us_county.shp* shapefile from the book's resources into a `us_counties_2019_shp` table in the `analysis` database, in your terminal you would move to the directory containing the shapefile and run the following command (all on one line):

```
shp2pgsql -I -s 4269 -W LATIN1 tl_2019_us_county.shp
us_counties_2019_shp | psql -d analysis -U postgres
```

The server should respond with a number of SQL `INSERT` statements before creating the index and returning you to the command line. It might take some time to construct the entire set of arguments the first time around. But after you've done one, subsequent imports should take less time because you can simply substitute file and table names into the syntax you already wrote.

Load your shapefile, and then you'll be ready to explore the data with queries.

Exploring the Census 2019 Counties Shapefile

Your new `us_counties_2019_shp` table contains columns including each county's name as well as the *Federal Information Processing Standards (FIPS)* codes uniquely assigned to each state and county. The `geom` column contains the spatial data for each county's boundary. To start, let's check what kind of spatial object `geom` contains using the `ST_AsText()` function. Use the code in Listing 15-14 to show the WKT representation of the first `geom` value in the table.

```
SELECT ST_AsText(geom)
FROM us_counties_2019_shp
ORDER BY gid
LIMIT 1;
```

Listing 15-14: Checking the `geom` column's WKT representation

The result is a MultiPolygon with hundreds of coordinate pairs. Here's a portion of the output:

```
MULTIPOLYGON(((-97.019516 42.004097,-97.019519
42.004933,-97.019527 42.007501,-97.019529
42.009755,-97.019529 42.009776,-97.019529
42.009939,-97.019529 42.010163,-97.019538 42.013931,-97.01955
42.014546,-97.01955 42.014565,-97.019551 42.014608,-97.019551
42.014632,-97.01958 42.016158,-97.019622 42.018384,-97.019629
42.018545,-97.01963 42.019475,-97.01963 42.019553,-97.019644
42.020927, --snip-- )))
```

Each coordinate pair marks a Point on the boundary of the county, and remember that a `MULTIPOLYGON` object can contain a set of polygons. In the case of US counties, that will enable storage of counties whose boundaries contain more than one distinct, separated area. Now, you're ready to analyze the data.

Finding the Largest Counties in Square Miles

Which county can claim the title of largest in area? To find the answer, Listing 15-15 uses the `ST_Area()` function, which returns the area of a Polygon or MultiPolygon object. If you're working with a `geography` data type, `ST_Area()` returns the result in square meters. With a `geometry` data type—as used with this shapefile—the function returns the area in SRID units. Typically, those units are not useful for practical analysis, so we'll cast the `geometry` type to `geography` to obtain square meters. It's an intensive calculation, so expect extra time for this query to complete.

```
SELECT name,
       statefp AS st,
       round(
             ( ST_Area(1geom::geography) / 22589988.110336
)::numeric, 2
            )  AS 3square_miles
FROM us_counties_2019_shp
ORDER BY square_miles 4DESC
LIMIT 5;
```

Listing 15-15: Finding the largest counties by area using `ST_Area()`

The `geom` column is data type `geometry`, so to find the area in square meters, we cast the `geom` column to a `geography` data type using the double-colon syntax 1. Then, to get square miles, we divide the area by 2589988.110336, which is the number of square meters in a square mile 2. To make the result easier to read, I've wrapped it in a `round()` function and named the resulting column `square_miles` 3. Finally, we list the results in descending order from the largest area to the smallest 4 and use `LIMIT 5` to show the first five results, which should look like this:

```
name                 st    square_miles
----------------     --    ------------
Yukon-Koyukuk        02       147871.00
North Slope          02        94827.92
Bethel               02        45559.08
Northwest Arctic     02        40619.78
Valdez-Cordova       02        40305.54
```

Congratulations to Alaska, where the boroughs (the name for county equivalents up there) are big. The five largest are all in Alaska, denoted by the state FIPS code `02`. Yukon-Koyukuk, located in the heart of Alaska, is more than 147,800 square miles. (Keep that information in mind for the "Try It Yourself" exercise at the end of the chapter.)

Note that the shapefile doesn't include a state name, just its FIPS code. Because the spatial data resides in a table, in the next section we'll join to another census table to obtain the state name.

Finding a County by Longitude and Latitude

If you've ever wondered how spammy online ads seem to know where you live ("This one trick helped a Boston man fix his old shoes!"), it's thanks to *geolocation services* that use various means, such as your phone's GPS, to find your longitude and latitude. Given your coordinates, a spatial query can then determine which geography (a city or town, for example) that point falls into.

You can replicate this technique using your census shapefile and the `ST_Within()` function, which returns `true` if one geometry is inside another on the coordinate grid. *Listing 15-16* shows an example using the longitude and latitude of downtown Hollywood, California.

```
SELECT sh.name,
       c.state_name
FROM us_counties_2019_shp sh JOIN us_counties_pop_est_2019 c
    ON sh.statefp = c.state_fips AND sh.countyfp =
c.county_fips
WHERE 1ST_Within(
         'SRID=4269;POINT(-118.3419063
34.0977076)'::geometry, geom
);
```

Listing 15-16: Using `ST_Within()` to find the county belonging to a pair of coordinates

The `ST_Within()` function 1 inside the `WHERE` clause requires two `geometry` inputs and evaluates whether the first is inside the second. For the function to work properly, both `geometry` inputs must have the same SRID. In this example, the first input is an extended WKT representation of a Point that includes the SRID `4269` (same as the census data), which is cast as a `geometry` type. The `ST_Within()` function doesn't accept a separate SRID input, so to set it for the supplied WKT, you must prefix it to the string like this: `'SRID=4269;POINT(-118.3419063 34.0977076)'`. The second input is the `geom` column from the table.

Run the query; you should see the following result:

```
name           state_name
-----------    -----------
Los Angeles    California
```

It shows that the Point you supplied is within Los Angeles county in California. We also see how this technique can gain value (or raise privacy concerns) by relating a Point to data about its surrounding area—as we did here by joining to county population estimates. Suddenly, we can tell a lot about someone based on data describing where they spend time.

Try supplying other longitude and latitude pairs to see which US county they fall in. If you provide coordinates outside the United States, the query should return no results because the shapefile contains only US areas.

Examining Demographics Within a Distance

A fundamental metric for planners trying to locate a new school, business, or other community amenity is the number of people who live within a certain distance of it. Will there be enough people nearby to make construction worthwhile? To find the answer, we can use spatial and demographics data to estimate the population contained in the geographies within a certain distance of the planned location.

Say we're considering building a restaurant in downtown Lincoln, Nebraska, and we want to understand how many people live within 50 miles of the potential location. The code in *Listing 15-17* uses the `ST_DWithin()` function to find counties that have any portion of their boundary within 50 miles of downtown Lincoln and sum their estimated 2019 population.

```
SELECT sum(c.pop_est_2019) AS pop_est_2019
FROM us_counties_2019_shp sh JOIN us_counties_pop_est_2019 c
    ON sh.statefp = c.state_fips AND sh.countyfp =
c.county_fips
WHERE ST_DWithin(sh.geom::geography,
          ST_GeogFromText('SRID=4269;POINT(-96.699656
40.811567)'),
          80467);
```

Listing 15-17: Using `ST_DWithin()` to count people near Lincoln, Nebraska

In *Listing 15-10*, we used `ST_DWithin()` to find farmers' markets close to Des Moines, Iowa. Here, we apply the same technique. We pass three arguments to `ST_DWithin()`: the census shapefile's `geom` column cast to the `geography` type; a point representing downtown Lincoln; and the distance of 50 miles using its equivalent in meters, 80,467.

The query should return a sum of 1,470,295, using the data from the joined census estimates table's `pop_est_2019` column.

Say we want to list the county names and visualize their borders in pgAdmin; we can modify our query, as in *Listing 15-18*.

```
SELECT sh.name,
       c.state_name,
       c.pop_est_2019,
       1 ST_Transform(sh.geom, 4326) AS geom
FROM us_counties_2019_shp sh JOIN us_counties_pop_est_2019 c
    ON sh.statefp = c.state_fips AND sh.countyfp =
c.county_fips
WHERE ST_DWithin(geom::geography,
          ST_GeogFromText('SRID=4269;POINT(-96.699656
40.811567)'),
          80467);
```

Listing 15-18: Displaying counties near Lincoln, Nebraska

This query should return 25 rows with the county name and its population. If you click the eye icon in the header of the `geom` column, you should see the counties displayed on a map in pgAdmin's Geometry Viewer, as in Figure 15-6.

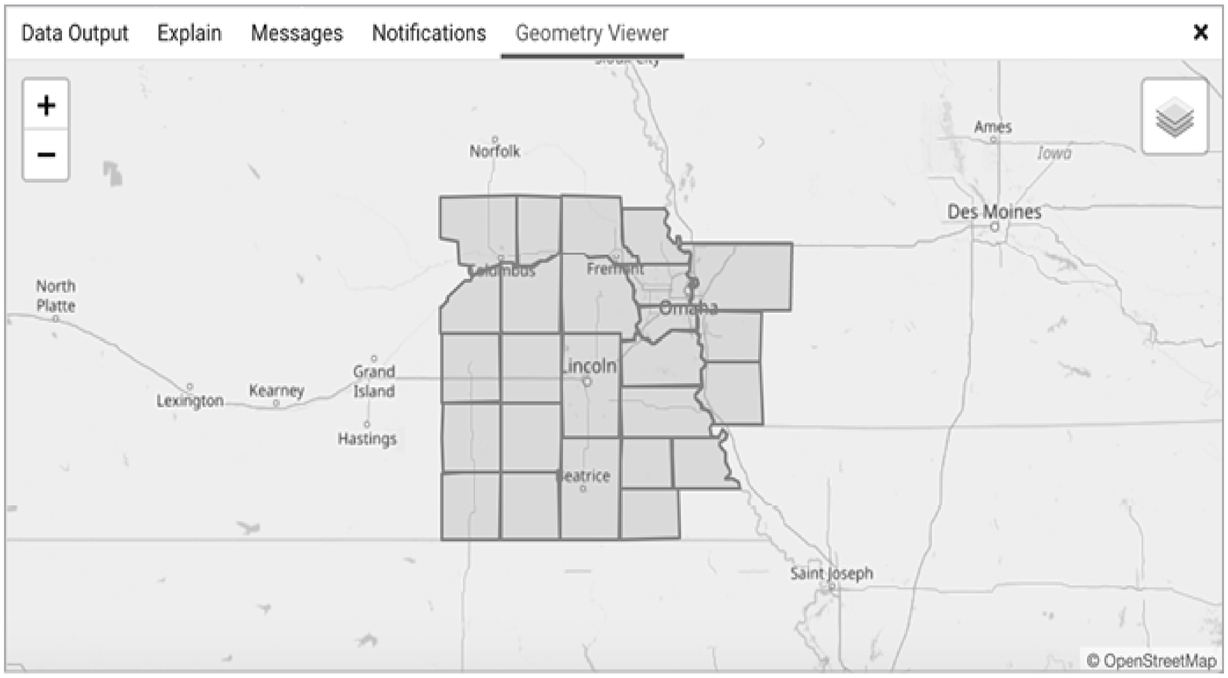


Figure 15-6: Counties that have a portion of their boundaries within 50 miles of Lincoln

These queries show counties that have any portion of their boundaries within 50 miles of Lincoln. Because counties tend to be large in area, they're a bit crude for determining an exact number of people within the distance of the point. For a more precise count, we could use smaller census geographies such as tracts or block groups, both of which are subcomponents of counties.

Finally, note that the pgAdmin Geometry Viewer's base map is the free OpenStreetMap, which uses the WGS 84 coordinate system. Our census shapefiles use a different coordinate system: North American Datum 83. For our data to display properly against the base map, we must use the `ST_Transform()` function 1 to convert the census geometry to the SRID of 4326. If we omit that function, the geographies will display on a blank canvas in the viewer because the coordinate systems don't match.

Performing Spatial Joins

Joining tables with spatial data opens up interesting opportunities for analysis. For example, you could join a table of coffee shops (which includes their longitude and latitude) to the counties table to find out how many shops exist in each county based on their location. In this section, we'll explore spatial joins with a detailed look at roads and waterways using census data.

Exploring Roads and Waterways Data

Much of the year, the Santa Fe River, which cuts through the New Mexico state capital, is a dry riverbed better described as an *intermittent stream*. According to the Santa Fe city website, the river is susceptible to flash flooding and was named the nation's most endangered river in 2007. If you were an urban planner, it would help to know where the river intersects roadways so you could plan for emergency response when it floods.

You can find these locations using another set of US Census TIGER/Line shapefiles that has details on roads and waterways in Santa Fe County. These shapefiles are also included with the book's resources. Download and unzip *tl_2019_35049_linearwater.zip* and *tl_2019_35049_roads.zip*, and then import both using the same steps from earlier in the chapter. Name the water table `santafe_linearwater_2019` and the roads table `santafe_roads_2019`.

Next, refresh your database and run a quick `SELECT * FROM` query on both tables to view the data. You should have 11,655 rows in the roads table and 1,148 in the linear water table.

As with the counties shapefile, both tables have an indexed `geom` column of type `geometry`. It's helpful to check the type of spatial object in the column so you know the type of spatial feature you're querying. You can do that using the `ST_AsText()` function or `ST_GeometryType()`, as shown in Listing 15-19.

```
SELECT ST_GeometryType(geom)
FROM santafe_linearwater_2019
LIMIT 1;

SELECT ST_GeometryType(geom)
FROM santafe_roads_2019
LIMIT 1;
```

Listing 15-19: Using `ST_GeometryType()` to determine geometry

Both queries should return one row with the same value: `ST_MultiLineString`. That tell us that waterways and roads are stored as MultiLineString objects, a set of LineStrings that can be noncontinuous.

Joining the Census Roads and Water Tables

To find all the roads in Santa Fe that intersect the Santa Fe River, we'll join the roads and waterway tables with a query that tells us where the objects touch. We'll do this using the `ST_Intersects()` function, which returns a Boolean `true` if two spatial objects contact each other. Inputs can be either `geometry` or `geography` types. Listing 15-20 joins the tables.

```
SELECT water.fullname AS waterway, 1
       roads.rttyp,
       roads.fullname AS road
FROM santafe_linearwater_2019 water JOIN santafe_roads_2019
roads 2
     3 ON ST_Intersects(water.geom, roads.geom)
WHERE water.fullname = 4'Santa Fe Riv'
      AND roads.fullname IS NOT NULL
ORDER BY roads.fullname;
```

Listing 15-20: Spatial join with `ST_Intersects()` to find roads crossing the Santa Fe River

The `SELECT` column list 1 includes the `fullname` column from the `santafe_linearwater_2019` table, which gets `water` as its alias in the `FROM` 2 clause. The column list includes the `rttyp` code, which represents the route type, and `fullname` columns from the `santafe_roads_2019` table, aliased as `roads`.

In the `ON` portion 3 of the `JOIN` construct, we use the `ST_Intersects()` function with the `geom` columns from both tables as inputs. Here, the expression evaluates as `true` if the geometries intersect. We use `fullname` to filter the results to show only those that have the full string `'Santa Fe Riv'` 4, which is how the Santa Fe River is listed in the water table. We also eliminate instances where road names are `NULL`. The query should return 37 rows; here are the first five:

```
waterway        rttyp    road
------------    -----    ----------------
Santa Fe Riv      M      Baca Ranch Ln
Santa Fe Riv      M      Baca Ranch Ln
Santa Fe Riv      M      Caja del Oro Grant Rd
Santa Fe Riv      M      Caja del Oro Grant Rd
Santa Fe Riv      M      Cam Carlos Rael
--snip--
```

Each road in the results intersects with a portion of the Santa Fe River. The route type code for each of the first results is `M`, which indicates that the road name shown is its *common* name as opposed to a county or state recognized name, for example. Other road names in the complete results carry route types of `C`, `S`, or `U` (for unknown). The full route type code list is available at *https://www.census.gov/library/reference/code-lists/route-type-codes.html*.

Finding the Location Where Objects Intersect

We successfully identified roads that intersect the Santa Fe River. That's good, but it would really help to know the precise location of each intersection. We can modify the query to include the `ST_Intersection()` function, which returns the location of the place where objects touch. I've added it as a column in Listing 15-21.

```
SELECT water.fullname AS waterway,
       roads.rttyp,
       roads.fullname AS road,
     1 ST_AsText(ST_Intersection(2water.geom, roads.geom))
FROM santafe_linearwater_2019 water JOIN santafe_roads_2019
roads
    ON ST_Intersects(water.geom, roads.geom)
WHERE water.fullname = 'Santa Fe Riv'
      AND roads.fullname IS NOT NULL
ORDER BY roads.fullname;
```

Listing 15-21: Using `ST_Intersection()` to show where roads cross the river

The function returns a geometry object, so to view its WKT representation, we must wrap it in `ST_AsText()` 1. The `ST_Intersection()` function takes two inputs: the `geom` columns 2 from both the `water` and `roads` tables. Run the query, and the results should now include the exact coordinate location, or locations, where the river crosses the roads (I've rounded the Point coordinates for brevity).

```
waterway       rttyp   road                   st_astext
------------   -----   ----------------       ----------------
------------
Santa Fe Riv     M     Baca Ranch Ln
POINT(-106.049802 35.642638)
Santa Fe Riv     M     Baca Ranch Ln
POINT(-106.049743 35.643126)
Santa Fe Riv     M     Caja del Oro Grant Rd
POINT(-106.024674 35.657624)
Santa Fe Riv     M     Caja del Oro Grant Rd
POINT(-106.024692 35.657644)
Santa Fe Riv     M     Cam Carlos Rael
POINT(-105.986934 35.672342)
--snip--
```

Much better than poring over a map with a pencil, and this might prompt more ideas for analyzing spatial data. For example, if you have a shapefile of building footprints, you could find buildings near the river and in danger of flooding during heavy rains. Governments and private organizations regularly use these techniques as part of their planning process.

Wrapping Up

Mapping is a powerful analysis tool, and the techniques you learned in this chapter give you a strong start toward exploring more with PostGIS. You may indeed want to visualize this data, and that's entirely possible with a GIS application such as Esri's ArcGIS (*https://www.esri.com/*) or the free open source QGIS (*https://qgis.org/*). Both can use a PostGIS-enabled PostgreSQL database as a data source, allowing you to visualize shapefile data in your tables or the results of queries.

You've now added working with geographic data to your analysis skills. Next, we'll explore another widely used data type called JavaScript Object Notation (JSON) and how PostgreSQL enables storing and querying it.

TRY IT YOURSELF

Use the spatial data you've imported in this chapter to try additional analysis:

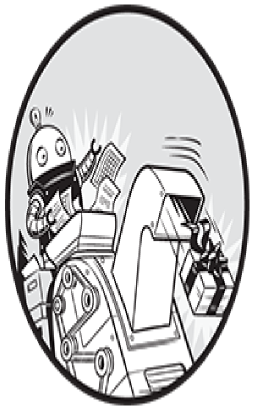
Earlier, you found which US county has the largest area. Now, aggregate the county data to find the area of each state in square miles. (Use the `statefp` column in the `us_counties_2019_shp` table.) How many states are bigger than the Yukon-Koyukuk area?

Using `ST_Distance()`, determine how many miles separate these two farmers' markets: The Oakleaf Greenmarket (9700 Argyle Forest Blvd, Jacksonville, Florida) and Columbia Farmers Market (1701 West Ash Street, Columbia, Missouri). You'll need to first find the coordinates for both in the `farmers_markets` table. Tip: you can also write this query using the Common Table Expression syntax you learned in Chapter 13.

More than 500 rows in the `farmers_markets` table are missing a value in the `county` column, which is an example of dirty government data. Using the `us_counties_2019_shp` table and the `ST_Intersects()` function, perform a spatial join to find the missing county names based on the longitude and latitude of each market. Because `geog_point` in `farmers_markets` is of the `geography` type and its SRID is `4326`, you'll need to cast `geom` in the census table to the `geography` type and change its SRID using `ST_SetSRID()`.

The `nyc_yellow_taxi_trips` table you created in Chapter 12 contains the longitude and latitude where each trip began and ended. Use PostGIS functions to turn the drop-off coordinates into a `geometry` type and count the state/county pairs where each drop-off occurred. As with the previous exercise, you'll need to join to the `us_counties_2019_shp` table and use its `geom` column for the spatial join.

16
WORKING WITH JSON DATA



JavaScript Object Notation (JSON) is a widely used text format for storing data in a platform-independent way so it can be shared between computer systems. In this chapter, you'll learn the structure of JSON as well as how to store and query JSON data types in PostgreSQL. After we explore PostgreSQL's JSON query operators, we'll analyze a month's worth of data about earthquakes.

The American National Standards Institute (ANSI) SQL standard added syntax definitions for JSON and specified functions for creating and accessing JSON objects in 2016. Major database systems have added JSON support in recent years as well, although implementations vary. PostgreSQL, for example, supports some of the ANSI standard while implementing a number of nonstandard operators. I'll note which aspects of PostgreSQL's JSON support are part of standard SQL as we work through exercises.

Understanding JSON Structure

JSON data primarily comprises two structures: an *object*, which is an unordered set of name/value pairs, and an *array*, which is an ordered collection of values. If you've used programming languages such as JavaScript, Python, or C#, these aspects of JSON should look familiar.

Inside an object, we use name/value pairs as a structure for storing and referencing individual data items. The object in its entirety is enclosed within curly brackets, and each name, more often referred to as a *key*, is enclosed in double quotes, followed by a colon and its corresponding value. The object can encapsulate multiple key/value pairs, separated by commas. Here's an example using movie information:

```
{"title": "The Incredibles", "year": 2004}
```

The keys are `title` and `year`, and their values are `"The Incredibles"` and `2004`. If the value is a string, it goes in double quotes. If it's a number, a Boolean value, or a `null`, we omit the quotes. If you're familiar with the Python language, you'll recognize this structure as a *dictionary*.

An array is an ordered list of values enclosed in square brackets. We separate each value in the array with a comma. For example, we might list movie genres like so:

```
["animation", "action"]
```

Arrays are common in programming languages, and we've used them already in SQL queries. In Python, this structure is called a *list*.

We can create many permutations of these structures, including nesting objects and arrays inside each other. For example, we can create an array of objects or use an array as the value of a key. We can add or omit key/value pairs or create additional arrays of objects without violating a preset schema. This flexibility—in contrast to the strict definition of a SQL table—is both part of the appeal of using JSON as a data store as well as one of the biggest difficulties in working with JSON data.

As an example, *Listing 16-1* shows information about two of my favorite films stored as JSON. The outermost structure is an array with two elements—one object for each film. We know the outermost structure is an array because the entire JSON begins and ends with square brackets.

```
[{1
    "title": "The Incredibles",
    "year": 2004,
  2"rating": {
        "MPAA": "PG"
    },
  3"characters": [{
        "name": "Mr. Incredible",
        "actor": "Craig T. Nelson"
    }, {
        "name": "Elastigirl",
        "actor": "Holly Hunter"
    }, {
        "name": "Frozone",
        "actor": "Samuel L. Jackson"
    }],
  4"genre": ["animation", "action", "sci-fi"]
}, {
    "title": "Cinema Paradiso",
    "year": 1988,
    "characters": [{
        "name": "Salvatore",
        "actor": "Salvatore Cascio"
    }, {
        "name": "Alfredo",
        "actor": "Philippe Noiret"
    }],
    "genre": ["romance", "drama"]
}]
```

Listing 16-1: JSON with information about two films

Inside the outermost array, each film object is surrounded by curly brackets. The open brace at 1 starts the object for the first film *The Incredibles*. For both films, we store the `title` and `year` as key/value pairs, and they have string and integer values, respectively. The third key, `rating` 2, has a JSON object for its value. That object contains a single key/value pair showing the film's rating from the Motion Picture Association of America.

Here we can see the flexibility JSON affords us as a storage medium. First, if we later wanted to add another country's rating for the film, we could easily add a second key/value pair to the `rating` value object.

Second, we're not required to include `rating`—or any key/value pair—in every film object. In fact, I omitted a `rating` for *Cinema Paradiso*. If a particular piece of data isn't available, in this case a rating, some systems that generate JSON might simply exclude that pair. Other systems might include `rating` but with a `null` value. Both are valid, and that flexibility is one of JSON's advantages: its data definition, or *schema*, can flex as needed.

The final two key/value pairs show other ways to structure JSON. For `characters` 3, the value is an array of objects, with each object surrounded by curly brackets and separated by a comma. The value for `genre` 4 is an array of strings.

Considering When to Use JSON with SQL

There are advantages to using *NoSQL* or *document* databases that store data in JSON or other text-based data formats, as opposed to the relational tables SQL uses. Document databases are flexible in terms of data definitions. You can redefine a data structure on the fly if needed. Document databases are often also used for high-volume applications because they can be scaled by adding servers. On the flip side, you may give up SQL advantages such as easily added constraints that enforce data integrity and support for transactions.

The arrival of JSON support in SQL has made it possible to enjoy the best of both worlds by adding JSON data as columns in relational tables. The decision to use a SQL or NoSQL database should be multifaceted. PostgreSQL performs favorably relative to NoSQL in terms of speed, but we must also consider the kinds and volume of data being stored, the applications being served, and more.

That said, some cases where you might want to take advantage of JSON in SQL include the following:

When users or applications need to arbitrarily create key/value pairs. For example, if tagging a collection of medical research papers, one user might want to add a key to track chemical names, and another user might want a key to track food names.

When storing related data in a JSON column instead of a separate table. An employees table could have the usual columns for name and contact information plus a JSON column with a flexible collection of key/value pairs that might hold additional attributes that don't apply to every employee, such as company awards or performance metrics.

When saving time by analyzing JSON data fetched from other systems without first parsing it into a set of tables.

Keep in mind that using JSON in PostgreSQL or other SQL databases can also present challenges. Constraints that are trivial to set up on regular SQL tables can be more difficult to set and enforce on JSON data. JSON data can consume more space as key names get repeated in text along with the quotes, commas, and braces that define its structure. Finally, the flexibility of JSON can create issues for the code that interacts with it—whether SQL or another language—if keys unexpectedly disappear or the data type of a value changes.

Keeping all this in mind, let's review PostgreSQL's two JSON data types and load some JSON into a table.

Using json and jsonb Data Types

PostgreSQL provides two data types for storing JSON. Both allow insertion of valid JSON only—text that includes required elements of the JSON specification, such as open and closing curly brackets around an object, commas separating objects, and proper quoting of keys. If you try to insert invalid JSON, the database will generate an error.

The main difference between the two is that one stores JSON as text and the other as binary data. The binary implementation is newer to PostgreSQL and generally preferred because it's faster at querying and has indexing capabilities.

The two types are as follows:

json

Stores JSON as text, keeping white space and maintaining the order of keys. If a single JSON object contains a particular key more than once (which is valid), the `json` type will preserve each of the repeated key/value pairs. Finally, each time a database function processes `json`-stored text, it must parse the object to interpret its structure. This can make reads from the database slower than with the `jsonb` type. Indexing is not supported. Typically, the `json` type is useful when an application has duplicate keys or needs to preserve the order of keys.

jsonb

Stores JSON in a binary format, removing white space and not maintaining the order of keys. If a single JSON object contains a particular key more than once, the `jsonb` type will preserve only the last of the key/value pairs. The binary format adds some overhead to writing data to the table, but processing is faster. Indexing is supported.

Neither `json` nor `jsonb` is part of the ANSI SQL standard, which doesn't specify a JSON data type and leaves it to database makers to decide how to implement support. The PostgreSQL documentation at *https://www.postgresql.org/docs/current/datatype-json.html* recommends using `jsonb` unless there's a need to preserve the order of key/value pairs.

We'll use `jsonb` exclusively in the remainder of the chapter, both because of speed considerations but also because many of PostgreSQL's JSON functions work the same way with both `json` and `jsonb`—and there are more functions available for `jsonb`. We'll continue by adding the films JSON from *Listing 16-1* to a table and exploring JSON query syntax.

