```
    ->  Parallel Seq Scan on new_york_addresses
(cost=0.00..13873.78 1
     rows=1293 width=46) (actual time=2.362..367.258
rows=1112 loops=3)
        Filter: (street = 'BROADWAY'::text)
        Rows Removed by Filter: 312346
Planning Time: 0.401 ms
Execution Time: 389.232 ms 2
```

Not all the output is relevant here, so I won't decode it all, but two lines are pertinent. The first indicates that to find any rows where `street = 'BROADWAY'`, the database will conduct a sequential scan 1 of the table. That's a synonym for a full table scan: the database will examine each row and remove any where `street` doesn't match `BROADWAY`. The execution time (on my computer about 389 milliseconds) 2 is how long the query took to run. Your time will depend on factors including your computer hardware.

For the test, run each query in *Listing 8-13* several times and record the fastest execution time for each. You'll notice that execution times for the same query will vary slightly on each run. That can be the result of several factors, from other processes running on the server to the effect of data being held in memory after a prior run of the query.

Adding the Index

Now, let's see how adding an index changes the query's search method and execution time. *Listing 8-14* shows the SQL statement for creating the index with PostgreSQL.

```
CREATE INDEX street_idx ON new_york_addresses (street);
```

Listing 8-14: Creating a B-tree index on the `new_york_addresses` *table*

Notice that it's similar to the commands for creating constraints. We give the `CREATE INDEX` keywords followed by a name we choose for the index, in this case `street_idx`. Then `ON` is added, followed by the target table and column.

Execute the `CREATE INDEX` statement, and PostgreSQL will scan the values in the `street` column and build the index from them. We need to create the index only once. When the task finishes, rerun each of the three queries in *Listing 8-13* and record the execution times reported by `EXPLAIN ANALYZE`. Here's an example:

```
Bitmap Heap Scan on new_york_addresses  (cost=76.47..6389.39
rows=3103 width=46) (actual time=1.355..4.802 rows=3336
loops=1)
  Recheck Cond: (street = 'BROADWAY'::text)
  Heap Blocks: exact=2157
  ->  Bitmap Index Scan on street_idx  (cost=0.00..75.70
rows=3103 width=0) 1
      (actual time=0.950..0.950 rows=3336 loops=1)
        Index Cond: (street = 'BROADWAY'::text)
Planning Time: 0.109 ms
Execution Time: 5.113 ms 2
```

Do you notice a change? First, we see that the database is now using an index scan on `street_idx` 1 instead of visiting each row in a sequential scan. Also, the query speed is now markedly faster 2. *Table 8-1* shows the fastest execution times (rounded) from my computer before and after adding the index.

Table 8-1: *Measuring Index Performance*

Query filter	Before index	After index
`WHERE street = 'BROADWAY'`	92 ms	5 ms
`WHERE street = '52 STREET'`	94 ms	1 ms
`WHERE street = 'ZWICKY AVENUE'`	93 ms	<1 ms

The execution times are much, much better, nearly a tenth of a second faster or more per query. Is a tenth of a second that impressive? Well, whether you're seeking answers in data using repeated querying or creating a database system for thousands of users, the time savings adds up.

If you ever need to remove an index from a table—perhaps if you're testing the performance of several index types—use the `DROP INDEX` command followed by the name of the index to remove.

Considerations When Using Indexes

You’ve seen that indexes have significant performance benefits, so does that mean you should add an index to every column in a table? Not so fast! Indexes are valuable, but they’re not always needed. In addition, they do enlarge the database and impose a maintenance cost on writing data. Here are a few tips for judging when to uses indexes:

Consult the documentation for the database system you’re using to learn about the kinds of indexes available and which to use on particular data types. PostgreSQL, for example, has five more index types in addition to B-tree. One, called GiST, is particularly suited to the geometry data types discussed later in the book. Full-text search, which you’ll learn in Chapter 14, also benefits from indexing.

Consider adding indexes to columns you’ll use in table joins. Primary keys are indexed by default in PostgreSQL, but foreign key columns in related tables are not and are a good target for indexes.

An index on a foreign key will help avoid an expensive sequential scan during a cascading delete.

Add indexes to columns that will frequently end up in a query `WHERE` clause. As you’ve seen, search performance is significantly improved via indexes.

Use `EXPLAIN ANALYZE` to test the performance under a variety of configurations. Optimization is a process! If an index isn’t being used by the database—and it’s not backing up a primary key or other constraint—you can drop it to reduce the size of your database and speed up inserts, updates, and deletes.

Wrapping Up

With the tools you’ve added to your toolbox in this chapter, you’re ready to ensure that the databases you build or inherit are best suited for your collection and exploration of data. It’s crucial to define constraints that match the data and the expectation of users by not allowing values that don’t make sense, making sure values are filled in, and setting up proper relationships between tables. You’ve also learned how to make your queries run faster and how to consistently organize your database objects. That’s a boon for you and for others who share your data.

This chapter concludes the first part of the book, which focused on giving you the essentials to dig into SQL databases. We'll continue building on these foundations as we explore more complex queries and strategies for data analysis. In the next chapter, we'll use SQL aggregate functions to assess the quality of a dataset and get usable information from it.

TRY IT YOURSELF

Are you ready to test yourself on the concepts covered in this chapter? Consider the following two tables from a database you're making to keep track of your vinyl LP collection. Start by reviewing these `CREATE TABLE` statements:

```
CREATE TABLE albums (
    album_id bigint GENERATED ALWAYS AS IDENTITY,
    catalog_code text,
    title text,
    artist text,
    release_date date,
    genre text,
    description text
);

CREATE TABLE songs (
    song_id bigint GENERATED ALWAYS AS IDENTITY,
    title text,
    composers text,
    album_id bigint
);
```

The `albums` table includes information specific to the overall collection of songs on the disc. The `songs` table catalogs each track on the album. Each song has a title and a column for its composers, who might be different than the album artist.

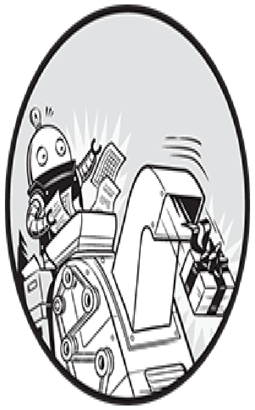
Use the tables to answer these questions:

Modify these `CREATE TABLE` statements to include primary and foreign keys plus additional constraints on both tables. Explain why you made your choices.

Instead of using `album_id` as a surrogate key for your primary key, are there any columns in `albums` that could be useful as a natural key? What would you have to know to decide?

To speed up queries, which columns are good candidates for indexes?

9
EXTRACTING INFORMATION BY GROUPING AND SUMMARIZING



Every dataset tells a story, and it's the data analyst's job to find it. In Chapter 3, you learned about interviewing data using `SELECT` statements by sorting columns, finding distinct values, and filtering results. You've also learned the fundamentals of SQL math, data types, table design, and joining tables. With these tools under your belt, you're ready to glean more insights by using *grouping* and *aggregate functions* to summarize your data.

By summarizing data, we can identify useful information we wouldn't see just by scanning the rows of a table. In this chapter, we'll use the well-known institution of your local library as our example.

Libraries remain a vital part of communities worldwide, but the internet and advancements in library technology have changed how we use them. For example, ebooks and online access to digital materials now have a permanent place in libraries along with books and periodicals.

In the United States, the Institute of Museum and Library Services (IMLS) measures library activity as part of its annual Public Libraries Survey. The survey collects data from about 9,000 library administrative entities, defined by the survey as agencies that provide library services to a particular locality. Some agencies are county library systems, and others are part of school districts. Data on each agency includes the number of branches, staff, books, hours open per year, and so on. The IMLS has been collecting data each year since 1988 and includes all public library agencies in the 50 states plus the District of Columbia and US territories such as American Samoa. (Read more about the program at *https://www.imls.gov/research-evaluation/data-collection/public-libraries-survey/*.)

For this exercise, we'll assume the role of an analyst who just received a fresh copy of the library dataset to produce a report describing trends from the data. We'll create three tables to hold data from the 2018, 2017, and 2016 surveys. (Often, it's helpful to assess multiple years of data to discern trends.) Then we'll summarize the more interesting data in each table and join the tables to see how measures changed over time.

Creating the Library Survey Tables

Let's create the three library survey tables and import the data. We'll use appropriate data types and constraints for each column and add indexes where appropriate. The code and three CSV files are available in the book's resources.

Creating the 2018 Library Data Table

We'll start by creating the table for the 2018 library data. Using the `CREATE TABLE` statement, *Listing 9-1* builds `pls_fy2018_libraries`, a table for the fiscal year 2018 Public Library System Data File from the Public Libraries Survey. The Public Library System Data File summarizes data at the agency level, counting activity at all agency outlets, which include central libraries, branch libraries, and bookmobiles. The annual survey generates two additional files we won't use: one summarizes data at the state level, and the other has data on individual outlets. For this exercise, those files are redundant, but you can read about the data they contain at https://www.imls.gov/sites/default/files/2018_pls_data_file_documentation.pdf.

For convenience, I've created a naming scheme for the tables: `pls` refers to the survey title, `fy2018` is the fiscal year the data covers, and `libraries` is the name of the particular file from the survey. For simplicity, I've selected 47 of the more relevant columns from the 166 in the original survey file to fill the `pls_fy2018_libraries` table, excluding data such as the codes that explain the source of individual responses. When a library didn't provide data, the agency derived the data using other means, but we don't need that information for this exercise.

Listing 9-1 is abbreviated for convenience, as indicated by the `--snip--` noted in the code, but the full version is included with the book's resources.

```
CREATE TABLE pls_fy2018_libraries (
    stabr text NOT NULL,
 1 fscskey text CONSTRAINT fscskey_2018_pkey PRIMARY KEY,
    libid text NOT NULL,
    libname text NOT NULL,
    address text NOT NULL,
    city text NOT NULL,
    zip text NOT NULL,
    --snip--
    longitude numeric(10,7) NOT NULL,
    latitude numeric(10,7) NOT NULL
);

2 COPY pls_fy2018_libraries
  FROM 'C:\YourDirectory\pls_fy2018_libraries.csv'
  WITH (FORMAT CSV, HEADER);

3 CREATE INDEX libname_2018_idx ON pls_fy2018_libraries
  (libname);
```

Listing 9-1: Creating and filling the 2018 Public Libraries Survey table

After finding the code and data file for Listing 9-1, connect to your `analysis` database in pgAdmin and run it. Make sure you remember to change *`C:\YourDirectory\`* to the path where you saved the *pls_fy2018_libraries.csv* file.

First, the code makes the table via `CREATE TABLE`. We assign a primary key constraint to the column named `fscskey` 1, a unique code the data dictionary says is assigned to each library. Because it's unique, present in each row, and unlikely to change, it can serve as a natural primary key.

The definition for each column includes the appropriate data type and `NOT NULL` constraints where the columns have no missing values. The `startdate` and `enddate` columns contain dates, but we've set their data type to `text` in the code; in the CSV file, those columns include nondate values, and our import will fail if we try to use a `date` data type. In Chapter 10, you'll learn how to clean up cases like these. For now, those columns are fine as is.

After creating the table, the `COPY` statement 2 imports the data from a CSV file named *pls_fy2018_libraries.csv* using the file path you provide. We add an index 3 to the `libname` column to provide faster results when we search for a particular library.

Creating the 2017 and 2016 Library Data Tables

Creating the tables for the 2017 and 2016 library surveys follows similar steps. I've combined the code to create and fill both tables in *Listing 9-2*. Note again that the listing shown is truncated, but the full code is in the book's resources at *https://nostarch.com/practical-sql-2nd-edition/*.

Update the file paths in the `COPY` statements for both imports and execute the code.

```
CREATE TABLE pls_fy2017_libraries (
    stabr text NOT NULL,
 1 fscskey text CONSTRAINT fscskey_17_pkey PRIMARY KEY,
    libid text NOT NULL,
    libname text NOT NULL,
    address text NOT NULL,
    city text NOT NULL,
    zip text NOT NULL,
    --snip--
    longitude numeric(10,7) NOT NULL,
    latitude numeric(10,7) NOT NULL
);

CREATE TABLE pls_fy2016_libraries (
        stabr text NOT NULL,
        fscskey text CONSTRAINT fscskey_16_pkey PRIMARY KEY,
        libid text NOT NULL,
        libname text NOT NULL,
        address text NOT NULL,
        city text NOT NULL,
        zip text NOT NULL,
        --snip--
        longitude numeric(10,7) NOT NULL,
        latitude numeric(10,7) NOT NULL
    );

2 COPY pls_fy2017_libraries
    FROM 'C:\YourDirectory\pls_fy2017_libraries.csv'
    WITH (FORMAT CSV, HEADER);

    COPY pls_fy2016_libraries
    FROM 'C:\YourDirectory\pls_fy2016_libraries.csv'
    WITH (FORMAT CSV, HEADER);

3 CREATE INDEX libname_2017_idx ON pls_fy2017_libraries
    (libname);
    CREATE INDEX libname_2016_idx ON pls_fy2016_libraries
    (libname);
```

Listing 9-2: Creating and filling the 2017 and 2016 Public Libraries Survey tables

We start by creating the two tables, and in both we again use `fscskey` 1 as the primary key. Next, we run `COPY` commands 2 to import the CSV files to the tables, and, finally, we create an index on the `libname` column 3 in both tables.

As you review the code, you'll notice that the three tables have an identical structure. Most ongoing surveys will have a handful of year-to-year changes because the makers of the survey either think of new questions or modify existing ones, but the columns I've selected for these three tables are consistent. The documentation for the survey years is at https://www.imls.gov/research-evaluation/data-collection/public-libraries-survey/. Now, let's mine this data to discover its story.

Exploring the Library Data Using Aggregate Functions

Aggregate functions combine values from multiple rows, perform an operation on those values, and return a single result. For example, you might return the average of values with the `avg()` aggregate function, as you learned in Chapter 6. Some aggregate functions are part of the SQL standard, and others are specific to PostgreSQL and other database managers. Most of the aggregate functions used in this chapter are part of standard SQL (a full list of PostgreSQL aggregates is at *https://www.postgresql.org/docs/current/functions-aggregate.html*).

In this section, we'll work through the library data using aggregates on single and multiple columns and then explore how you can expand their use by grouping the results they return with values from additional columns.

Counting Rows and Values Using count()

After importing a dataset, a sensible first step is to make sure the table has the expected number of rows. The IMLS documentation says the file we imported for the 2018 data has 9,261 rows; 2017 has 9,245; and 2016 has 9,252. The difference likely reflects library openings, closings, or mergers. When we count the number of rows in those tables, the results should match those counts.

The `count()` aggregate function, which is part of the ANSI SQL standard, makes it easy to check the number of rows and perform other counting tasks. If we supply an asterisk as an input, such as `count(*)`, the asterisk acts as a wildcard, so the function returns the number of table rows regardless of whether they include `NULL` values. We do this in all three statements in *Listing 9-3*.

```
SELECT count(*)
FROM pls_fy2018_libraries;

SELECT count(*)
FROM pls_fy2017_libraries;

SELECT count(*)
FROM pls_fy2016_libraries;
```

Listing 9-3: Using `count()` for table row counts

Run each of the commands in Listing 9-3 one at a time to see the table row counts. For `pls_fy2018_libraries`, the result should be as follows:

```
count
-----
 9261
```

For `pls_fy2017_libraries`, you should see the following:

```
count
-----
 9245
```

Finally, the result for `pls_fy2016_libraries` should be this:

```
count
-----
 9252
```

All three results match the number of rows we expected. This is a good first step because it will alert us to issues such as missing rows or a case where we might have imported the wrong file.

NOTE

*You can also check the row count using the pgAdmin interface, but it's clunky. Right-clicking the table name in pgAdmin's object browser and selecting **View/Edit Data▸All Rows** executes a SQL query for all rows. Then, a pop-up message in the results pane shows the row count, but it disappears after a few seconds.*

Counting Values Present in a Column

If we supply a column name instead of an asterisk to `count()`, it will return the number of rows that are not `NULL`. For example, we can count the number of non-`NULL` values in the `phone` column of the `pls_fy2018_libraries` table using `count()` as in Listing 9-4.

```
SELECT count(phone)
FROM pls_fy2018_libraries;
```

Listing 9-4: Using `count()` for the number of values in a column

The result shows 9,261 rows have a value in `phone`, the same as the total rows we found earlier.

```
count
-----
 9261
```

This means every row in the `phone` column has a value. You may have suspected this already, given that the column has a `NOT NULL` constraint in the `CREATE TABLE` statement. But running this check is worthwhile because the absence of values might influence your decision on whether to proceed with analysis at all. To fully vet the data, checking with topical experts and digging deeper into the data is usually a good idea; I recommend seeking expert advice as part of a broader analysis methodology (for more on this topic, see Chapter 20).