Importing and Indexing JSON Data

The file *films.json* in the Chapter 16 folder of the book's resources at *https://nostarch.com/practical-sql-2nd-edition/* contains a modified form of the JSON in *Listing 16-1*. View the file with a text editor, and you'll see each film's JSON object is placed on a single line, with no line breaks between elements. I've also removed the outermost square brackets and the comma separating the two film objects. Each remains a valid JSON object:

```
{"title": "The Incredibles", "year": 2004, --snip-- }
{"title": "Cinema Paradiso", "year": 1988, --snip-- }
```

I set up the file this way so that PostgreSQL's `COPY` command will interpret each film's JSON object as a separate row on import, the same way it does when importing a CSV file. The code in *Listing 16-2* makes a simple `films` table with a surrogate primary key and a `jsonb` column called `film`.

```
  CREATE TABLE films (
      id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
      film jsonb NOT NULL
  );

  COPY films (film)
1 FROM C:\YourDirectory\films.json';

2 CREATE INDEX idx_film ON films USING GIN (film);
```

Listing 16-2: Creating a table to hold JSON data and adding an index

Note that the `COPY` statement ends with the `FROM` clause 1 instead of continuing to include a `WITH` statement as in previous examples. The reason we no longer need the `WITH` statement, which we've used to specify options for file headers and CSV formatting, is that this file has no header and isn't delimited. We just want the database to read each line and process it.

After import, we add an index 2 to the `jsonb` column using the GIN index type. We discussed the generalized inverted index (GIN) with full-text search in Chapter 14. GIN's implementation of indexing the location of words or key values within text is particularly suited to JSON data. Note that because index entries point to rows in a table, `jsonb` column indexing works best when each row contains a relatively small chunk of JSON—as opposed to a table with one row that has a single, enormous JSON value and repeated keys.

Execute the commands to create and fill the table and add the index. Run `SELECT * FROM films;` and you should see two rows containing the autogenerated `id` and the JSON object text. Now you're ready to explore querying the data using with PostgreSQL's JSON operators.

Using json and jsonb Extraction Operators

To retrieve values from our stored JSON, we can use PostgreSQL-specific *extraction operators*, which return either a JSON object, an element of an array, or an element that exists at a path in the JSON structure we specify. Table 16-1 shows the operators and their functions, which can vary based on the data type of the input. Each works with `json` and `jsonb` data types.

Table 16-1: *`json` and `jsonb` Extraction Operators*

Operator, syntax	Function	Returns

`json -> text`
 	
`jsonb -> text`
	Extracts a key value, specified as text	`json` or `jsonb` (matching the input)

`json ->> text`
 	
`jsonb ->> text`
	Extracts a key value, specified as text	`text`

`json -> integer`
 	
`jsonb -> integer`
	Extracts an array element, specified as an integer denoting its array position	`json` or `jsonb` (matching the input)

`json ->> integer`
 	
`jsonb ->> integer`
	Extracts an array element, specified as an integer denoting its array position	`text`

`json #> text array`
 	
`jsonb #> text array`
	Extracts a JSON object at a specified path	`json` or `jsonb` (matching the input)

`json #>> text array`
 	
`jsonb #>> text array`
	Extracts a JSON object at a specified path	`text`

Let’s try the operators with our films JSON to learn more about how they vary in function.

Key Value Extraction

In Listing 16-3 we use the -> and ->> operators followed by text naming the key value to retrieve. In that context, with text input, these are called *field extraction operators* because they extract a field, or key value, from the JSON. The difference between the two is that -> returns the key value as JSON in the same type as stored, and ->> returns the key value as text.

```
SELECT id, film ->1 'title' AS title
FROM films
ORDER BY id;

SELECT id, film ->>2 'title' AS title
FROM films
ORDER BY id;

SELECT id, film ->3 'genre' AS genre
FROM films
ORDER BY id;
```

Listing 16-3: Retrieving a JSON key value with field extraction operators

In the `SELECT` list, we specify our JSON column name followed by the operator and the key name in single quotes. In the first example, the syntax `-> 'title'` 1 returns the value of the `title` key as JSON in the same data type as stored, `jsonb`. Run the first query, and you should see the output like this:

```
id       title
-- -----------------
 1 "The Incredibles"
 2 "Cinema Paradiso"
```

In pgAdmin, the data type listed in the `title` column header should indicate `jsonb`, and the film titles remain quoted, as they are in the JSON object.

Changing the field extraction operator to ->> 2 returns the film titles as text instead:

```
id       title
-- ---------------
 1 The Incredibles
 2 Cinema Paradiso
```

Finally, we'll return an array. In our films JSON, the value of the key `genre` is an array of values. Using the field extraction operator `->` 3 returns the array as `jsonb`:

```
id          genre
-- ---------------------------------
 1 ["animation", "action", "sci-fi"]
 2 ["romance", "drama"]
```

If we used `->>` here, we'd return the arrays as text. Let's look at how to extract elements from an array.

Array Element Extraction

To retrieve a specific value from an array, we follow the `->` and `->>` operators with an integer specifying the value's position, or *index*, in the array. We call these *element extraction operators* because they retrieve an element from a JSON array. As with field extraction, `->` returns the value as JSON in the same type as stored, and `->>` returns it as text.

Listing 16-4 shows four examples using the array values of `"genre"`.

```
SELECT id, film -> 'genre' -> 01  AS genres
FROM films
ORDER BY id;

SELECT id, film -> 'genre' -> -12 AS genres
FROM films
ORDER BY id;

SELECT id, film -> 'genre' -> 23 AS genres
FROM films
ORDER BY id;

SELECT id, film -> 'genre' ->> 04 AS genres
```

```
FROM films
ORDER BY id;
```

Listing 16-4: Retrieving a JSON array value with element extraction operators

We must first retrieve the array value from the key as JSON and then retrieve the desired element from the array. In the first example, we specify the JSON column `film`, followed by the field extraction operator `->` and the `genre` key name in single quotes. This returns the `genre` value as `jsonb`. We follow the key name with `->` and the integer `0` 1 to get the first element.

Why not use `1` for the first value in the array? In many languages, including Python and JavaScript, index values start at zero, and that's also true when accessing JSON arrays with SQL.

NOTE

SQL arrays have a different ordering scheme than JSON arrays in PostgreSQL. The first element in a SQL array is at position `1`; in a JSON array, the first element is at position `0`.

Run the first query, and your results should look like this, showing the first element in each film's `genre` array, returned as `jsonb`:

```
id   genres
-- -----------
 1 "animation"
 2 "romance"
```

We can also access the last element of the array, even if we aren't sure of its index, because the number of genres per film can vary. We count backward from the end of the list using a negative index number. Supplying `-1` 2 tells `->` to get the first element from the end of the list:

```
id  genres
-- --------
 1 "sci-fi"
 2 "drama"
```

We can count back further if we want—an index of -2 will get the next-to-last element.

Note that PostgreSQL won't return an error if there's no element at the supplied index position; it will simply return a NULL for that row. For example, if we supply 2 3 for the index, we see results for one of our films and a NULL for the other:

```
id  genres
-- --------
 1 "sci-fi"
 2
```

We get a NULL back for *Cinema Paradiso* because it has only two elements in its genre value array, and index 2 (since we count up starting with zero) represents the third element. Later in the chapter, we'll learn how to count array lengths.

Finally, changing the element extraction operator to ->> 4 returns the desired element as a text data type rather than JSON:

```
id  genres
-- ---------
 1 animation
 2 romance
```

This is the same pattern as we saw when extracting key values: -> returns a JSON data type, and ->> returns text.

Path Extraction

Both #> and #>> are *path extraction operators* that return an object located at a JSON path. A path is a series of keys or array indices that lead to the location of a value. In our example JSON, it might be just the title key if we want the name of the film. Or it could be more complex, such as the characters key followed by an index value of 1, then the actor key; this would provide the path to the name of the actor at index 1. The #> path extraction operator returns a JSON data type matching the stored data, and #>> returns text.

Consider the MPAA rating for the film *The Incredibles*, which appears in our JSON like this:

```
"rating": {
    "MPAA": "PG"
}
```

The structure is a key named `rating` with an object for its value; inside that object is a key/value pair with `MPAA` as the key name. Thus, the path to the film's MPAA rating begins with the `rating` key and ends with the `MPAA` key. To denote the path's elements, we use the PostgreSQL string syntax for arrays, creating a comma-separated list inside curly brackets and single quotes. We then feed that string to the path extraction operators. Listing 16-5 shows three examples of setting paths.

```
SELECT id, film #> '{rating, MPAA}'1 AS mpaa_rating
FROM films
ORDER BY id;

SELECT id, film #> '{characters, 0, name}'2 AS name
FROM films
ORDER BY id;

SELECT id, film #>> '{characters, 0, name}'3 AS name
FROM films
ORDER BY id;
```

Listing 16-5: Retrieving a JSON key value with path extraction operators

To get each film's MPAA rating, we specify the path in an array: `{rating, MPAA}` 1 with each item separated by commas. Run the query, and you should see these results:

```
id mpaa_rating
-- -----------
 1 "PG"
 2
```

The query returns the PG rating for *The Incredibles* and a `NULL` for *Cinema Paradiso* because, in our data, the latter film has no MPAA rating present.

The second example works with the array of `characters`, which in our JSON looks like this:

```
"characters": [{
    "name": "Salvatore",
    "actor": "Salvatore Cascio"
}, {
    "name": "Alfredo",
    "actor": "Philippe Noiret"
}]
```

The `characters` array shown is for the second movie, but both films have a similar structure. Array objects each represent a character and the name and the actor who played them. To locate the name of the first character in the array, we specify a path 2 that starts at the `characters` key, continues to the first element of the array using the index `0`, and ends at the `name` key. The query results should look like this:

```
id       name
-- ----------------
 1 "Mr. Incredible"
 2 "Salvatore"
```

The `#>` operator returns results as a JSON data type, in our case `jsonb`. If we want the results as text, we use `#>>` 3 with the same path.

Containment and Existence

The final collection of operators we'll explore performs two kinds of evaluations. The first concerns *containment* and checks whether a specified JSON value contains a second specified JSON value. The second tests for *existence*: whether a string of text within a JSON object exists as a top-level key (or as an element of an array nested inside a deeper object). Both kinds of operators return a Boolean value, which means we can use them in a `WHERE` clause to filter query results.

This set of operators works only with the `jsonb` data type—another good reason to favor `jsonb` over `json`—and can make use of our GIN index for efficient searching. *Table 16-2* lists the operators with their syntax and function.

Table 16-2: *`jsonb` Containment and Existence Operators*

Operator, syntax	Function	Returns
`jsonb` `@>` *`jsonb`*	Tests whether the first JSON value contains the second JSON value	`boolean`
`jsonb` `<@` *`jsonb`*	Tests whether the second JSON value contains the first JSON value	`boolean`
`jsonb` `?` *`text`*	Tests whether the text exists as a top-level (not nested) key or an array value	`boolean`
`jsonb` `?|` *`text array`*	Tests whether any of the text elements in the array exist as a top-level (not nested) key or as an array value	`boolean`
`jsonb` `?&` *`text array`*	Tests whether all of the text elements in the array exist as a top-level (not nested) key or as an array value	`boolean`

Using Containment Operators

In *Listing 16-6*, we use `@>` to evaluate whether one JSON value contains a second JSON value.

```
SELECT id, film ->> 'title' AS title,
       film @>1 '{"title": "The Incredibles"}'::jsonb AS
is_incredible
FROM films
ORDER BY id;
```

Listing 16-6: Demonstrating the `@>` containment operator

In our `SELECT` list, we check whether the JSON stored in the `film` column in each row contains the key/value pair for *The Incredibles*. We use the `@>` containment operator 1 in an expression that generates a column with the Boolean result `true` if `film` contains `"title": "The Incredibles"`. We give the name of our JSON column, `film`, then the `@>` operator, and then a string (cast to `jsonb`) specifying the key/value pair. In our `SELECT` list, we also return the text of the film title as a column. Running the query should produce these results:

```
id      title      is_incredible
-- ---------------- -------------
 1 The Incredibles  true
 2 Cinema Paradiso  false
```

As expected, the expression evaluates to `true` for *The Incredibles* and `false` for *Cinema Paradiso*.

Because the expression evaluates to a Boolean result, we can use it in a query's `WHERE` 2 clause, as shown in *Listing 16-7*.

```
SELECT film ->> 'title' AS title,
       film ->> 'year' AS year
FROM films
2 WHERE film @> '{"title": "The Incredibles"}'::jsonb;
```

Listing 16-7: Using a containment operator in a `WHERE` clause

Here we again check that the JSON in the `film` column contains the key/value pair for the title of *The Incredibles*. By placing the evaluation in a `WHERE` clause, the query should return just the row where the expression returns `true`:

```
     title      year
--------------- ----
The Incredibles 2004
```

Finally, in *Listing 16-8*, we flip the order of evaluation to check whether the key/value pair specified is contained within the `film` column.

```
SELECT film ->> 'title' AS title,
       film ->> 'year' AS year
FROM films
WHERE '{"title": "The Incredibles"}'::jsonb <@3 film;
```

Listing 16-8: Demonstrating the `<@` containment operator

Here we use the `<@` operator 3 instead of `@>` to flip the order of evaluation. This expression also evaluates to `true`, returning the same result as the previous query.

Using Existence Operators

Next, in *Listing 16-9*, we explore three existence operators. These check whether the text we supply exists as a top-level key or as an element of an array. All return a Boolean value.

```
SELECT film ->> 'title' AS title
FROM films
WHERE film ?1 'rating';

SELECT film ->> 'title' AS title,
       film ->> 'rating' AS rating,
       film ->> 'genre' AS genre
FROM films
WHERE film ?|2 '{rating, genre}';

SELECT film ->> 'title' AS title,
       film ->> 'rating' AS rating,
       film ->> 'genre' AS genre
FROM films
WHERE film ?&3 '{rating, genre}';
```

Listing 16-9: Demonstrating existence operators

The `?` operator checks for the existence of a single key or array element. In the first query's `WHERE` clause, we give the `film` column, the `?` operator 1, and then the string `rating`. This syntax says, "In each row, does `rating` exist as a key in the JSON in the `film` column?" When we run the query, the results show the one film that has a `rating` key, *The Incredibles*.

The `?|` and `?&` operators act as `or` and `and`. For example, using `?|` 2 tests whether either `rating` or `genre` exist as top-level keys. Running that second query returns both films, because both have at least one of those keys. Using `?&` 3, however, tests whether both `rating` and `genre` exist as keys, and that's true for only *The Incredibles*.

All these operators provide options for fine-tuning your exploration of your JSON data. Now, let's use some of them on a larger dataset.

Analyzing Earthquake Data

In this section, we'll analyze a collection of JSON data about earthquakes compiled by the US Geological Survey, an agency of the US Department of the Interior that monitors natural phenomenon including volcanoes, landslides, and water quality. The USGS uses a network of seismographs that record the earth's vibrations, compiling data on each seismic event's location and intensity. Minor earthquakes occur around the world many times a day; the big ones are less frequent but potentially devastating.

For our exercise, I fetched a month's worth of JSON-formatted earthquake data from a USGS *application programming interface*, better known as an API. An *API* is a resource for transmitting data and commands between computers, and JSON is often used for APIs. You'll find the data in the file *earthquakes.json* in the folder for this chapter included in the book's resources.

Exploring and Loading the Earthquake Data

Listing 16-10 shows the data structure for each earthquake record in the file, along with a selection of its key/value pairs (your *Chapter_16.sql* file has the nonsnipped version).

```
{
    "type": "Feature", 1
    "properties":2 {
        "mag": 1.44,
        "place": "134 km W of Adak, Alaska",
        "time": 1612051063470,
        "updated": 1612139465880,
        "tz": null,
        --snip--
        "felt": null,
        "cdi": null,
        "mmi": null,
        "alert": null,
        "status": "reviewed",
        "tsunami": 0,
        "sig": 32,
        "net": "av",
        "code": "91018173",
        "ids": ",av91018173,",
        "sources": ",av,",
        "types": ",origin,phase-data,",
        "nst": 10,
        "dmin": null,
        "rms": 0.15,
        "gap": 174,
        "magType": "ml",
        "type": "earthquake",
        "title": "M 1.4 - 134 km W of Adak, Alaska"
    },
    "geometry":3 {
        "type": "Point",
        "coordinates": [-178.581, 51.8418333333333, 22.48]
    },
    "id": "av91018173"
}
```

Listing 16-10: JSON with data on one earthquake

This data is in *GeoJSON* format, a JSON-based specification for spatial data. GeoJSON will include one or more `Feature` objects, denoted by inclusion of the key/value pair `"type": "Feature"` 1. Each `Feature` describes a single spatial object and contains both descriptive attributes (such as event time or related codes) under `properties` 2 plus a `geometry` 3 key that includes the coordinates of the spatial object. In our data, each `geometry` is a Point, a simple feature with the coordinates of one earthquake's longitude, latitude, and depth in kilometers. We discussed Points and simple features in Chapter 15 when working with PostGIS; GeoJSON incorporates it and other spatial simple features. You can read more about the GeoJSON specification at *https://geojson.org/* and see definitions of the keys in the USGS documentation at *https://earthquake.usgs.gov/data/comcat/data-eventterms.php/*.

Let's load our data into a table called `earthquakes` using the code in Listing 16-11.

```
CREATE TABLE earthquakes (
    id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    earthquake jsonb1 NOT NULL
);

COPY earthquakes (earthquake)
FROM C:\YourDirectory\earthquakes.json';
```

```
2 CREATE INDEX idx_earthquakes ON earthquakes USING GIN
(earthquake);
```

Listing 16-11: *Creating and loading an earthquakes table*

As with our `films` table, we use `COPY` to copy the data into a single `jsonb` column 1 and add a GIN index 2. Running `SELECT * FROM earthquakes;` should return 12,899 rows. Now let's see what we can learn from the data.

Working with Earthquake Times

The `time` key/value pair represents the moment the earthquake occurred. In Listing 16-12, we retrieve the value of `time` using a path extraction operator.

```
SELECT id, earthquake #>> '{properties, time}'1 AS time
FROM earthquakes
ORDER BY id LIMIT 5;
```

Listing 16-12: *Retrieving the earthquake time*

In the `SELECT` list, we give the `earthquake` column followed by a `#>>` path extraction operator and the path 1 to the time value denoted as an array. The `#>>` operator will return our value as text. Running the query should return five rows:

```
id     time
-- -------------
 1 1612137592990
 2 1612137479762
 3 1612136740672
 4 1612136207600
 5 1612135893550
```

If those values don't look like times to you, that's not surprising. By default, the USGS represents time as milliseconds since the Unix epoch at 00:00 UTC on January 1, 1970. That's a variant of the standard epoch time we covered in Chapter 12, which measures seconds since the epoch. We can convert this USGS `time` value to something understandable using `to_timestamp()` and a little math, as shown in Listing 16-13.

```
SELECT id, earthquake #>> '{properties, time}' as time,
     1 to_timestamp(
           (earthquake #>> '{properties, time}')::bigint /
10002
                 ) AS time_formatted
FROM earthquakes
ORDER BY id LIMIT 5;
```

Listing 16-13: Converting the `time` value to a timestamp

Inside the parentheses of the `to_timestamp()` 1 function, we repeat the code to extract the `time` value. The `to_timestamp()` function requires a number representing seconds, but the extracted value is text and in milliseconds, so we also cast the extracted text to `bigint` and divide by 1,000 2 to convert it to seconds.

On my machine, the query generates the following results showing the extracted `time` value and its converted timestamp (your values will vary depending on your PostgreSQL server's time zone, so `time_formatted` will show when the earthquake occurred in your server's time zone time):

```
id     time          time_formatted
-- ------------- ----------------------
 1 1612137592990 2021-01-31 18:59:52-05
 2 1612137479762 2021-01-31 18:57:59-05
 3 1612136740672 2021-01-31 18:45:40-05
 4 1612136207600 2021-01-31 18:36:47-05
 5 1612135893550 2021-01-31 18:31:33-05
```

Now that we have an understandable timestamp, let's find the oldest and newest earthquake times using the `min()` and `max()` aggregate functions in Listing 16-14.

```
SELECT min1(to_timestamp(
              (earthquake #>> '{properties, time}')::bigint /
1000
                         )) AT TIME ZONE 'UTC'2 AS
min_timestamp,
       max3(to_timestamp(
              (earthquake #>> '{properties, time}')::bigint /
1000
```

```
                          )) AT TIME ZONE 'UTC' AS
max_timestamp
FROM earthquakes;
```

Listing 16-14: Finding the minimum and maximum earthquake times

We place `to_timestamp()` and our milliseconds-to-seconds conversion inside both the `min()` 1 and `max()` 3 functions in our `SELECT` list. This time, we add the keywords `AT TIME ZONE 'UTC'` 2 after both functions; regardless of our server time zone settings, the results will display the timestamps in UTC, as USGS records them. Your results should look like this:

```
   min_timestamp        max_timestamp
------------------- -------------------
2021-01-01 00:01:39 2021-01-31 23:59:52
```

This collection of earthquakes spans a month—from early morning January 1, 2021, through the end of day on January 31. That's helpful context as we continue to dig for usable information.

Finding the Largest and Most-Reported Earthquakes

Next, we'll look at two data points that measure an earthquake's size and the degree to which citizens reported feeling it and apply JSON extraction techniques to simple sorting of results.

Extracting by Magnitude

The USGS reports each earthquake's magnitude in the `mag` key, beneath `properties`. Magnitude, according to the USGS, is a number representing the size of an earthquake at its source. Its scale is logarithmic: a magnitude 4 earthquake has seismic waves whose amplitude is about 10 times bigger than a quake with a magnitude of 3. With that context, let's find the five largest earthquakes in our data using the code in Listing 16-15.

```
SELECT earthquake #>> '{properties, place}'1 AS place,
       to_timestamp((earthquake #>> '{properties,
time}')::bigint / 1000)
           AT TIME ZONE 'UTC' AS time,
       (earthquake #>> '{properties, mag}')::numeric AS
magnitude
FROM earthquakes
2 ORDER BY (earthquake #>> '{properties, mag}')::numeric3 DESC
NULLS LAST
LIMIT 5;
```

Listing 16-15: Finding the five earthquakes with the largest magnitude

We again use path extraction operators to retrieve our desired elements, including values for `place` 1 and `mag`. To show the largest five in our results, we add an `ORDER BY` clause 2 with `mag`. We cast the value to numeric 3 here and in the `SELECT` because we want to display and sort the value as a number rather than as text. We also add the `DESC NULLS LAST` keywords, which sorts the results in descending order and places `NULL` values (of which there are two) last. Your results should look like this:

```
               place                        time
magnitude
------------------------------------- ------------------- ---
------
211 km SE of Pondaguitan, Philippines 2021-01-21 12:23:04
7
South Shetland Islands                2021-01-23 23:36:50
6.9
30 km SSW of Turt, Mongolia           2021-01-11 21:32:59
6.7
28 km SW of Pocito, Argentina         2021-01-19 02:46:21
6.4
Kermadec Islands, New Zealand         2021-01-08 00:28:50
6.3
```

The largest, of magnitude 7, was located beneath the ocean southeast of the small city of Pondaguitan in the Philippines. The second was in the Antarctic near the South Shetland Islands.

Extracting by Citizen Reports

The USGS operates a Did You Feel It? website at https://earthquake.usgs.gov/data/dyfi/ where people can report their earthquake experiences. Our JSON includes the number of reports for each earthquake under the key `felt`, beneath `properties`. Let’s see which earthquakes in our data generated the most reports using the code in Listing 16-16.

```
SELECT earthquake #>> '{properties, place}' AS place,
       to_timestamp((earthquake #>> '{properties,
time}')::bigint / 1000)
           AT TIME ZONE 'UTC' AS time,
       (earthquake #>> '{properties, mag}')::numeric AS
magnitude,
       (earthquake #>> '{properties, felt}')::integer1 AS
felt
FROM earthquakes
ORDER BY (earthquake #>> '{properties, felt}')::integer2 DESC
NULLS LAST
LIMIT 5;
```

Listing 16-16: Finding earthquakes with the most Did You Feel It? reports

Structurally, this query is similar to Listing 16-15 that found the largest quakes. We add a path extraction operator for the `felt` 1 key, casting the returned text value to an `integer` type. We cast to `integer` so the extracted text is treated as a number for sorting and display. Finally, we place the extraction code in `ORDER BY` 2, using `NULLS LAST` because there are many earthquakes with no reports and we want those to appear last in the list. You should see these results:

```
          place                      time           magnitude felt
------------------------- ------------------- ---------- -----
4km SE of Aromas, CA      2021-01-17 04:01:27        4.2 19907
2km W of Concord, CA      2021-01-14 19:18:10       3.63  5101
10km NW of Pinnacles, CA  2021-01-02 14:42:23       4.32  3076
2km W of Willowbrook, CA  2021-01-20 16:31:58       3.52  2086
3km NNW of Santa Rosa, CA 2021-01-19 04:22:20       2.68  1765
```

The top five are in California, which makes sense. Did You Feel It? is a US government-run system, so we’d expect more US reports—particularly in earthquake-prone California. Also, some of the largest quakes in our data occurred beneath oceans or in remote regions. The quake with more than 19,900 reports was moderate, but its nearness to cities meant more chance for people to notice it.

Converting Earthquake JSON to Spatial Data

Our JSON data has longitude and latitude values for each earthquake, meaning we can perform spatial analysis using the GIS techniques discussed in Chapter 15. For example, we'll use a PostGIS distance function to locate earthquakes that occurred within 50 miles from a city. First, though, we must convert the coordinates stored in JSON to a PostGIS data type.

The longitude and latitude values are found in the array of the `coordinates` key, under `geometry`. Here's an example:

```
    "geometry": {
        "type": "Point",
        "coordinates": [-178.581, 51.8418333333333, 22.48]
    }
```

The first coordinate, at position `0` in the array, represents longitude; the second, at position `1`, is latitude. The third value denotes depth in kilometers, which we won't use. To extract these elements as text, we make use of a `#>>` path operator, as in Listing 16-17.

```
SELECT id,
       earthquake #>> '{geometry, coordinates}' AS
coordinates,
       earthquake #>> '{geometry, coordinates, 0}' AS
longitude,
       earthquake #>> '{geometry, coordinates, 1}' AS
latitude
FROM earthquakes
ORDER BY id
LIMIT 5;
```

Listing 16-17: Extracting the earthquake's location data

The query should return five rows:

```
id             coordinates              longitude    latitude
-- -------------------------------- ------------ ----------
 1 [-122.852, 38.8228333, 2.48]     -122.852     38.8228333
 2 [-148.3859, 64.2762, 16.2]       -148.3859    64.2762
 3 [-152.489, 59.0143, 73]          -152.489     59.0143
 4 [-115.82, 32.7493333, 9.85]      -115.82      32.7493333
 5 [-115.6446667, 33.1711667, 5.89] -115.6446667 33.1711667
```

A quick visual compare of our result to the JSON `longitude` and `latitude` values tells us we've extracted the values properly. Next, we'll use a PostGIS function to convert those values to a Point in the `geography` data type.

NOTE

Your `analysis` database must have PostGIS enabled for this part of the chapter. If you skipped Chapter 15, you can enable PostGIS by running `CREATE EXTENSION postgis;` in pgAdmin.