Counting Distinct Values in a Column

In Chapter 3, I covered the `DISTINCT` keyword—part of the SQL standard—which with `SELECT` returns a list of unique values. We can use it to see unique values in a single column, or we can see unique combinations of values from multiple columns. We also can add `DISTINCT` to the `count()` function to return a count of distinct values from a column.

Listing 9-5 shows two queries. The first counts all values in the 2018 table's `libname` column. The second does the same but includes `DISTINCT` in front of the column name. Run them both, one at a time.

```
SELECT count(libname)
FROM pls_fy2018_libraries;
```

```
SELECT count(DISTINCT libname)
FROM pls_fy2018_libraries;
```

Listing 9-5: Using `count()` for the number of distinct values in a column

The first query returns a row count that matches the number of rows in the table that we found using Listing 9-3:

```
count
-----
 9261
```

That's good. We expect to have the library agency name listed in every row. But the second query returns a smaller number:

```
count
-----
 8478
```

Using `DISTINCT` to remove duplicates reduces the number of library names to the 8,478 that are unique. Closer inspection of the data shows that 526 library agencies in the 2018 survey shared their name with one or more other agencies. Ten library agencies are named `OXFORD PUBLIC LIBRARY`, each one in a city or town named Oxford in different states, including Alabama, Connecticut, Kansas, and Pennsylvania, among others. We'll write a query to see combinations of distinct values in the "Aggregating Data Using GROUP BY" section.

Finding Maximum and Minimum Values Using max() and min()

The `max()` and `min()` functions give us the largest and smallest values in a column and are useful for a couple of reasons. First, they help us get a sense of the scope of the values reported. Second, the functions can reveal unexpected issues with data, as you'll see now.

Both `max()` and `min()` work the same way, with the name of a column as input. Listing 9-6 uses `max()` and `min()` on the 2018 table, taking the `visits` column that records the number of annual visits to the library agency and all of its branches. Run the code.

```
SELECT max(visits), min(visits)
FROM pls_fy2018_libraries;
```

Listing 9-6: Finding the most and fewest visits using `max()` and `min()`

The query returns the following results:

```
max          min
--------     ---
16686945      -3
```

Well, that's interesting. The maximum value of more than 16.6 million is reasonable for a large city library system, but `-3` as the minimum? On the surface, that result seems like a mistake, but it turns out that the creators of the library survey are employing a common but potentially problematic convention in data collection by placing a negative number or some artificially high value in a column to indicate some condition.

In this case, negative values in number columns indicate the following:

A value of `-1` indicates a "nonresponse" to that question.

A value of `-3` indicates "not applicable" and is used when a library agency has closed either temporarily or permanently.

We'll need to account for and exclude negative values as we explore the data, because summing a column and including the negative values will result in an incorrect total. We can do this using a `WHERE` clause to filter them. It's a good reminder to always read the documentation for the data to get ahead of the issue instead of having to backtrack after spending a lot of time on deeper analysis!

NOTE

A better alternative for this negative value scenario is to use `NULL` in rows in the `visits` column where response data is absent and then create a separate `visits_flag` column to hold codes explaining why.

Aggregating Data Using GROUP BY

When you use the `GROUP BY` clause with aggregate functions, you can group results according to the values in one or more columns. This allows us to perform operations such as `sum()` or `count()` for every state in the table or for every type of library agency.

Let's explore how using `GROUP BY` with aggregate functions works. On its own, `GROUP BY`, which is also part of standard ANSI SQL, eliminates duplicate values from the results, similar to `DISTINCT`. Listing 9-7 shows the `GROUP BY` clause in action.

```
  SELECT stabr
  FROM pls_fy2018_libraries
1 GROUP BY stabr
  ORDER BY stabr;
```

Listing 9-7: Using `GROUP BY` on the `stabr` column

We add the `GROUP BY` clause 1 after the `FROM` clause and include the column name to group. In this case, we're selecting `stabr`, which contains the state abbreviation, and grouping by that same column. We then use `ORDER BY` stabr as well so that the grouped results are in alphabetical order. This will yield a result with unique state abbreviations from the 2018 table. Here's a portion of the results:

```
stabr
-----
AK
AL
AR
AS
AZ
CA
--snip--
WV
WY
```

Notice that there are no duplicates in the 55 rows returned. These standard two-letter postal abbreviations include the 50 states plus

Washington, DC, and several US territories, such as Guam and the US Virgin Islands.

You’re not limited to grouping just one column. In Listing 9-8, we use the `GROUP BY` clause on the 2018 data to specify the `city` and `stabr` columns for grouping.

```
SELECT city, stabr
FROM pls_fy2018_libraries
GROUP BY city, stabr
ORDER BY city, stabr;
```

Listing 9-8: Using `GROUP BY` on the `city` and `stabr` columns

The results get sorted by city and then by state, and the output shows unique combinations in that order:

```
city           stabr
----------     -----
ABBEVILLE      AL
ABBEVILLE      LA
ABBEVILLE      SC
ABBOTSFORD     WI
ABERDEEN       ID
ABERDEEN       SD
ABERNATHY      TX
--snip--
```

This grouping returns 9,013 rows, 248 fewer than the total table rows. The result indicates that the file includes multiple instances where there’s more than one library agency for a particular city and state combination.

Combining GROUP BY with count()

If we combine `GROUP BY` with an aggregate function, such as `count()`, we can pull more descriptive information from our data. For example, we know 9,261 library agencies are in the 2018 table. We can get a count of agencies by state and sort them to see which states have the most. Listing 9-9 shows how to do this.

```
1 SELECT stabr, count(*)
  FROM pls_fy2018_libraries
2 GROUP BY stabr
3 ORDER BY count(*) DESC;
```

Listing 9-9: Using `GROUP BY` with `count()` on the `stabr` column

We're now asking for the values in the `stabr` column and a count of how many rows have a given `stabr` value. In the list of columns to query 1, we specify `stabr` and `count()` with an asterisk as its input, which will cause `count()` to include `NULL` values. Also, when we select individual columns along with an aggregate function, we must include the columns in a `GROUP BY` clause 2. If we don't, the database will return an error telling us to do so, because you can't group values by aggregating and have ungrouped column values in the same query.

To sort the results and have the state with the largest number of agencies at the top, we can use an `ORDER BY` clause 3 that includes the `count()` function and the `DESC` keyword.

Run the code in Listing 9-9. The results show New York, Illinois, and Texas as the states with the greatest number of library agencies in 2018:

```
stabr    count
-----    -----
NY         756
IL         623
TX         560
IA         544
PA         451
MI         398
WI         381
MA         369
--snip--
```

Remember that our table represents library agencies that serve a locality. Just because New York, Illinois, and Texas have the greatest number of library agencies doesn't mean they have the greatest number of outlets where you can walk in and peruse the shelves. An agency might have one central library only, or it might have no central libraries but 23 branches spread around a county. To count outlets, each row in the table also has values in the columns `centlib` and `branlib`, which record the number of central and branch libraries, respectively. To find totals, we would use the `sum()` aggregate function on both columns.

Using GROUP BY on Multiple Columns with count()

We can glean yet more information from our data by combining `GROUP BY` with `count()` and multiple columns. For example, the `stataddr` column in all three tables contains a code indicating whether the agency's address changed in the last year. The values in `stataddr` are as follows:

00	No change from last year
07	Moved to a new location
15	Minor address change

Listing 9-10 shows the code for counting the number of agencies in each state that moved, had a minor address change, or had no change using `GROUP BY` with `stabr` and `stataddr` and adding `count()`.

```
1 SELECT stabr, stataddr, count(*)
   FROM pls_fy2018_libraries
2 GROUP BY stabr, stataddr
3 ORDER BY stabr, stataddr;
```

Listing 9-10: Using `GROUP BY` with `count()` of the `stabr` and `stataddr` columns

The key sections of the query are the column names and the `count()` function after `SELECT` 1, and making sure both columns are reflected in the `GROUP BY` clause 2 to ensure that `count()` will show the number of unique combinations of `stabr` and `stataddr`.

To make the output easier to read, let's sort first by the state and address status codes in ascending order 3. Here are the results:

```
stabr    stataddr    count
-----    --------    -----
AK       00          82
```

```
AL        00          220
AL        07            3
AL        15            1
AR        00           58
AR        07            1
AR        15            1
AS        00            1
--snip--
```

The first few rows show that code `00` (no change in address) is the most common value for each state. We'd expect that because it's likely there are more library agencies that haven't changed address than those that have. The result helps assure us that we're analyzing the data in a sound way. If code `07` (moved to a new location) was the most frequent in each state, that would raise a question about whether we've written the query correctly or whether there's an issue with the data.

Revisiting sum() to Examine Library Activity

Now let's expand our techniques to include grouping and aggregating across joined tables using the 2018, 2017, and 2016 libraries data. Our goal is to identify trends in library visits spanning that three-year period. To do this, we need to calculate totals using the `sum()` aggregate function.

Before we dig into these queries, let's address the values `-3` and `-1`, which indicate "not applicable" and "nonresponse." To prevent these negative numbers from affecting the analysis, we'll filter them out using a `WHERE` clause to limit the queries to rows where values in `visits` are zero or greater.

Let's start by calculating the sum of annual visits to libraries from the individual tables. Run each `SELECT` statement in *Listing 9-11* separately.

```
SELECT sum(visits) AS visits_2018
FROM pls_fy2018_libraries
WHERE visits >= 0;

SELECT sum(visits) AS visits_2017
FROM pls_fy2017_libraries
WHERE visits >= 0;

SELECT sum(visits) AS visits_2016
```

```
FROM pls_fy2016_libraries
WHERE visits >= 0;
```

Listing 9-11: Using the `sum()` *aggregate function to total visits to libraries in 2016, 2017, and 2018*

For 2018, visits totaled approximately 1.29 billion:

```
visits_2018
-----------
 1292348697
```

For 2017, visits totaled approximately 1.32 billion:

```
visits_2017
-----------
 1319803999
```

And for 2016, visits totaled approximately 1.36 billion:

```
visits_2016
-----------
 1355648987
```

We're onto something here, but it may not be good news for libraries. The trend seems to point downward with visits dropping about 5 percent from 2016 to 2018.

Let's refine this approach. These queries sum visits recorded in each table. But from the row counts we ran earlier in the chapter, we know that each table contains a different number of library agencies: 9,261 in 2018; 9,245 in 2017; and 9,252 in 2016. The differences are likely due to agencies opening, closing, or merging. So, let's determine how the sum of visits will differ if we limit the analysis to library agencies that exist in all three tables and have a non-negative value for `visits`. We can do that by joining the tables, as shown in Listing 9-12.

```
1 SELECT sum(pls18.visits) AS visits_2018,
         sum(pls17.visits) AS visits_2017,
         sum(pls16.visits) AS visits_2016
```

```
2 FROM pls_fy2018_libraries pls18
          JOIN pls_fy2017_libraries pls17 ON pls18.fscskey =
   pls17.fscskey
          JOIN pls_fy2016_libraries pls16 ON pls18.fscskey =
   pls16.fscskey
3 WHERE pls18.visits >= 0
          AND pls17.visits >= 0
          AND pls16.visits >= 0;
```

Listing 9-12: Using `sum()` to total visits on joined 2018, 2017, and 2016 tables

This query pulls together a few concepts we covered in earlier chapters, including table joins. At the top, we use the `sum()` aggregate function 1 to total the `visits` columns from each of the three tables. When we join the tables on the tables' primary keys, we're declaring table aliases 2 as we explored in Chapter 7—and here, we're omitting the optional `AS` keyword in front of each alias. For example, we declare `pls18` as the alias for the 2018 table to avoid having to write its lengthier full name throughout the query.

Note that we use a standard `JOIN`, also known as an `INNER JOIN`, meaning the query results will only include rows where the values in the `fscskey` primary key match in all three tables.

As we did in *Listing 9-11*, we specify with a `WHERE` clause 3 that the result should include only those rows where `visits` are greater than or equal to 0 in the tables. This will prevent the artificial negative values from impacting the sums.

Run the query. The results should look like this:

```
visits_2018    visits_2017    visits_2016
-----------    -----------    -----------
 1278148838     1319325387     1355078384
```

The results are similar to what we found by querying the tables separately, although these totals are as much as 14 million smaller in 2018. Still, the downward trend holds.

For a full picture of how library use is changing, we'd want to run a similar query on all of the columns that contain performance indicators to chronicle the trend in each. For example, the column `wifisess` shows how many times users connected to the library's wireless internet. If we use `wifisess` instead of `visits` in *Listing 9-11*, we get this result:

```
wifi_2018  wifi_2017  wifi_2016
---------  ---------  ---------
349767271  311336231  234926102
```

So, though visits were down, libraries saw a sharp increase in Wi-Fi network use. That provides a keen insight into how the role of libraries is changing.

NOTE

Although we joined the tables on `fscskey`, it's entirely possible that some library agencies that appear in all three tables merged or split during the three years. A call to the IMLS asking about caveats for working with this data is a good idea.

Grouping Visit Sums by State

Now that we know library visits dropped for the United States as a whole between 2016 and 2018, you might ask yourself, "Did every part of the country see a decrease, or did the degree of the trend vary by region?" We can answer this question by modifying our preceding query to group by the state code. Let's also use a percent-change calculation to compare the trend by state. *Listing 9-13* contains the full code.

```
1 SELECT pls18.stabr,
         sum(pls18.visits) AS visits_2018,
         sum(pls17.visits) AS visits_2017,
         sum(pls16.visits) AS visits_2016,
         round( (sum(pls18.visits::numeric) -
  sum(pls17.visits)) /
           2 sum(pls17.visits) * 100, 1 ) AS chg_2018_17,
         round( (sum(pls17.visits::numeric) -
  sum(pls16.visits)) /
              sum(pls16.visits) * 100, 1 ) AS chg_2017_16
  FROM pls_fy2018_libraries pls18
```

```
       JOIN pls_fy2017_libraries pls17 ON pls18.fscskey =
pls17.fscskey
       JOIN pls_fy2016_libraries pls16 ON pls18.fscskey =
pls16.fscskey
WHERE pls18.visits >= 0
       AND pls17.visits >= 0
       AND pls16.visits >= 0
3 GROUP BY pls18.stabr
4 ORDER BY chg_2018_17 DESC;
```

Listing 9-13: Using `GROUP BY` *to track percent change in library visits by state*

We follow the `SELECT` keyword with the `stabr` column 1 from the 2018 table; that same column appears in the `GROUP BY` clause 3. It doesn't matter which table's `stabr` column we use because we're only querying agencies that appear in all three tables. After the `visits` columns, we include the now-familiar percent-change calculation you learned in Chapter 6. We use this twice, giving the aliases `chg_2018_17` 2 and `chg_2017_16` for clarity. We end the query with an `ORDER BY` clause 4, sorting by the `chg_2018_17` column alias.

When you run the query, the top of the results shows 10 states with an increase in visits from 2017 to 2018. The rest of the results show a decline. American Samoa, at the bottom of the ranking, had a 28 percent drop!

```
stabr visits_2018 visits_2017 visits_2016 chg_2018_17
chg_2017_16
----- ----------- ----------- ----------- ----------- -------
----
SD        3824804     3699212     3722376         3.4
-0.6
MT        4332900     4215484     4298268         2.8
-1.9
FL       68423689    66697122    70991029         2.6
-6.0
ND        2216377     2162189     2201730         2.5
-1.8
ID        8179077     8029503     8597955         1.9
-6.6
DC        3632539     3593201     3930763         1.1
-8.6
ME        6746380     6731768     6811441         0.2
```

```
-1.2
NH        7045010     7028800     7236567           0.2
-2.9
UT       15326963    15295494    16096911           0.2
-5.0
DE        4122181     4117904     4125899           0.1
-0.2
OK       13399265    13491194    13112511          -0.7
2.9
WY        3338772     3367413     3536788          -0.9
-4.8
MA       39926583    40453003    40427356          -1.3
0.1
WA       37338635    37916034    38634499          -1.5
-1.9
MN       22952388    23326303    24033731          -1.6
-2.9
--snip--
GA       26835701    28816233    27987249          -6.9
3.0
AR        9551686    10358181    10596035          -7.8
-2.2
GU          75119       81572       71813          -7.9
13.6
MS        7602710     8581994     8915406         -11.4
-3.7
HI        3456131     4135229     4490320         -16.4
-7.9
AS          48828       67848       63166         -28.0
7.4
```

It's helpful, for context, to also see the percent change in `visits` from 2016 to 2017. Many of the states, such as Minnesota, show consecutive declines. Others, including several at the top of the list, show gains after substantial decreases the year prior.

This is when it's a good idea investigate what's driving the changes. Data analysis can sometimes raise as many questions as it answers, but that's part of the process. It's always worth a phone call to a person who works closely with the data to review your findings. Sometimes, they'll have a good explanation. Other times, an expert will say, "That doesn't sound right." That answer might send you back to the keeper of the data or the documentation to find out if you overlooked a code or a nuance with the data.

Filtering an Aggregate Query Using HAVING

To refine our analysis, we can examine a subset of states and territories that share similar characteristics. With percent change in visits, it makes sense to separate large states from small states. In a small state like Rhode Island, a single library closing for six months for repairs could have a significant effect. A single closure in California might be scarcely noticed in a statewide count. To look at states with a similar volume in visits, we could sort the results by either of the `visits` columns, but it would be cleaner to get a smaller result set by filtering our query.

To filter the results of aggregate functions, we need to use the `HAVING` clause that's part of standard ANSI SQL. You're already familiar with using `WHERE` for filtering, but aggregate functions, such as `sum()`, can't be used within a `WHERE` clause because they operate at the row level, and aggregate functions work across rows. The `HAVING` clause places conditions on groups created by aggregating. The code in *Listing 9-14* modifies the query in *Listing 9-13* by inserting the `HAVING` clause after `GROUP BY`.

```
SELECT pls18.stabr,
       sum(pls18.visits) AS visits_2018,
       sum(pls17.visits) AS visits_2017,
       sum(pls16.visits) AS visits_2016,
       round( (sum(pls18.visits::numeric) -
sum(pls17.visits)) /
            sum(pls17.visits) * 100, 1 ) AS chg_2018_17,
       round( (sum(pls17.visits::numeric) -
sum(pls16.visits)) /
            sum(pls16.visits) * 100, 1 ) AS chg_2017_16
FROM pls_fy2018_libraries pls18
       JOIN pls_fy2017_libraries pls17 ON pls18.fscskey =
pls17.fscskey
       JOIN pls_fy2016_libraries pls16 ON pls18.fscskey =
pls16.fscskey
WHERE pls18.visits >= 0
       AND pls17.visits >= 0
       AND pls16.visits >= 0
GROUP BY pls18.stabr
1 HAVING sum(pls18.visits) > 50000000
ORDER BY chg_2018_17 DESC;
```

Listing 9-14: Using a `HAVING` clause to filter the results of an aggregate query

In this case, we've set our query results to include only rows with a sum of visits in 2018 greater than 50 million. That's an arbitrary value I chose to show only the very largest states. Adding the `HAVING` clause 1 reduces the number of rows in the output to just six. In practice, you might experiment with various values. Here are the results:

```
stabr visits_2018 visits_2017 visits_2016 chg_2018_17
chg_2017_16
----- ----------- ----------- ----------- ----------- -------
----
FL       68423689    66697122    70991029         2.6
-6.0
NY       97921323   100012193   103081304        -2.1
-3.0
CA      146656984   151056672   155613529        -2.9
-2.9
IL       63466887    66166082    67336230        -4.1
-1.7
OH       68176967    71895854    74119719        -5.2
-3.0
TX       66168387    70514138    70975901        -6.2
-0.7
```

All but one of the six states experienced a decline in visits, but notice that the percent-change variation isn't as wide as in the full set of states and territories. Depending on what we learn from library experts, looking at the states with the most activity as a group might be helpful in describing trends, as would looking at other groupings. Think of a sentence you might write that would say, "Among states with the most library visits, Florida was the only one to see an increase in activity between 2017 and 2018; the rest saw visits decrease between 2 percent and 6 percent." You could write similar sentences about medium-sized states and small states.

Wrapping Up

If you're now inspired to visit your local library and check out a couple of books, ask a librarian whether their branch has seen a rise or drop in visits over the last few years. You can probably guess the answer. In this chapter, you learned how to use standard SQL techniques to summarize data in a table by grouping values and using a handful of aggregate functions. By joining datasets, you were able to identify some interesting trends.

You also learned that data doesn't always come perfectly packaged. The presence of negative values in columns, used as an indicator rather than as an actual numeric value, forced us to filter out those rows. Unfortunately, those sorts of challenges are part of the data analyst's everyday world, so we'll spend the next chapter learning how to clean up a dataset that has a number of issues. Later in the book, you'll also discover more aggregate functions to help you find the stories in your data.

TRY IT YOURSELF

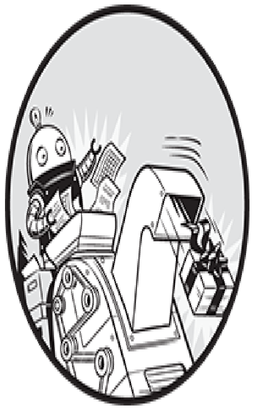
Put your grouping and aggregating skills to the test with these challenges:

We saw that library visits have declined recently in most places. But what is the pattern in library employment? All three library survey tables contain the column `totstaff`, which is the number of paid full-time equivalent employees. Modify the code in Listings 9-13 and 9-14 to calculate the percent change in the sum of the column over time, examining all states as well as states with the most visitors. Watch out for negative values!

The library survey tables contain a column called `obereg`, a two-digit Bureau of Economic Analysis Code that classifies each library agency according to a region of the United States, such as New England, Rocky Mountains, and so on. Just as we calculated the percent change in visits grouped by state, do the same to group percent changes in visits by US region using `obereg`. Consult the survey documentation to find the meaning of each region code. For a bonus challenge, create a table with the `obereg` code as the primary key and the region name as text, and join it to the summary query to group by the region name rather than the code.

Thinking back to the types of joins you learned in Chapter 7, which join type will show you all the rows in all three tables, including those without a match? Write such a query and add an `IS NULL` filter in a `WHERE` clause to show agencies not included in one or more of the tables.

10
INSPECTING AND MODIFYING DATA



If I were to propose a toast to a newly minted class of data analysts, I'd raise my glass and say, "May your data arrive perfectly structured and free of errors!" In reality, you'll sometimes receive data in such a sorry state that it's hard to analyze without modifying it. This is called *dirty data*, a general label for data with errors, missing values, or poor organization that makes standard queries ineffective. In this chapter, you'll use SQL to clean a set of dirty data and perform other useful maintenance tasks to make data workable.

Dirty data can have multiple origins. Converting data from one file type to another or giving a column the wrong data type can cause information to be lost. People also can be careless when inputting or editing data, leaving behind typos and spelling inconsistencies. Whatever the cause may be, dirty data is the bane of the data analyst.

You'll learn how to examine data to assess its quality and how to modify data and tables to make analysis easier. But the techniques you'll learn will be useful for more than just cleaning data. The ability to make changes to data and tables gives you options for updating or adding new information to your database as it becomes available, elevating your database from a static collection to a living record.

Let's begin by importing our data.

Importing Data on Meat, Poultry, and Egg Producers

For this example, we'll use a directory of US meat, poultry, and egg producers. The Food Safety and Inspection Service (FSIS), an agency within the US Department of Agriculture, compiles and updates this database regularly. The FSIS is responsible for inspecting animals and food at more than 6,000 meat processing plants, slaughterhouses, farms, and the like. If inspectors find a problem, such as bacterial contamination or mislabeled food, the agency can issue a recall. Anyone interested in agriculture business, food supply chain, or outbreaks of foodborne illnesses will find the directory useful. Read more about the agency on its site at *https://www.fsis.usda.gov/*.

The data we'll use comes from *https://www.data.gov/*, a website run by the US federal government that catalogs thousands of datasets from various federal agencies (*https://catalog.data.gov/dataset/fsis-meat-poultry-and-egg-inspection-directory-by-establishment-name/*). I've converted the Excel file posted on the site to CSV format, and you'll find a link to the file *MPI_Directory_by_Establishment_Name.csv* along with other resources for this book at *https://nostarch.com/practical-sql-2nd-edition/*.

NOTE

Because FSIS updates the data regularly, you will see different results than those shown in this chapter if you download directly from https://www.data.gov/.

To import the file into PostgreSQL, use the code in *Listing 10-1* to create a table called `meat_poultry_egg_establishments` and use `COPY` to add the CSV file to the table. As in previous examples, use pgAdmin to connect to your `analysis` database, and then open the Query Tool to run the code. Remember to change the path in the `COPY` statement to reflect the location of your CSV file.

```
CREATE TABLE meat_poultry_egg_establishments (
  1 establishment_number text CONSTRAINT est_number_key
PRIMARY KEY,
    company text,
    street text,
    city text,
    st text,
    zip text,
    phone text,
    grant_date date,
  2 activities text,
    dbas text
);

3 COPY meat_poultry_egg_establishments
  FROM
  'C:\YourDirectory\MPI_Directory_by_Establishment_Name.csv'
  WITH (FORMAT CSV, HEADER);

4 CREATE INDEX company_idx ON meat_poultry_egg_establishments
  (company);
```

Listing 10-1: Importing the FSIS Meat, Poultry, and Egg Inspection Directory

The table has 10 columns. We add a natural primary key constraint to the `establishment_number` column 1, which will hold unique values that identify each establishment. Most of the remaining columns relate to the company's name and location. You'll use the `activities` column 2, which describes activities at the company, in the "Try It Yourself" exercise at the end of this chapter. We set most columns to `text`. In PostgreSQL, `text` is a varying length data type that affords us up to 1GB of data (see Chapter 4). The column `dbas` contains strings of more than 1,000 characters in its rows, so we're prepared to handle that. We import the CSV file 3 and then create an index on the `company` column 4 to speed up searches for particular companies.

For practice, let's use the `count()` aggregate function introduced in Chapter 9 to check how many rows are in the `meat_poultry_egg_establishments` table:

```
SELECT count(*) FROM meat_poultry_egg_establishments;
```

The result should show 6,287 rows. Now let's find out what the data contains and determine whether we can glean useful information from it as is, or if we need to modify it in some way.

Interviewing the Dataset

Interviewing data is my favorite part of analysis. We interview a dataset to discover its details—what it holds, what questions it can answer, and how suitable it is for our purposes—the same way a job interview reveals whether a candidate has the skills required.

The aggregate queries from Chapter 9 are a useful interviewing tool because they often expose the limitations of a dataset or raise questions you may want to ask before drawing conclusions and assuming the validity of your findings.

For example, the `meat_poultry_egg_establishments` table's rows describe food producers. At first glance, we might assume that each company in each row operates at a distinct address. But it's never safe to assume in data analysis, so let's check using the code in Listing 10-2.

```
SELECT company,
       street,
       city,
       st,
       count(*) AS address_count
FROM meat_poultry_egg_establishments
GROUP BY company, street, city, st
HAVING count(*) > 1
ORDER BY company, street, city, st;
```

Listing 10-2: Finding multiple companies at the same address

Here, we group companies by unique combinations of the `company`, `street`, `city`, and `st` columns. Then we use `count(*)`, which returns the number of rows for each combination of those columns and gives it the alias `address_count`. Using the `HAVING` clause introduced in Chapter 9, we filter the results to show only cases where more than one row has the same combination of values. This should return all duplicate addresses for a company.

The query returns 23 rows, which means there are close to two dozen cases where the same company is listed multiple times at the same address:

```
company                    street                     city
st    address_count
-----------------------    -----------------------    -------
---    --    -------------
Acre Station Meat Farm     17076 Hwy 32 N
Pinetown      NC                2
Beltex Corporation         3801 North Grove Street    Fort
Worth    TX                2
Cloverleaf Cold Storage    111 Imperial Drive         Sanford
NC                 2
--snip--
```

This is not necessarily a problem. There may be valid reasons for a company to appear multiple times at the same address. For example, two types of processing plants could exist with the same name. On the other hand, we may have found data entry errors. Either way, it's a wise practice to eliminate concerns about the validity of a dataset before relying on it, and this result should prompt us to investigate individual cases before we draw conclusions. However, this dataset has other issues that we need to look at before we can get meaningful information from it. Let's work through a few examples.

Checking for Missing Values

Next, we'll check whether we have values from all states and whether any rows are missing a state code by asking a basic question: How many meat, poultry, and egg processing companies are there in each state? We'll use the aggregate function `count()` along with `GROUP BY` to determine this, as shown in *Listing 10-3*.

```
SELECT st,
       count(*) AS st_count
FROM meat_poultry_egg_establishments
GROUP BY st
ORDER BY st;
```

Listing 10-3: Grouping and counting states

The query is a simple count that tallies the number of times each state postal code (`st`) appears in the table. Your result should include 57 rows, grouped by the state postal code in the column `st`. Why more than the 50 US states? Because the data includes Puerto Rico and other unincorporated US territories, such as Guam and American Samoa. Alaska (`AK`) is at the top of the results with a count of `17` establishments:

```
st    st_count
--    --------
AK          17
AL          93
AR          87
AS           1
--snip--
WA         139
WI         184
WV          23
WY           1
             3
```

However, the row at the bottom of the list has a `NULL` value in the `st` column and a `3` in `st_count`. That means three rows have a `NULL` in `st`. To see the details of those facilities, let's query those rows.

NOTE

Depending on the database implementation, `NULL` values will appear either first or last in a sorted column. In PostgreSQL, they appear last by default. The ANSI SQL standard doesn't specify one or the other, but it lets you add `NULLS FIRST` or `NULLS LAST` to an `ORDER BY` clause to specify a preference. For example, to make `NULL` values appear first in the preceding query, the clause would read `ORDER BY st NULLS FIRST`.

In Listing 10-4, we add a `WHERE` clause with the `st` column and the `IS NULL` keywords to find which rows are missing a state code.

```
SELECT establishment_number,
       company,
       city,
       st,
       zip
FROM meat_poultry_egg_establishments
WHERE st IS NULL;
```

Listing 10-4: *Using `IS NULL` to find missing values in the `st` column*

This query returns three rows that don't have a value in the `st` column:

```
est_number            company                             city
st    zip
-----------------     -------------------------------     -----
-    --    -----
V18677A               Atlas Inspection, Inc.
Blaine           55449
M45319+P45319         Hall-Namie Packing Company, Inc
36671
M263A+P263A+V263A     Jones Dairy Farm
53538
```

That's a problem, because any counts that include the `st` column will be incorrect, such as the number of establishments per state. When you spot an error such as this, it's worth making a quick visual check of the original file you downloaded. Unless you're working with files in the gigabyte range, you can usually open a CSV file in one of the text editors I noted in Chapter 1 and search for the row. If you're working with larger files, you might be able to examine the source data using utilities such as `grep` (on Linux and macOS) or `findstr` (on Windows). In this case, a visual check of the file from *https://www.data.gov/* confirms that, indeed, there was no state listed in those rows in the file, so the error is organic to the data, not one introduced during import.

In our interview of the data so far, we've discovered that we'll need to add missing values to the `st` column to clean up this table. Let's look at what other issues exist in our dataset and make a list of cleanup tasks.

Checking for Inconsistent Data Values

Inconsistent data is another factor that can hamper our analysis. We can check for inconsistently entered data within a column by using `GROUP BY` with `count()`. When you scan the unduplicated values in the results, you might be able to spot variations in the spelling of names or other attributes.