Listing 16-18 generates a Point of type `geography` for each earthquake, which we can use as input for PostGIS spatial functions.

```
SELECT ST_SetSRID(
         ST_MakePoint1(
            (earthquake #>> '{geometry, coordinates,
0}')::numeric,
            (earthquake #>> '{geometry, coordinates,
1}')::numeric
         ),
             43262)::geography AS earthquake_point
FROM earthquakes
ORDER BY id;
```

Listing 16-18: Converting JSON location data to PostGIS geography

Inside `ST_MakePoint()` 1, we place our code to extract longitude and latitude, casting both values to type `numeric` as required by the function. We nest that function inside `ST_SetSRID()` to set a spatial reference system identifier (SRID) for the resulting Point. In Chapter 15, you learned that the SRID specifies a coordinate grid for plotting spatial objects. The SRID value `4326` 2 denotes the commonly used WGS 84 coordinate system. Finally, we cast the entire output to the `geography` type. The first several rows should look like this:

```
                  earthquake_point
--------------------------------------------------
0101000020E61000004A0C022B87B65EC0A6C7009A52694340
0101000020E6100000D8F0F44A598C62C0EFC9C342AD115040
0101000020E6100000CFF753E3A50F63C0992A1895D4814D40
--snip--
```

We can’t interpret those strings of digits and letters directly, but we can use pgAdmin’s Geometry Viewer to see the Points plotted on a map. With your query results visible in the pgAdmin Data Output pane, click the eye icon in the `earthquake_point` result header. You should see the earthquakes plotted on a map that uses OpenStreetMap as the base layer, as in *Figure 16-1*.

Even with only a month of data, it’s easy to see the abundance of earthquakes concentrated around the edges of the Pacific Ocean, in the so-called Ring of Fire where tectonic plates meet and volcanos are more active.

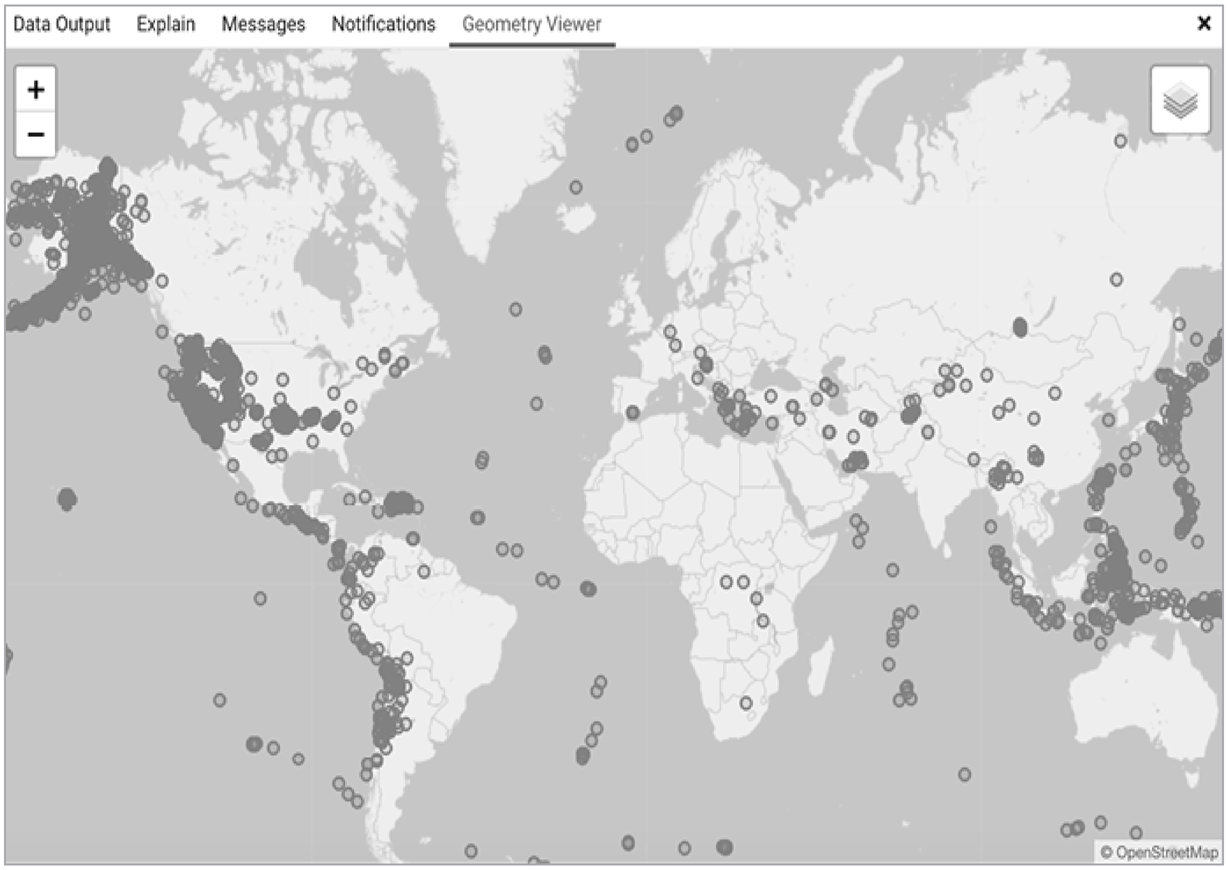


Figure 16-1: Viewing earthquake locations in pgAdmin

Finding Earthquakes Within a Distance

Next, let’s narrow our study to earthquakes that occurred near Tulsa, Oklahoma—a part of the country that has seen increased seismic activity since 2009 as a result of oil and gas processing, according to the USGS.

To perform more complex GIS tasks like this, it’s easier if we permanently convert the JSON coordinates to a column of PostGIS type `geography` in the `earthquakes` table. That allows us to avoid the clutter of adding conversion code in each query.

Listing 16-19 adds a column called `earthquake_point` to the `earthquakes` table and fills the new column with the JSON coordinates converted to type `geography`.

```
1 ALTER TABLE earthquakes ADD COLUMN earthquake_point
   geography(POINT, 4326);

2 UPDATE earthquakes
   SET earthquake_point =
           ST_SetSRID(
               ST_MakePoint(
                   (earthquake #>> '{geometry, coordinates,
   0}')::numeric,
                   (earthquake #>> '{geometry, coordinates,
   1}')::numeric
                ),
                    4326)::geography;

3 CREATE INDEX quake_pt_idx ON earthquakes USING GIST
    (earthquake_point);
```

Listing 16-19: Converting JSON coordinates to a PostGIS geometry column

We use `ALTER TABLE` 1 to add a column `earthquake_point` of type `geography`, specifying that the column will hold Points with an SRID of `4326`. Next, we `UPDATE` 2 the table, setting the `earthquake_point` column using the same syntax as in *Listing 16-18*, and add a spatial index using GIST 3 to the new column.

That done, we can use Listing 16-20 to find earthquakes within 50 miles of Tulsa.

```
SELECT earthquake #>> '{properties, place}' AS place,
       to_timestamp((earthquake -> 'properties' ->>
'time')::bigint / 1000)
           AT TIME ZONE 'UTC' AS time,
       (earthquake #>> '{properties, mag}')::numeric AS
magnitude,
       earthquake_point
FROM earthquakes
1 WHERE ST_DWithin(earthquake_point,
                 2 ST_GeogFromText('POINT(-95.989505
36.155007)'),
                   80468)
ORDER BY time;
```

Listing 16-20: Finding earthquakes within 50 miles of downtown Tulsa, Oklahoma

In the `WHERE` clause 1, we employ the `ST_DWithin()` function, which returns a Boolean value of `true` if one spatial object is within a specified distance of another object. Here, we want to evaluate each earthquake Point to check whether it's within 50 miles of downtown Tulsa. We designate the city's coordinates in `ST_GeogFromText()` 2 and supply the value of 50 miles using its meters equivalent, `80468`, as meters is the required input. The query should return 19 rows (I've omitted the `earthquake_point` column and truncated the results for brevity):

```
              place                       time           magnitude
------------------------------ -------------------- ----------
4 km SE of Owasso, Oklahoma    2021-01-04 19:46:58       1.53
6 km SSE of Cushing, Oklahoma  2021-01-05 08:04:42       0.91
2 km SW of Hulbert, Oklahoma   2021-01-05 21:08:28       1.95
--snip--
```

View the earthquake locations by clicking the eye icon atop the `earthquake_point` column in the results in pgAdmin. You should see 19 dots around the city, as in Figure 16-2 (and you can adjust the underlying map style by clicking the layer icon at top right).

Achieving these results required some coding gymnastics that would have been unnecessary if the data had arrived in a shapefile or in a typical SQL table. Nevertheless, it's possible to extract meaningful insights from JSON data using PostgreSQL's support for the format. In the last part of the chapter, we'll cover useful PostgreSQL functions for generating and manipulating JSON.

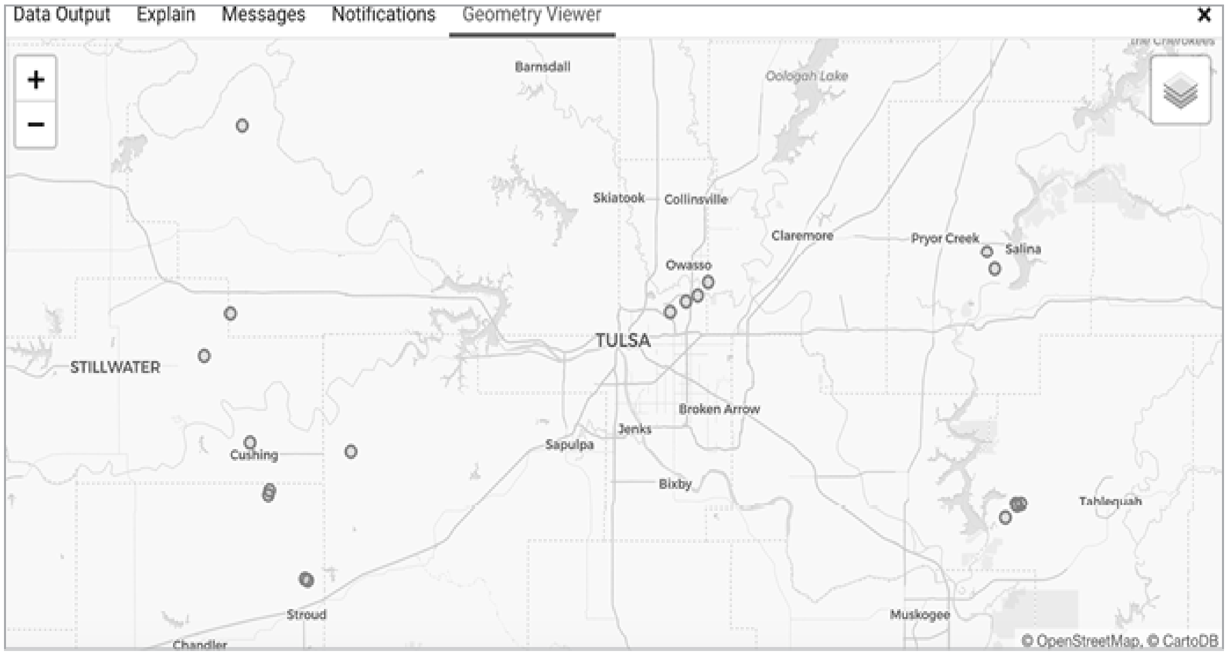


Figure 16-2: Viewing earthquakes near Tulsa, Oklahoma, in pgAdmin

Generating and Manipulating JSON

We can use PostgreSQL functions to create JSON from existing rows in a SQL table or to modify JSON stored in a table to add, subtract, or change keys and values. The PostgreSQL documentation at *https://www.postgresql.org/docs/current/functions-json.html* lists several dozen JSON-related functions—we'll work through a few you might find handy.

Turning Query Results into JSON

Because JSON is primarily a format for sharing data, it's useful to be able to quickly convert the results of a SQL query into JSON for delivery to another computer system. *Listing 16-21* uses the PostgreSQL-specific `to_json()` function to turn rows from the `employees` table you made in Chapter 7 into JSON.

```
1 SELECT to_json(employees) AS json_rows
  FROM employees;
```

Listing 16-21: Turning query results into JSON with `to_json()`

The `to_json()` function does what it says: transforms a supplied SQL value to JSON. To convert all values in each row of the `employees` table, we use `to_json()` in a `SELECT` 1 and supply the table name as the function's argument; that returns each row as a JSON object with column names as keys:

```
                                     json_rows
---------------------------------------------------------------
-------------------------
{"emp_id":1,"first_name":"Julia","last_name":"Reyes","salary"
:115300.00,"dept_id":1}
{"emp_id":2,"first_name":"Janet","last_name":"King","salary":
98000.00,"dept_id":1}
{"emp_id":3,"first_name":"Arthur","last_name":"Pappas","salar
y":72700.00,"dept_id":2}
{"emp_id":4,"first_name":"Michael","last_name":"Taylor","sala
ry":89500.00,"dept_id":2}
```

We can modify our query a few ways to limit which columns to include in the results. In *Listing 16-22*, we use a `row()` constructor as the argument for `to_json()`.

```
SELECT to_json(row(emp_id, last_name))1 AS json_rows
FROM employees;
```

Listing 16-22: Specifying columns to convert to JSON

A `row()` constructor (which is ANSI SQL compliant) builds a row value from the arguments passed to it. In this case, we supply the column names `emp_id` and `last_name` 1 and place `row()` inside `to_json()`. This syntax returns just those columns in the JSON result:

```
      json_rows
----------------------
{"f1":1,"f2":"Reyes"}
{"f1":2,"f2":"King"}
{"f1":3,"f2":"Pappas"}
{"f1":4,"f2":"Taylor"}
```

Notice, however, that the keys are named `f1` and `f2` instead of their source column names. That's a side effect of `row()`, which doesn't preserve column names when it builds the row record. We can set the names of the keys, which is often done to keep the names short and reduce JSON file size, improving transfer speeds. Listing 16-23 shows how via a subquery.

```
SELECT to_json(employees) AS json_rows
FROM (
    1 SELECT emp_id, last_name AS ln2 FROM employees
) AS employees;
```

Listing 16-23: Generating key names with a subquery

We write a subquery 1 that grabs the columns we want and alias the result as `employees`. In the process, we alias a column name 2 to shorten its appearance as a key in the JSON.

The results should look like this:

```
        json_rows
--------------------------
{"emp_id":1,"ln":"Reyes"}
{"emp_id":2,"ln":"King"}
{"emp_id":3,"ln":"Pappas"}
{"emp_id":4,"ln":"Taylor"}
```

Finally, Listing 16-24 shows how to compile all the rows of JSON into a single array of objects. You may want to do this if you're providing this data to another application that will iterate over the array of objects to perform a task, such as a calculation, or to render data on a device.

```
1 SELECT json_agg(to_json(employees)) AS json
  FROM (
    SELECT emp_id, last_name AS ln FROM employees
) AS employees;
```

Listing 16-24: Aggregating the rows and converting to JSON

We wrap `to_json()` in the PostgreSQL-specific `json_agg()` 1 function, which aggregates values, including `NULL`, into a JSON array. Its output should look like this:

```
                                                       json
-------------------------------------------------------------
-------------------------------
[{"emp_id":1,"ln":"Reyes"}, {"emp_id":2,"ln":"King"},
{"emp_id":3,"ln":"Pappas"}, --snip-- ]
```

These are simple examples, but you can build more complex JSON structures using subqueries to generate nested objects. We'll consider one way to do that as part of our "Try It Yourself" exercises at the end of the chapter.

Adding, Updating, and Deleting Keys and Values

We can add to, update, and delete from JSON with a combination of concatenation and PostgreSQL-specific functions. Let's work through some examples.

Adding or Updating a Top-Level Key/Value Pair

In Listing 16-25, we return to our `films` table and add a top-level key/value pair `"studio": "Pixar"` to the film *The Incredibles* using two different techniques:

```
UPDATE films
SET film = film ||1 '{"studio": "Pixar"}'::jsonb
WHERE film @> '{"title": "The Incredibles"}'::jsonb;

UPDATE films
SET film = film || jsonb_build_object('studio', 'Pixar')2
WHERE film @> '{"title": "The Incredibles"}'::jsonb;
```

Listing 16-25: Adding a top-level key/value pair via concatenation

Both examples use `UPDATE` statements to set new values for the `jsonb` column `film`. In the first, we use the PostgreSQL concatenation operator `||` 1 to combine the existing film JSON with the new key value/pair that we cast to `jsonb`. In the second, we use concatenation again but with `jsonb_build_object()`. This function takes a series of key and value names as arguments and returns a `jsonb` object, letting us concatenate several key/value pairs at a time if we wanted.

Both statements will insert the new key/value pair if the key doesn't exist in the JSON being concatenated; it will overwrite a key that's present. There's no functional difference between the two statements, so feel free to use whichever you prefer. Note that this behavior is specific to `jsonb`, which doesn't allow duplicate key names.

If you `SELECT * FROM films;` and double-click the updated data in the `film` column, you should see the new key/value pair:

```
--snip--
    "rating": {
        "MPAA": "PG"
    },
    "studio": "Pixar",
    "characters": [
--snip--
```

Updating a Value at a Path

Currently we have two entries for the `genre` key for *Cinema Paradiso*:

```
"genre": ["romance", "drama"]
```

To add a third entry to the array, we use the function `jsonb_set()`, which allows us to specify a value to update at a specific JSON path. In Listing 16-26, we use the `UPDATE` statement and `jsonb_set()` to add the genre `World War II`.

```
UPDATE films
SET film = jsonb_set(film, 1
                 '{genre}', 2
                  film #> '{genre}' || '["World War II"]', 3
                  true4)
WHERE film @> '{"title": "Cinema Paradiso"}'::jsonb;
```

Listing 16-26: Adding an array value at a path with `jsonb_set()`

In `UPDATE`, we `SET` the value of `film` to the result of `jsonb_set()` and use `WHERE` to limit the update to just the row with *Cinema Paradiso*. The function's first argument 1 is the target JSON we want to modify, here `film`. The second argument is the path 2 to the array value—the `genre` key. Third, we give the new value for `genre`, which we specify as the current value of `genre` concatenated with an array 3 with one value, `"World War II"`. That concatenation will produce an array with three elements. The final argument is an optional Boolean value 4 that dictates whether `jsonb_set()` should create the value if it's not already present. It's redundant here since `genre` already exists; I've shown it for reference.

Run the query and then perform a quick `SELECT` to check the updated JSON. You should see the `genre` array including three values: `["romance", "drama", "World War II"]`.

Deleting a Value

We can remove keys and values from a JSON object by pairing two operators. Listing 16-27 shows two `UPDATE` examples.

```
UPDATE films
SET film = film -1 'studio'
WHERE film @> '{"title": "The Incredibles"}'::jsonb;

UPDATE films
SET film = film #-2 '{genre, 2}'
WHERE film @> '{"title": "Cinema Paradiso"}'::jsonb;
```

Listing 16-27: Deleting values from JSON

The minus sign 1 acts as a *deletion operator*, removing the key `studio` and its value, which we added earlier for *The Incredibles*. Supplying a text string after the minus sign indicates we want to remove a key and its value; supplying an integer will remove the element at that index.

The `#-` 2 sign is a *path deletion operator* that removes the JSON element that exists at a path we specify. The syntax is similar to that of the path extraction operators `#>` and `#>>`. Here, we use `{genre, 2}` to indicate the third element of the array for `genre` (remember, JSON array indexes begin counting at zero). This will remove the value `World War II` that we added earlier to *Cinema Paradiso*.

Run both statements and then use `SELECT` to view the altered film JSON. You should see both elements removed.

Using JSON Processing Functions

To finish our JSON studies, we'll review a selection of PostgreSQL-specific functions for processing JSON data, including expanding array values into table rows and formatting output. You can find a complete listing of functions in the PostgreSQL documentation at *https://www.postgresql.org/docs/current/functions-json.html*.

Finding the Length of an Array

Counting the number of items in an array is a routine programming and analysis task. We might, for example, want to know how many actors are stored for each film in our JSON. To do this, we can use the `jsonb_array_length()` function in Listing 16-28.

```
SELECT id,
       film ->> 'title' AS title,
     1 jsonb_array_length(film -> 'characters') AS
num_characters
FROM films
ORDER BY id;
```

Listing 16-28: Finding the length of an array

As its only argument, the function 1 takes an expression that extracts the value of the `character` key from `film`. Running the query should produce these results:

```
id      title      num_characters
-- ---------------- --------------
 1 The Incredibles               3
 2 Cinema Paradiso               2
```

The output correctly shows that we have three characters for *The Incredibles* and two for *Cinema Paradiso*. Note there's a similar `json_array_length()` function for the `json` type.

Returning Array Elements as Rows

The `jsonb_array_elements()` and `jsonb_array_elements_text()` functions convert array elements into rows, with one row per element. This is a useful tool for data processing. To convert JSON into structured SQL data, for example, we could use this function to generate the rows to `INSERT` into a table or to generate rows that we can aggregate by grouping and counting.

Listing 16-29 uses both functions to turn the `genre` key's array values into rows. Each function takes a `jsonb` array as an argument. The difference between the two is that `jsonb_array_elements()` returns the array elements as rows of `jsonb` values, while `jsonb_array_elements_text()` returns elements as, you guessed it, `text`.

```
SELECT id,
       jsonb_array_elements(film -> 'genre') AS genre_jsonb,
       jsonb_array_elements_text(film -> 'genre') AS
genre_text
FROM films
ORDER BY id;
```

Listing 16-29: Returning array elements as rows

Running the code should produce these results:

```
id genre_jsonb genre_text
-- ----------- ----------
 1 "animation" animation
 1 "action"    action
 1 "sci-fi"    sci-fi
 2 "romance"   romance
 2 "drama"     drama
```

On an array with a simple list of values, that works nicely, but if an array contains a collection of JSON objects with their own key/value pairs, like `character` in our `film` JSON, we need additional processing to unpack the values first. *Listing 16-30* walks through the process.

```
SELECT id,
       jsonb_array_elements(film -> 'characters') 1
FROM films
ORDER BY id;

2 WITH characters (id, json) AS (
     SELECT id,
            jsonb_array_elements(film -> 'characters')
     FROM films
)
3 SELECT id,
       json ->> 'name' AS name,
       json ->> 'actor' AS actor
FROM characters
ORDER BY id;
```

Listing 16-30: Returning key values from each item in an array

We use `jsonb_array_elements()` to return the elements of the `characters` 1 array, which should return each JSON object in the array as a row:

```
id                  jsonb_array_elements
-- -----------------------------------------------------
 1 {"name": "Mr. Incredible", "actor": "Craig T. Nelson"}
 1 {"name": "Elastigirl", "actor": "Holly Hunter"}
 1 {"name": "Frozone", "actor": "Samuel L. Jackson"}
 2 {"name": "Salvatore", "actor": "Salvatore Cascio"}
 2 {"name": "Alfredo", "actor": "Philippe Noiret"}
```

To convert the `name` and `actor` values to columns, we employ a common table expression (CTE) as covered in Chapter 13. Our CTE 2 uses `jsonb_array_elements()` to generate a simple temporary `characters` table with two columns: the film’s `id` and the unpacked array values in a column called `json`. We follow with a `SELECT` statement 3 that queries the temporary table, extracting the values of `name` and `actor` from the `json` column:

```
id      name           actor
-- -------------- -----------------
 1 Mr. Incredible Craig T. Nelson
 1 Elastigirl     Holly Hunter
 1 Frozone        Samuel L. Jackson
 2 Salvatore      Cascio
 2 Alfredo        Philippe Noiret
```

Those values are neatly parsed into a standard SQL structure and suitable for further analysis using standard SQL.

Wrapping Up

JSON is such a ubiquitous format that it’s likely you’ll encounter it often in your journey analyzing data. You’ve learned that PostgreSQL easily handles loading, indexing, and parsing JSON, but JSON sometimes requires extra steps to process that aren’t needed with data handled via standard SQL conventions. As with many areas of coding, your decision on whether to make use of JSON will depend on your specific circumstances. Now, you’re equipped to understand the context.

JSON itself is a standard, but the data types and the majority of functions and syntax in this chapter were PostgreSQL-specific. That’s because the ANSI SQL standard leaves it to database vendors to decide how to implement most JSON support. If your work involves using Microsoft SQL Server, MySQL, SQLite, or another system, consult their documentation. You’ll find many similarities in capabilities even if the function names differ.

TRY IT YOURSELF

Use your new JSON skills to answer these questions:

The earthquakes JSON has a key `tsunami` that's set to a value of `1` for large earthquakes in oceanic regions (though it doesn't mean a tsunami actually happened). Using either path or field extraction operators, find earthquakes with a `tsunami` value of `1` and include their location, time, and magnitude in your results.

Use the following `CREATE TABLE` statement to add the table `earthquakes_from_json` to your `analysis` database:

```
CREATE TABLE earthquakes_from_json (
    id text PRIMARY KEY,
    title text,
    type text,
    quake_date timestamp with time zone,
    mag numeric,
    place text,
    earthquake_point geography(POINT, 4326),
    url text
);
```

Using field and path extraction operators, write an `INSERT` statement to fill the table with the correct values for each earthquake. Refer to the full sample earthquake JSON in your *Chapter_16.sql* file for any key names and paths you need.

Bonus (difficult) question: Try writing a query to generate the following JSON using the data in the `teachers` and `teachers_lab_access` tables from Chapter 13:

```
{
    "id": 6,
    "fn": "Kathleen",
    "ln": "Roush",
    "lab_access": [{
        "lab_name": "Science B",
        "access_time": "2022-12-17T16:00:00-05:00"
    }, {
        "lab_name": "Science A",
        "access_time": "2022-12-07T10:02:00-05:00"
    }]
}
```

It's helpful to remember that the `teachers` table has a one-to-many relationship with `teachers_lab_access`; the first three keys must come from `teachers`, and the JSON objects in the array of `lab_access` will come from `teachers_lab_access`. Hint: you'll need to use a subquery in your `SELECT` list and a function called `json_agg()` to create the `lab_access` array. If you're stumped, take a peek in your *Try_It_Yourself.sql* file included with the book's resources, where I've placed the answers to all these exercises.

17
SAVING TIME WITH VIEWS, FUNCTIONS, AND TRIGGERS

One advantage of using a programming language is that we can automate repetitive, boring tasks. That's what this chapter is about: taking the queries or steps you might do over and over and turning them into reusable database objects that you code once and can call later to let the database do the work. Programmers call this the DRY principle: Don't Repeat Yourself.

You'll start by learning to store queries as reusable database *views*. Next, you'll explore how to create database functions you can use to operate on your data, the same way you've used built-in functions like `round()` and `upper()`. Then you'll set up *triggers* to run your functions automatically when certain events occur on a table. All these techniques are useful not only for reducing repetitive work but for ensuring data integrity too.

We'll practice these techniques on tables created from examples in earlier chapters. All the code for this chapter is available for download along with the book's resources at *https://nostarch.com/practical-sql-2nd-edition/*.

Using Views to Simplify Queries

A *view* is essentially a stored query with a name that you can work with as if it were a table. For example, a view might store a query that calculates total population by state. As with a table, you could query that view, join the view to tables (or to other views), and use the view to update or insert data into a table it's based on, albeit with some caveats. The stored query in a view can be simple, referencing just one table, or complex, with multiple table joins.

Views are especially useful in the following scenarios:

Avoiding duplicate effort: They let you write a complex query once and access the results when needed.

Reducing clutter: They can trim the amount of information you need to wade through by showing only columns relevant to your needs.

Providing security: Views can limit access to only certain columns in a table.

In this section, we'll look at two kinds of views. The first—a standard view—contains PostgreSQL syntax that's largely in line with the ANSI SQL standard for views. Every time you access a standard view, the stored query runs and generates a temporary set of results. The second is a *materialized view*, which is specific to PostgreSQL, Oracle, and a limited number of other database systems. When you create a materialized view, the data returned by its query is stored permanently in the database like a table; you can refresh the view to update the stored data if needed.

NOTE

To ensure data security and prevent users from seeing sensitive information, you must restrict access by setting account permissions in PostgreSQL and also define the view using a `security_barrier` *attribute. Typically, a database administrator handles these tasks, but if you want to explore this issue further, read the PostgreSQL documentation on user roles at https://www.postgresql.org/docs/current/sql-createrole.html, on the* `GRANT` *command at https://www.postgresql.org/docs/current/sql-grant.html, and on* `security_barrier` *at https://www.postgresql.org/docs/current/rules-privileges.html.*

Views are easy to create and maintain. Let's work through several examples to see how they work.