For example, many of the 6,200 companies in our table are multiple locations owned by just a few multinational food corporations, such as Cargill or Tyson Foods. To find out how many locations each company owns, we count the values in the `company` column. Let's see what happens when we do, using the query in *Listing 10-5*.

```
SELECT company,
       count(*) AS company_count
FROM meat_poultry_egg_establishments
GROUP BY company
ORDER BY company ASC;
```

Listing 10-5: Using `GROUP BY` and `count()` to find inconsistent company names

Scrolling through the results reveals a number of cases in which a company's name is spelled in several different ways. For example, notice the entries for the Armour-Eckrich brand:

```
company                        company_count
---------------------------    -------------
```

```
--snip--
Armour - Eckrich Meats, LLC                1
Armour-Eckrich Meats LLC                   3
Armour-Eckrich Meats, Inc.                 1
Armour-Eckrich Meats, LLC                  2
--snip--
```

At least four different spellings are shown for seven establishments that are likely owned by the same company. If we later perform any aggregation by company, it would help to standardize the names so all the items counted or summed are grouped properly. Let's add that to our list of items to fix.

Checking for Malformed Values Using length()

It's a good idea to check for unexpected values in a column that should be consistently formatted. For example, each entry in the `zip` column in the meat_poultry_egg_establishments table should be formatted in the style of US ZIP codes with five digits. However, that's not what is in our dataset.

Solely for the purpose of this example, I replicated a common error I've committed before. When I converted the original Excel file to a CSV file, I stored the ZIP code in the default "General" number format instead of as a text value, and any ZIP code that begins with a zero lost its leading zero because an integer can't start with a zero. As a result, 07502 appears in the table as `7502`. You can make this error in a variety of ways, including by copying and pasting data into Excel columns set to "General." After being burned a few times, I learned to take extra caution with numbers that should be formatted as text.

My deliberate error appears when we run the code in *Listing 10-6*. The example introduces `length()`, a *string function* that counts the number of characters in a string. We combine `length()` with `count()` and `GROUP BY` to determine how many rows have five characters in the `zip` field and how many have a value other than five. To make it easy to scan the results, we use `length()` in the `ORDER BY` clause.

```
SELECT length(zip),
       count(*) AS length_count
FROM meat_poultry_egg_establishments
GROUP BY length(zip)
ORDER BY length(zip) ASC;
```

Listing 10-6: Using `length()` and `count()` to test the `zip` column

The results confirm the formatting error. As you can see, `496` of the ZIP codes are four characters long, and `86` are three characters long, which likely means these numbers originally had two leading zeros that my conversion erroneously eliminated:

```
length    length_count
------    ------------
     3              86
     4             496
     5            5705
```

Using the `WHERE` clause, we can see which states these shortened ZIP codes correspond to, as shown in Listing 10-7.

```
SELECT st,
       count(*) AS st_count
FROM meat_poultry_egg_establishments
1 WHERE length(zip) < 5
GROUP BY st
ORDER BY st ASC;
```

Listing 10-7: Filtering with `length()` to find short `zip` values

We use the `length()` function inside the `WHERE` clause 1 to return a count of rows where the ZIP code is less than five characters for each state code. The result is what we would expect. The states are largely in the Northeast region of the United States where ZIP codes often start with a zero:

```
st    st_count
--    --------
CT          55
MA         101
ME          24
NH          18
NJ         244
PR          84
RI          27
VI           2
VT          27
```

Obviously, we don't want this error to persist, so we'll add it to our list of items to correct. So far, we need to correct the following issues in our dataset:

Missing values for three rows in the `st` column

Inconsistent spelling of at least one company's name

Inaccurate ZIP codes due to file conversion

Next, we'll look at how to use SQL to fix these issues by modifying your data.

Modifying Tables, Columns, and Data

Almost nothing in a database, from tables to columns and the data types and values they contain, is set in concrete after it's created. As your needs change, you can use SQL to add columns to a table, change data types on existing columns, and edit values. Given the issues we discovered in the `meat_poultry_egg_establishments` table, being able to modify our database will come in handy.

We'll use two SQL commands. The first, `ALTER TABLE`, is part of the ANSI SQL standard and provides options to `ADD COLUMN`, `ALTER COLUMN`, and `DROP COLUMN`, among others.

NOTE

Typically, PostgreSQL and other databases include implementation-specific extensions to `ALTER TABLE` that provide an array of options for managing database objects (see https://www.postgresql.org/docs/current/sql-altertable.html). For our exercises, we'll stick with the core options.

The second command, `UPDATE`, also included in the SQL standard, allows you to change values in a table's columns. You can supply criteria using `WHERE` to choose which rows to update.

Let’s explore the basic syntax and options for both commands and then use them to fix the issues in our dataset.

WHEN TO TOSS YOUR DATA

If your interview of the data reveals too many missing values or values that defy common sense—such as numbers ranging in the billions when you expected thousands—it’s time to reevaluate your use of it. The data may not be reliable enough to serve as the foundation of your analysis.

If you suspect as much, the first step is to revisit the original data file. Make sure you imported it correctly and that values in all the source columns are located in the same columns in the table. You might need to open the original spreadsheet or CSV file and do a visual comparison. The second step is to call the agency or company that produced the data to confirm what you see and seek an explanation. You might also ask for advice from others who have used the same data.

More than once I’ve had to toss a dataset after determining that it was poorly assembled or simply incomplete. Sometimes, the amount of work required to make a dataset usable undermines its usefulness. These situations require you to make a tough judgment call. But it’s better to start over or find an alternative than to use bad data that can lead to faulty conclusions.

Modifying Tables with ALTER TABLE

We can use the `ALTER TABLE` statement to modify the structure of tables. The following examples show standard ANSI SQL syntax for common operations, starting with the code for adding a column to a table:

```
ALTER TABLE table ADD COLUMN column data_type;
```

We can remove a column with the following syntax:

```
ALTER TABLE table DROP COLUMN column;
```

To change the data type of a column, we would use this code:

```
ALTER TABLE table ALTER COLUMN column SET DATA TYPE
data_type;
```

We add a `NOT NULL` constraint to a column like so:

```
ALTER TABLE table ALTER COLUMN column SET NOT NULL;
```

Note that in PostgreSQL and some other systems, adding a constraint to the table causes all rows to be checked to see whether they comply with the constraint. If the table has millions of rows, this could take a while.

Removing the `NOT NULL` constraint looks like this:

```
ALTER TABLE table ALTER COLUMN column DROP NOT NULL;
```

When you execute `ALTER TABLE` with the placeholders filled in, you should see a message that reads `ALTER TABLE` in the pgAdmin output screen. If an operation violates a constraint or if you attempt to change a column's data type and the existing values in the column won't conform to the new data type, PostgreSQL returns an error. But PostgreSQL won't give you any warning about deleting data when you drop a column, so use extra caution before dropping a column.

Modifying Values with UPDATE

The `UPDATE` statement, part of the ANSI SQL standard, modifies the data in a column that meets a condition. It can be applied to all rows or a subset of rows. Its basic syntax for updating the data in every row in a column follows this form:

```
UPDATE table
SET column = value;
```

We first pass `UPDATE` the name of the table. Then to `SET` we pass the column we want to update. The new *`value`* to place in the column can be a string, number, the name of another column, or even a query or expression that generates a value. The new value must be compatible with the column data type.

We can update values in multiple columns by adding additional columns and source values and separating each with a comma:

```
UPDATE table
SET column_a = value,
    column_b = value;
```

To restrict the update to particular rows, we add a WHERE clause with some criteria that must be met before the update can happen, such as rows where values equal a date or match a string:

```
UPDATE table
SET column = value
WHERE criteria;
```

We can also update one table with values from another table. Standard ANSI SQL requires that we use a *subquery*, a query inside a query, to specify which values and rows to update:

```
UPDATE table
SET column = (SELECT column
              FROM table_b
              WHERE table.column = table_b.column)
WHERE EXISTS (SELECT column
              FROM table_b
              WHERE table.column = table_b.column);
```

The value portion of SET, inside the parentheses, is a subquery. A SELECT statement inside parentheses generates the values for the update by joining columns in both tables on matching row values. Similarly, the WHERE EXISTS clause uses a SELECT statement to ensure that we only update rows where both tables have matching values. If we didn't use WHERE EXISTS, we might inadvertently set some values to NULL without planning to. (If this syntax looks somewhat complicated, that's okay. I'll cover subqueries in detail in Chapter 13.)

Some database managers offer additional syntax for updating across tables. PostgreSQL supports the ANSI standard but also a simpler syntax using a FROM clause:

```
UPDATE table
SET column = table_b.column
FROM table_b
WHERE table.column = table_b.column;
```

When you execute an UPDATE statement, you'll get a message stating UPDATE along with the number of rows affected.

Viewing Modified Data with RETURNING

If you add an optional `RETURNING` clause to `UPDATE`, you can view the values that were modified without having to run a second, separate query. The syntax of the clause uses the `RETURNING` keyword followed by a list of columns or a wildcard in the same manner that we name columns following `SELECT`. Here's an example:

```
UPDATE table
SET column_a = value
RETURNING column_a, column_b, column_c;
```

Instead of just noting the number of rows modified, `RETURNING` directs the database to show the columns you specify for the rows modified. This is a PostgreSQL-specific implementation that you also can use with `INSERT` and `DELETE FROM`. We'll try it with some of our examples.

Creating Backup Tables

Before modifying a table, it's a good idea to make a copy for reference and backup in case you accidentally destroy some data. Listing 10-8 shows how to use a variation of the familiar `CREATE TABLE` statement to make a new table from the table we want to duplicate.

```
CREATE TABLE meat_poultry_egg_establishments_backup AS
SELECT * FROM meat_poultry_egg_establishments;
```

Listing 10-8: Backing up a table

The result should be a pristine copy of your table with the new specified name. You can confirm this by counting the number of records in both tables at once:

```
SELECT
    (SELECT count(*) FROM meat_poultry_egg_establishments) AS
original,
    (SELECT count(*) FROM
meat_poultry_egg_establishments_backup) AS backup;
```

The results should return the same count from both tables, like this:

```
original    backup
--------    ------
    6287      6287
```

If the counts match, you can be sure your backup table is an exact copy of the structure and contents of the original table. As an added measure and for easy reference, we'll use `ALTER TABLE` to make copies of column data within the table we're updating.

NOTE

Indexes are not copied when creating a table backup using the `CREATE TABLE` statement. If you decide to run queries on the backup, be sure to create a separate index on that table.

Restoring Missing Column Values

The query in *Listing 10-4* earlier revealed that three rows in the `meat_poultry_egg_establishments` table don't have a value in the `st` column:

```
est_number           company                            city
st    zip
-----------------    -------------------------------    -----
-    --    -----
V18677A              Atlas Inspection, Inc.
Blaine          55449
M45319+P45319        Hall-Namie Packing Company, Inc
36671
M263A+P263A+V263A    Jones Dairy Farm
53538
```

To get a complete count of establishments in each state, we need to fill those missing values using an `UPDATE` statement.

Creating a Column Copy

Even though we've backed up this table, let's take extra caution and make a copy of the `st` column within the table so we still have the original data if we make some dire error somewhere. Let's create the copy and fill it with the existing `st` column values as in *Listing 10-9*.

```
1 ALTER TABLE meat_poultry_egg_establishments ADD COLUMN st_copy
   text;

   UPDATE meat_poultry_egg_establishments
2 SET st_copy = st;
```

Listing 10-9: Creating and filling the `st_copy` column with `ALTER TABLE` and `UPDATE`

The `ALTER TABLE` statement 1 adds a column called `st_copy` using the same `text` data type as the original `st` column. Next, the `SET` clause 2 in `UPDATE` fills our new `st_copy` column with the values in column `st`. Because we don't specify any criteria using `WHERE`, values in every row are updated, and PostgreSQL returns the message `UPDATE 6287`. Again, it's worth noting that on a very large table, this operation could take some time and also substantially increase the table's size. Making a column copy in addition to a table backup isn't entirely necessary, but if you're the patient, cautious type, it can be worthwhile.

We can confirm the values were copied properly with a simple `SELECT` query on both columns, as in *Listing 10-10*.

```
SELECT st,
       st_copy
FROM meat_poultry_egg_establishments
WHERE st IS DISTINCT FROM st_copy
ORDER BY st;
```

Listing 10-10: Checking values in the `st` and `st_copy` columns

To check for differences between values in the columns, we use `IS DISTINCT FROM` in the `WHERE` clause. You've used `DISTINCT` before to find unique values in a column (Chapter 3); in this context, `IS DISTINCT FROM` tests whether values in `st` and `st_copy` are different. This keeps us from having to scan every row ourselves. Running this query will return zero rows, meaning the values match throughout the table.

NOTE

Because `IS DISTINCT FROM` treats `NULL` as a known value, comparisons between values always will evaluate to `true` or `false`. That's different than the `<>` operator, in which a comparison that includes a `NULL` will return `NULL`. Run `SELECT 'a' <> NULL;` to see this behavior.

Now, with our original data safely stored, we can update the three rows with missing state codes. This is now our in-table backup, so if something goes drastically wrong while we're updating the original column, we can easily copy the original data back in. I'll show you how after we apply the first updates.

Updating Rows Where Values Are Missing

To update those rows' missing values, we first find the values we need with a quick online search: Atlas Inspection is located in Minnesota; Hall-Namie Packing is in Alabama; and Jones Dairy is in Wisconsin. We add those states to the appropriate rows in Listing 10-11.

```
  UPDATE meat_poultry_egg_establishments
  SET st = 'MN'
1 WHERE establishment_number = 'V18677A';

  UPDATE meat_poultry_egg_establishments
  SET st = 'AL'
  WHERE establishment_number = 'M45319+P45319';

  UPDATE meat_poultry_egg_establishments
  SET st = 'WI'
  WHERE establishment_number = 'M263A+P263A+V263A'
2 RETURNING establishment_number, company, city, st, zip;
```

Listing 10-11: Updating the `st` column for three establishments

Because we want each `UPDATE` statement to affect a single row, we include a `WHERE` clause 1 for each that identifies the company's unique `establishment_number`, which is the table's primary key. When we run the first two queries, PostgreSQL responds with the message `UPDATE 1`, showing that only one row was updated for each query. When we run the third, the `RETURNING` clause 2 directs the database to show several columns from the row that was updated:

```
establishment_number   company            city      st    zip
--------------------   ----------------   --------  --    ----
-
M263A+P263A+V263A      Jones Dairy Farm             WI
53538
```

If we rerun the code in *Listing 10-4* to find rows where `st` is `NULL`, the query should return nothing. Success! Our count of establishments by state is now complete.

Restoring Original Values

What happens if we botch an update by providing the wrong values or updating the wrong rows? We'll just copy the data back from either the full table backup or the column backup. *Listing 10-12* shows the two options.

```
1 UPDATE meat_poultry_egg_establishments
  SET st = st_copy;

2 UPDATE meat_poultry_egg_establishments original
  SET st = backup.st
  FROM meat_poultry_egg_establishments_backup backup
  WHERE original.establishment_number =
  backup.establishment_number;
```

Listing 10-12: Restoring original `st` column values

To restore the values from the backup column in `meat_poultry_egg_establishments`, run an `UPDATE` query 1 that sets `st` to the values in `st_copy`. Both columns should again have the identical original values. Alternatively, you can create an `UPDATE` 2 that sets `st` to values in the `st` column from the `meat_poultry_egg_establishments_backup` table you made in *Listing 10-8*. This will obviate the fixes you made to add missing state values, so if you want to try this query, you'll need to redo the fixes using *Listing 10-11*.

Updating Values for Consistency

In *Listing 10-5*, we discovered several cases where a single company's name was entered inconsistently. These inconsistencies will hinder us if we want to aggregate data by company name, so we'll fix them.