Creating and Querying Views

In this section, we'll return to the census estimates table `us_counties_pop_est_2019` you imported in Chapter 5. *Listing 17-1* creates a standard view that returns just the population of Nevada counties. The original table has sixteen columns; the view will return just four of them. This would be useful for making a subset of Nevada census data quickly accessible when we're referring to it often or using the data in an application.

```
1 CREATE OR REPLACE VIEW nevada_counties_pop_2019 AS
     2 SELECT county_name,
              state_fips,
              county_fips,
              pop_est_2019
       FROM us_counties_pop_est_2019
       WHERE state_name = 'Nevada';
```

Listing 17-1: Creating a view that displays Nevada 2019 counties

We define the view using the keywords `CREATE OR REPLACE VIEW` 1 followed by the view's name, `nevada_counties_pop_2019`, and then `AS`.

(We can name the view any way we'd like; I prefer a name that's descriptive of the view's results.) Next, we use a standard SQL `SELECT` 2 to fetch the 2019 population estimate (the `pop_est_2019` column) for each Nevada county from the `us_counties_pop_est_2019` table.

Notice the `OR REPLACE` keywords after `CREATE`. These are optional and tell the database that if a view with this name already exists, then replace it with the new definition. It's helpful to include these keywords if you're iterating on creating a view and want to refine the query. There is one caveat: if you're replacing an existing view, the new query 2 must generate the same column names with the same data types and in the same order as the one it's replacing. You can add columns, but they must be placed at the end of the column list. If you try to do otherwise, the database will respond with an error message.

Run the code in *Listing 17-1* using pgAdmin. The database should respond with the message `CREATE VIEW`. To find the new view, in pgAdmin's object browser, right-click the `analysis` database and click **Refresh**. Choose **Schemas▶public▶Views** to see all views. When you right-click your new view and click **Properties**, you should see a more verbose version of the query (with the table name prepended to each column name) on the Code tab in the dialog that opens. That's a handy way to inspect views you might find in a database.

NOTE

As with other database objects, you can delete a view using the `DROP` *command. In this example, the syntax would be* `DROP VIEW nevada_counties_pop_2019;`*.*

This type of view—one that isn't materialized—holds no data at this point; instead, the stored `SELECT` query it contains will run when you access the view from another query. For example, the code in *Listing 17-2* returns all columns in the view. As with a typical `SELECT` query, we can use `ORDER BY` to sort results, this time using the county's Federal Information Processing Standards (FIPS) code—the standard designator the US Census

Bureau and other federal agencies use to specify each county and state. We also add a `LIMIT` clause to display just five rows.

```
SELECT *
FROM nevada_counties_pop_2019
ORDER BY county_fips
LIMIT 5;
```

Listing 17-2: Querying the `nevada_counties_pop_2010` *view*

Aside from the five-row limit, the result should be the same as if you had run the `SELECT` query used to create the view in Listing 17-1:

```
     geo_name      | state_fips  | county_fips  | pop_2010
-------------------+-------------+--------------+-----------
 Churchill County | 32          | 001          |   24909
 Clark County     | 32          | 003          | 2266715
 Douglas County   | 32          | 005          |   48905
 Elko County      | 32          | 007          |   52778
 Esmeralda County | 32          | 009          |     873
```

This simple example isn't useful unless quickly listing Nevada county population is a task you'll perform frequently. So, let's imagine a question data-minded analysts in a political research organization might ask often: what was the percent change in population for each county in Nevada (or any other state) from 2010 to 2019?

We wrote a query to answer this question in Chapter 7, and though it wasn't onerous to create, it did require joining tables on two columns and using a percent change formula that involved rounding and type casting. To avoid repeating that work, we can create a view that stores a query similar to the one in Chapter 7 as a view, as shown in Listing 17-3.

```
1 CREATE OR REPLACE VIEW county_pop_change_2019_2010 AS
    2 SELECT c2019.county_name,
           c2019.state_name,
           c2019.state_fips,
           c2019.county_fips,
           c2019.pop_est_2019 AS pop_2019,
           c2010.estimates_base_2010 AS pop_2010,
         3 round( (c2019.pop_est_2019::numeric -
c2010.estimates_base_2010)
               / c2010.estimates_base_2010 * 100, 1 ) AS
pct_change_2019_2010
 4 FROM us_counties_pop_est_2019 AS c2019
        JOIN us_counties_pop_est_2010 AS c2010
    ON c2019.state_fips = c2010.state_fips
       AND c2019.county_fips = c2010.county_fips;
```

Listing 17-3: Creating a view showing population change for US counties

We start the view definition with `CREATE OR REPLACE VIEW` 1, followed by the name of the view and `AS`. The `SELECT` query 2 names columns from the census tables and includes a column definition with a percent change calculation 3 that you learned about in Chapter 6. Then we join the 2019 and 2010 census tables 4 using the state and county FIPS codes. Run the code, and the database should again respond with `CREATE VIEW`.

Now that we've created the view, we can use the code in Listing 17-4 to run a simple query using the new view that retrieves data for Nevada counties.

```
SELECT county_name,
       state_name,
       pop_2019,
     1 pct_change_2019_2010
FROM county_pop_change_2019_2010
2 WHERE state_name = 'Nevada'
ORDER BY county_fips
LIMIT 5;
```

Listing 17-4: Selecting columns from the `county_pop_change_2019_2010` *view*

In Listing 17-2, in the query that referenced our `nevada_counties_pop_2019` view, we retrieved every column in the view by using the asterisk wildcard after `SELECT`. Listing 17-4 shows that, as with a query on a table, we can name specific columns when querying a view. Here, we specify four of the `county_pop_change_2019_2010` view's seven columns. One is `pct_change_2019_2010` 1, which returns the result of the percent change calculation we're looking for. As you can see, it's much simpler to write the column name like this than the whole formula. We're also filtering the results using a `WHERE` clause 2, similar to how we'd filter any query.

After querying the four columns from the view, the results should look like this:

```
  county_name      state_name  pop_2019  pct_change_2019_2010
----------------  ----------  --------  --------------------
Churchill County  Nevada         24909                   0.1
Clark County      Nevada       2266715                  16.2
Douglas County    Nevada         48905                   4.1
Elko County       Nevada         52778                   7.8
Esmeralda County  Nevada           873                  11.4
```

Now we can revisit this view as often as we like to pull data for presentations or to answer questions about the percent change in population for any county in America from 2010 to 2019.

Looking at just these five rows, you can see a couple of interesting stories emerge: the continued rapid growth of Clark County, which includes the city of Las Vegas, as well as a strong percent increase in Esmeralda County, one of the smallest counties in the United States and home to several ghost towns.

Creating and Refreshing a Materialized View

A materialized view differs from a standard view in that upon its creation, the materialized view's stored query is executed, and the results it generates are saved in the database. In effect, this creates a new table. The view retains its stored query, so you can update the saved data by issuing a command to refresh the view. A good use for materialized views is to preprocess complex queries that take a while to run and make those results available for faster querying.

Let's drop the `nevada_counties_pop_2019` view and re-create it as a materialized view using the code in *Listing 17-5*.

```
1 DROP VIEW nevada_counties_pop_2019;
```

```
2 CREATE MATERIALIZED VIEW nevada_counties_pop_2019 AS
      SELECT county_name,
             state_fips,
             county_fips,
             pop_est_2019
      FROM us_counties_pop_est_2019
      WHERE state_name = 'Nevada';
```

Listing 17-5: Creating a materialized view

First, we use a `DROP VIEW` 1 statement to remove the `nevada_counties_pop_2019` view from the database. Then, we run `CREATE MATERIALIZED VIEW` 2 to make the view. Notice that the syntax is the same as the one for making a standard view, except for the added `MATERIALIZED` keyword and the omission of `OR REPLACE`, which is not available in the materialized view syntax. After running the statement, the database should respond with the message `SELECT 17`, telling you that the view's query produced 17 rows to be stored in the view. We can now query this data as with a standard view.

Let's say that the population estimates stored in `us_counties_pop_est_2019` are revised. To update the data stored in the materialized view, we can use the `REFRESH` keyword, as in Listing 17-6.

```
REFRESH MATERIALIZED VIEW nevada_counties_pop_2019;
```

Listing 17-6: Refreshing a materialized view

Executing this statement reruns the query stored in the `nevada_counties_pop_2019` view; the server will respond with the message `REFRESH MATERIALIZED VIEW`. The view will now reflect any updates to the data referenced by the view's query. When you have a query that takes some time to run, you can save time by storing its results in a materialized view that's refreshed periodically, letting users quickly access the stored data rather than run a lengthy query.

NOTE

Using `REFRESH MATERIALIZED VIEW CONCURRENTLY` will prevent locking out `SELECT` statements that are executed against the view during the refresh. See https://www.postgresql.org/docs/current/sql-refreshmaterializedview.html for details.

To delete a materialized view, we use a `DROP MATERIALIZED VIEW` statement. Also, note that materialized views appear in a different part of pgAdmin's object browser, under **Schemas▶public▶Materialized Views**.

Inserting, Updating, and Deleting Data Using a View

With nonmaterialized views, you can update or insert data in the underlying table being queried as long as the view meets certain conditions. One requirement is that the view must reference a single table or updatable view. If the view's query joins tables, as with the population change view we just built in the previous section, you can't perform inserts or updates to the original table directly. Also, the view's query can't contain `DISTINCT`, `WITH`, `GROUP BY`, or other clauses. (See a complete list of restrictions at https://www.postgresql.org/docs/current/sql-createview.html.)

You already know how to directly insert and update data on a table, so why do it through a view? One reason is that a view is one way you can exercise control over which data a user can update. Let's work through an example to see how this works.

Creating a View of Employees

In the Chapter 7 lesson on joins, we created and filled the `departments` and `employees` tables with four rows about people and where they work (if you skipped that section, you can revisit *Listing 7-1*). Running a quick `SELECT * FROM employees ORDER BY emp_id;` query shows the table's contents, as you can see here:

```
emp_id first_name last_name  salary    dept_id
------ ---------- --------- --------- -------
     1 Julia      Reyes     115300.00       1
     2 Janet      King       98000.00       1
     3 Arthur     Pappas     72700.00       2
     4 Michael    Taylor     89500.00       2
```

Let's say we want to use a view to give users in the Tax Department (its `dept_id` is `1`) the ability to add, remove, or update their employees' names without letting them change salary information or the data of employees in another department. To do this, we can set up a view using Listing 17-7.

```
CREATE OR REPLACE VIEW employees_tax_dept WITH
(security_barrier)1 AS
     SELECT emp_id,
            first_name,
            last_name,
            dept_id
     FROM employees
   2 WHERE dept_id = 1
   3 WITH LOCAL CHECK OPTION;
```

Listing 17-7: Creating a view on the `employees` table

This view is similar to others we've created so far, but with a few additions. First, in the `CREATE OR REPLACE VIEW` statement, we add the keywords `WITH (security_barrier)` 1. This enables a level of database security to prevent a malicious user from getting around restrictions that the view places on rows and columns. (See *https://www.postgresql.org/docs/current/rules-privileges.html* for how someone might subvert a view if you omit this type of security.)

In the view's `SELECT` query, we pick the columns we want to show from the `employees` table and use `WHERE` to filter the results on `dept_id = 1` 2 to list only Tax Department staff. The view itself will restrict updates or deletes to rows matching the condition in the `WHERE` clause. Adding the keywords `WITH LOCAL CHECK OPTION` 3 restricts inserts as well, allowing users to add new Tax Department employees only (if the view definition omitted those keywords, you could use it to insert a row with a `dept_id` of `3`, for example). The `LOCAL CHECK OPTION` also prevents a user from changing an employee's `dept_id` to a value other than `1`.

Create the `employees_tax_dept` view by running the code in *Listing 17-7*. Then run `SELECT * FROM employees_tax_dept ORDER BY emp_id;`, which should provide these two rows:

```
emp_id first_name last_name dept_id
------ ---------- --------- -------
     1 Julia      Reyes           1
     2 Janet      King            1
```

The result shows the employees who work in the Tax Department; they're two of the four rows in the entire `employees` table.

Now, let's look at how inserts and updates work via this view.

Inserting Rows Using the employees_tax_dept View

We can use a view to insert or update data, but instead of using the table name in the `INSERT` or `UPDATE` statement, we substitute the view name. After we add or change data using a view, the change is applied to the underlying table, which in this case is `employees`. The view then reflects the change via the query it runs.

Listing 17-8 shows two examples that attempt to add new employee records via the `employees_tax_dept` view. The first succeeds, but the second fails.

```
1 INSERT INTO employees_tax_dept (emp_id, first_name, last_name,
   dept_id)
   VALUES (5, 'Suzanne', 'Legere', 1);

2 INSERT INTO employees_tax_dept (emp_id, first_name, last_name,
   dept_id)
   VALUES (6, 'Jamil', 'White', 2);

3 SELECT * FROM employees_tax_dept ORDER BY emp_id;

4 SELECT * FROM employees ORDER BY emp_id;
```

Listing 17-8: Successful and rejected inserts via the `employees_tax_dept` *view*

In the first `INSERT` 1, which uses the insert syntax you learned in Chapter 2, we supply the first and last names of Suzanne Legere plus her `emp_id` and `dept_id`. Because the new row will satisfy the `LOCAL CHECK` in the view —it contains the same columns and `dept_id` is `1`—the insert succeeds when it executes.

But when we run the second `INSERT` 2 to add an employee named Jamil White using a `dept_id` of `2`, the operation fails with the error message `new row violates check option for view "employees_tax_dept"`. The reason is that when we created the view, we used a `WHERE` clause to return only rows with `dept_id = 1`. The `dept_id` of `2` doesn't pass the `LOCAL CHECK`, so it's prevented from being inserted.

Run the `SELECT` statement 3 on the view to check that Suzanne Legere was successfully added:

```
emp_id first_name last_name dept_id
------ ---------- --------- -------
     1 Julia      Reyes           1
     2 Janet      King            1
     5 Suzanne    Legere          1
```

We also query the `employees` table 4 to see that, in fact, Suzanne Legere was added to the full table. The view queries the `employees` table each time we access it.

```
emp_id first_name last_name  salary   dept_id
------ ---------- --------- --------- -------
     1 Julia      Reyes     115300.00       1
     2 Janet      King       98000.00       1
     3 Arthur     Pappas     72700.00       2
     4 Michael    Taylor     89500.00       2
     5 Suzanne    Legere                    1
```

As you can see from the addition of Suzanne Legere, the data we add using a view is also added to the underlying table. However, because the view doesn't include the `salary` column, the value in her row is `NULL`. If you attempt to insert a salary value using this view, you would receive the error message `column "salary" of relation "employees_tax_dept" does not exist`. The reason is that even though the `salary` column exists in the underlying `employees` table, it's not referenced in the view. Again, this is one way to limit access to sensitive data. Check the links I provided in the note in the section "Using Views to Simplify Queries" to learn more about granting permissions to users and adding `WITH (security_barrier)` if you plan to take on database administrator responsibilities.

Updating Rows Using the employees_tax_dept View

The same restrictions on accessing data in an underlying table apply when we update data using the `employees_tax_dept` view. *Listing 17-9* shows a standard query to change the spelling of Suzanne's last name using `UPDATE` (as a person with more than one uppercase letter in their last name, I can confirm such corrections aren't unusual).

```
UPDATE employees_tax_dept
SET last_name = 'Le Gere'
WHERE emp_id = 5;

SELECT * FROM employees_tax_dept ORDER BY emp_id;
```

Listing 17-9: Updating a row via the `employees_tax_dept` view

Run the code, and the result from the `SELECT` query should show the updated last name, which occurs in the underlying `employees` table:

```
emp_id first_name last_name dept_id
------ ---------- --------- -------
     1 Julia      Reyes           1
     2 Janet      King            1
     5 Suzanne    Le Gere         1
```

Suzanne's last name is now correctly spelled as Le Gere, not Legere.

However, if we try to update the name of an employee who's not in the Tax Department, the query fails just as it did when we tried to insert Jamil White in *Listing 17-8*. Trying to use this view to update the salary of an employee—even one in the Tax Department—will also fail. If the view doesn't reference a column in the underlying table, you can't access that column through the view. Again, the fact that updates on views are restricted in this way offers ways to secure and hide certain pieces of data.

Deleting Rows Using the employees_tax_dept View

Now, let's explore how to delete rows using a view. The restrictions on which data you can affect apply here as well. For example, if Suzanne Le Gere gets a better offer from another firm and decides to leave, you could remove her from `employees` through the `employees_tax_dept` view. Listing 17-10 shows the query in the standard `DELETE` syntax.

```
DELETE FROM employees_tax_dept
WHERE emp_id = 5;
```

Listing 17-10: Deleting a row via the `employees_tax_dept` *view*

Run the query, and PostgreSQL should respond with `DELETE 1`. However, when you try to delete a row for an employee in a department other than the Tax Department, PostgreSQL won't allow it and will report `DELETE 0`.

In summary, views not only give you control over access to data, but also give you shortcuts for working with data. Next, let's explore how to use functions to save keystrokes and time.

Creating Your Own Functions and Procedures

You've used functions throughout the book, such as to capitalize letters with `upper()` or add numbers with `sum()`. Behind these functions is a significant amount of (sometimes complex) programming that executes a series of actions and may, depending on the job of the function, return a response. We'll avoid complicated code here, but we'll build some basic functions that you can use as a launchpad for your own ideas. Even simple functions can help you avoid repeating code.

Much of the syntax in this section is specific to PostgreSQL, which supports both user-defined functions and *procedures* (the difference between the two is subtle, and I'll give examples of both). You can define functions and procedures using plain SQL, but you also can choose from other options. One is a PostgreSQL-specific *procedural language* called PL/pgSQL that adds features not found in standard SQL, such as logical control structures (`IF ... THEN ... ELSE`). Other options include PL/Python and PL/R for the Python and R programming languages.

Note that major database systems including Microsoft SQL Server, Oracle, and MySQL implement their own variations of functions and procedures. If you're using another database management system, this section will be useful for understanding concepts related to functions, but you'll need to check your database's documentation for specifics on its implementation of functions.

Creating the percent_change() Function

A function processes data and returns a value. As an example, let's write a function to simplify a staple of data analysis: calculating the percent change between two values. In Chapter 6, you learned that we express the percent change formula this way:

```
percent change = (New Number - Old Number) / Old Number
```

Rather than writing that formula each time we need it, we can create a function called `percent_change()` that takes the new and old numbers as inputs and returns the result rounded to a user-specified number of decimal places. Let's walk through the code in Listing 17-11 to see how to declare a simple function that uses SQL.

```
1 CREATE OR REPLACE FUNCTION
2 percent_change(new_value numeric,
                 old_value numeric,
                 decimal_places integer 3DEFAULT 1)
4 RETURNS numeric AS
5 'SELECT round(
          ((new_value - old_value) / old_value) * 100,
   decimal_places
   );'
6 LANGUAGE SQL
7 IMMUTABLE
8 RETURNS NULL ON NULL INPUT;
```

Listing 17-11: Creating a `percent_change()` function

A lot is happening in this code, but it's not as complicated as it looks. We start with the command `CREATE OR REPLACE FUNCTION` 1. As with the syntax to create a view, the `OR REPLACE` keywords are optional. We then give the name of the function 2 and, in parentheses, a list of *arguments* that determine the function's inputs. Each argument will serve as an input to the function and gets a name and data type. For example, `new_value` and `old_value` are `numeric` and require that the user of the function supply input values matching that type, whereas `decimal_places` (which specifies the number of places to round results) is `integer`. For `decimal_places`, we specify `1` as the `DEFAULT` 3 value—this makes the argument optional and, if it's omitted by the user, will set the argument to `1` by default.

We then use the keywords `RETURNS numeric AS` 4 to tell the function to return its calculation as type `numeric`. If this were a function to concatenate strings, we might return `text`.

Next, we write the meat of the function that performs the calculation. Inside single quotes, we place a `SELECT` query 5 that includes the percent change calculation nested inside a `round()` function. In the formula, we use the function's argument names instead of numbers.

We then supply a series of keywords that define the function's attributes and behavior. The `LANGUAGE` 6 keyword specifies that we've written this function using plain SQL as opposed to one of other languages PostgreSQL supports for creating functions. Next, the `IMMUTABLE` keyword 7 indicates that the function cannot modify the database and will always return the same result for a given set of arguments. The line `RETURNS NULL ON NULL INPUT` 8 guarantees that the function will supply a `NULL` response if any input that is not supplied by default is a `NULL`.

Run the code using pgAdmin to create the `percent_change()` function. The server should respond with the message `CREATE FUNCTION`.

Using the percent_change() Function

To test the new `percent_change()` function, run it by itself using `SELECT`, as shown in *Listing 17-12*.

```
SELECT percent_change(110, 108, 2);
```

Listing 17-12: Testing the `percent_change()` function

This example uses a value of `110` for the new number, `108` for the old number, and `2` as the desired number of decimal places to round the result.

Run the code; the result should look like this:

```
 percent_change
----------------
           1.85
```

The result tells us there's a 1.85 percent increase between 108 and 110. Experiment with other numbers to see how the results change. Also, try changing the `decimal_places` argument to values including `0`, or omit it, to see how that affects the output. You should see results that have more or fewer numbers after the decimal point, based on your input.

We created this function to avoid writing the full percent change formula in queries. Let's use it to calculate percent change using a version of the census estimates population change query we wrote in Chapter 7, as shown in Listing 17-13.

```
SELECT c2019.county_name,
       c2019.state_name,
       c2019.pop_est_2019 AS pop_2019,
     1 percent_change(c2019.pop_est_2019,
                      c2010.estimates_base_2010) AS
pct_chg_func,
     2 round( (c2019.pop_est_2019::numeric -
c2010.estimates_base_2010)
           / c2010.estimates_base_2010 * 100, 1 ) AS
pct_change_formula
FROM us_counties_pop_est_2019 AS c2019
    JOIN us_counties_pop_est_2010 AS c2010
ON c2019.state_fips = c2010.state_fips
   AND c2019.county_fips = c2010.county_fips
ORDER BY pct_chg_func DESC
LIMIT 5;
```

Listing 17-13: Testing `percent_change()` on census data

Listing 17-13 modifies the original query from Chapter 7 to add the `percent_change()` function 1 as a column in `SELECT`. We also include the explicit percent change formula 2 so we can compare results. As inputs, we use the 2019 population estimate column (`c2019.pop_est_2019`) as the new number and the 2010 estimates base as the old (`c2010.estimates_base_2010`).

The query results should display the five counties with the greatest percent change in population, and the results from the function should match the results from the formula entered directly into the query. Note that each value in the `pct_chg_func` column has one decimal place, the function's default value, because we didn't provide the optional third argument. Here's the result with both the function and the formula:

```
  county_name    state_name   pop_2019 pct_chg_func
pct_chg_formula
--------------- ------------ -------- ------------ ----------
-----
McKenzie County North Dakota    15024        136.3
136.3
Loving County   Texas             169        106.1
106.1
Williams County North Dakota    37589         67.8
67.8
Hays County     Texas          230191         46.5
46.5
Wasatch County  Utah            34091         44.9
44.9
```

Now that we know the function works as intended, we can use `percent_change()` any time we need to solve that calculation—and that's much faster than writing out the formula!

Updating Data with a Procedure

As implemented in PostgreSQL, a *procedure* is a close relative of a function, albeit with some significant differences. Both procedures and functions can perform data operations that don't return a value, such as an update. Procedures, on the other hand, don't have a clause to return a value, while functions do. Also, procedures can incorporate the transaction commands we covered in Chapter 10 such as `COMMIT` and `ROLLBACK`, and functions cannot. Many database managers implement procedures, which are sometimes referred to as *stored procedures*. PostgreSQL added procedures as of version 11 and are part of the SQL standard, though PostgreSQL syntax is not fully compatible.

We can simplify routine updates to data using procedures. In this section, we'll write a procedure that updates a record of the correct number of personal days off a teacher gets (in addition to vacation days) based on the time elapsed since their hire date.

For this exercise, we'll return to the `teachers` table from the first lesson in Chapter 2. If you skipped "Creating a Table" in that chapter, create the `teachers` table and insert the data now using the example code in Listings 2-2 and 2-3.

Let's add a column to `teachers` to hold the teachers' personal days using the code in Listing 17-14. The new column will be empty until we fill it later using a procedure.

```
ALTER TABLE teachers ADD COLUMN personal_days integer;

SELECT first_name,
       last_name,
       hire_date,
       personal_days
FROM teachers;
```

Listing 17-14: Adding a column to the `teachers` table and seeing the data

Listing 17-14 updates the teachers table using `ALTER` and adds the `personal_days` column using the keywords `ADD COLUMN`. We then run the `SELECT` statement to view the data, in which we also include the names and hire dates of each teacher. When both queries finish, you should see the following six rows:

```
first_name last_name hire_date  personal_days
---------- --------- ---------- -------------
Janet      Smith     2011-10-30
Lee        Reynolds  1993-05-22
Samuel     Cole      2005-08-01
Samantha   Bush      2011-10-30
```

```
Betty       Diaz       2005-08-30
Kathleen    Roush      2010-10-22
```

The `personal_days` column contains only `NULL` values because we haven't inserted anything yet.

Now, let's create a procedure called `update_personal_days()` that populates the `personal_days` column with their earned personal days (in addition to vacation days). We'll use the following criteria:

Less than 10 years since hire: 3 personal days

10 to less than 15 years since hire: 4 personal days

15 to less than 20 years since hire: 5 personal days

20 years to less than 25 years since hire: 6 personal days

25 years or more since hire: 7 personal days

The code in *Listing 17-15* creates a procedure. This time, instead of using plain SQL, we'll incorporate elements of the PL/pgSQL procedural language, which is an additional language PostgreSQL supports for writing functions. Let's walk through some differences.

```
CREATE OR REPLACE PROCEDURE update_personal_days()
AS 1$$
2 BEGIN
     UPDATE teachers
     SET personal_days =
       3 CASE WHEN (now() - hire_date) >= '10 years'::interval
                   AND (now() - hire_date) < '15
years'::interval THEN 4
              WHEN (now() - hire_date) >= '15 years'::interval
                   AND (now() - hire_date) < '20
years'::interval THEN 5
              WHEN (now() - hire_date) >= '20 years'::interval
                   AND (now() - hire_date) < '25
years'::interval THEN 6
              WHEN (now() - hire_date) >= '25 years'::interval
THEN 7
              ELSE 3
         END;
   4 RAISE NOTICE 'personal_days updated!';
END;
```

```
5 $$
6 LANGUAGE plpgsql;
```

Listing 17-15: Creating an `update_personal_days()` function

We begin with `CREATE OR REPLACE PROCEDURE` and give the procedure a name. This time, we provide no arguments because no user input is required—the procedure operates on predetermined columns with set values for calculating intervals.

Often, when writing PL/pgSQL-based functions, the PostgreSQL convention is to use the non-ANSI SQL standard dollar-quote (`$$`) to mark the start 1 and end 5 of the string that contains all the function's commands. (As with the `percent_change()` SQL function earlier, you could use single quote marks to enclose the string, but then any single quotes in the string would need to be doubled, and that not only looks messy but can be confusing.) So, everything between the pair of `$$` is the code that does the work. You can also add some text between the dollar signs, like `$namestring$`, to create a unique pair of beginning and ending quotes. This is useful, for example, if you need to quote a query inside the function.

Right after the first `$$` we start a `BEGIN ... END;` 2 block. This is a PL/pgSQL convention that delineates the start and end of a section of code within a function or procedure; as with dollar quotes, it is possible to nest one `BEGIN ... END;` inside another to facilitate logical groupings of code. Inside that block, we place an `UPDATE` statement that uses a `CASE` statement 3 to determine the number of days each teacher gets. We subtract the `hire_date` from the current date, which is retrieved from the server by the `now()` function. Depending on which range `now() - hire_date` falls into, the `CASE` statement returns the number of personal days corresponding to the range. We use the PL/pgSQL keywords `RAISE NOTICE` 4 to display a message that the procedure is done. Finally, we use the `LANGUAGE` keyword 6 so the database knows to interpret what we've written according to the syntax specific to PL/pgSQL.