Here are the spelling variations of Armour-Eckrich Meats in *Listing 10-5*:

```
--snip--
Armour - Eckrich Meats, LLC
Armour-Eckrich Meats LLC
Armour-Eckrich Meats, Inc.
Armour-Eckrich Meats, LLC
--snip--
```

We can standardize the spelling using an `UPDATE` statement. To protect our data, we'll create a new column for the standardized spellings, copy the names in `company` into the new column, and work in the new column. *Listing 10-13* has the code for both actions.

```
ALTER TABLE meat_poultry_egg_establishments ADD COLUMN
company_standard text;

UPDATE meat_poultry_egg_establishments
SET company_standard = company;
```

Listing 10-13: Creating and filling the `company_standard` column

Now, let's say we want any name in `company` that starts with the string `Armour` to appear in `company_standard` as `Armour-Eckrich Meats`. (This assumes we've checked all Armour entries and want to standardize them.) With *Listing 10-14*, we can update all the rows matching the string `Armour` using `WHERE`.

```
 UPDATE meat_poultry_egg_establishments
 SET company_standard = 'Armour-Eckrich Meats'
1 WHERE company LIKE 'Armour%'
2 RETURNING company, company_standard;
```

Listing 10-14: Using an `UPDATE` statement to modify column values that match a string

The important piece of this query is the `WHERE` clause that uses the `LIKE` keyword 1 for case-sensitive pattern matching introduced in Chapter 3. Including the wildcard syntax `%` at the end of the string `Armour` updates all rows that start with those characters regardless of what comes after them. The clause lets us target all the varied spellings used for the company's name. The `RETURNING` clause 2 causes the statement to provide the results of the updated `company_standard` column next to the original `company` column:

```
company                           company_standard
---------------------------       --------------------
Armour-Eckrich Meats LLC          Armour-Eckrich Meats
Armour - Eckrich Meats, LLC       Armour-Eckrich Meats
Armour-Eckrich Meats LLC          Armour-Eckrich Meats
Armour-Eckrich Meats LLC          Armour-Eckrich Meats
Armour-Eckrich Meats, Inc.        Armour-Eckrich Meats
Armour-Eckrich Meats, LLC         Armour-Eckrich Meats
Armour-Eckrich Meats, LLC         Armour-Eckrich Meats
```

The values for Armour-Eckrich in `company_standard` are now standardized with consistent spelling. To standardize other company names in the table, we would create an `UPDATE` statement for each case. We would also keep the original `company` column for reference.

Repairing ZIP Codes Using Concatenation

Our final fix repairs values in the `zip` column that lost leading zeros. Zip codes in Puerto Rico and the US Virgin Islands begin with two zeros, so we need to restore two leading zeros to the values in `zip`. For the other states, located mostly in New England, we'll restore a single leading zero.

We'll use `UPDATE` in conjunction with the double-pipe *string concatenation operator* (`||`). Concatenation combines two string values into one (it will also combine a string and a number into a string). For example, inserting `||` between the strings `abc` and `xyz` results in `abcxyz`. The double-pipe operator is a SQL standard for concatenation supported by PostgreSQL. You can use it in many contexts, such as `UPDATE` queries and `SELECT`, to provide custom output from existing as well as new data.

First, Listing 10-15 makes a backup copy of the `zip` column as we did earlier.

```
ALTER TABLE meat_poultry_egg_establishments ADD COLUMN
zip_copy text;

UPDATE meat_poultry_egg_establishments
SET zip_copy = zip;
```

Listing 10-15: *Creating and filling the `zip_copy` column*

Next, we use the code in Listing 10-16 to perform the first update.

```
 UPDATE meat_poultry_egg_establishments
1 SET zip = '00' || zip
2 WHERE st IN('PR','VI') AND length(zip) = 3;
```

Listing 10-16: *Modifying codes in the `zip` column missing two leading zeros*

We use `SET` to set the value in the `zip` column 1 to the result of the concatenation of `00` and the existing value. We limit the `UPDATE` to only those rows where the `st` column has the state codes `PR` and `VI` 2 using the `IN` comparison operator from Chapter 3 and add a test for rows where the length of `zip` is `3`. This entire statement will then only update the `zip` values for Puerto Rico and the Virgin Islands. Run the query; PostgreSQL should return the message `UPDATE 86`, which is the number of rows we expect to change based on our earlier count in Listing 10-6.

Let's repair the remaining ZIP codes using a similar query in Listing 10-17.

```
UPDATE meat_poultry_egg_establishments
SET zip = '0' || zip
WHERE st IN('CT','MA','ME','NH','NJ','RI','VT') AND
length(zip) = 4;
```

Listing 10-17: *Modifying codes in the `zip` column missing one leading zero*

PostgreSQL should return the message `UPDATE 496`. Now, let's check our progress. Earlier in *Listing 10-6*, when we aggregated rows in the `zip` column by length, we found `86` rows with three characters and `496` with four.

Using the same query now returns a more desirable result: all the rows have a five-digit ZIP code.

```
length    count
------    -----
     5     6287
```

I'll discuss additional string functions in Chapter 14 when we consider advanced techniques for working with text.

Updating Values Across Tables

In "Modifying Values with UPDATE" earlier in the chapter, I showed the standard ANSI SQL and PostgreSQL-specific syntax for updating values in one table based on values in another. This syntax is particularly valuable in a relational database where primary keys and foreign keys establish table relationships. In those cases, we may need information in one table to update values in another table.

Let's say we're setting an inspection deadline for each of the companies in our table. We want to do this by US regions, such as Northeast, Pacific, and so on, but those regional designations don't exist in our table. However, they *do* exist in the file *state_regions.csv*, included with the book's resources, that contains matching `st` state codes. Once we load that file into a table, we can use that data in an `UPDATE` statement. Let's begin with the New England region to see how this works.

Enter the code in *Listing 10-18*, which contains the SQL statements to create a `state_regions` table and fill the table with data:

```
CREATE TABLE state_regions (
    st text CONSTRAINT st_key PRIMARY KEY,
    region text NOT NULL
);

COPY state_regions
```

```
FROM 'C:\YourDirectory\state_regions.csv'
WITH (FORMAT CSV, HEADER);
```

Listing 10-18: Creating and filling a `state_regions` table

We'll create two columns in a `state_regions` table: one containing the two-character state code `st` and the other containing the `region` name. We set the primary key constraint to the `st` column, which holds a unique `st_key` value to identify each state. In the data you're importing, each state is present and assigned to a census region, and territories outside the United States are labeled as outlying areas. We'll update the table one region at a time.

Next, let's return to the `meat_poultry_egg_establishments` table, add a column for inspection dates, and then fill in that column with the New England states. *Listing 10-19* shows the code.

```
ALTER TABLE meat_poultry_egg_establishments
    ADD COLUMN inspection_deadline timestamp with time zone;

1 UPDATE meat_poultry_egg_establishments establishments
2 SET inspection_deadline = '2022-12-01 00:00 EST'
3 WHERE EXISTS (SELECT state_regions.region
                FROM state_regions
                WHERE establishments.st = state_regions.st
                      AND state_regions.region = 'New
England');
```

Listing 10-19: Adding and updating an `inspection_deadline` column

The `ALTER TABLE` statement creates the `inspection_deadline` column in the `meat_poultry_egg_establishments` table. In the `UPDATE` statement, we give the table an alias of `establishments` to make the code easier to read 1 (and do so omitting the optional `AS` keyword). Next, `SET` assigns a timestamp value of `2022-12-01 00:00 EST` to the new `inspection_deadline` column 2. Finally, `WHERE EXISTS` includes a subquery that connects the `meat_poultry_egg_establishments` table to the `state_regions` table we created in *Listing 10-18* and specifies which rows to update 3. The subquery (in parentheses, beginning with `SELECT`) looks for rows in the `state_regions` table where the `region` column matches the string `New England`. At the same time, it joins the `meat_poultry_egg_establishments` table with the `state_regions` table using the `st` column from both tables. In effect, the query is telling the database to find all the `st` codes that correspond to the New England region and use those codes to filter the update.

When you run the code, you should receive a message of `UPDATE 252`, which is the number of companies in New England states. You can use the code in *Listing 10-20* to see the effect of the change.

```
SELECT st, inspection_deadline
FROM meat_poultry_egg_establishments
GROUP BY st, inspection_deadline
ORDER BY st;
```

Listing 10-20: Viewing updated `inspection_date` values

The results should show the updated inspection deadlines for all New England companies. The top of the output shows Connecticut has received a deadline timestamp, for example, but states outside New England remain `NULL` because we haven't updated them yet:

```
st    inspection_deadline
--    ----------------------
--snip--
CA
CO
CT    2022-12-01 00:00:00-05
DC
--snip--
```

To fill in deadlines for additional regions, substitute a different region for `New England` in *Listing 10-19* and rerun the query.

Deleting Unneeded Data

The most irrevocable way to modify data is to remove it entirely. SQL includes options to remove rows and columns along with options to delete an entire table or database. We want to perform these operations with caution, removing only data or tables we don't need. Without a backup, your data is gone for good.

NOTE

It's easy to exclude unwanted data in queries using a `WHERE` *clause, so decide whether you truly need to delete the data or can just filter it out. Cases where deleting may be the best solution include data with errors, data imported incorrectly, or almost no disk space.*

In this section, we'll use a variety of SQL statements to delete data. If you didn't back up the `meat_poultry_egg_establishments` table using *Listing 10-8*, now is a good time to do so.

Writing and executing these statements is fairly simple, but doing so comes with a caveat. If deleting rows, a column, or a table would cause a violation of a constraint, such as the foreign key constraint covered in Chapter 8, you need to deal with that constraint first. That might involve removing the constraint, deleting data in another table, or deleting another table. Each case is unique and will require a different way to work around the constraint.

Deleting Rows from a Table

To remove rows from a table, we can use either `DELETE FROM` or `TRUNCATE`, which are both part of the ANSI SQL standard. Each offers options that are useful depending on your goals.

Using `DELETE FROM`, we can remove all rows from a table, or we can add a `WHERE` clause to delete only the portion that matches an expression we supply. To delete all rows from a table, use the following syntax:

```
DELETE FROM table_name;
```

To remove only selected rows, add a `WHERE` clause along with the matching value or pattern to specify which ones you want to delete:

```
DELETE FROM table_name WHERE expression;
```

For example, to exclude US territories from our processors table, we can remove the companies in those locations using the code in Listing 10-21.

```
DELETE FROM meat_poultry_egg_establishments
WHERE st IN('AS','GU','MP','PR','VI');
```

Listing 10-21: Deleting rows matching an expression

Run the code; PostgreSQL should return the message `DELETE 105`. This means the 105 rows where the `st` column held any of the codes designating a territory that you supplied via the `IN` keyword have been removed from the table.

With large tables, using `DELETE FROM` to remove all rows can be inefficient because it scans the entire table as part of the process. In that case, you can use `TRUNCATE`, which skips the scan. To empty the table using `TRUNCATE`, use the following syntax:

```
TRUNCATE table_name;
```

A handy feature of `TRUNCATE` is the ability to reset an `IDENTITY` sequence, such as one you may have created to serve as a surrogate primary key, as part of the operation. To do that, add the `RESTART IDENTITY` keywords to the statement:

```
TRUNCATE table_name RESTART IDENTITY;
```

We'll skip truncating any tables for now as we need the data for the rest of the chapter.

Deleting a Column from a Table

Earlier we created a backup `zip` column called `zip_copy`. Now that we've finished working on fixing the issues in `zip`, we no longer need `zip_copy`. We can remove the backup column, including all the data within the column, from the table using the `DROP` keyword in the `ALTER TABLE` statement.

The syntax for removing a column is similar to other `ALTER TABLE` statements:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

The code in *Listing 10-22* removes the `zip_copy` column:

```
ALTER TABLE meat_poultry_egg_establishments DROP COLUMN
zip_copy;
```

Listing 10-22: Removing a column from a table using `DROP`

PostgreSQL returns the message `ALTER TABLE`, and the `zip_copy` column should be deleted. The database doesn't actually rewrite the table to remove the column; it just marks the column as deleted in its internal catalog and no longer shows it or adds data to it when new rows are added.

Deleting a Table from a Database

The `DROP TABLE` statement is a standard ANSI SQL feature that deletes a table from the database. This statement might come in handy if, for example, you have a collection of backups, or *working tables*, that have outlived their usefulness. It's also useful when you need to change the structure of a table significantly; in that case, rather than using too many `ALTER TABLE` statements, you can just remove the table and create a fresh one by running a new `CREATE TABLE` statement and re-importing the data.

The syntax for the `DROP TABLE` command is simple:

```
DROP TABLE table_name;
```

For example, *Listing 10-23* deletes the backup version of the `meat_poultry_egg_establishments` table.

```
DROP TABLE meat_poultry_egg_establishments_backup;
```

Listing 10-23: Removing a table from a database using `DROP`

Run the query; PostgreSQL should respond with the message `DROP TABLE` to indicate the table has been removed.

Using Transactions to Save or Revert Changes

So far, our alterations in this chapter have been final. That is, after you run a `DELETE` or `UPDATE` query (or any other query that alters your data or database structure), the only way to undo the change is to restore from a backup. However, there is a way to check your changes before finalizing them and cancel the change if it's not what you intended. You do this by enclosing the SQL statement within a *transaction*, which includes keywords that allow you to commit your changes if they are successful or roll them back if not. You define a transaction using the following keywords at the beginning and end of the query:

`START TRANSACTION` Signals the start of the transaction block. In PostgreSQL, you can also use the non-ANSI SQL `BEGIN` keyword.

`COMMIT` Signals the end of the block and saves all changes.

`ROLLBACK` Signals the end of the block and reverts all changes.

You can include multiple statements between `BEGIN` and `COMMIT` to define a sequence of operations that perform one unit of work in a database. An example is when you buy concert tickets, which might involve two steps: charging your credit card and reserving your seats so someone else can't buy them. A database programmer would want either both steps in the transaction to happen (say, when your card charge goes through) or neither to happen (if you cancel at checkout). Defining both steps as one transaction—also called a *transaction block*—keeps them as a unit; if one step is canceled or throws an error, the other gets canceled too. You can learn more details about transactions and PostgreSQL at *https://www.postgresql.org/docs/current/tutorial-transactions.html*.

We can use a transaction block to review changes a query makes and then decide whether to keep or discard them. In our table, let's say we're cleaning dirty data related to the company AGRO Merchants Oakland LLC.

The table has three rows listing the company, but one row has an extra comma in the name:

```
Company
---------------------------
AGRO Merchants Oakland LLC
AGRO Merchants Oakland LLC
AGRO Merchants Oakland, LLC
```

We want the name to be consistent, so we'll remove the comma from the third row using an `UPDATE` query, as we did earlier. But this time we'll check the result of our update before we make it final (and we'll purposely make a mistake we want to discard). Listing 10-24 shows how to do this using a transaction block.

```
1 START TRANSACTION;

  UPDATE meat_poultry_egg_establishments
2 SET company = 'AGRO Merchantss Oakland LLC'
  WHERE company = 'AGRO Merchants Oakland, LLC';

3 SELECT company
  FROM meat_poultry_egg_establishments
  WHERE company LIKE 'AGRO%'
  ORDER BY company;

4 ROLLBACK;
```

Listing 10-24: Demonstrating a transaction block

Beginning with `START TRANSACTION;` 1, we'll run each statement separately. The database responds with the message `START TRANSACTION`, letting you know that any succeeding changes you make to data will not be made permanent unless you issue a `COMMIT` command. Next, we run the `UPDATE` statement, which changes the company name in the row where it has an extra comma. I intentionally added an extra `s` in the name used in the `SET` clause 2 to introduce a mistake.

When we view the names of companies starting with the letters `AGRO` using the `SELECT` statement 3, we see that, oops, one company name is misspelled now.

```
Company
----------------------------
AGRO Merchants Oakland LLC
AGRO Merchants Oakland LLC
AGRO Merchantss Oakland LLC
```

Instead of rerunning the `UPDATE` statement to fix the typo, we can simply discard the change by running the `ROLLBACK;` 4 command. When we rerun the `SELECT` statement to view the company names, we're back to where we started:

```
Company
----------------------------
AGRO Merchants Oakland LLC
AGRO Merchants Oakland LLC
AGRO Merchants Oakland, LLC
```

From here, you correct your `UPDATE` statement by removing the extra `s` and rerun it, beginning with the `START TRANSACTION` statement again. If you're happy with the changes, run `COMMIT;` to make them permanent.

NOTE

When you start a transaction in PostgreSQL, any changes you make to the data aren't visible to other database users until you execute `COMMIT`*. Other databases may behave differently depending on their settings.*

Transaction blocks are often used for more complex situations rather than checking simple changes. Here you've used them to test whether a query behaves as desired, saving you time and headaches. Next, let's look at another way to save time when updating lots of data.

Improving Performance When Updating Large Tables

With PostgreSQL, adding a column to a table and filling it with values can quickly inflate the table's size because the database creates a new version of the existing row each time a value is updated, but it doesn't delete the old row version. That essentially doubles the table's size. (You'll learn how to clean up these old rows when I discuss database maintenance in "Recovering Unused Space with VACUUM" in Chapter 19.) For small datasets, the increase is negligible, but for tables with hundreds of thousands or millions of rows, the time required to update rows and the resulting extra disk usage can be substantial.