Run the code in Listing 17-15 to create the `update_personal_days()` procedure. To invoke the procedure, we use the `CALL` command, which is part of the ANSI SQL standard:

```
CALL update_personal_days();
```

When the procedure runs, the server responds with the notice it raises, which is `personal_days updated!`.

When you rerun the `SELECT` statement in Listing 17-14, you should see that each row of the `personal_days` column is filled with the appropriate values. Note that results will vary depending on when you run this function, because calculations using `now()` change as time passes.

```
first_name last_name hire_date  personal_days
---------- --------- ---------- -------------
Janet      Smith     2011-10-30             3
Lee        Reynolds  1993-05-22             7
Samuel     Cole      2005-08-01             5
Samantha   Bush      2011-10-30             3
Betty      Diaz      2005-08-30             5
Kathleen   Roush     2010-10-22             4
```

You could use the `update_personal_days()` function to regularly update data manually after performing certain tasks, or you could use a task scheduler such as pgAgent (a separate open source tool) to run it automatically. You can learn about pgAgent and other tools in "PostgreSQL Utilities, Tools, and Extensions" in the appendix.

Using the Python Language in a Function

Previously, I mentioned that PL/pgSQL is the default procedural language within PostgreSQL, but the database also supports creating functions using open source languages, such as Python and R. This support allows you to take advantage of features and modules from those languages within functions you create. For example, with Python, you can use the `pandas` library for analysis. The documentation at https://www.postgresql.org/docs/current/server-programming.html provides a comprehensive review of the languages included with PostgreSQL, but here I'll show you a simple function using Python.

To enable PL/Python, you must create the extension using the code in Listing 17-16.

```
CREATE EXTENSION plpython3u;
```

Listing 17-16: Enabling the PL/Python procedural language

If you get an error, such as `image not found`, that means the PL/Python extension is not installed on your system. Depending on the operating system, installation of PL/Python typically requires installation of Python and additional configuration beyond the basic PostgreSQL install. For this, refer to the installation instructions for your operating system in Chapter 1.

After enabling the extension, we can create a function using syntax similar to the examples you've tried so far, but using Python for the body of the function. Listing 17-17 shows how to use PL/Python to create a function called `trim_county()` that removes the word *County* from the end of a string. We'll use this function to clean up names of counties in the census data.

```
  CREATE OR REPLACE FUNCTION trim_county(input_string text)
1 RETURNS text AS $$
      import re2
    3 cleaned = re.sub(r' County', '', input_string)
      return cleaned
  $$
4 LANGUAGE plpython3u;
```

Listing 17-17: Using PL/Python to create the `trim_county()` function

The structure should look familiar. After naming the function and its text input, we use the `RETURNS` keyword 1 to specify that the function will send text back. After the opening `$$` quotes, we get straight to the Python code, starting with a statement to import the Python regular expressions module, `re` 2. Even if you don't know much about Python, you can probably deduce that the next two lines of code 3 set a variable called `cleaned` to the results of a Python regular expression function called `sub()`. That function looks for a space followed by the word *County* in the `input_string` passed into the function and substitutes an empty string, which is denoted by two apostrophes. Then the function returns the content of the variable `cleaned`.

To end, we specify `LANGUAGE plpython3u` 4 to note we're writing the function with PL/Python.

Run the code to create the function, and then execute the `SELECT` statement in *Listing 17-18* to see it in action.

```
SELECT county_name,
       trim_county(county_name)
FROM us_counties_pop_est_2019
ORDER BY state_fips, county_fips
LIMIT 5;
```

Listing 17-18: Testing the `trim_county()` function

We use the `county_name` column in the `us_counties_pop_est_2019` table as input to `trim_county()`. That should return these results:

```
  county_name      trim_county
----------------   -------------
 Autauga County    Autauga
 Baldwin County    Baldwin
 Barbour County    Barbour
 Bibb County       Bibb
 Blount County     Blount
```

As you can see, the `trim_county()` function evaluated each value in the `county_name` column and removed a space and the word *County* when present. Although this is a trivial example, it shows how easy it is to use Python—or one of the other supported procedural languages—inside a function.

Next, you'll learn how to use triggers to automate your database.

Automating Database Actions with Triggers

A database *trigger* executes a function whenever a specified event, such as an `INSERT`, `UPDATE`, or `DELETE`, occurs on a table or a view. You can set a trigger to fire before, after, or instead of the event, and you can also set it to fire once for each row affected by the event or just once per operation. For example, let’s say you delete 20 rows from a table. You could set the trigger to fire once for each of the 20 rows deleted or just one time.

We’ll work through two examples. The first example keeps a log of changes made to grades at a school. The second automatically classifies temperatures each time we collect a reading.

Logging Grade Updates to a Table

Let’s say we want to automatically track changes made to a student `grades` table in our school’s database. Every time a row is updated, we want to record the old and new grade plus the time the change occurred (search online for *David Lightman and grades* and you’ll see why this might be worth tracking). To handle this task automatically, we’ll need three items:

A `grades_history` table to record the changes to grades in a `grades` table

A trigger to run a function every time a change occurs in the `grades` table, which we’ll name `grades_update`

The function the trigger will execute, which we’ll call `record_if_grade_changed()`

Creating Tables to Track Grades and Updates

Let’s start by making the tables we need. *Listing 17-19* includes the code to first create and fill `grades` and then create `grades_history`.

```
1 CREATE TABLE grades (
      student_id bigint,
      course_id bigint,
      course text NOT NULL,
      grade text NOT NULL,
  PRIMARY KEY (student_id, course_id)
  );

2 INSERT INTO grades
  VALUES
      (1, 1, 'Biology 2', 'F'),
      (1, 2, 'English 11B', 'D'),
      (1, 3, 'World History 11B', 'C'),
      (1, 4, 'Trig 2', 'B');
```

```
3 CREATE TABLE grades_history (
      student_id bigint NOT NULL,
      course_id bigint NOT NULL,
      change_time timestamp with time zone NOT NULL,
      course text NOT NULL,
      old_grade text NOT NULL,
      new_grade text NOT NULL,
  PRIMARY KEY (student_id, course_id, change_time)
  );
```

Listing 17-19: Creating the `grades` *and* `grades_history` *tables*

These commands are straightforward. We use `CREATE` to make a `grades` table 1 and add four rows using `INSERT` 2, where each row represents a student's grade in a class. Then we use `CREATE TABLE` to make the `grades_history` table 3 to hold the data we log each time an existing grade is altered. The `grades_history` table has columns for the new grade, old grade, and the time of the change. Run the code to create the tables and fill the `grades` table. We insert no data into `grades_history` here because the trigger process will handle that task.

Creating the Function and Trigger

Next, let's write the `record_if_grade_changed()` function that the trigger will execute (note that the PostgreSQL documentation refers to such functions as *trigger procedures*). We must write the function before naming it in the trigger. Let's go through the code in Listing 17-20.

```
CREATE OR REPLACE FUNCTION record_if_grade_changed()
  1 RETURNS trigger AS
$$
BEGIN
  2 IF NEW.grade <> OLD.grade THEN
    INSERT INTO grades_history (
        student_id,
        course_id,
        change_time,
        course,
        old_grade,
        new_grade)
    VALUES
        (OLD.student_id,
          OLD.course_id,
          now(),
          OLD.course,
        3 OLD.grade,
        4 NEW.grade);
    END IF;
  5 RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Listing 17-20: Creating the `record_if_grade_changed()` function

The `record_if_grade_changed()` function follows the pattern of earlier examples but with differences specific to working with triggers. First, we specify `RETURNS trigger` 1 instead of a data type. We use dollar-quotes to delineate the code portion of the function, and because `record_if_grade_changed()` is a PL/pgSQL function, we also place the code to execute inside a `BEGIN ... END;` block. Next, we start the procedure using an `IF ... THEN` statement 2, which is one of the control structures PL/pgSQL provides. We use it here to run the `INSERT` statement only if the updated grade is different from the old grade, which we check using the <> operator.

When a change occurs to the `grades` table, the trigger (which we'll create next) will execute. For each row that's changed, the trigger will pass two collections of data into `record_if_grade_changed()`. The first is the row values *before* they were changed, noted with the prefix `OLD`. The second is the row values *after* they were changed, noted with the prefix `NEW`. The function can access the original row values and the updated row values, which it will use for a comparison. If the `IF ... THEN` statement evaluates as `true`, indicating that the old and new `grade` values are different, we use `INSERT` to add a row to `grades_history` that contains both `OLD.grade` 3 and `NEW.grade` 4. Finally, we include a `RETURN` statement 5 with a value of `NULL`; the trigger procedure performs a database `INSERT`, so we do not need a value returned.

NOTE

Sometimes a `RETURN` *value is ignored; the PostgreSQL documentation at https://www.postgresql.org/docs/current/plpgsql-trigger.html details the scenarios in which this occurs.*

Run the code in [Listing 17-20](#) to create the function. Then, add the `grades_update` trigger to the `grades` table using [Listing 17-21](#).

```
1 CREATE TRIGGER grades_update
  2 AFTER UPDATE
    ON grades
  3 FOR EACH ROW
  4 EXECUTE PROCEDURE record_if_grade_changed();
```

Listing 17-21: Creating the `grades_update` *trigger*

In PostgreSQL, the syntax for creating a trigger follows the ANSI SQL standard (although not all aspects of the standard are supported, per the documentation at https://www.postgresql.org/docs/current/sql-createtrigger.html). The code begins with a `CREATE TRIGGER` 1 statement, followed by clauses that control when the trigger runs and how it behaves. We use `AFTER UPDATE` 2 to specify that we want the trigger to fire after the update occurs on the `grades` row. We could also use the `BEFORE` or `INSTEAD OF` keywords depending on the need.

We write `FOR EACH ROW` 3 to tell the trigger to execute the procedure once for each row updated in the table. For example, if someone runs an update that affects three rows, the procedure will run three times. The alternate (and default) is `FOR EACH STATEMENT`, which runs the procedure once. If we didn't care about capturing changes to each row and simply wanted to record that grades were changed at a certain time, we could use that option. Finally, we use `EXECUTE PROCEDURE` 4 to name `record_if_grade_changed()` as the function the trigger should run.

Create the trigger by running the code in [Listing 17-21](#) in pgAdmin. The database should respond with the message `CREATE TRIGGER`.

Testing the Trigger

Now that we've created the trigger and the function, it should run when data in the `grades` table changes; let's see what the process does. First, let's check the current status of our data. When you run `SELECT * FROM grades_history;`, you'll see that the table is empty because we haven't made any changes to the `grades` table yet and there's nothing to track. Next, when you run `SELECT * FROM grades ORDER BY student_id, course_id;`, you should see the grade data that you inserted in *Listing 17-19*, as shown here:

```
student_id course_id      course       grade
---------- --------- ----------------- -----
         1         1 Biology 2         F
         1         2 English 11B       D
         1         3 World History 11B C
         1         4 Trig 2            B
```

That Biology 2 grade doesn't look very good. Let's update it using the code in *Listing 17-22*.

```
UPDATE grades
SET grade = 'C'
WHERE student_id = 1 AND course_id = 1;
```

Listing 17-22: Testing the `grades_update` trigger

When you run the `UPDATE`, pgAdmin doesn't display anything to let you know that the trigger executed in the background. It just reports `UPDATE 1`, meaning a row was updated. But our trigger did run, which we can confirm by examining columns in `grades_history` using this `SELECT` query:

```
SELECT student_id,
       change_time,
       course,
       old_grade,
       new_grade
FROM grades_history;
```

When you run this query, you should see that the `grades_history` table, which contains all changes to grades, now has one row:

```
student_id          change_time           course
old_grade new_grade
---------- ----------------------------- ---------    -------
-- ---------
         1 2023-09-01 15:50:43.291164-04 Biology 2    F
C
```

This row displays the old Biology 2 grade of `F`, the new value `C`, and `change_time`, showing the time of update (your result should reflect your date and time). Note that the addition of this row to `grades_history` happened in the background without the knowledge of the person making the update. But the `UPDATE` event on the table caused the trigger to fire, which executed the `record_if_grade_changed()` function.

If you've ever used a content management system, such as WordPress or Drupal, this sort of revision tracking might be familiar. It provides a helpful record of changes made to content for reference, auditing, and, unfortunately, occasional finger-pointing. Regardless, the ability to trigger actions on a database automatically gives you more control over your data.

Automatically Classifying Temperatures

In Chapter 13, we used the SQL `CASE` statement to reclassify temperature readings into descriptive categories. The `CASE` statement is also part of the PL/pgSQL procedural language, and we can use its capability to assign values to variables to automatically store those category names in a table each time we add a temperature reading. If we're routinely collecting temperature readings, using this technique to automate the classification spares us from having to handle the task manually.

We'll follow the same steps we used for logging the grade changes: we first create a function to classify the temperatures and then create a trigger to run the function each time the table is updated. Use *Listing 17-23* to create a `temperature_test` table for the exercise.

```
CREATE TABLE temperature_test (
    station_name text,
    observation_date date,
    max_temp integer,
    min_temp integer,
    max_temp_group text,
PRIMARY KEY (station_name, observation_date)
);
```

Listing 17-23: Creating a `temperature_test` *table*

The `temperature_test` table contains columns to hold the name of the station and date of the temperature observation. Let's imagine that we have some process to insert a row once a day that provides the maximum and minimum temperature for that location, and we need to fill the `max_temp_group` column with a descriptive classification of the day's high reading to provide text to a weather forecast we're distributing.

To do this, we first make a function called `classify_max_temp()`, as shown in Listing 17-24.

```
CREATE OR REPLACE FUNCTION classify_max_temp()
    RETURNS trigger AS
$$
BEGIN
  1 CASE
       WHEN NEW.max_temp >= 90 THEN
           NEW.max_temp_group := 'Hot';
       WHEN NEW.max_temp >= 70 AND NEW.max_temp < 90 THEN
           NEW.max_temp_group := 'Warm';
       WHEN NEW.max_temp >= 50 AND NEW.max_temp < 70 THEN
           NEW.max_temp_group := 'Pleasant';
       WHEN NEW.max_temp >= 33 AND NEW.max_temp < 50 THEN
           NEW.max_temp_group := 'Cold';
       WHEN NEW.max_temp >= 20 AND NEW.max_temp < 33 THEN
           NEW.max_temp_group := 'Frigid';
       WHEN NEW.max_temp < 20 THEN
           NEW.max_temp_group := 'Inhumane';
       ELSE NEW.max_temp_group := 'No reading';
    END CASE;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Listing 17-24: Creating the `classify_max_temp()` *function*

By now, these functions should look familiar. What's new here is the PL/pgSQL version of the `CASE` syntax 1, which differs slightly from the SQL syntax. The PL/pgSQL syntax includes a semicolon after each `WHEN`

`. . . THEN` clause. Also new is the *assignment operator* `:=`, which we use to assign the descriptive name to the `NEW.max_temp_group` column based on the outcome of the `CASE` function. For example, the statement `NEW.max_temp_group := 'Cold'` assigns the string `'Cold'` to `NEW.max_temp_group` when the temperature value is greater than or equal to 33 degrees but less than 50 degrees Fahrenheit. When the function returns the `NEW` row to be inserted in the table, it will include the string value `Cold`. Run the code to create the function.

Next, using the code in Listing 17-25, create a trigger to execute the function each time a row is added to `temperature_test`.

```
CREATE TRIGGER temperature_insert
  1 BEFORE INSERT
    ON temperature_test
  2 FOR EACH ROW
  3 EXECUTE PROCEDURE classify_max_temp();
```

Listing 17-25: Creating the `temperature_insert` trigger

In this example, we classify `max_temp` and create a value for `max_temp_group` prior to inserting the row into the table. Doing so is more efficient than performing a separate update after the row is inserted. To specify that behavior, we set the `temperature_insert` trigger to fire `BEFORE INSERT` 1.

We also want the trigger to fire `FOR EACH ROW` 2 because we want each `max_temp` recorded in the table to get a descriptive classification. The final `EXECUTE PROCEDURE` statement names the `classify_max_temp()` function 3 we just created. Run the `CREATE TRIGGER` statement in pgAdmin, and then test the setup using Listing 17-26.

```
INSERT INTO temperature_test
VALUES
    ('North Station', '1/19/2023', 10, -3),
    ('North Station', '3/20/2023', 28, 19),
    ('North Station', '5/2/2023', 65, 42),
    ('North Station', '8/9/2023', 93, 74),
    ('North Station', '12/14/2023', NULL, NULL);
```

```
SELECT * FROM temperature_test ORDER BY observation_date;
```

Listing 17-26: Inserting rows to test the `temperature_insert` trigger

Here we insert five rows into `temperature_test`, and we expect the `temperature_insert` trigger to fire for each row—and it does! The `SELECT` statement in the listing should display these results:

```
station_name  observation_date max_temp min_temp
max_temp_group
------------- ---------------- -------- -------- ------------
--
North Station 2023-01-19             10       -3 Inhumane
North Station 2023-03-20             28       19 Frigid
North Station 2023-05-02             65       42 Pleasant
North Station 2023-08-09             93       74 Hot
North Station 2023-12-14                         No reading
```

Thanks to the trigger and function, each `max_temp` inserted automatically receives the appropriate classification in the `max_temp_group` column—including the instance where we had no reading for that value. Note that the trigger's update of the column will override any user-supplied values during insert.

This temperature example and the earlier grade-change auditing example are rudimentary, but they give you a glimpse of how useful triggers and functions can be in simplifying data maintenance.

Wrapping Up

Although the techniques you learned in this chapter begin to merge with those of a database administrator, you can apply the concepts to reduce the amount of time you spend repeating certain tasks. I hope these approaches will help you free up more time to find interesting stories in your data.

This chapter concludes our discussion of analysis techniques and the SQL language. The next two chapters offer workflow tips to help you increase your command of PostgreSQL. They include how to connect to a database and run queries from your computer’s command line and how to maintain your database.

TRY IT YOURSELF

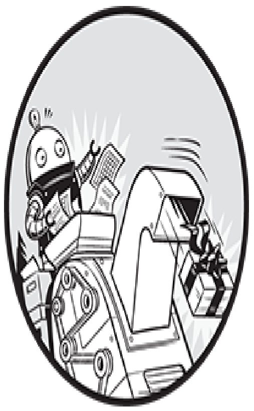
Review the concepts in the chapter with these exercises:

Create a materialized view that displays the number of New York City taxi trips per hour. Use the taxi data from Chapter 12 and the query in *Listing 12-8*. How do you refresh the view if you need to?

In Chapter 11, you learned how to calculate a rate per thousand. Turn that formula into a `rate_per_thousand()` function that takes three arguments to calculate the result: `observed_number`, `base_number`, and `decimal_places`.

In Chapter 10, you worked with the `meat_poultry_egg_establishments` table that listed food processing facilities. Write a trigger that automatically adds an inspection deadline timestamp six months in the future whenever you insert a new facility into the table. Use the `inspection_deadline` column added in *Listing 10-19*. You should be able to describe the steps needed to implement a trigger and how the steps relate to each other.

18
USING POSTGRESQL FROM THE COMMAND LINE



In this chapter, you’ll learn how to work with PostgreSQL from the *command line*, a text-based interface where you enter names of programs or other commands to perform tasks, such as editing files or listing the contents of a file directory.

The command line—also called a *command line interface*, a *console*, a *shell*, or the *terminal*—existed long before computers had a graphical user interface (GUI) with menus, icons, and buttons you could click for navigation. Back in college, to edit a file, I had to enter commands into a terminal connected to an IBM mainframe computer. Working that way felt mysterious, as though I’d attained new powers—and I had! Today, even in a GUI world, familiarity with the command line is essential for a programmer moving toward expert-level skills. Perhaps that’s why when a movie wants to convey that a character really knows what they’re doing on a computer, they’re shown typing cryptic, text-only commands.

As we learn about this text-only world, pay attention to these advantages of mastering working from the command line instead of a GUI, such as pgAdmin:

You can often work faster by entering short commands instead of clicking through layers of menu items.

You gain access to functions that only the command line provides.

If command line access is all you have to work with (for example, when you've connected to a remote computer), you can still get work done.

We'll use `psql`, a command-line tool included with PostgreSQL that lets you run queries, manage database objects, and interact with the computer's operating system via text command. You'll learn how to set up and access your computer's command line and then launch `psql`. Along the way, we'll cover general command line syntax and additional commands for database tasks. Patience is key: even experienced experts often resort to documentation to recall the available command line options.

Setting Up the Command Line for psql

To start, we'll access the command line on our operating system and, if needed, set an environment variable called `PATH` that tells our system where to find `psql`. *Environment variables* hold parameters that specify system or application configurations, such as where to store temporary files; they also allow you to enable or disable options. The `PATH` environment variable stores the names of one or more directories containing executable programs, and in this instance will tell the command line interface the location of `psql`, avoiding the hassle of having to enter its full directory path each time you launch it.

Windows psql Setup

On Windows, you'll run `psql` within the *Command Prompt*, the application that provides that system's command line interface. Before we do that, we need to tell Command Prompt where to find *psql.exe*—the full name of the `psql` application on Windows.

Adding psql and Utilities to the Windows PATH

The following steps assume that you installed PostgreSQL according to the instructions described in "Windows Installation" in Chapter 1. (If you installed PostgreSQL another way, use the Windows File Explorer to search your C: drive to find the directory that holds *psql.exe*, and then replace

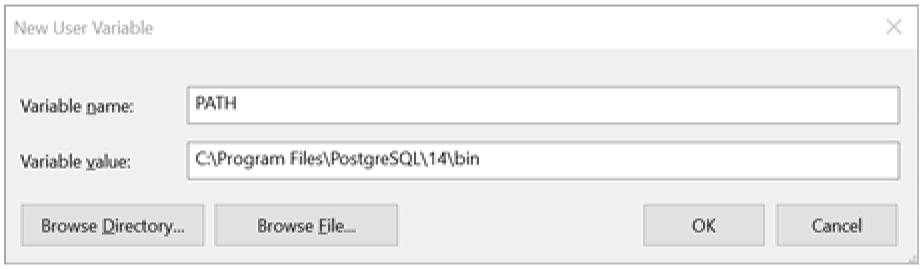
*C:\Program Files\PostgreSQL\*x*\bin* in the following steps with your own path.)

Open the Windows Control Panel by clicking the **Search** icon on the Windows taskbar, entering **Control Panel**, and then clicking the **Control Panel** icon.

Inside the Control Panel app, enter **Environment** in the search box. In the list of search results displayed, click **Edit the System Environment Variables**. A System Properties dialog should appear.

In the System Properties dialog, on the Advanced tab, click **Environment Variables**. The dialog that opens should have two sections: User variables and System variables. In the User variables section, if you don't see a `PATH` variable, continue to step a to create a new one. If you do see an existing `PATH` variable, continue to step b to modify it.

If you don't see `PATH` in the User variables section, click **New** to open a New User Variable dialog, shown in *Figure 18-1*.



Variable name: PATH
Variable value: C:\Program Files\PostgreSQL\14\bin

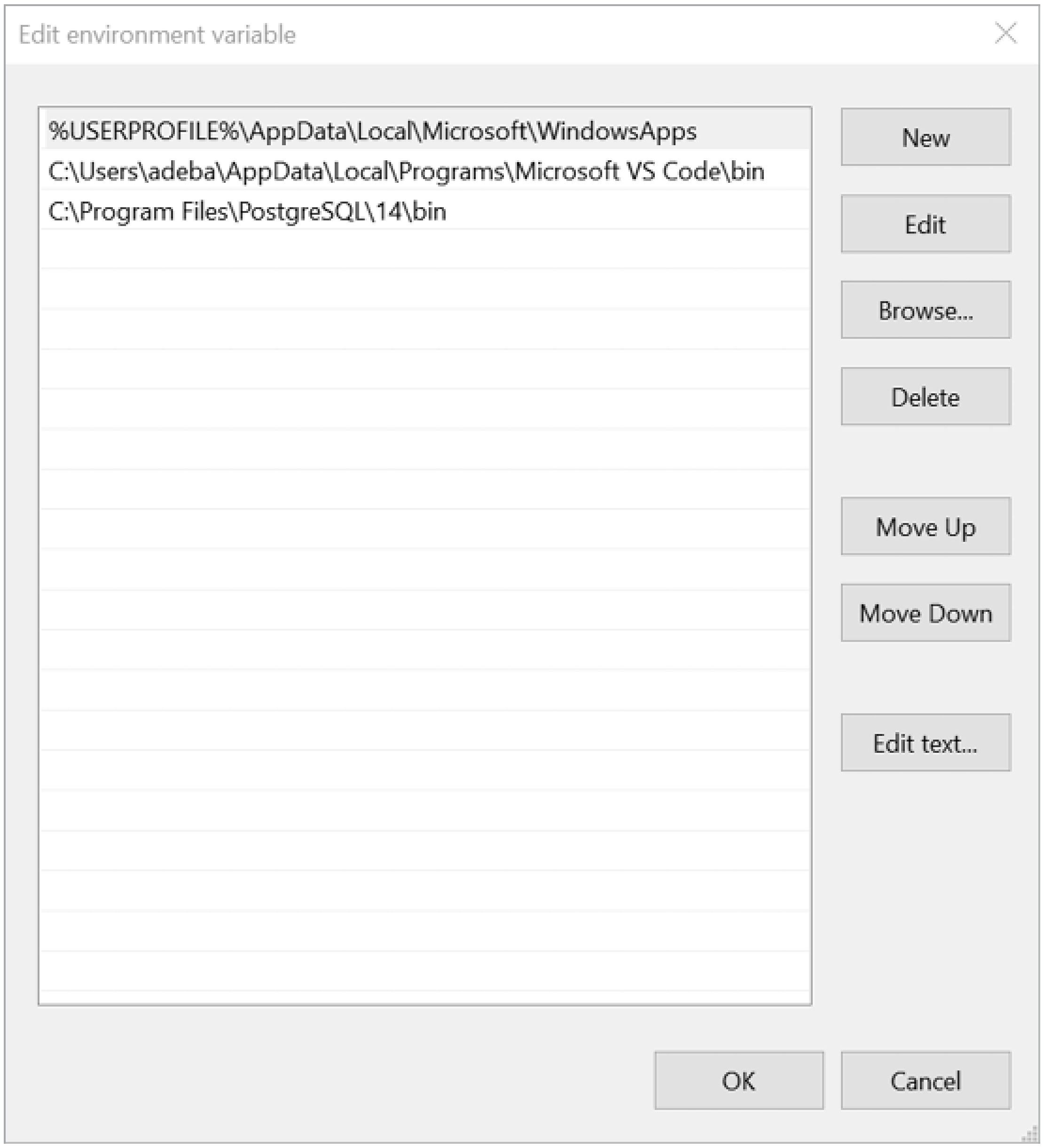
Figure 18-1: Creating a new `PATH` environment variable in Windows 10

In the Variable name box, enter **PATH**. In the Variable value box, enter **C:\Program Files\PostgreSQL\*x*\bin**, where *x* is the version of PostgreSQL you're using. (Instead of typing, you can click **Browse Directory** and navigate to the directory in the Browse For Folder dialog.) When you've either entered the path manually or browsed to it, click **OK** in all dialogs to close them.

If you do see an existing `PATH` variable in the User variables section, highlight it and click **Edit**. In the list of variables that displays, click **New** and enter **C:\Program Files\PostgreSQL\*x*\bin**, where *x* is the version of PostgreSQL you're using. (Instead of typing, you can click **Browse Directory** and navigate to the directory in the Browse For Folder dialog.)

The result should look like the highlighted line in *Figure 18-2*. When you're finished, click **OK** in all dialogs to close them.