Instead of adding a column and filling it with values, we can save disk space by copying the entire table and adding a populated column during the operation. Then, we rename the tables so the copy replaces the original, and the original becomes a backup. Thus, we have a fresh table without the added old rows.

Listing 10-25 shows how to copy `meat_poultry_egg_establishments` into a new table while adding a populated column. To do this, if you didn't already drop the `meat_poultry_egg_establishments_backup` table as shown in *Listing 10-23*, go ahead and drop it. Then run the `CREATE TABLE` statement.

```
 CREATE TABLE meat_poultry_egg_establishments_backup AS
1 SELECT *,
       2 '2023-02-14 00:00 EST'::timestamp with time zone AS
 reviewed_date
 FROM meat_poultry_egg_establishments;
```

Listing 10-25: Backing up a table while adding and filling a new column

The query is a modified version of the backup script in *Listing 10-8*. Here, in addition to selecting all the columns using the asterisk wildcard 1, we also add a column called `reviewed_date` by providing a value cast as a `timestamp` data type 2 and the `AS` keyword. That syntax adds and fills `reviewed_date`, which we might use to track the last time we checked the status of each plant.

Then we use *Listing 10-26* to swap the table names.

```
1 ALTER TABLE meat_poultry_egg_establishments
      RENAME TO meat_poultry_egg_establishments_temp;
2 ALTER TABLE meat_poultry_egg_establishments_backup
      RENAME TO meat_poultry_egg_establishments;
3 ALTER TABLE meat_poultry_egg_establishments_temp
      RENAME TO meat_poultry_egg_establishments_backup;
```

Listing 10-26: Swapping table names using `ALTER TABLE`

Here we use `ALTER TABLE` with a `RENAME TO` clause to change a table name. The first statement changes the original table name to one that ends with `_temp` 1. The second statement renames the copy we made with *Listing 10-24* to the original name of the table 2. Finally, we rename the table that ends with `_temp` to the ending `_backup` 3. The original table is now called `meat_poultry_egg_establishments_backup`, and the copy with the added column is called `meat_poultry_egg_establishments`. This process avoids updating rows and thus inflating the table.

Wrapping Up

Gleaning useful information from data sometimes requires modifying the data to remove inconsistencies, fix errors, and make it more suitable for supporting an accurate analysis. In this chapter you learned some useful tools to help you assess dirty data and clean it up. In a perfect world, all datasets would arrive with everything clean and complete. But such a perfect world doesn't exist, so the ability to alter, update, and delete data is indispensable.

Let me restate the important tasks of working safely. Be sure to back up your tables before you start making changes. Make copies of your columns, too, for an extra level of protection. When I discuss database maintenance for PostgreSQL later in the book, you'll learn how to back up entire databases. These few steps of precaution will save you a world of pain.

In the next chapter, we'll return to math to explore some of SQL's advanced statistical functions and techniques for analysis.

TRY IT YOURSELF

In this exercise, you'll turn the `meat_poultry_egg_establishments` table into useful information. You need to answer two questions: how many of the plants in the table process meat, and how many process poultry?

The answers to these two questions lie in the `activities` column. Unfortunately, the column contains an assortment of text with inconsistent input. Here's an example of the kind of text you'll find in the `activities` column:

```
Poultry Processing, Poultry Slaughter
Meat Processing, Poultry Processing
Poultry Processing, Poultry Slaughter
```

The mishmash of text makes it impossible to perform a typical count that would allow you to group processing plants by activity. However, you can make some modifications to fix this data. Your tasks are as follows:

Create two new columns called `meat_processing` and `poultry_processing` in your table. Each can be of the type `boolean`.

Using `UPDATE`, set `meat_processing = TRUE` on any row in which the `activities` column contains the text *Meat Processing*. Do the same update on the `poultry_processing` column, but this time look for the text *Poultry Processing* in `activities`.

Use the data from the new, updated columns to count how many plants perform each type of activity. For a bonus challenge, count how many plants perform both activities.

11
STATISTICAL FUNCTIONS IN SQL

In this chapter, we'll explore SQL statistical functions along with guidelines for using them. A SQL database usually isn't the first tool a data analyst chooses when they need to do more than calculate sums and averages. Typically, the software of choice is a full-featured statistics package, such as SPSS or SAS, the programming languages R or Python, or even Excel. But you don't have to discount your database. Standard ANSI SQL, including PostgreSQL's implementation, offers powerful stats functions and capabilities that reveal a lot about your data without having to export your dataset to another program.

Statistics is a vast subject worthy of its own book, so we'll only skim the surface here. Nevertheless, you'll learn how to apply high-level statistical concepts to help you derive meaning from your data using a new dataset from the US Census Bureau. You'll also learn to use SQL to create rankings, calculate rates using data about business establishments, and smooth out time-series data using rolling averages and sums.

Creating a Census Stats Table

Let’s return to one of my favorite data sources, the US Census Bureau. This time, you’ll use county data from the 2014–2018 American Community Survey (ACS) 5-Year Estimates, another product from the bureau.

Use the code in Listing 11-1 to create the table `acs_2014_2018_stats` and import the CSV file *acs_2014_2018_stats.csv*. The code and data are available with all the book’s resources via https://nostarch.com/practical-sql-2nd-edition/. Remember to change *C:\YourDirectory\* to the location of the CSV file.

```
CREATE TABLE acs_2014_2018_stats (
  1 geoid text CONSTRAINT geoid_key PRIMARY KEY,
    county text NOT NULL,
    st text NOT NULL,
  2 pct_travel_60_min numeric(5,2),
    pct_bachelors_higher numeric(5,2),
    pct_masters_higher numeric(5,2),
    median_hh_income integer,
  3 CHECK (pct_masters_higher <= pct_bachelors_higher)
);

COPY acs_2014_2018_stats
FROM 'C:\YourDirectory\acs_2014_2018_stats.csv'
WITH (FORMAT CSV, HEADER);

4 SELECT * FROM acs_2014_2018_stats;
```

Listing 11-1: Creating a 2014–2018 ACS 5-Year Estimates table and importing data

The `acs_2014_2018_stats` table has seven columns. The first three 1 include a unique `geoid` that serves as the primary key, the name of the `county`, and the state name `st`. Both `county` and `st` carry the `NOT NULL` constraint because each row should contain a value. The next four columns display certain percentages 2 I derived for each county from estimates in the ACS release, plus one more economic indicator:

`pct_travel_60_min`

The percentage of workers ages 16 and older who commute more than 60 minutes to work.

`pct_bachelors_higher`

The percentage of people ages 25 and older whose level of education is a bachelor's degree or higher. (In the United States, a bachelor's degree is usually awarded upon completing a four-year college education.)

`pct_masters_higher`

The percentage of people ages 25 and older whose level of education is a master's degree or higher. (In the United States, a master's degree is the first advanced degree earned after completing a bachelor's degree.)

`median_hh_income`

The county's median household income in 2018 inflation-adjusted dollars. As you learned in Chapter 6, a median value is the midpoint in an ordered set of numbers, where half the values are larger than the midpoint and half are smaller. Because averages can be skewed by a few very large or very small values, government reporting on economic data, such as income, tends to use medians.

We include a `CHECK` constraint 3 to ensure that the figures for the bachelor's degree are equal to or higher than those for the master's degree, because in the United States, a bachelor's degree is earned before or concurrently with a master's degree. A county showing the opposite could indicate data imported incorrectly or a column mislabeled. Our data checks out: upon import, there are no errors showing a violation of the `CHECK` constraint.

We use the `SELECT` statement 4 to view all 3,142 rows imported, each corresponding to a county surveyed in this census release.

Next, we'll use statistics functions in SQL to better understand the relationships among the percentages.

THE US CENSUS: ESTIMATES VS. THE COMPLETE COUNT

Each US Census Bureau data product has its own methodology. The Decennial Census, the most well-known, is a full count of the US population conducted every 10 years using forms mailed to each household in the country and visits by census workers. One of its primary purposes is to determine the number of seats each state holds in the US House of Representatives. The census population estimates we've used build off the decennial count and use births, deaths, migration, and other factors to create population totals for the years between the decennial counts.

In contrast, the American Community Survey is an ongoing annual survey of about 3.5 million US households. It asks about topics including income, education, employment, ancestry, and housing. Private and public organizations use ACS data to track trends that drive decision-making. Currently, the US Census Bureau packages ACS data into two releases: a one-year dataset with estimates for geographies with populations of 65,000 or more, and a five-year dataset that includes all geographies. Because it's a survey, ACS results are estimates and have a margin of error, which I've omitted for brevity but which you'll see included in a full ACS dataset.

Measuring Correlation with corr(Y, X)

Correlation describes the statistical relationship between two variables, measuring the extent to which a change in one is associated with a change in the other. In this section, we'll use the SQL `corr(Y, X)` function to measure what relationship exists, if any, between the percentage of people in a county who've attained a bachelor's degree and the median household income in that county. We'll also determine whether, according to our data, a better-educated population typically equates to higher income and, if it does, the strength of that relationship.

First, some background. The *Pearson correlation coefficient* (generally denoted as *r*) measures the strength and direction of a *linear relationship* between two variables. Variables that have a strong linear relationship cluster along a line when graphed on a scatterplot. The Pearson value of *r* falls between −1 and 1; either end of the range indicates a perfect correlation, whereas values near zero indicate a random distribution with little correlation. A positive *r* value indicates a *direct relationship*: as one variable increases, the other does too. When graphed, the data points representing each pair of values in a direct relationship would slope upward from left to right. A negative *r* value indicates an *inverse relationship*: as one variable increases, the other decreases. Dots representing an inverse relationship would slope downward from left to right on a scatterplot.

Table 11-1 provides general guidelines for interpreting positive and negative *r* values, although different statisticians may offer different interpretations.

Table 11-1: *Interpreting Correlation Coefficients*

Correlation coefficient (+/−)	What it could mean
0	No relationship
.01 to .29	Weak relationship
.3 to .59	Moderate relationship
.6 to .99	Strong to nearly perfect relationship
1	Perfect relationship

In standard ANSI SQL and PostgreSQL, we calculate the Pearson correlation coefficient using `corr(Y, X)`. It's one of several *binary aggregate functions* in SQL and is so named because these functions accept two inputs. The input `Y` is the *dependent variable* whose variation depends on the value of another variable, and `X` is the *independent variable* whose value doesn't depend on another variable.

NOTE

Even though SQL specifies the `Y` *and* `X` *inputs for the* `corr()` *function, correlation calculations don't distinguish between dependent and independent variables. Switching the order of inputs in* `corr()` *produces the same result. However, for convenience and readability, these examples order the input variables according to dependent and independent.*

We'll use `corr(Y, X)` to discover the relationship between education level and income, with income as our dependent variable and education as our independent variable. Enter the code in *Listing 11-2* to use `corr(Y, X)` with `median_hh_income` and `pct_bachelors_higher` as inputs.

```
SELECT corr(median_hh_income, pct_bachelors_higher)
    AS bachelors_income_r
FROM acs_2014_2018_stats;
```

Listing 11-2: Using `corr(Y, X)` to measure the relationship between education and income

Run the query; your result should be an *r* value of just below 0.70 given as the floating-point `double precision` data type:

```
bachelors_income_r
------------------
0.6999086502599159
```

This positive *r* value indicates that as a county's educational attainment increases, household income tends to increase. The relationship isn't perfect, but the *r* value shows the relationship is fairly strong. We can visualize this pattern by plotting the variables on a scatterplot using Excel, as shown in *Figure 11-1*. Each data point represents one US county; the data point's position on the x-axis shows the percentage of the population ages 25 and older that has a bachelor's degree or higher. The data point's position on the y-axis represents the county's median household income.

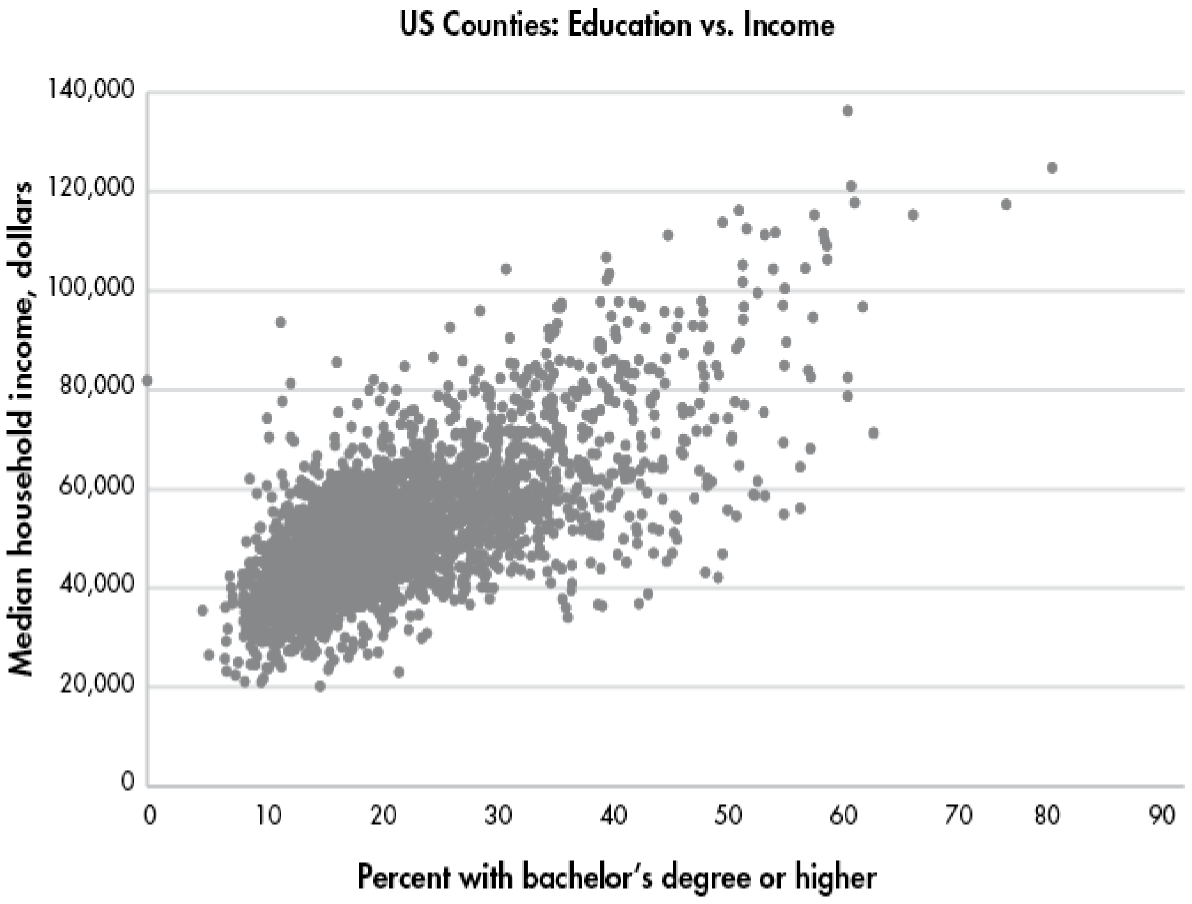


Figure 11-1: A scatterplot showing the relationship between education and income

Notice that although most of the data points are grouped together in the bottom-left corner of the graph, they do generally slope upward from left to right. Also, the points spread out rather than strictly follow a straight line. If they were in a straight line sloping up from left to right, the *r* value would be 1, indicating a perfect positive linear relationship.

Checking Additional Correlations

Now let's calculate the correlation coefficients for the remaining variable pairs using the code in Listing 11-3.

```
SELECT
  1 round(
      corr(median_hh_income, pct_bachelors_higher)::numeric,
2
```

```
      ) AS bachelors_income_r,
    round(
      corr(pct_travel_60_min, median_hh_income)::numeric, 2
      ) AS income_travel_r,
    round(
      corr(pct_travel_60_min, pct_bachelors_higher)::numeric,
2
      ) AS bachelors_travel_r
FROM acs_2014_2018_stats;
```

Listing 11-3: Using `corr(Y, X)` on additional variables

This time we'll round off the decimal values to make the output more readable by wrapping the `corr(Y, X)` function inside SQL's `round()` function 1, which takes two inputs: the `numeric` value to be rounded and an `integer` value indicating the number of decimal places to round the first value. If the second parameter is omitted, the value is rounded to the nearest whole integer. Because `corr(Y, X)` returns a floating-point value by default, we cast it to the `numeric` type using the `::` notation you learned in Chapter 4. Here's the output:

```
bachelors_income_r    income_travel_r    bachelors_travel_r
------------------    ---------------    ------------------
              0.70               0.06                 -0.14
```

The `bachelors_income_r` value is `0.70`, which is the same as our first run but rounded up to two decimal places. Compared to `bachelors_income_r`, the other two correlations are weak.

The `income_travel_r` value shows that the correlation between income and the percentage of those who commute more than an hour to work is practically zero. This indicates that a county's median household income bears little connection to how long it takes people to get to work.

The `bachelors_travel_r` value shows that the correlation of bachelor's degrees and lengthy commutes is also low at `-0.14`. The negative value indicates an inverse relationship: as education increases, the percentage of the population that travels more than an hour to work decreases. Although this is interesting, a correlation coefficient that is this close to zero indicates a weak relationship.

When testing for correlation, we need to note some caveats. The first is that even a strong correlation does not imply causality. We can't say that a change in one variable causes a change in the other, only that the changes move together. The second is that correlations should be subject to testing to determine whether they're statistically significant. Those tests are beyond the scope of this book but worth studying on your own.

Nevertheless, the SQL `corr(Y, X)` function is a handy tool for quickly checking correlations between variables.

Predicting Values with Regression Analysis

Researchers also want to predict values using available data. For example, let's say 30 percent of a county's population has a bachelor's degree or higher. Given the trend in our data, what would we expect that county's median household income to be? Likewise, for each percent increase in education, how much increase, on average, would we expect in income?

We can answer both questions using *linear regression*. Simply put, the regression method finds the best linear equation, or straight line, that describes the relationship between an independent variable (such as education) and a dependent variable (such as income). We can then look at points along this line to predict values where we don't have observations. Standard ANSI SQL and PostgreSQL include functions that perform linear regression.

Figure 11-2 shows our previous scatterplot with a regression line added.

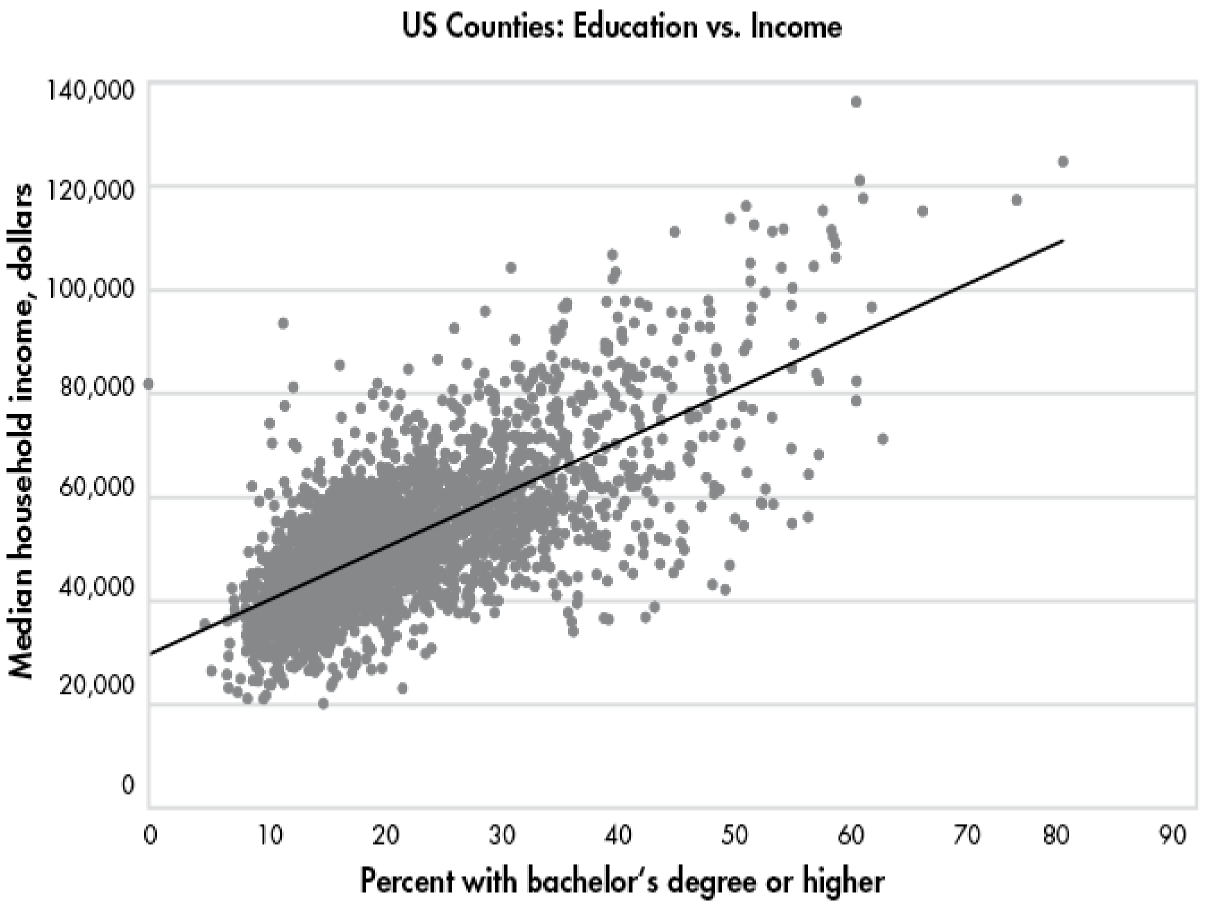


Figure 11-2: Scatterplot with least squares regression line showing the relationship between education and income

The straight line running through the middle of all the data points is called the *least squares regression line*, which approximates the "best fit" for a straight line that best describes the relationship between the variables. The equation for the regression line is like the *slope-intercept* formula you might remember from high school math but written using differently named variables: $Y = bX + a$. Here are the formula's components:

Y is the predicted value, which is also the value on the y-axis, or dependent variable.

b is the slope of the line, which can be positive or negative. It measures how many units the y-axis value will increase or decrease for each unit of the x-axis value.

X represents a value on the x-axis, or independent variable.

a is the y-intercept, the value at which the line crosses the y-axis when the *X* value is zero.

Let's apply this formula using SQL. Earlier, we questioned the expected median household income in a county where than 30 percent or more of the population had a bachelor's degree. In our scatterplot, the percentage with bachelor's degrees falls along the x-axis, represented by *X* in the calculation. Let's plug that value into the regression line formula in place of *X*:

$$Y = b(30) + a$$

To calculate *Y*, which represents the predicted median household income, we need the line's slope, *b*, and the y-intercept, *a*. To get these values, we'll use the SQL functions `regr_slope(Y, X)` and `regr_intercept(Y, X)`, as shown in *Listing 11-4*.

```
SELECT
    round(
        regr_slope(median_hh_income,
pct_bachelors_higher)::numeric, 2
        ) AS slope,
    round(
        regr_intercept(median_hh_income,
pct_bachelors_higher)::numeric, 2
        ) AS y_intercept
FROM acs_2014_2018_stats;
```

Listing 11-4: Regression slope and intercept functions

Using the `median_hh_income` and `pct_bachelors_higher` variables as inputs for both functions, we'll set the resulting value of the `regr_slope(Y, X)` function as `slope` and the output for the `regr_intercept(Y, X)` function as `y_intercept`.

Run the query; the result should show the following:

```
slope      y_intercept
-------    -----------
1016.55       29651.42
```

The `slope` value shows that for every one-unit increase in bachelor's degree percentage, we can expect a county's median household income will increase by $1,016.55. The `y_intercept` value shows that when the regression line crosses the y-axis, where the percentage with bachelor's degrees is at 0, the y-axis value is 29,651.42. Now let's plug both values into the equation to get our predicted value *Y*:

$$Y = 1016.55(30) + 29651.42$$
$$Y = 60147.92$$

Based on our calculation, in a county in which 30 percent of people age 25 and older have a bachelor's degree or higher, we can expect a median household income to be about $60,148. Of course, our data includes counties whose median income falls above and below that predicted value, but we expect this to be the case because our data points in the scatterplot don't line up perfectly along the regression line. Recall that the correlation coefficient we calculated was 0.70, indicating a strong but not perfect relationship between education and income. Other factors likely contributed to variations in income, such as the types of jobs available in each county.

Finding the Effect of an Independent Variable with r-Squared

Beyond determining the direction and strength of the relationship between two variables, we can also calculate the extent that the variation in the *x* (independent) variable explains the variation in the *y* (dependent) variable. To do this we square the *r* value to find the *coefficient of determination*, better known as *r-squared*. An *r*-squared indicates the percentage of the variation that is explained by the independent variable, and is a value between zero and one. For example, if *r*-squared equals 0.1, we would say that the independent variable explains 10 percent of the variation in the dependent variable, or not much at all.

To find *r*-squared, we use the `regr_r2(Y, X)` function in SQL. Let's apply it to our education and income variables using the code in Listing 11-5.

```
SELECT round(
        regr_r2(median_hh_income,
pct_bachelors_higher)::numeric, 3
        ) AS r_squared
FROM acs_2014_2018_stats;
```

Listing 11-5: Calculating the coefficient of determination, or r*-squared*

This time we'll round off the output to the nearest thousandth place and alias the result to `r_squared`. The query should return the following result:

```
r_squared
---------
    0.490
```

The *r*-squared value of `0.490` indicates that about 49 percent of the variation in median household income among counties can be explained by the percentage of people with a bachelor's degree or higher in that county. Any number of factors could explain the other 51 percent, and statisticians will typically test numerous combinations of variables to determine what they are.

Before you use these numbers in a headline or presentation, it's worth revisiting the following points:

Correlation doesn't prove causality. For verification, do a Google search on "correlation and causality." Many variables correlate well but have no meaning. (See *https://www.tylervigen.com/spurious-correlations* for examples of correlations that don't prove causality, including the correlation between divorce rate in Maine and margarine consumption.) Statisticians usually perform *significance testing* on the results to make sure values are not simply the result of randomness.

Statisticians also apply additional tests to data before accepting the results of a regression analysis, including whether the variables follow the standard bell curve distribution and meet other criteria for a valid result.

Let's explore two additional concepts before wrapping up our look at statistical functions.

Finding Variance and Standard Deviation

Variance and *standard deviation* describe the degree to which a set of values varies from the average of those values. Variance, often used in finance, is the average of each number's squared distance from the average. The more dispersion in a set of values, the greater the variance. A stock market trader can use variance to measure the volatility of a particular stock —how much its daily closing values tend to vary from the average. That could indicate how risky an investment the stock might be.

Standard deviation is the square root of the variance and is most useful for assessing data whose values form a normal distribution, usually visualized as a symmetrical *bell curve*. In a *normal distribution*, about two-thirds of values fall within one standard deviation of the average; 95 percent are within two standard deviations. The standard deviation of a set of values, therefore, helps us understand how close most of our values are to the average. For example, consider a study that found the average height of adult US women is about 65.5 inches with a standard deviation of 2.5 inches. Given that heights are normally distributed, that means about two-thirds of women are within 2.5 inches of the average, or 63 inches to 68 inches tall.

When calculating variance and standard deviation, note that they report different units. Standard deviation is expressed in the same units as the values, while variance is not—it reports a number that is larger than the units, on a scale of its own.

These are the functions for calculating variance:

`var_pop(numeric)` Calculates the population variance of the input values. In this context, *population* refers to a dataset that contains all possible values, as opposed to a sample that just contains a portion of all possible values.

`var_samp(numeric)` Calculates the sample variance of the input values. Use this with data that is sampled from a population, as in a random sample survey.

For calculating standard deviation, we use these:

`stddev_pop(numeric)` Calculates the population standard deviation.

`stddev_samp(numeric)` Calculates the sample standard deviation.

With functions covering correlation, regression, and other descriptive statistics, you have a basic toolkit for obtaining a preliminary survey of your data before doing more rigorous analysis. All these topics are worth in-depth study to better understand when you might use them and what they measure. A classic, easy-to-understand resource I recommend is the book *Statistics* by David Freedman, Robert Pisani, and Roger Purves.

Creating Rankings with SQL

Rankings make the news often. You'll see them used anywhere from weekend box-office charts to sports teams' league standings. With SQL you can create numbered rankings in your query results, which are useful for tasks such as tracking changes over several years. You can also simply use a ranking as a fact on its own in a report. Let's explore how to create rankings using SQL.

Ranking with rank() and dense_rank()

Standard ANSI SQL includes several ranking functions, but we'll just focus on two: `rank()` and `dense_rank()`. Both are *window functions*, which are defined as functions that perform calculations across a set of rows relative to the current row. Unlike aggregate functions, which combine rows to calculate values, with window functions the query first generates a set of rows, and then the window function runs across the result set to calculate the value it will return.

The difference between `rank()` and `dense_rank()` is the way they handle the next rank value after a tie: `rank()` includes a gap in the rank order, but `dense_rank()` does not. This concept is easier to understand in action, so let's look at an example. Consider a Wall Street analyst who covers the highly competitive widget manufacturing market. The analyst wants to rank companies by their annual output. The SQL statements in *Listing 11-6* create and fill a table with this data and then rank the companies by widget output.

```
CREATE TABLE widget_companies (
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    company text NOT NULL,
    widget_output integer NOT NULL
);

INSERT INTO widget_companies (company, widget_output)
VALUES
    ('Dom Widgets', 125000),
    ('Ariadne Widget Masters', 143000),
    ('Saito Widget Co.', 201000),
    ('Mal Inc.', 133000),
    ('Dream Widget Inc.', 196000),
    ('Miles Amalgamated', 620000),
    ('Arthur Industries', 244000),
    ('Fischer Worldwide', 201000);

SELECT
    company,
    widget_output,
  1 rank() OVER (ORDER BY widget_output DESC),
  2 dense_rank() OVER (ORDER BY widget_output DESC)
FROM widget_companies
ORDER BY widget_output DESC;
```

Listing 11-6: Using the `rank()` and `dense_rank()` window functions

Notice the syntax in the `SELECT` statement that includes `rank()` 1 and `dense_rank()` 2. After the function names, we use the `OVER` clause and in parentheses place an expression that specifies the "window" of rows the function should operate on. The *window* is the set of rows relative to the current row, and in this case, we want both functions to work on all rows of the `widget_output` column, sorted in descending order. Here's the output:

```
company                     widget_output    rank
dense_rank
--------------------------  -------------    ----    ------
----
Miles Amalgamated                  620000       1         1
Arthur Industries                  244000       2         2
Fischer Worldwide                  201000       3         3
Saito Widget Co.                   201000       3         3
Dream Widget Inc.                  196000       5         4
Ariadne Widget Masters             143000       6         5
Mal Inc.                           133000       7         6
Dom Widgets                        125000       8         7
```

The columns produced by `rank()` and `dense_rank()` show each company's ranking based on the `widget_output` value from highest to lowest, with Miles Amalgamated at number one. To see how `rank()` and `dense_rank()` differ, check the fifth-row listing, Dream Widget Inc.

With `rank()`, Dream Widget Inc. is the fifth-highest-ranking company. Because `rank()` allows a gap in the order when a tie occurs, Dream placing fifth tells us there are four companies with more output. In contrast, `dense_rank()` doesn't allow a gap in the rank order so it places Dream Widget Inc. in fourth place. This reflects the fact that Dream has the fourth-highest widget output regardless of how many companies produced more.

Both ways of handling ties have merit, but in practice `rank()` is used most often. It's also what I recommend using, because it more accurately reflects the total number of companies ranked, shown by the fact that Dream Widget Inc. has four companies ahead of it in total output, not three.

Let's look at a more complex ranking example.

Ranking Within Subgroups with PARTITION BY

The ranking we just did was a simple overall ranking based on widget output. But sometimes you'll want to produce ranks within groups of rows in a table. For example, you might want to rank government employees by salary within each department or rank movies by box-office earnings within each genre.

To use window functions in this way, we'll add `PARTITION BY` to the `OVER` clause. A `PARTITION BY` clause divides table rows according to values in a column we specify.