Now when you launch Command Prompt, the `PATH` should include the directory. Note that any time you make changes to the `PATH`, you must close and reopen Command Prompt for the changes to take effect. Next, let's set up Command Prompt.



```
%USERPROFILE%\AppData\Local\Microsoft\WindowsApps
C:\Users\adeba\AppData\Local\Programs\Microsoft VS Code\bin
C:\Program Files\PostgreSQL\14\bin
```

Figure 18-2: Editing existing `PATH` environment variables in Windows 10

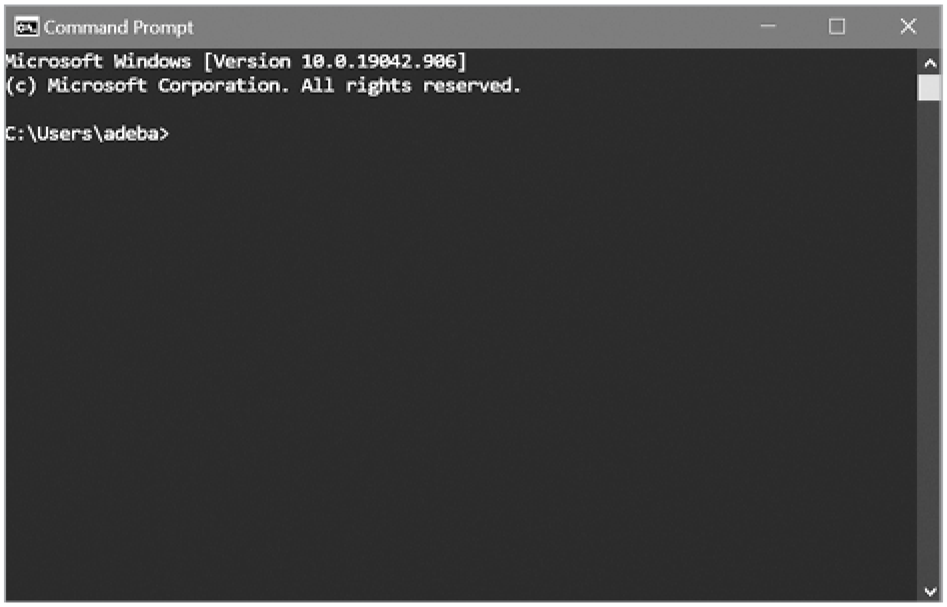
Launching and Configuring Windows Command Prompt

Command Prompt is an executable file named *cmd.exe*. To launch it, select **Start▶Windows System▶Command Prompt** or enter **cmd** into the search bar. When the application opens, you should see a window with a black background that displays version and copyright information along with a prompt showing your current directory. On my Windows 10 system, Command Prompt opens to my default user directory and displays `C:\Users\adeba>`, as shown in *Figure 18-3*.

This line is known as the *prompt*, and it shows the current working directory. For me, this is my C: drive, which is typically the main hard drive on a Windows system, and the `\Users\adeba` directory on that drive. The greater-than sign `>` indicates the area where you enter your commands.

NOTE

*For fast access to Command Prompt, you can add it to your Windows taskbar. When Command Prompt is running, right-click its icon on the taskbar and then click **Pin to taskbar**.*



```
Microsoft Windows [Version 10.0.19042.906]
(c) Microsoft Corporation. All rights reserved.

C:\Users\adeba>
```

Figure 18-3: My Command Prompt in Windows 10

You can customize the font and colors plus access other settings by clicking the Command Prompt icon at the left of its window bar and selecting **Properties** from the menu. To make Command Prompt more suited for query output, I recommend setting the window size (on the Layout tab) to a width of at least 80 and a height of 25. For the font, the official PostgreSQL documentation recommends using Lucida Console to properly display all the characters.

Entering Instructions on Windows Command Prompt

Now you're ready to enter instructions in Command Prompt. Enter **help** at the prompt, and press ENTER on your keyboard to see a list of available Windows system commands. You can view information about a command by including its name after `help`. For example, enter **`help time`** to display information about using the `time` command to set or view the system time.

Exploring the full workings of Command Prompt is an enormous topic beyond the scope of this book; however, I encourage you to try some of the commands in *Table 18-1*, which contains useful frequently used commands that aren't actually necessary for the exercises in this chapter.

Table 18-1: *Useful Windows Commands*

Command	Function	Example	Action
`cd`	Change directory	`cd C:\my-stuff`	Changes to the *my-stuff* directory on the C: drive
`copy`	Copy a file	`copy C:\my-stuff\song.mp3 C:\Music\song_favorite.mp3`	Copies the *song.mp3* file from *my-stuff* to a new file called *song_favorite.mp3* in the *Music* directory
`del`	Delete	`del *.jpg`	Deletes all files with a *.jpg* extension in the current directory (asterisk wildcard)
`dir`	List directory contents	`dir /p`	Shows directory contents one screen at a time (using the `/p` option)
`findstr`	Find strings in text files matching a regular expression	`findstr "peach" *.txt`	Searches for the text *peach* in all *.txt* files in the current directory
`mkdir`	Make a new directory	`makedir C:\my-stuff\Salad`	Creates a *Salad* directory inside the *my-stuff* directory
`move`	Move a file	`move C:\my-stuff\song.mp3 C:\Music\`	Moves the file *song.mp3* to the *C:\Music* directory

With your Command Prompt open and configured, you're ready to roll. Skip ahead to the section "Working with psql."

macOS psql Setup

On macOS, you'll run `psql` within Terminal, the application that provides access to that system's command line via one of several *shell* programs such as `bash` or `zsh`. Shell programs on Unix- or Linux-based systems, including macOS, provide not only the command prompt where users enter instructions, but also their own programming language for automating tasks. For example, you can use `bash` commands to write a program to log in to a remote computer, transfer files, and log out.

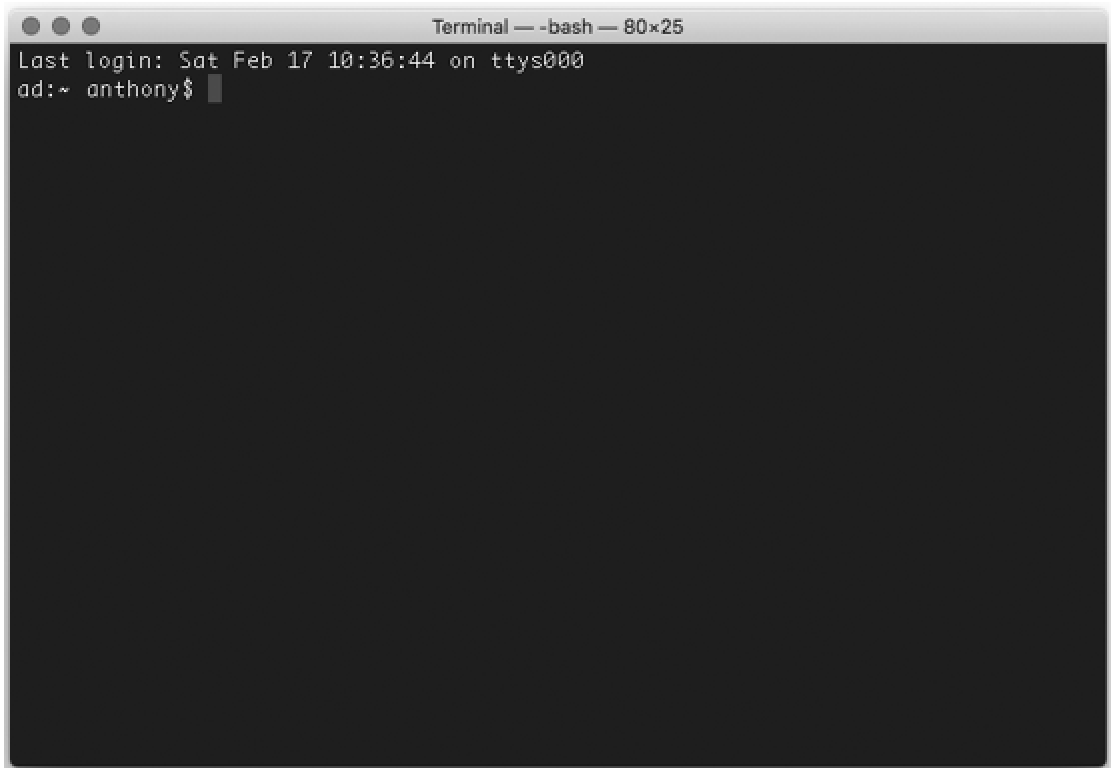
If you followed the Postgres.app setup instructions for macOS in Chapter 1—including running the commands at your Terminal—you shouldn't need additional configuration to use `psql` and associated commands. Instead, we'll proceed to launching Terminal.

Launching and Configuring the macOS Terminal

Launch Terminal by navigating to **Applications▶Utilities▶Terminal**. When it opens, you should see a window that displays the date and time of your last login followed by a prompt that includes your computer name, current working directory, and username. On my Mac, which is set to the `bash` shell, the prompt displays as `ad:~ anthony$` and ends with a dollar sign (`$`), as shown in *Figure 18-4*.

The tilde (`~`) represents the system's home directory, which is `/Users/anthony`. Terminal doesn't display the full directory path, but you can see that information at any time by entering the `pwd` command (short for *print working directory*) and pressing ENTER on your keyboard. The area after the dollar sign is where you enter commands.

If your Mac is set to another shell such as `zsh`, your prompt may look different. With `zsh`, for example, the prompt ends with a percent sign. The particular shell you're using does not make a difference for these exercises.



```
Last login: Sat Feb 17 10:36:44 on ttys000
ad:~ anthony$
```

Figure 18-4: Terminal command line in macOS

NOTE

*For fast access to Terminal, add it to your macOS Dock. While Terminal is running, right-click its icon and select **Options▶Keep in Dock**.*

If you've never used Terminal, its default black-and-white color scheme might seem boring. You can change fonts, colors, and other settings by selecting **Terminal▶Preferences**. To make Terminal bigger to better fit the query output display, I recommend setting the window size (on the Window tab under Profiles) to a width of at least 80 columns and a height of 25 rows. My preferred font (on the Text tab) is Monaco 14, but experiment to find one you like.

Exploring the full workings of Terminal and related commands is an enormous topic beyond the scope of this book, but take some time to try several commands. *Table 18-2* lists some useful commonly used commands that aren't actually necessary for the exercises in this chapter. Enter **`man`** (short for *manual*) followed by a command name to get help on any command. For example, you can use **`man ls`** to find out how to use the `ls` command to list directory contents.

Table 18-2: *Useful Terminal Commands*

Command	Function	Example	Action
`cd`	Change directory	`cd /Users/pparker/my-stuff/`	Changes to the *my-stuff* directory
`cp`	Copy files	`cp song.mp3 song_backup.mp3`	Copies the file *song.mp3* to *song_backup.mp3* in the current directory
`grep`	Find strings in a text file matching a regular expression	`grep 'us_counties_2010' *.sql`	Finds all lines in files with a *.sql* extension that have the text *us_counties_2010*
`ls`	List directory contents	`ls -al`	Lists all files and directories (including hidden) in "long" format
`mkdir`	Make a new directory	`mkdir resumes`	Makes a directory named *resumes* under the current working directory
`mv`	Move a file	`mv song.mp3 /Users/pparker/songs`	Moves the file *song.mp3* from the current directory to a */songs* directory under a user directory
`rm`	Remove (delete) files	`rm *.jpg`	Deletes all files with a *.jpg* extension in the current directory (asterisk wildcard)

With your Terminal open and configured, you're ready to roll. Skip ahead to the section "Working with psql."

Linux psql Setup

Recall from "Linux Installation" in Chapter 1 that methods for installing PostgreSQL vary according to your Linux distribution. Nevertheless, `psql` is part of the standard PostgreSQL install, and you probably already ran `psql` commands as part of the installation process via your distribution's command line terminal application. Even if you didn't, standard Linux installations of PostgreSQL will automatically add `psql` to your `PATH`, so you should be able to access it.

Launch a terminal application and proceed to the next section, "Working with psql." On some distributions, such as Ubuntu, you can open a terminal by pressing CTRL-ALT-T. Also note that the Mac Terminal commands in *Table 18-2* apply to Linux as well and may be useful to you.

Working with psql

Now that you've identified your command line interface and set it up to recognize the location of `psql`, let's launch `psql` and connect to a database on your local installation of PostgreSQL. Then we'll explore executing queries and special commands for retrieving database information.

Launching psql and Connecting to a Database

Regardless of the operating system you're using, you start `psql` in the same way. Open your command line interface (Command Prompt on Windows, Terminal on macOS or Linux). To launch `psql` and connect to a database, we use the following pattern at the command prompt:

```
psql -d database_name -U user_name
```

Following the `psql` application name, we provide the database name after a `-d` database argument and a username after the `-U` username argument.

For the database name, we'll use `analysis`, which is where we created our tables and other objects for the book's exercises. For username, we'll use `postgres`, which is the default user created during installation. So, to connect to the `analysis` database on your local server, enter this at the command line:

```
psql -d analysis -U postgres
```

Note that you can connect to a database on a remote server by specifying the `-h` argument followed by the hostname. For example, you would use the following line if you were connecting to a database named `analysis` on a server called `example.com`:

```
psql -d analysis -U postgres -h example.com
```

Either way, if you set a password during installation, you should receive a password prompt when `psql` launches. If so, enter your password. After `psql` connects to the database, you should then see a prompt that looks like this:

```
psql (13.3)
Type "help" for help.

analysis=#
```

Here, the first line lists the version number of `psql` and the server you're connected to. Your version will vary depending on when you installed PostgreSQL. The prompt where you'll enter commands is `analysis=#`, which refers to the name of the database, followed by an equal sign (`=`) and a hash mark (`#`). The hash mark indicates that you're logged in with *superuser* privileges, which give you unlimited ability to access and create objects and set up accounts and security. If you're logged in as a user without superuser privileges, the last character of the prompt will be a greater-than sign (`>`). As you can see, the user account you logged in with here (`postgres`) is a superuser.

NOTE

PostgreSQL installations create a default superuser account called `postgres`*. If you're running Postgres.app on macOS, that installation created an additional superuser account that has your system username and no password.*

Finally, on Windows systems, you may see a warning message after launching `psql` about the console code page differing from the Windows code page. This is related to a mismatch in character sets between Command Prompt and the rest of the Windows system. For the exercises in this book, you can safely ignore that warning. If you prefer, you can eliminate it on a per-session basis by entering the command **`cmd.exe /c chcp 1252`** in your Windows Command Prompt before launching `psql`.

Getting Help or Exiting

At the `psql` prompt, you can get help for both `psql` and general SQL with a set of *meta-commands*, detailed in Table 18-3. Meta-commands, which begin with a backslash (`\`), go beyond offering help—we'll cover several that return information about the database, let you adjust settings, or process data.

Table 18-3: *Help Commands Within* `psql`

Command	Displays
`\?`	Commands available within `psql`, such as `\dt` to list tables.
`\? options`	Options for use with the `psql` command, such as `-U` to specify username.
`\? variables`	Variables for use with `psql`, such as `VERSION` for the current `psql` version.
`\h`	List of SQL commands. Add a command name to see detailed help for it (for example, `\h INSERT`).

Even experienced users often need a refresher on commands and options, so having the details in the `psql` application is handy. To exit `psql`, use the meta-command `\q` (for *quit*).

Changing the Database Connection

When working with SQL, it's not unusual to be working with multiple databases, so you need a way to switch between databases. You can do this easily at the `psql` prompt using the `\c` meta-command.

For example, while connected to your `analysis` database, at the `psql` prompt enter the following command to create a database named `test`:

```
analysis=# CREATE DATABASE test;
```

Then, to connect to the new `test` database you just created, enter **`\c`** followed by the name of the database at the `psql` prompt (and provide your PostgreSQL password if asked):

```
analysis=# \c test
```

The application should respond with the following message:

```
You are now connected to database "test" as user "postgres".
test=#
```

The prompt will show you which database you're connected to. To log in as a different user—for example, using a username the macOS installation created—you could add that username after the database name. On my Mac, the syntax looks like this:

```
analysis-# \c test anthony
```

The response would be as follows:

```
You are now connected to database "test" as user "anthony".
test=#
```

To reduce clutter, you can remove the `test` database you created. First, use `\c` to disconnect from `test` and connect to the `analysis` database (we can remove a database only if no one is connected to it). Once you've connected to `analysis`, enter **`DROP DATABASE test;`** at the `psql` prompt.

Setting Up a Password File

If you'd rather not see a password prompt when starting `psql`, you can set up a file to store database connection information that includes the server name, your username, and password. On startup, `psql` will read the file and bypass the password prompt if the file contains an entry that matches the database connection and username.

On Windows 10, the file must be named *pgpass.conf* and must reside in the following directory: C:\USERS\YourUsername\AppData\Roaming\postgresql\\. You may need to create the `postgresql` directory. On macOS and Linux, the file must be named *.pgpass* and must reside in your user home directory. The documentation at *https://www.postgresql.org/docs/current/libpq-pgpass.html* notes that on macOS and Linux, you may need to set file permissions after creating the file by running `chmod 0600 ~/.pgpass` at the command line.

Create the file using a text editor and save it with the correct name and location for your system. Inside, you'll need to add a line for each database connection in the following format:

```
hostname:port:database:username:password
```

For example, to set up a connection for the `analysis` database and `postgres` username, enter this line, substituting your password:

```
localhost:5432:analysis:postgres:password
```

You can substitute an asterisk in any of the first four parameters to serve as a wildcard. For example, to supply a password for any local database with the `postgres` username, substitute an asterisk for the database name:

```
localhost:5432:*:postgres:password
```

Saving your password will save you some typing, but be mindful of best practices regarding security. Always secure your computer with a strong password and/or physical security key, and don't create a password file on any public or shared system.

Running SQL Queries on psql

We've configured `psql` and connected to a database, so now let's run some SQL queries. We'll start with a single-line query and then run a multiple-line query.

You enter SQL into `psql` directly at the prompt. For example, to see a few rows from the 2019 census table we've used throughout the book, enter a query at the prompt, as shown in Listing 18-1.

```
analysis=# SELECT county_name FROM us_counties_pop_est_2019
ORDER BY county_name LIMIT 3;
```

Listing 18-1: Entering a single-line query in `psql`

Press ENTER to execute the query, and `psql` should display the following results in your terminal in text including the number of rows returned:

```
   county_name
------------------
 Abbeville County
 Acadia Parish
 Accomack County
(3 rows)

analysis=#
```

Below the result, you can see the `analysis=#` prompt again, ready for further input. You can use the up and down arrows on your keyboard to scroll through recent queries and press ENTER to execute them again, avoiding having to retype them.

Entering a Multiline Query

You're not limited to single-line queries. If you have a query that spans multiple lines, you can enter lines one at a time, pressing ENTER after each, and `psql` knows not execute the query until you provide a semicolon. Let's reenter the query in *Listing 18-1* over multiple lines, shown in *Listing 18-2*.

```
analysis=# SELECT county_name
analysis-# FROM us_counties_pop_est_2019
analysis-# ORDER BY county_name
analysis-# LIMIT 3;
```

Listing 18-2: Entering a multiline query in `psql`

Note that when your query extends past one line, the symbol between the database name and the hash mark changes from an equal sign to a hyphen. This multiline query executes only when you press ENTER after the final line, which ends with a semicolon.

Checking for Open Parentheses in the psql Prompt

Another helpful feature of `psql` is that it shows when you haven't closed a pair of parentheses. *Listing 18-3* shows this in action.

```
analysis=# CREATE TABLE wineries (
analysis(# id bigint,
analysis(# winery_name text
analysis(# );
CREATE TABLE
```

Listing 18-3: Showing open parentheses in the `psql` prompt

Here, you create a simple table called `wineries` that has two columns. After entering the first line of the `CREATE TABLE` statement and an open parenthesis (`(`), the prompt then changes from `analysis=#` to `analysis(#` to include an open parenthesis. This reminds you an open parenthesis needs closing. The prompt maintains that configuration until you add the closing parenthesis.

NOTE

If you have a lengthy query saved in a text file, such as one from this book's resources, you can copy it to your computer clipboard and paste it into `psql` (CTRL-V on Windows, COMMAND-V on macOS, and SHIFT-CTRL-V on Linux). That saves you from typing the whole query. After you paste the query text into `psql`, press ENTER to execute it.

Editing Queries

To modify the most recent query you've executed in `psql`, you can edit it using the `\e` or `\edit` meta-command. Enter **`\e`** to open the last-executed query in a text editor. The editor `psql` uses by default depends on your operating system.

On Windows, `psql` will open Notepad, a simple GUI text editor. Edit your query, save it by choosing **File▸Save**, and then exit Notepad using **File▸Exit**. When Notepad closes, `psql` should execute your revised query.

On macOS and Linux, `psql` uses a command line application called `vim`, which is a favorite among programmers but can seem inscrutable for beginners. Check out a helpful `vim` cheat sheet at https://vim.rtorr.com/. For now, you can use the following steps to make simple edits:

- When vim opens the query in your terminal, press I to activate insert mode.
- Make your edits to the query.
- Press ESC and then SHIFT-: to display a colon command prompt at the bottom left of the vim screen, which is where you enter commands to control vim.
- Enter **wq** (for *write*, *quit*) and press ENTER to save your changes.

Now when vim exits, the psql prompt should execute your revised query. Press the up-arrow key to see the revised text.

Navigating and Formatting Results

The query you ran in Listings 18-1 and 18-2 returned only one column and a handful of rows, so its output was contained nicely in your command line interface. But for queries with more columns or rows, the output can fill more than one screen, making it difficult to navigate. Fortunately, you have several ways to customize the display style of the output using the \pset meta-command.

Setting Paging of Results

One way to adjust the output format is to specify how psql handles scrolling through lengthy query results, known as *paging*. By default, if your results return more rows than will fit on one screen, psql will display the first screen's worth of rows and then let you scroll through the rest. For example, *Listing 18-4* shows what happens at the psql prompt when we remove the LIMIT clause from the query in *Listing 18-1*.

```
analysis=# SELECT county_name FROM us_counties_pop_est_2019
ORDER BY county_name;

            county_name
-----------------------------------
 Abbeville County
 Acadia Parish
 Accomack County
 Ada County
 Adair County
 Adair County
 Adair County
 Adair County
 Adams County
 Adams County
 Adams County
 Adams County
-- More --
```

Listing 18-4: A query with scrolling results

Recall that this table has 3,142 rows. Listing 18-4 shows only the first 12 on the current screen (the number of visible rows depends on your terminal configuration). On Windows, the indicator `-- More --` tells you that additional results are available, and you can press ENTER to scroll through them. On macOS and Linux, the indicator will be a colon. Scrolling through a few thousand rows will take a while. Press Q to exit the results and return to the `psql` prompt.

To bypass manual scrolling and have all your results immediately display, you can change the `pager` setting using the `\pset pager` meta-command. Run that command at your `psql` prompt, and it should return the message `Pager usage is off`. Now when you rerun the query in Listing 18-3 with the `pager` setting turned off, you should see results like this:

```
 --snip--
 York County
 York County
 York County
 York County
 Young County
 Yuba County
 Yukon-Koyukuk Census Area
 Yuma County
 Yuma County
 Zapata County
 Zavala County
 Ziebach County
(3142 rows)

analysis=#
```

You're immediately taken to the end of the results without having to scroll. To turn paging back on, run `\pset pager` again.

Formatting the Results Grid

You also can use `\pset` with the following options to format the results:

border int

Use this option to specify whether the results grid has no border (`0`), internal lines dividing columns (`1`), or lines around all cells (`2`). For example, `\pset border 2` sets lines around all cells.

format unaligned

Use the option `\pset format unaligned` to display the results in lines separated by a delimiter rather than in columns, similar to what you would see in a CSV file. The separator defaults to a pipe symbol (`|`). You can set a different separator using the `fieldsep` command. For example, to set a comma as the separator, run `\pset fieldsep ','`. To revert to a column view, run `\pset format aligned`. You can use the `psql` meta-command `\a` to toggle between aligned and unaligned views.

footer

Use this option to toggle the results footer, which displays the result row count, on or off.

null

Use this option to set how `psql` displays `NULL` values. By default, they show as blanks. You can run `\pset null '(null)'` to replace blanks with `(null)` when the column value is `NULL`.

You can explore additional options in the PostgreSQL documentation at *https://www.postgresql.org/docs/current/app-psql.html*. In addition, it's possible to set up a *.psqlrc* file on macOS or Linux or a *psqlrc.conf* file on Windows to hold your configuration preferences and load them each time `psql` starts. A good example is provided at *https://www.citusdata.com/blog/2017/07/16/customizing-my-postgres-shell-using-psqlrc/*.

Viewing Expanded Results

Sometimes, it's helpful to view results arranged not in the typical table style of rows and columns, but instead in a vertical list. This is useful when the number of columns is too big to fit on-screen in the normal horizontal results grid and also for scanning values in columns row by row. In `psql`, you can switch to a vertical list view using the `\x` (for *expanded*) meta-command. The best way to understand the difference between normal and expanded view is by looking at an example. Listing 18-5 shows the normal display you see when querying the `grades` table in Chapter 17 using `psql`.

```
analysis=# SELECT * FROM grades ORDER BY student_id,
course_id;
 student_id | course_id |       course       | grade
------------+-----------+--------------------+-------
          1 |         1 | Biology 2          | C
          1 |         2 | English 11B        | D
          1 |         3 | World History 11B  | C
          1 |         4 | Trig 2             | B
(4 rows)
```

Listing 18-5: Normal display of the `grades` table query

To change to the expanded view, enter **`\x`** at the `psql` prompt, which should display the message `Expanded display is on`. Then, when you run the same query again, you should see the expanded results, as shown in Listing 18-6.

```
analysis=# SELECT * FROM grades ORDER BY student_id,
course_id;
-[ RECORD 1 ]-----------------
student_id | 1
course_id  | 1
course     | Biology 2
grade      | C
-[ RECORD 2 ]-----------------
student_id | 1
course_id  | 2
course     | English 11B
grade      | D
-[ RECORD 3 ]-----------------
student_id | 1
course_id  | 3
```

```
course     | World History 11B
grade      | C
-[ RECORD 4 ]-----------------
student_id | 1
course_id  | 4
course     | Trig 2
grade      | B
```

Listing 18-6: Expanded display of the `grades` table query

The results appear in vertical blocks separated by record numbers. Depending on your needs and the type of data you're working with, this format might be easier to read. You can revert to column display by entering **`\x`** again at the `psql` prompt. In addition, setting `\x auto` will make PostgreSQL automatically display the results in a table or expanded view based on the size of the output.

Next, let's explore how to use `psql` to dig into database information.

Meta-Commands for Database Information

You can have `psql` display details about databases, tables, and other objects via a particular set of meta-commands. To see how these work, we'll explore the meta-command that displays the tables in your database, including how to append a plus sign (+) to the command to expand the output and add an optional pattern to filter the output.

To see a list of tables, you can enter **`\dt`** at the `psql` prompt. Here's a snippet of the output on my system:

```
                           List of relations
 Schema |                  Name                    | Type  |
Owner
--------+----------------------------------------+-------+---
--------
 public | acs_2014_2018_stats                    | table |
anthony
 public | cbp_naics_72_establishments            | table |
anthony
 public | char_data_types                        | table |
anthony
 public | check_constraint_example               | table |
anthony
```

```
 public | crime_reports                           | table |
anthony
 --snip--
```

This result lists all tables in the current database alphabetically.

You can filter the output by adding a pattern the database object name must match. For example, use `\dt us*` to show only tables whose names begin with `us` (the asterisk acts as a wildcard). The results should look like this:

```
                  List of relations
 Schema |          Name            | Type  |   Owner
--------+--------------------------+-------+-----------
 public | us_counties_2019_shp     | table | anthony
 public | us_counties_2019_top10   | table | anthony
 public | us_counties_pop_est_2010 | table | anthony
 public | us_counties_pop_est_2019 | table | anthony
 public | us_exports               | table | anthony
```

Table 18-4 shows several additional commands you might find helpful, including `\l`, which lists the databases on your server. Adding a plus sign to each command, as in `\dt+`, adds more information to the output, including object sizes.

Table 18-4: *Example of `psql` `\d` Commands*

Command	Displays
`\d [pattern]`	Columns, data types, plus other information on objects
`\di [pattern]`	Indexes and their associated tables
`\dt [pattern]`	Tables and the account that owns them
`\du [pattern]`	User accounts and their attributes
`\dv [pattern]`	Views and the account that owns them
`\dx [pattern]`	Installed extensions
`\l [pattern]`	Databases

The entire list of commands is available in the PostgreSQL documentation at *https://www.postgresql.org/docs/current/app-psql.html*, or you can see details by using the `\?` command noted earlier.

Importing, Exporting, and Using Files

In this section, we'll explore how to use `psql` to import and export data from the command line, which can be necessary when you're connected to remote servers, such as Amazon Web Services instances of PostgreSQL. We'll also use `psql` to read and execute SQL commands stored in a file and learn the syntax for sending `psql` output to a file.

Using \copy for Import and Export

In Chapter 5, you learned how to use the PostgreSQL `COPY` command to import and export data. It's a straightforward process, but it has one significant limitation: the file you're importing or exporting must be on the same machine as the PostgreSQL server. That's fine if you're working on your local machine, as you've been doing with these exercises. But if you're connecting to a database on a remote computer, you might not have access to its file system. You can get around this restriction by using the `\copy` meta-command in `psql`.

The `\copy` meta-command works just like the PostgreSQL `COPY`, except when you execute it at the `psql` prompt, it will route data from your machine to the server you're connected to, whether local or remote. We won't actually connect to a remote server to try this since it's rare to find a public remote server we could connect to, but you can still learn the syntax by using the commands on our local `analysis` database.

In *Listing 18-7*, at the `psql` prompt we use a `DELETE` statement to remove all the rows from the small `state_regions` table you created in Chapter 10 and then import data using `\copy`. You'll need to change the file path to match the location of the file on your computer.

```
analysis=# DELETE FROM state_regions;
DELETE 56
analysis=# \copy state_regions FROM
'C:\YourDirectory\state_regions.csv' WITH (FORMAT CSV,
HEADER);
COPY 56
```

Listing 18-7: Importing data using `\copy`

Next, to import the data, we use `\copy` with the same syntax used with PostgreSQL `COPY`, including a `FROM` clause with the file path on your machine, and a `WITH` clause that specifies the file is a CSV and has a header row. When you execute the statement, the server should respond with `COPY 56`, letting you know the rows have been successfully imported.

If you're connected to a remote server via `psql`, you would use the same `\copy` syntax, and the command would route your local file to the remote server for importing. In this example, we used `\copy FROM` to import a file. We could also use `\copy TO` for exporting. Let's look at an alternate way to import or export data (or run other SQL commands) via `psql`.

Passing SQL Commands to psql

By placing a command in quotes after the `-c` argument, we can send it to our connected server, local or remote. The command can be a single SQL statement, multiple SQL statements separated by semicolons, or a meta-command. This can allow us to run `psql`, connect to a server, and execute a command in a single command line statement—handy if we want to incorporate `psql` statements into shell scripts to automate tasks.

For example, we can import data to the `state_regions` table with the statement in Listing 18-8, which must be entered on one line at your command prompt (and not inside `psql`).

```
psql -d analysis -U postgres -c1 'COPY state_regions FROM
STDIN2 WITH (FORMAT CSV, HEADER);' <3
C:\YourDirectory\state_regions.csv
```

Listing 18-8: Importing data using `psql` with `COPY`

To try it, you'll need to first run `DELETE FROM state_regions;` inside `psql` to clear the table. Then exit `psql` by typing the meta-command `\q`.

At your command prompt, enter the statement in Listing 18-8. We first use `psql` and the `-d` and `-U` commands to connect to your `analysis` database. Then comes the `-c` command 1, which we follow with the PostgreSQL statement for importing the data. The statement is similar to `COPY` statements we've used with one exception: after `FROM`, we use the keyword `STDIN` 2 instead of the complete file path and filename. `STDIN` means "standard input," which is a stream of input data that can come from a device, a keyboard, or in this case the file *state_regions.csv*, which we direct 3 to `psql` using the less-than (<) symbol. You'll need to supply the full path to the file.

Running this entire command at your command prompt should import the CSV file and generate the message `COPY 56`.

Saving Query Output to a File

It's sometimes helpful to save the query results and messages generated during a `psql` session to a file, such as to keep a history of your work or to use the output in a spreadsheet or other application. To send query output to a file, you can use the `\o` meta-command along with the full path and name of an output file that `psql` will create.

NOTE

On Windows, file paths for the `\o` command must use either Linux-style forward slashes, such as `C:/my-stuff/my-file.txt`, or double backslashes, such as `C:\\my-stuff\\my-file.txt`.

For example, in Listing 18-9 we change the `psql` format style from a table to CSV and then output query results directly to a file.

```
1 analysis=# \pset format csv
  Output format is csv.

  analysis=# SELECT * FROM grades ORDER BY student_id,
  course_id;
2 student_id,course_id,course,grade
  1,1,Biology 2,F
  1,2,English 11B,D
  1,3,World History 11B,C
  1,4,Trig 2,B

3 analysis=# \o 'C:/YourDirectory/query_output.csv'
```

```
  analysis=# SELECT * FROM grades ORDER BY student_id,
  course_id;
4 analysis=#
```

Listing 18-9: Saving query output to a file

First, we set the output format 1 using the meta-command `\pset format csv`. When you run a simple `SELECT` on the `grades` table, the output 2 should return as values separated by commas. Next, to send that data to a file the next time you run the query, use the `\o` meta-command and then provide a complete path to a file called `query_output.csv` 3. When you run the `SELECT` query again, there should be no output to the screen 4. Instead, you'll find a file with the contents of the query in the directory specified at 3.

Note that every time you run a query from this point, the output is appended to the same file specified after the `\o` (for *output*) command. To stop saving output to that file, you can either specify a new file or enter **`\o`** with no filename to resume having results output to the screen.

Reading and Executing SQL Stored in a File

To run SQL stored in a text file, you execute `psql` on the command line and supply the filename after an `-f` (for file) argument. This syntax lets you quickly run a query or table update from the command line or in conjunction with a system scheduler to run a job at regular intervals.

Let's say you saved the `SELECT` query from Listing 18-9 in a file called *display-grades.sql*. To run the saved query, use the following `psql` syntax at your command line:

```
psql -d analysis -U postgres -f C:\YourDirectory\display-
grades.sql
```

When you press ENTER, `psql` should launch, run the stored query in the file, display the results, and exit. For repetitive tasks, this workflow can save considerable time because you avoid launching pgAdmin or rewriting a query. You also can stack multiple queries in the file so they run in succession, which, for example, you might do if you want to run several updates on your database.

Additional Command Line Utilities to Expedite Tasks

PostgreSQL also has its own set of command line utilities that you can enter in your command line interface without launching `psql`. A listing is available at *https://www.postgresql.org/docs/current/reference-client.html*, and I'll explain several in Chapter 19 that are specific to database maintenance. Here I'll cover two that are particularly useful: creating a database at the command line with the `createdb` utility and loading shapefiles into a PostGIS database via the `shp2pgsql` utility.

Adding a Database with createdb

Earlier in the chapter, you used `CREATE DATABASE` to add the database `test` to your PostgreSQL server. We can achieve the same thing using `createdb` at the command line. For example, to create a new database on your server named `box_office`, run the following at your command line:

```
createdb -U postgres -e box_office
```

The `-U` argument tells the command to connect to the PostgreSQL server using the `postgres` account. The `-e` argument (for *echo*) prints the commands generated by `createdb` as output. Running this command creates the database and prints output to the screen ending with `CREATE DATABASE box_office;`. You can then connect to the new database via `psql` using the following line:

```
psql -d box_office -U postgres
```

The `createdb` command accepts arguments to connect to a remote server (just like `psql` does) and to set options for the new database. A full list of arguments is available at *https://www.postgresql.org/docs/current/app-createdb.html*. Again, the `createdb` command is a time-saver that comes in handy when you don't have access to a GUI.

Loading Shapefiles with shp2pgsql

In Chapter 15, you learned about shapefiles, which contain data describing spatial objects. On Windows and some Linux distributions, you can import shapefiles into a PostGIS-enabled database using the Shapefile Import/Export Manager GUI tool (generally) included with PostGIS. However, the Shapefile Import/Export Manager is not always included with PostGIS on macOS or some flavors of Linux. In those cases (or if you'd rather work at the command line), you can import a shapefile using the PostGIS command line tool `shp2pgsql`.

To import a shapefile into a new table from the command line, use the following syntax:

```
shp2pgsql -I -s SRID -W encoding shapefile_name table_name |
psql -d database -U user
```

A lot is happening in this single line. Here's a breakdown of the arguments (if you skipped Chapter 15, you might need to review it now):

`-I` Uses GiST to add an index on the new table's geometry column.

`-s` Lets you specify an SRID for the geometric data.

`-W` Lets you specify encoding. (Recall that we used `Latin1` for census shapefiles.)

`shapefile_name` The name (including full path) of the file ending with the *.shp* extension.

`table_name` The name of the table the shapefile is imported to.

Following these arguments, you place a pipe symbol (`|`) to direct the output of `shp2pgsql` to `psql`, which has the arguments for naming the database and user. For example, to load the *tl_2019_us_county.shp* shapefile into a `us_counties_2019_shp` table in the `analysis` database, you can run the following command. Note that although this command wraps onto two lines here, it should be entered as one line in the command line:

```
shp2pgsql -I -s 4269 -W Latin1 tl_2019_us_county.shp
us_counties_2019_shp | psql -d analysis -U postgres
```

The server should respond with a number of SQL `INSERT` statements before creating the index and returning you to the command line. It might take some time to construct the entire set of arguments the first time around, but after you've done one, subsequent imports should take less time. You can simply substitute file and table names into the syntax you already wrote.

Wrapping Up

Feeling mysterious and powerful yet? Indeed, when you delve into a command line interface and make the computer do your bidding using text commands, you enter a world of computing that resembles a sci-fi movie sequence. Not only does working from the command line save you time, it also helps you overcome barriers you might hit when working in environments that don't support graphical tools. In this chapter, you learned the basics of working with the command line plus PostgreSQL specifics. You discovered your operating system's command line application and set it up to work with `psql`. Then you connected `psql` to a database and learned how to run SQL queries via the command line. Many experienced computer users prefer to use the command line for its simplicity and speed once they become familiar with using it. You might, too.

In Chapter 19, we'll review common database maintenance tasks including backing up data, changing server settings, and managing the growth of your database. These tasks will give you more control over your working environment and help you better manage your data analysis projects.

TRY IT YOURSELF

To reinforce the techniques in this chapter, choose an example from an earlier chapter and try working through it using only the command line. Chapter 15, "Analyzing Spatial Data with PostGIS," is a good choice because it gives you the opportunity to work with `psql` and the shapefile loader `shp2pgsql`. That said, I encourage you to choose any example that you think you would benefit from reviewing.

19
MAINTAINING YOUR DATABASE

To wrap up our exploration of SQL, we'll look at key database maintenance tasks and options for customizing PostgreSQL. In this chapter, you'll learn how to track and conserve space in your databases, how to change system settings, and how to back up and restore databases. How often you'll need to perform these tasks depends on your current role and interests. If you want to be a *database administrator* or a *backend developer*, the topics covered here are vital.

It's worth noting that database maintenance and performance tuning are large enough topics that they often occupy entire books, and this chapter mainly serves as an introduction to a handful of essentials. If you want to learn more, a good place to begin is with the resources in the appendix.

Let's start with the PostgreSQL `VACUUM` feature, which lets you shrink the size of tables by removing unused rows.

Recovering Unused Space with VACUUM

The PostgreSQL `VACUUM` command helps manage the size of a database, which—as discussed in “Improving Performance When Updating Large Tables” in Chapter 10—can grow as a result of routine operations.

For example, when you update a row value, the database creates a new version of that row with the updated value and retains (but hides) the old version of the row. The PostgreSQL documentation refers to these rows that you can’t see as *dead tuples*, with *tuples*—an ordered list of elements—being the name for the internal implementation of rows in a PostgreSQL database. The same thing happens when you delete a row. Though the row is no longer visible to you, it lives on as a dead row in the table.

This is by design so the database can provide certain features in environments where multiple transactions are occurring, and an old version of a row might be needed by transactions other than the current one.

The `VACUUM` command cleans up these dead rows. Running `VACUUM` on its own designates the space occupied by dead rows as available for the database to use again (assuming that any transactions using the rows have been completed). In most cases, `VACUUM` doesn’t return the space to your system’s disk; it just flags that space as available for new data. To actually shrink the size of the data file, you can run `VACUUM FULL`, which rewrites the table to a new version that doesn’t include the dead row space. It drops the old version.

Although `VACUUM FULL` frees space on your system’s disk, there are a couple of caveats to keep in mind. First, `VACUUM FULL` takes more time to complete than `VACUUM`. Second, it must have exclusive access to the table while rewriting it, which means that no one can update data during the operation. The regular `VACUUM` command can run while updates and other operations are happening. Finally, not all dead space in a table is bad. In many cases, having available space to put new tuples instead of needing to ask the operating system for more disk space can improve performance.

You can run either `VACUUM` or `VACUUM FULL` on demand, but PostgreSQL by default runs an *autovacuum* background process that monitors the database and runs `VACUUM` as needed. Later in this chapter, I’ll show you how to monitor autovacuum as well as run the `VACUUM` command manually. But first, let’s look at how a table grows as a result of updates and how you can track this growth.

Tracking Table Size

We'll create a small test table and monitor its growth as we fill it with data and perform an update. The code for this exercise, as with all resources for the book, is available at *https://nostarch.com/practical-sql-2nd-edition/*.

Creating a Table and Checking Its Size

Listing 19-1 creates a `vacuum_test` table with a single column to hold an integer. Run the code, and then we'll measure the table's size.

```
CREATE TABLE vacuum_test (
    integer_column integer
);
```

Listing 19-1: Creating a table to test vacuuming

Before we fill the table with test data, let's check how much space it occupies on disk to establish a reference point. We can do so in two ways: check the table properties via the pgAdmin interface or run queries using PostgreSQL administrative functions. In pgAdmin, click once on a table to highlight it, and then click the **Statistics** tab. Table size is one of about two dozen indicators in the list.

I'll focus on the running queries technique here because knowing these queries is helpful if for some reason pgAdmin isn't available or you're using another graphical user interface (GUI). *Listing 19-2* shows how to check the `vacuum_test` table size using PostgreSQL functions.

```
SELECT 1pg_size_pretty(
           2pg_total_relation_size('vacuum_test')
       );
```

Listing 19-2: Determining the size of `vacuum_test`

The outermost function, `pg_size_pretty()` 1, converts bytes to a more easily understandable format in kilobytes, megabytes, or gigabytes. Wrapped inside is the `pg_total_relation_size()` function 2, which reports how many bytes a table, its indexes, and any offline compressed data takes up on disk. Because the table is empty at this point, running the code in pgAdmin should return a value of `0 bytes`, like this:

```
 pg_size_pretty
----------------
 0 bytes
```

You can get the same information using the command line. Launch `psql` as you learned in Chapter 18. Then, at the prompt, enter the meta-command **`\dt+ vacuum_test`**, which should display the following information including table size (I've omitted one column for space):

```
                      List of relations
 Schema |     Name     | Type  |  Owner   | Persistence |
Size
--------+-------------+-------+----------+-------------+-----
----
 public | vacuum_test | table | postgres | permanent   | 0
bytes
```

Again, the current size of the `vacuum_test` table should display `0 bytes`.

Checking Table Size After Adding New Data

Let's add some data to the table and then check its size again. We'll use the `generate_series()` function introduced in Chapter 12 to fill the table's `integer_column` with 500,000 rows. Run the code in *Listing 19-3* to do this.

```
INSERT INTO vacuum_test
SELECT * FROM generate_series(1,500000);
```

Listing 19-3: Inserting 500,000 rows into `vacuum_test`

This standard `INSERT INTO` statement adds the results of `generate_series()`, which is a series of values from 1 to 500,000, as rows to the table. After the query completes, rerun the query in *Listing 19-2* to check the table size. You should see the following output:

```
 pg_size_pretty
----------------
```

```
 17 MB
```

The query reports that the `vacuum_test` table, now with a single column of 500,000 integers, uses 17MB of disk space.

Checking Table Size After Updates

Now, let's update the data to see how that affects the table size. We'll use the code in *Listing 19-4* to update every row in the `vacuum_test` table by adding `1` to the `integer_column` values, replacing the existing value with a number that's one greater.

```
UPDATE vacuum_test
SET integer_column = integer_column + 1;
```

Listing 19-4: Updating all rows in `vacuum_test`

Run the code, and then test the table size again.

```
 pg_size_pretty
----------------
 35 MB
```

The table size doubled from 17MB to 35MB! The increase seems excessive, because the `UPDATE` simply replaced existing numbers with values of a similar size. As you might have guessed, the reason for this increase in table size is that for every updated value, PostgreSQL creates a new row, and the dead row remains in the table. Even though you see only 500,000 rows, the table has double that number. This behavior can lead to surprises for database owners who don't monitor disk space.

Before looking at how using `VACUUM` and `VACUUM FULL` affects the table's size on disk, let's review the process that runs `VACUUM` automatically as well as how to check on statistics related to table vacuums.

Monitoring the Autovacuum Process

PostgreSQL's autovacuum process monitors the database and launches `VACUUM` automatically when it detects a large number of dead rows in a table. Although autovacuum is enabled by default, you can turn it on or off and configure it using the settings I'll cover later in "Changing Server Settings." Because autovacuum runs in the background, you won't see any immediately visible indication that it's working, but you can check its activity by querying data that PostgreSQL collects about system performance.

PostgreSQL has its own *statistics collector* that tracks database activity and usage. You can look at the statistics by querying one of several views the system provides. (See a complete list of views for monitoring the state of the system in the PostgreSQL documentation under "The Statistics Collector": *https://www.postgresql.org/docs/current/monitoring-stats.html*.) To check the activity of autovacuum, we query the `pg_stat_all_tables` view, as shown in Listing 19-5.

```
SELECT 1relname,
       2last_vacuum,
       3last_autovacuum,
       4vacuum_count,
       5autovacuum_count
FROM pg_stat_all_tables
WHERE relname = 'vacuum_test';
```

Listing 19-5: Viewing autovacuum statistics for `vacuum_test`

As you learned in Chapter 17, a view provides the results of a stored query. The query stored by the view `pg_stat_all_tables` returns a column called `relname`1, which is the name of the table, plus columns with statistics related to index scans, rows inserted and deleted, and other data. For this query, we're interested in `last_vacuum` 2 and `last_autovacuum` 3, which contain the last time the table was vacuumed manually and automatically, respectively. We also ask for `vacuum_count` 4 and `autovacuum_count` 5, which show the number of times the vacuum was run manually and automatically.

By default, autovacuum checks tables every minute. So, if a minute has passed since you last updated `vacuum_test`, you should see details of vacuum activity when you run the query in Listing 19-5. Here's what my system shows (note that I've removed the seconds from the time to save space here):

```
   relname    | last_vacuum | last_autovacuum   | vacuum_count
| autovacuum_count
-------------+-------------+------------------+--------------
+------------------
 vacuum_test |             | 2021-09-02 14:46 |            0
|                1
```

The table shows the date and time of the last autovacuum, and the `autovacuum_count` column shows one occurrence. This result indicates that autovacuum executed a `VACUUM` command on the table once. However, because we've not vacuumed manually, the `last_vacuum` column is empty, and the `vacuum_count` is `0`.

NOTE

The autovacuum process also runs the `ANALYZE` *command, which gathers data on the contents of tables. PostgreSQL stores this information and uses it to execute queries efficiently in the future. You can run* `ANALYZE` *manually if needed.*

Recall that `VACUUM` designates dead rows as available for the database to reuse but typically doesn't reduce the size of the table on disk. You can confirm this by rerunning the code in Listing 17-2, which shows the table remains at 35MB even after the automatic vacuum.

Running VACUUM Manually

To run `VACUUM` manually, you can use the single line of code in Listing 19-6.

```
VACUUM vacuum_test;
```

Listing 19-6: Running `VACUUM` *manually*

This command should return the message `VACUUM` from the server. Now when you fetch statistics again using the query in Listing 17-5, you should see that the `last_vacuum` column reflects the date and time of the manual vacuum you just ran, and the number in the `vacuum_count` column should increase by one.

In this example, we executed `VACUUM` on our test table, but you can also run `VACUUM` on the entire database by omitting the table name. In addition, you can add the `VERBOSE` keyword to return information such as the number of rows found in a table and the number of rows removed, among other information.

Reducing Table Size with VACUUM FULL

Next, we'll run `VACUUM` with the `FULL` option, which actually returns the space taken up by dead tuples back to disk. It does this by creating a new version of a table with the dead rows discarded.

To see how `VACUUM FULL` works, run the command in Listing 19-7.

```
VACUUM FULL vacuum_test;
```

Listing 19-7: Using `VACUUM FULL` *to reclaim disk space*

After the command executes, test the table size again. It should be back down to 17MB, the size it was when we first inserted data.

It's never prudent or safe to run out of disk space, so minding the size of your database files as well as your overall system space is a worthwhile routine to establish. Using `VACUUM` to prevent database files from growing bigger than they have to is a good start.

Changing Server Settings

You can alter the settings for your PostgreSQL server by editing values in *postgresql.conf*, one of several configuration text files that control server settings. Other files include *pg_hba.conf*, which controls connections to the server, and *pg_ident.conf*, which database administrators can use to map usernames on a network to usernames in PostgreSQL. See the PostgreSQL documentation on these files for details; here we'll just cover *postgresql.conf* because it contains settings you may likely want to change.

Most of the values in the file are set to defaults you may never need to adjust, but it's worth exploring in case you're fine-tuning your system. Let's start with the basics.

Locating and Editing postgresql.conf

The location of *postgresql.conf* varies depending on your operating system and install method. You can run the command in Listing 19-8 to locate the file.

```
SHOW config_file;
```

Listing 19-8: *Showing the location of* postgresql.conf

When I run the command on macOS, it shows the path to the file, as shown here:

```
/Users/anthony/Library/Application Support/Postgres/var-
13/postgresql.conf
```

To edit *postgresql.conf*, navigate in your file system to the directory displayed by `SHOW config_file;` and open the file using a text editor. Don't use a rich-text editor like Microsoft Word, as it may add additional formatting to the file.

NOTE

It's a good idea to save an unaltered copy of postgresql.conf *for reference in case you make a change that breaks the system and you need to revert to the original version.*

When you open the file, the first several lines should read as follows:

```
# -----------------------------
# PostgreSQL configuration file
# -----------------------------
#
# This file consists of lines of the form:
#
```

```
#   name = value
--snip--
```

The *postgresql.conf* file is organized into sections that specify settings for file locations, security, logging of information, and other processes. Many lines begin with a hash mark (`#`), which indicates the line is commented out and the setting shown is the active default.

For example, in the *postgresql.conf* file section "Autovacuum Parameters," the default is for autovacuum to be turned on (which is a good, standard practice). The hash mark (`#`) in front of the line means that the line is commented out and the default value is in effect:

```
#autovacuum = on                # Enable autovacuum
subprocess?  'on'
```

To change this or other default settings, you would remove the hash mark, adjust the setting value, and save *postgresql.conf*. Some changes, such as to memory allocations, require a restart of the server; they're noted in *postgresql.conf*. Other changes require only a reload of settings files. You can reload settings files by running the function `pg_reload_conf()` under an account with superuser permissions or by executing the `pg_ctl` command, which we'll cover in the next section.

Listing 19-9 shows settings you may want to change, excerpted from the *postgresql.conf* section "Client Connection Defaults." Use your text editor to search the file for the following.

```
1 datestyle = 'iso, mdy'

2 timezone = 'America/New_York'

3 default_text_search_config = 'pg_catalog.english'
```

Listing 19-9: Sample postgresql.conf *settings*

You can use the `datestyle` setting 1 to specify how PostgreSQL displays dates in query results. This setting takes two parameters separated by a comma: the output format and the ordering of month, day, and year. The default for the output format is the ISO format *`YYYY-MM-DD`* we've used throughout this book, which I recommend for its cross-national portability. However, you can also use the traditional SQL format *`MM/DD/YYYY`*, the expanded Postgres format `Sun Nov 12 22:30:00 2023 EST`, or the German format *`DD.MM.YYYY`* with dots between the date, month, and year (`12.11.2023`). To specify the format using the second parameter, arrange *`m`*, *`d`*, and *`y`* in the order you prefer.

The `timezone` 2 parameter sets the server time zone. *Listing 19-9* shows the value `America/New_York`, which reflects the time zone on my machine when I installed PostgreSQL. Yours should vary based on your location. When setting up PostgreSQL for use as the backend to a database application or on a network, administrators often set this value to `UTC` and use that as a standard on machines across multiple locations.

The `default_text_search_config` 3 value sets the language used by the full-text search operations. Here, mine is set to `english`. Depending on your needs, you can set this to `spanish`, `german`, `russian`, or another language of your choice.

These three examples represent only a handful of settings available for adjustment. Unless you end up deep in system tuning, you probably won't have to tweak much else. Also, use caution when changing settings on a network server used by multiple people or applications; changes can have unintended consequences, so it's worth communicating with colleagues first.

Next, let's look at how to use `pg_ctl` to make changes take effect.

NOTE

The PostgreSQL `ALTER SYSTEM` command can also be used to update settings. The command creates settings in the file postgresql.auto.conf *that will override values in* postgresql.conf. *See https://www.postgresql.org/docs/current/sql-altersystem.html for details.*

Reloading Settings with pg_ctl

The command line utility `pg_ctl` allows you to perform actions on a PostgreSQL server, such as starting and stopping it and checking its status. Here, we'll use the utility to reload the settings files so the changes we make will take effect. Running the command reloads all settings files at once.

You'll need to open and configure a command line prompt the same way you did in Chapter 18 when you learned how to set up and use `psql`. After you launch a command prompt, use one of the following commands to reload, replacing the path with the path to the PostgreSQL data directory:

On Windows, use `pg_ctl reload -D "C:\path\to\data\directory\"`.

On macOS or Linux, use `pg_ctl reload -D '/path/to/data/directory/'`.

To find the location of your PostgreSQL data directory, run the query in Listing 19-10.

```
SHOW data_directory;
```

Listing 19-10: Showing the location of the data directory

Place the path after the `-D` argument, between double quotes on Windows and single quotes on macOS or Linux. You run this command on your system's command prompt, not inside the `psql` application. Enter the command and press ENTER; it should respond with the message `server signaled`. The settings files will be reloaded, and changes should take effect.

If you've changed settings that require a server restart, replace `reload` in Listing 19-10 with `restart`.

NOTE

*On Windows, you may need to run Command Prompt with administrator privileges to execute `pg_ctl` statements. Navigate to Command Prompt in your Start menu, right-click, and select **Run as Administrator**.*

Backing Up and Restoring Your Database

You might want to back up your entire database either for safekeeping or for transferring data to a new or upgraded server. PostgreSQL offers command line tools that make backup and restore operations easy. The next few sections show examples of how to export data from a database or a single table to a file, as well as how to restore data from an export files.

Using pg_dump to Export a Database or Table

The PostgreSQL command line tool `pg_dump` creates an output file that contains all the data from your database; SQL commands for re-creating tables, views, functions, and other database objects; and commands for loading the data into tables. You can also use `pg_dump` to save only selected tables in your database. By default, `pg_dump` outputs a text file; I'll discuss an alternate custom compressed format first and then discuss other options.

To export the `analysis` database we've used for our exercises to a file, run the command in Listing 19-11 at your system's command prompt (not in `psql`).

```
pg_dump -d analysis -U user_name -Fc -v -f
analysis_backup.dump
```

Listing 19-11: Exporting the `analysis` database with `pg_dump`

Here, we start the command with `pg_dump` and use similar connection arguments as with `psql`. We specify the database to export with the `-d` argument, followed by the `-U` argument and your username. Next, we use the `-Fc` argument to specify that we want to generate this export in a custom PostgreSQL compressed format and the `-v` argument to generate verbose output. Then we use the `-f` argument to direct the output of `pg_dump` to a text file named *analysis_backup.dump*. To place the file in a directory other than the one your terminal prompt is currently open to, you can specify the complete directory path before the filename.

When you execute the command, depending on your installation, you might see a password prompt. Fill in that password, if prompted. Then, depending on the size of your database, the command could take a few minutes to complete. You'll see a series of messages about the objects the command is reading and outputting. When it's done, it should return you to a new command prompt, and you should see a file named *analysis_backup.dump* in your current directory.

To limit the export to one or more tables that match a particular name, use the `-t` argument followed by the name of the table in single quotes. For example, to back up just the `train_rides` table, use the following command:

```
pg_dump -t 'train_rides' -d analysis -U user_name -Fc -v -f
train_backup.dump
```

Now let's look at how to restore the data from the export file, and then we'll explore additional `pg_dump` options.

Restoring a Database Export with pg_restore

The `pg_restore` utility restores data from your exported database file. You might need to restore your database when migrating data to a new server or when upgrading to a new major version of PostgreSQL. To restore the `analysis` database (assuming you're on a server where `analysis` doesn't exist), at the command prompt, run the command in Listing 19-12.

```
pg_restore -C -v -d postgres -U user_name
analysis_backup.dump
```

Listing 19-12: Restoring the `analysis` database with `pg_restore`

After `pg_restore`, you add the `-C` argument, which tells the utility to create the `analysis` database on the server. (It gets the database name from the export file.) Then, as you saw previously, the `-v` argument provides verbose output, and `-d` specifies the name of the database to connect to, followed by the `-U` argument and your username. Press ENTER, and the restore will begin. When it's done, you should be able to view your restored database via `psql` or in pgAdmin.

Exploring Additional Backup and Restore Options

You can configure `pg_dump` with multiple options to include or exclude certain database objects, such as tables matching a name pattern, or to specify the output format. For example, when we backed up the `analysis` database, we specified the `-Fc` argument with `pg_dump` to generate the backup in a custom PostgreSQL compressed format. By excluding the `-Fc` argument, the utility will output in plain text, and you can view the contents of the backup with a text editor. For details, check the full `pg_dump` documentation at *https://www.postgresql.org/docs/current/app-pgdump.html*. For corresponding restore options, check the `pg_restore` documentation at *https://www.postgresql.org/docs/current/app-pgrestore.html*.

You also may want to explore the `pg_basebackup` command, which can back up multiple databases running on a PostgreSQL server. See *https://www.postgresql.org/docs/current/app-pgbasebackup.html* for details. An even more robust backup solution is pgBackRest (*https://pgbackrest.org/*), a free, open source application with options such as cloud integration for storage and the ability to create full, incremental, or differential backups.

Wrapping Up

In this chapter, you learned how to track and conserve space in your databases using the `VACUUM` feature in PostgreSQL. You also learned how to change system settings as well as back up and restore databases using other command line tools. You may not need to perform these tasks every day, but the maintenance tricks you learned here can help enhance the performance of your databases. Note that this is not a comprehensive overview of the topic; see the appendix for more resources on database maintenance.

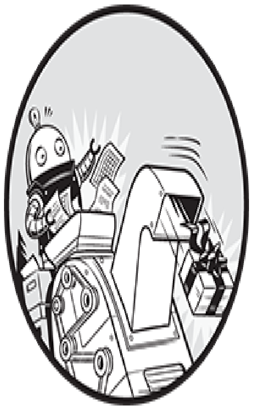
In the next and final chapter of this book, I'll share guidelines for identifying hidden trends and telling an effective story using your data.

TRY IT YOURSELF

Using the techniques you learned in this chapter and earlier in the book, create a database and add a small table and some data. Then back up the database, delete it, and restore it using `pg_dump` and `pg_restore`.

If you make your backup using the default text format instead of compressed, you can use a text editor to explore the file created by `pg_dump` to examine how it organizes the statements to create objects and insert data.

20
TELLING YOUR DATA'S STORY



Learning SQL can be fun in and of itself, but it serves a greater purpose: it helps unearth the stories in your data. As you've learned, SQL gives you the tools to find interesting trends, insights, or anomalies in your data and then make smart decisions based on what you've learned. But how do you identify these trends from just a collection of rows and columns? And how can you glean meaningful insights from these trends after identifying them?

In this chapter, I outline a process I've used as a journalist and product developer to discover stories in data and communicate my findings. I'll start with how to generate ideas by asking good questions and gathering and exploring data. Then I explain the analysis process, which culminates in presenting your findings clearly. Identifying trends in your dataset and creating a narrative of your findings sometimes requires considerable experimentation and enough fortitude to weather the occasional dead end. Regard these tips as less of a checklist and more of a guideline to help ensure a thorough analysis that minimizes mistakes.

Start with a Question

Curiosity, intuition, or sometimes just dumb luck can often spark ideas for data analysis. If you're a keen observer of your surroundings, you might notice changes in your community over time and wonder if you can measure that change. Consider your local real estate market. If you see "For Sale" signs popping up around town more than usual, you might start asking questions. Is there a dramatic increase in home sales this year compared with last year? If so, by how much? Which neighborhoods are riding the wave? These questions create a great opportunity for data analysis. If you're a journalist, you might find a story. If you run a business, you might see a marketing opportunity.

Likewise, if you surmise that a trend is occurring in your industry, confirming it might provide you with a business opportunity. For example, if you suspect that sales of a particular product are sluggish, you can analyze data to confirm the hunch and adjust inventory or marketing efforts appropriately.

Keep track of these ideas and prioritize them according to their potential value. Analyzing data to satisfy your curiosity is perfectly fine, but if the answers can make your institution more effective or your company more profitable, that's a sign they're worth pursuing.

Document Your Process

Before you delve into analysis, consider how to make your process transparent and reproducible. For the sake of credibility, others in your organization as well as those outside it should be able to reproduce your work. In addition, make sure you document enough of your process so that if you set the project aside for several weeks, you won't have trouble getting up to speed when you return to it.

There isn't one right way to document your work. Taking notes on research or creating step-by-step SQL queries that another person could follow to replicate your data import, cleaning, and analysis can make it easier for others to verify your findings. Some analysts store notes and code in a text file. Others use version control systems, such as GitHub, or work in code notebooks. What’s important is you create a system of documentation and use it consistently.

Gather Your Data

After you’ve hatched an idea for analysis, the next step is to find data that relates to the trend or question. If you’re working in an organization that already has its own data on the topic, lucky you—you’re set! In that case, you might be able to tap internal marketing or sales databases, customer relationship management (CRM) systems, or subscriber or event registration data. But if your topic encompasses broader issues involving demographics, the economy, or industry-specific subjects, you’ll need to do some digging.

A good place to start is to ask experts about the sources they use. Analysts, government decision-makers, and academics can point you to available data and describe its usefulness. Federal, state, and local governments, as you’ve seen throughout the book, produce volumes of data on all kinds of topics. In the United States, check out the federal government’s data catalog site at *https://www.data.gov/* or individual federal agency sites, such as the National Center for Education Statistics (NCES) at *https://nces.ed.gov/* or the Bureau of Labor Statistics at *https://www.bls.gov/*.

You can also browse local government websites. Any time you see a form for users to fill out or a report formatted in rows and columns, those are signs that structured data might be available for analysis. All is not lost if you have access only to unstructured data, though—as you learned in Chapter 14, you can even mine unstructured data, such as text files, for analysis.

If the data you want to analyze was collected over multiple years, I recommend examining five or ten years or more, instead of just one or two, if possible. Analyzing a snapshot of data collected over a month or a year can yield interesting results, but many trends play out over a longer period of time and may not be evident if you look at a single year of data. I’ll discuss this further in the section “Identify Key Indicators and Trends over Time.”

No Data? Build Your Own Database

Sometimes, no one has the data you need in a format you can use. If you have time, patience, and a methodology, you might be able to build your own dataset. That is what my *USA Today* colleague Robert Davis and I did when we wanted to study issues related to the deaths of college students on campuses in the United States. Not a single organization—not the schools or state or federal officials—could tell us how many college students were dying each year from accidents, overdoses, or illnesses on campus. We decided to collect our own data and structure the information into tables in a database.

We started by researching news articles, police reports, and lawsuits related to student deaths. After finding reports of more than 600 student deaths from 2000 to 2005, we followed up with interviews with education experts, police, school officials, and parents. From each report, we cataloged details such as each student's age, school, cause of death, year in school, and whether drugs or alcohol played a role. Our findings led to the publication of the article "In College, First Year Is by Far the Riskiest" in *USA Today* in 2006. The story featured the key finding from the analysis of our SQL database: freshmen were particularly vulnerable and accounted for the highest percentage of the student deaths we studied.

You too can create a database if you lack the data you need. The key is to identify the pieces of information that matter and then systematically collect them.

Assess the Data's Origins

After you've identified a dataset, find as much information about its origins and maintenance methods as you can. Governments and institutions gather data in all sorts of ways, and some methods produce data that is more credible and standardized than others.

For example, you've already seen that US Department of Agriculture (USDA) food producer data included the same company names spelled in multiple ways. It's worth knowing why. (Perhaps the data is manually copied from a written form to a computer.) Similarly, the New York City taxi data you analyzed in Chapter 12 records the start and end times of each trip. This begs the question of when the timer starts and stops—when the passenger gets in and out of the vehicle, or is there some other trigger? You should know these details not only to draw better conclusions from analysis but also to pass them along to others who might be interpreting your analysis.

The origins of a dataset might also affect how you analyze the data and report your findings. For example, with US Census Bureau data, it's important to know that the decennial census conducted every 10 years is a complete count of the population, whereas the American Community Survey (ACS) is drawn from only a sample of households. As a result, ACS counts have a margin of error, but the decennial census doesn't. It would be irresponsible to report on the ACS without considering that the margin of error could render differences between numbers insignificant.

Interview the Data with Queries

Once you have your data, understand its origins, and have it loaded into your database, you can explore it with queries. Throughout the book, I call this step *interviewing data*, which is what you should do to find out more about the contents of your data and whether they contain any red flags.

A good place to start is with aggregates. Counts, sums, sorting, and grouping by column values should reveal minimum and maximum values, potential issues with duplicate entries, and a sense of the general scope of your data. If your database contains multiple, related tables, try joins to make sure you understand how the tables relate. Using `LEFT JOIN` and `RIGHT JOIN`, as you learned in Chapter 7, should show whether key values from one table are missing in another. That may or may not be a concern, but at least you'll be able to identify potential problems to address. Jot down a list of questions or concerns you have, and then move on to the next step.

Consult the Data's Owner

After exploring your database and forming early conclusions about the quality and trends you observed, take time to bring questions or concerns to a person who knows the data well. That person could work at the government agency or firm that gave you the data, or the person might be an analyst who has worked with the data before. This step is your chance to clarify your understanding of the data, verify initial findings, and discover whether the data has any issues that make it unsuitable for your needs.

For example, if you're querying a table and notice values in columns that seem to be gross outliers (such as dates in the future for events that were supposed to have happened in the past), you should ask about that discrepancy. If you expect to find someone's name in a table (perhaps even your own name) and it's not there, that should prompt another question. Is it possible you don't have the whole dataset, or is there a problem with data collection?

The goal is to get expert help to do the following:

Understand the limits of the data. Make sure you know what the data includes, what it excludes, and any caveats about content that might affect how you perform your analysis.

Make sure you have a complete dataset. Verify that you have all the records you should expect to see and that if any data is missing, you understand why.

Determine whether the dataset suits your needs. Consider looking elsewhere for more reliable data if your source acknowledges problems with the data's quality.

Every dataset and situation is unique, but consulting another user or owner of the data can help you avoid unnecessary missteps.

Identify Key Indicators and Trends over Time

When you're satisfied that you understand the data and are confident in its trustworthiness, completeness, and appropriateness to your analysis, the next step is to run queries to identify key indicators and, if possible, trends over time.

Your goal is to unearth data that you can summarize in a sentence or present as a slide in a presentation. An example of a finding would be something like this: “After five years of declines, the number of people enrolling in Widget University has increased by 5 percent for two consecutive semesters.”

To identify this type of trend, you’ll follow a two-step process:

- Choose an indicator to track. In census data, it might be the percentage of the population that is over age 60. Or in the New York City taxi data, it could be the median number of weekday trips over the span of one year.
- Track that indicator over multiple years to see how it has changed, if at all.

In fact, these are the steps we used in Chapter 7 to apply percent change calculations to multiple years of census data contained in joined tables. In that case, we looked at the change in population in counties between 2010 and 2019. The population estimate was the key indicator, and the percent change showed the trend over the nine-year span for each county.

One caveat about measuring change over time: even when you see a dramatic change between any two years, it’s worth digging into as many years’ worth of data as possible to understand the shorter-term change in the context of a long-term trend. Any year-to-year change might seem dramatic, but seeing it in context of multiyear activity can help you assess its true significance.

For example, the US National Center for Health Statistics releases data on the number of babies born each year. As a data nerd, this is one of the indicators I like to keep tabs on, because births often reflect broader trends in culture or the economy. *Figure 20-1* shows the annual number of births from 1910 to 2020.

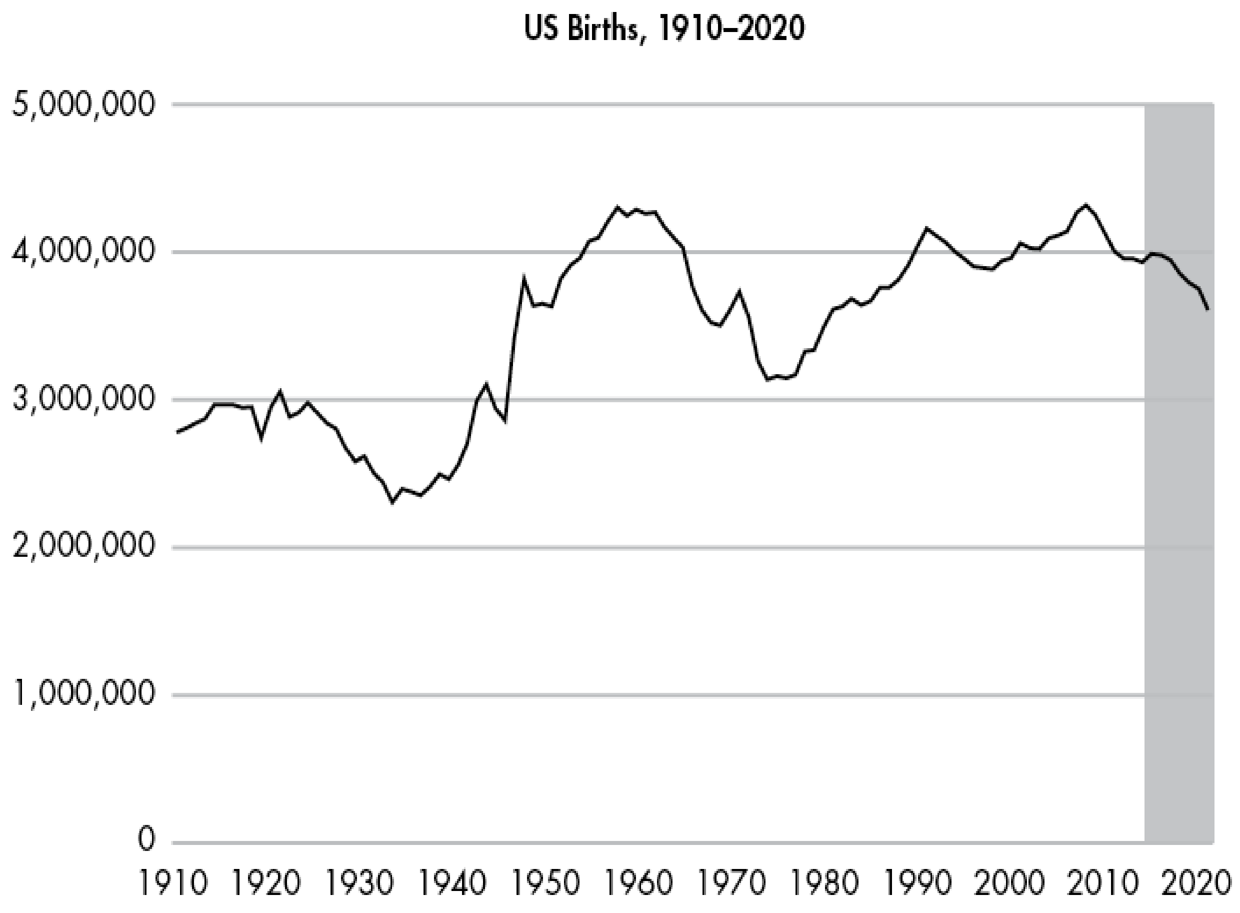


Figure 20-1: US births from 1910 to 2020. Source: US National Center for Health Statistics

Looking at only the last five years of this graph (shaded in gray), we see that the number of births has declined steadily, to 3.61 million in 2020 from 3.95 million in 2016. The recent drops are indeed noteworthy (reflecting continuing decreases in birth rates and an aging population). But in the long-term context, we see that the nation has experienced several baby booms and busts in the past 100 years. One example you can see in *Figure 20-1* is the major rise in the mid-1940s following World War II, which signaled the start of the Baby Boom generation.

By identifying key indicators and looking at change over time, both short term and long term, you might uncover one or more findings worth presenting to others or acting on.

NOTE

Any time you work with data from a survey, poll, or other sample, it's important to test for statistical significance. Are the results actually a trend or just the result of chance? Significance testing is a statistical concept beyond the scope of this book but one that data analysts should know. See the appendix for PostgreSQL resources for advanced statistics.

Ask Why

Data analysis can tell you what happened, but it doesn't always indicate why something happened. To learn the *why*, it's worth revisiting the data with experts in the topic or the owners of the data. In the US births data, it's easy to calculate year-to-year percent change from those numbers. But the data doesn't tell us why births steadily increased from the early 1980s to 1990. For that information, you could consult a demographer who would most likely explain that the rise in births during those years coincided with more Baby Boomers entering their child-bearing years.

As you share your findings and methodology with experts, ask them to note anything that seems unlikely or worthy of further examination. For the findings that they can corroborate, ask them to help you understand the forces behind those findings. If they're willing to be cited, you can use their comments to supplement your report or presentation. Quoting experts' insights about trends in this way is a standard approach journalists use.

Communicate Your Findings

How you share the results of your analysis depends on your role. A student might present their results in a paper or dissertation. A person who works in a corporate setting might present their findings using PowerPoint, Keynote, or Google Slides. A journalist might write a story or produce a data visualization. Regardless of the end product, here are tips for presenting the information well, using a fictional home sales analysis as an example:

Identify an overarching theme based on your findings. Make the theme the title of your presentation, paper, or visualization. For example, you might title a presentation on real estate “Home Sales Rise in Suburban Neighborhoods, Fall in Cities.”

Present overall numbers to show the general trend. Highlight the key findings from your analysis. For example, “All suburban neighborhoods saw sales rise 5 percent each of the last two years, reversing three years of declines. Meanwhile, city neighborhoods saw a 2 percent decline.”

Highlight specific examples that support the trend. Describe one or two relevant cases. For example, “In Smithtown, home sales increased 15 percent following the relocation of XYZ Corporation’s headquarters last year.”

Acknowledge examples counter to the overall trend. Use one or two relevant cases here as well. For example, “Two city neighborhoods did show growth in home sales: Arvis (up 4.5%) and Zuma (up 3%).”

Stick to the facts. Never distort or exaggerate any findings.

Provide expert insights. Use quotes or citations.

Visualize numbers using bar charts. line charts, or maps. Tables are helpful for giving your audience specific numbers, but it’s easier to understand trends from a visualization.

Cite the source of the data and what your analysis includes or omits. Provide dates covered, the name of the provider, and any distinctions that affect the analysis, for example, “Based on Walton County tax filings in 2022 and 2023. Excludes commercial properties.”

Share your data. Post data online for download, including a description of the steps you took to analyze it. Nothing says transparency more than sharing your data with others so they can perform their own analysis and corroborate your findings.

Generally, a short presentation that communicates your findings clearly and succinctly, and then invites dialogue from your audience, works best. Of course, you can follow your own preferred pattern for working with data and presenting your conclusions. But over the years, these steps have helped me avoid data errors and mistaken assumptions.

Wrapping Up

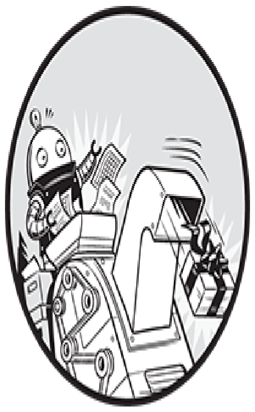
At last, you've reached the end of our practical exploration of SQL! Thank you for reading this book, and I welcome your suggestions and feedback via email at *practicalsqlbook@gmail.com*. At the end of this book is an appendix that lists additional PostgreSQL-related tools you might want to try.

I hope you've come away with data analysis skills you can start using immediately on the everyday data you encounter. More importantly, I hope you've seen that each dataset has a story, or several stories, to tell. Identifying and telling these stories is what makes working with data worthwhile; it's more than just combing through a collection of rows and columns. I look forward to hearing about what you discover!

TRY IT YOURSELF

It's your turn to find and tell a story using the SQL techniques we've covered. Using the process outlined in this chapter, consider a local or national topic and search for available data. Assess its quality, the questions it might answer, and its timeliness. Consult with an expert who knows the data and the topic well. Load the data into PostgreSQL and interview it using aggregate queries and filters. What trends can you discover? Summarize your findings in a short presentation.

APPENDIX
ADDITIONAL POSTGRESQL RESOURCES



This appendix contains resources to help you stay informed about PostgreSQL developments, find additional software, and get help. Because software resources are likely to change, I'll maintain a copy of this appendix at the GitHub repository that contains all the book's resources. You can find a link to GitHub via *https://nostarch.com/practical-sql-2nd-edition/*.

PostgreSQL Development Environments

Throughout the book, we've used the graphical user interface pgAdmin to connect to PostgreSQL, run queries, and view database objects. Although pgAdmin is free, open source, and popular, it's not your only choice for working with PostgreSQL. The wiki entry "PostgreSQL Clients" at *https://wiki.postgresql.org/wiki/PostgreSQL_Clients* catalogs many alternatives.

The following list shows several tools I've tried, including free and paid options. The free tools work well for general analysis work. If you wade deeper into database development, you might want to upgrade to the paid options, which typically offer advanced features and support.

Beekeeper Studio Free and open source GUI for PostgreSQL, MySQL, Microsoft SQL Server, SQLite, and other platforms. Beekeeper works on Windows, macOS, and Linux and features one of the more refined app designs among database GUIs (see *https://www.beekeeperstudio.io/*).

DBeaver Described as a “universal database tool” that works with PostgreSQL, MySQL, and many other databases, DBeaver includes a visual query builder, code autocompletion, and other advanced features. There are paid and free versions for Windows, macOS, and Linux (see *https://dbeaver.com/*).

DataGrip A SQL development environment that offers code completion, bug detection, and suggestions for streamlining code, among many other features. It’s a paid product, but the company, JetBrains, offers discounts and free versions for students, educators, and nonprofits (see *https://www.jetbrains.com/datagrip/*).

Navicat A richly featured SQL development environment with versions that support PostgreSQL as well as other databases, including MySQL, Oracle, MongoDB, and Microsoft SQL Server. There is no free version of Navicat, but the company offers a 14-day free trial (see *https://www.navicat.com/*).

Postbird A simple cross-platform PostgreSQL GUI for writing queries and viewing objects. Free and open source (see *https://github.com/Paxa/postbird/*).

Postico A macOS-only client from the maker of Postgres.app that takes its cues from Apple design. The full version is paid, but a restricted-feature version is available with no time limit (see *https://eggerapps.at/postico/*).

A trial version can help you decide whether the product is right for you.

PostgreSQL Utilities, Tools, and Extensions

You can expand the capabilities of PostgreSQL via numerous third-party utilities, tools, and extensions. These range from additional backup and import/export options to improved formatting for the command line to powerful statistics packages. You'll find a curated list online at *https://github.com/dhamaniasad/awesome-postgres/*, but here are several to highlight:

Devart Excel Add-in for PostgreSQL An Excel add-in that lets you load and edit data from PostgreSQL directly in Excel workbooks (see *https://www.devart.com/excel-addins/postgresql.html*).

MADlib A machine learning and analytics library for large data sets that integrates with PostgreSQL (see *https://madlib.apache.org/*).

pgAgent A job manager that lets you run queries at scheduled times, among other tasks (see *https://www.pgadmin.org/docs/pgadmin4/latest/pgagent.html*).

pgBackRest An advanced database backup and restore management tool (see *https://pgbackrest.org/*).

`pgcli` A substitute command-line interface for `psql` that includes autocompletion and syntax highlighting (see *https://github.com/dbcli/pgcli/*).

pgRouting Enables a PostGIS-enabled PostgreSQL database to perform network analysis tasks, such as finding driving distance along roadways (see *https://pgrouting.org/*).

PL/R A loadable procedural language that provides the ability to use the R statistical programming language within PostgreSQL functions and triggers (see *https://www.joeconway.com/plr.html*).

`pspg` Formats the output of `psql` into sortable, scrollable tables with support for several color themes (see *https://github.com/okbob/pspg/*).

PostgreSQL News and Community

Now that you're a bona fide PostgreSQL user, it's wise to stay on top of community news. The PostgreSQL development team updates the software on a regular basis, and changes might affect code you've written or tools you're using. You might even find new opportunities for analysis.

Here's a collection of online resources to help you stay informed:

Crunchy Data blog Posts from the team at Crunchy Data, which provides enterprise PostgreSQL support and solutions (see *https://blog.crunchydata.com/blog/*).

The EDB Blog Posts from the team at EDB, a PostgreSQL services company that provides the Windows installer referenced in this book and also leads development of pgAdmin (see *https://www.enterprisedb.com/blog/*).

Planet PostgreSQL Aggregates blog posts and announcements from the database community (see *https://planet.postgresql.org/*).

Postgres Weekly An email newsletter that rounds up announcements, blog posts, and product announcements (see *https://postgresweekly.com/*).

PostgreSQL mailing lists These lists are useful for asking questions of community experts. The pgsql-novice and pgsql-general lists are particularly good for beginners, although note that email volume can be heavy (see *https://www.postgresql.org/list/*).

PostgreSQL news archive Official news from the PostgreSQL team (see *https://www.postgresql.org/about/newsarchive/*).

PostgreSQL nonprofits PostgreSQL-related charitable organizations include the United States PostgreSQL Association and PostgreSQL Europe. Both provide education, events, and advocacy around the product (see *https://postgresql.us/* and *https://www.postgresql.eu/*).

PostgreSQL user groups A list of community groups that offer meetups and other activities (see *https://www.postgresql.org/community/user-groups/*).

PostGIS blog Announcements and updates about the PostGIS extension (see *https://postgis.net/blog/*).

Additionally, I recommend paying attention to developer notes for any of the PostgreSQL-related software you use, such as pgAdmin.

Documentation

Throughout this book, I've made frequent reference to pages in the official PostgreSQL documentation. You can find documentation for each version of the software along with an FAQ and wiki on the main page at *https://www.postgresql.org/docs/*. It's worth reading sections of the manual as you learn more about a topic, such as indexes, or search for all the options that come with functions. In particular, the "Preface," "Tutorial," and "SQL Language" sections cover much of the material presented in the book's chapters.

Other good resources for documentation are the Postgres Guide at *http://postgresguide.com/* and Stack Overflow, where you can find questions and answers posted by developers at *https://stackoverflow.com/questions/tagged/postgresql/*. You can also check out the Q&A site for PostGIS at *https://gis.stackexchange.com/questions/tagged/postgis/*.

Index

Please note that index links to approximate location of each term.