Here's an example using made-up data about grocery stores. Enter the code in *Listing 11-7* to fill a table called `store_sales`.

```
CREATE TABLE store_sales (
    store text NOT NULL,
    category text NOT NULL,
    unit_sales bigint NOT NULL,
    CONSTRAINT store_category_key PRIMARY KEY (store,
category)
);
```

```
INSERT INTO store_sales (store, category, unit_sales)
VALUES
    ('Broders', 'Cereal', 1104),
    ('Wallace', 'Ice Cream', 1863),
    ('Broders', 'Ice Cream', 2517),
    ('Cramers', 'Ice Cream', 2112),
    ('Broders', 'Beer', 641),
    ('Cramers', 'Cereal', 1003),
    ('Cramers', 'Beer', 640),
    ('Wallace', 'Cereal', 980),
    ('Wallace', 'Beer', 988);

SELECT
    category,
    store,
    unit_sales,
  1 rank() OVER (PARTITION BY category ORDER BY unit_sales
DESC)
FROM store_sales
2 ORDER BY category, rank() OVER (PARTITION BY category
         ORDER BY unit_sales DESC);
```

Listing 11-7: Applying `rank()` within groups using `PARTITION BY`

In the table, each row includes a store's product category and sales for that category. The final `SELECT` statement creates a result set showing how each store's sales ranks within each category. The new element is the addition of `PARTITION BY` in the `OVER` clause 1. In effect, the clause tells the program to create rankings one category at a time, using the store's unit sales in descending order.

To display the results by category and rank, we add an `ORDER BY` clause 2 that includes the `category` column and the same `rank()` function syntax. Here's the output:

```
category     store      unit_sales    rank
---------    -------    ----------    ----
Beer         Wallace           988       1
Beer         Broders           641       2
Beer         Cramers           640       3
Cereal       Broders          1104       1
Cereal       Cramers          1003       2
Cereal       Wallace           980       3
Ice Cream    Broders          2517       1
Ice Cream    Cramers          2112       2
Ice Cream    Wallace          1863       3
```

Rows for each category are ordered by category unit sales with the `rank` column displaying the ranking.

Using this table, we can see at a glance how each store ranks in a food category. For instance, Broders tops sales for cereal and ice cream, but Wallace wins in the beer category. You can apply this concept to many other scenarios: for each auto manufacturer, finding the vehicle with the most consumer complaints; figuring out which month had the most rainfall in each of the last 20 years; finding the team with the most wins against left-handed pitchers; and so on.

Calculating Rates for Meaningful Comparisons

Rankings based on raw counts aren't always meaningful; in fact, they can be misleading. Consider birth statistics: the US National Center for Health Statistics (NCHS) reported that in 2019, there were 377,599 babies born in the state of Texas and 46,826 born in the state of Utah. So, women in Texas are more likely to have babies, right? Not so fast. In 2019, Texas' estimated population was nine times as much as Utah's. Given that context, comparing the plain number of births in the two states isn't very meaningful.

A more accurate way to compare these numbers is to convert them to rates. Analysts often calculate a rate per 1,000 people, or some multiple of that number, to allow an apples-to-apples comparison. For example, the fertility rate—the number of births per 1,000 women ages 15 to 44—was 62.5 for Texas in 2019 and 66.7 for Utah, according to the NCHS. So, despite the smaller number of births, on a per-1,000 rate, women in Utah actually had more children.

The math behind this is simple. Let's say your town had 115 births and a population of 2,200 women ages 15 to 44. You can find the per-1,000 rate as follows:

$$(115 / 2{,}200) \times 1{,}000 = 52.3$$

In your town, there were 52.3 births per 1,000 women ages 15 to 44, which you can now compare to other places regardless of their size.

Finding Rates of Tourism-Related Businesses

Let's try calculating rates using SQL and census data. We'll join two tables: the census population estimates you imported in Chapter 5 plus data I compiled about tourism-related businesses from the census' County Business Patterns program. (You can read about the program methodology at *https://www.census.gov/programs-surveys/cbp/about.html*.)

Listing 11-8 contains the code to create and fill the business patterns table. Remember to point the script to the location in which you've saved the CSV file *cbp_naics_72_establishments.csv*, which you can download from GitHub via the link at *https://nostarch.com/practical-sql-2nd-edition/*.

```
CREATE TABLE cbp_naics_72_establishments (
    state_fips text,
    county_fips text,
    county text NOT NULL,
    st text NOT NULL,
    naics_2017 text NOT NULL,
    naics_2017_label text NOT NULL,
    year smallint NOT NULL,
    establishments integer NOT NULL,
    CONSTRAINT cbp_fips_key PRIMARY KEY (state_fips,
county_fips)
);

COPY cbp_naics_72_establishments
FROM 'C:\YourDirectory\cbp_naics_72_establishments.csv'
WITH (FORMAT CSV, HEADER);

SELECT *
FROM cbp_naics_72_establishments
ORDER BY state_fips, county_fips
LIMIT 5;
```

Listing 11-8: Creating and filling a table for census county business pattern data

Once you've imported the data, run the final `SELECT` statement to view the first few rows of the table. Each row contains descriptive information about a county along with the number of business establishments that fall under code 72 of the North American Industry Classification System (NAICS). Code 72 covers "Accommodation and Food Services" establishments, mainly hotels, inns, bars, and restaurants. The number of those businesses in a county is a good proxy for the amount of tourist and recreation activity in the area.

Let's find out which counties have the highest concentration of such businesses per 1,000 population, using the code in Listing 11-9.

```
SELECT
    cbp.county,
    cbp.st,
    cbp.establishments,
    pop.pop_est_2018,
  1 round( (cbp.establishments::numeric / pop.pop_est_2018) *
1000, 1 )
        AS estabs_per_1000
FROM cbp_naics_72_establishments cbp JOIN
us_counties_pop_est_2019 pop
    ON cbp.state_fips = pop.state_fips
    AND cbp.county_fips = pop.county_fips
2 WHERE pop.pop_est_2018 >= 50000
ORDER BY cbp.establishments::numeric / pop.pop_est_2018 DESC;
```

Listing 11-9: Finding business rates per thousand population in counties with 50,000 or more people

Overall, this syntax should look familiar. In Chapter 5, you learned that when dividing an integer by an integer, one of the values must be a `numeric` or `decimal` for the result to include decimal places. We do that in the rate calculation 1 with PostgreSQL's double-colon shorthand. Because we don't need many decimal places, we wrap the statement in the `round()` function to round off the output to the nearest tenth. Then we give the calculated column an alias of `estabs_per_1000` for easy reference.

Also, we use a `WHERE` clause 2 to limit our results to counties with 50,000 or more people. That's an arbitrary value that lets us see how rates compare within a group of more-populous, better-known counties. Here’s a portion of the results, sorted with highest rates at top:

```
    county             st      establishments  pop_est_2018
estabs_per_1000
------------------ ----------- --------------- -------------
---------------
Cape May County    New Jersey              925         92446
10.0
Worcester County   Maryland                453         51960
8.7
Monroe County      Florida                 540         74757
7.2
Warren County      New York                427         64215
6.6
New York County    New York              10428       1629055
6.4
Hancock County     Maine                   337         54734
6.2
Sevier County      Tennessee               570         97895
5.8
Eagle County       Colorado                309         54943
5.6
--snip--
```

The counties that have the highest rates make sense. Cape May County, New Jersey, is home to numerous beach resort towns on the Atlantic Ocean and Delaware Bay. Worcester County, Maryland, contains Ocean City and other beach attractions. And Monroe County, Florida, is best known for its vacation hotspot, the Florida Keys. Sense a pattern?

Smoothing Uneven Data

A *rolling average* is an average calculated for each time period in a dataset, using a moving window of rows as input each time. Think of a hardware store: it might sell 20 hammers on Monday, 15 hammers on Tuesday, and just a few the rest of the week. The next week, hammer sales might spike on Friday. To find the big-picture story in such uneven data, we can smooth numbers by calculating the rolling average, sometimes called a *moving average*.

Here are two weeks of hammer sales at that hypothetical hardware store:

```
Date         Hammer sales  Seven-day average
----------   ------------  -----------------
2022-05-01        0
2022-05-02       20
2022-05-03       15
2022-05-04        3
2022-05-05        6
2022-05-06        1
1 2022-05-07      1             6.6
2 2022-05-08      2             6.9
2022-05-09       18               6.6
2022-05-10       13               6.3
2022-05-11        2               6.1
2022-05-12        4               5.9
2022-05-13       12               7.4
2022-05-14        2               7.6
```

Let's say that for every day we want to know the average sales over the last seven days (we can choose any period, but a week is an intuitive unit). Once we have seven days of data 1, we calculate the average of sales over the seven-day period that includes the current day. The average of hammer sales from May 1 to May 7, 2022, is `6.6` per day.

The next day 2, we again average sales over the most recent seven days, from May 2 to May 8, 2022. The result is `6.9` per day. As we continue each day, despite the ups and downs in the daily sales, the seven-day average remains fairly steady. Over a long period of time, we'll be able to better discern a trend.

Let's use the window function syntax again to perform this calculation using the code in *Listing 11-10*. The code and data are available with all the book's resources in GitHub, available via *https://nostarch.com/practical-sql-2nd-edition/*. Remember to change *`C:\YourDirectory\`* to the location of the CSV file.

```
1 CREATE TABLE us_exports (
      year smallint,
      month smallint,
      citrus_export_value bigint,
      soybeans_export_value bigint
  );
```

```
2 COPY us_exports
   FROM 'C:\YourDirectory\us_exports.csv'
   WITH (FORMAT CSV, HEADER);

3 SELECT year, month, citrus_export_value
   FROM us_exports
   ORDER BY year, month;

4 SELECT year, month, citrus_export_value,
       round(
          5 avg(citrus_export_value)
               6 OVER(ORDER BY year, month
                    7 ROWS BETWEEN 11 PRECEDING AND CURRENT
   ROW), 0)
           AS twelve_month_avg
   FROM us_exports
   ORDER BY year, month;
```

Listing 11-10: Creating a rolling average for export data

We create a table 1 and use `COPY` 2 to insert data from *us_exports.csv*. This file contains data showing the monthly dollar value of US exports of citrus fruit and soybeans, two commodities whose sales are tied to the growing season. The data comes from the US Census Bureau's international trade division at *https://usatrade.census.gov/*.

The first `SELECT` statement 3 lets you view the monthly citrus export data, which covers every month from 2002 through summer 2020. The last dozen rows should look like this:

```
year month citrus_export_value
---- ----- -------------------
--snip--
2019     9            14012305
2019    10            26308151
2019    11            60885676
2019    12            84873954
2020     1           110924836
2020     2           171767821
2020     3           201231998
2020     4           122708243
2020     5            75644260
2020     6            36090558
```

```
2020     7           20561815
2020     8           15510692
```

Notice the pattern: the value of citrus fruit exports is highest in winter months, when the growing season is paused in the northern hemisphere and countries need imports to meet demand. We'll use the second `SELECT` statement 4 to compute a 12-month rolling average so we can see, for each month, the annual trend in exports.

In the `SELECT` values list, we place an `avg()` 5 function to calculate the average of the values in the `citrus_export_value` column. We follow the function with an `OVER` clause 6 that has two elements in parentheses: an `ORDER BY` clause that sorts the data for the period we plan to average, and the number of rows to average, using the keywords `ROWS BETWEEN 11 PRECEDING AND CURRENT ROW` 7. This tells PostgreSQL to limit the window to the current row and the 11 rows before it—12 total.

We wrap the entire statement, from the `avg()` function through the `OVER` clause, in a `round()` function to limit the output to whole numbers. The last dozen rows of your query result should be as follows:

```
year month citrus_export_value twelve_month_avg
---- ----- ------------------- ----------------
--snip--
2019     9            14012305         74465440
2019    10            26308151         74756757
2019    11            60885676         74853312
2019    12            84873954         74871644
2020     1           110924836         75099275
2020     2           171767821         78874520
2020     3           201231998         79593712
2020     4           122708243         78278945
2020     5            75644260         77999174
2020     6            36090558         78045059
2020     7            20561815         78343206
2020     8            15510692         78376692
```

Notice the 12-month average is far more consistent. If we want to see the trend, it's helpful to graph the results using Excel or a stats program. *Figure 11-3* shows the monthly totals from 2015 through August 2020 in bars, with the 12-month average as a line.

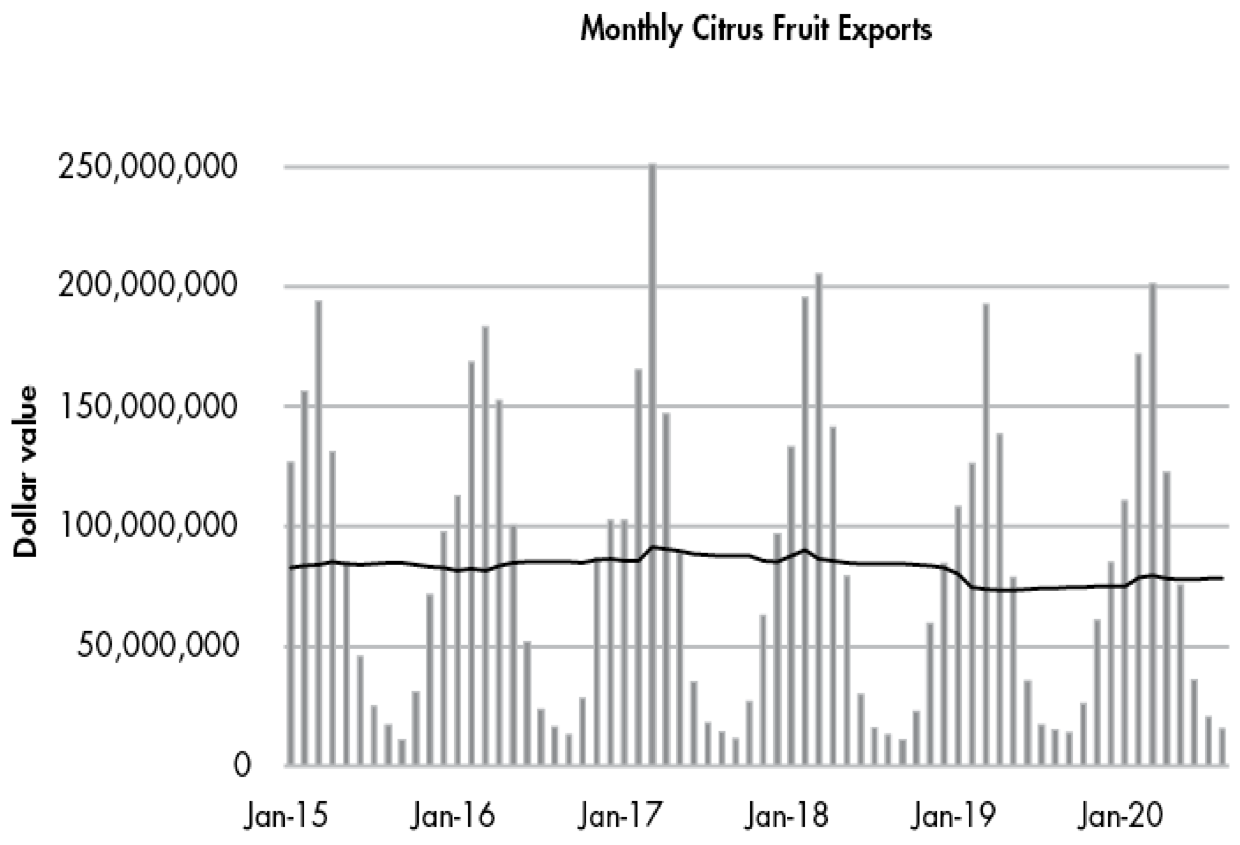


Figure 11-3: Monthly citrus fruit exports with 12-month rolling average

Based on the rolling average, citrus fruit exports were generally steady until 2019 and then trended down before recovering slightly in 2020. It's difficult to discern that movement from the monthly data, but the rolling average makes it apparent.

The window function syntax offers multiple options for analysis. For example, instead of calculating a rolling average, you could substitute the `sum()` function to find the rolling total over a time period. If you calculated a seven-day rolling sum, you'd know the weekly total ending on any day in your dataset.

NOTE

Calculating rolling averages or sums works best when there are no breaks in the time periods in your data. A missing month, for example, will turn a 12-month sum into a 13-month sum because the window function pays attention to rows, not dates.

SQL offers additional window functions. Check the official PostgreSQL documentation at *https://www.postgresql.org/docs/current/tutorial-window.html* for an overview of window functions, and check *https://www.postgresql.org/docs/current/functions-window.html* for a listing of window functions.

Wrapping Up

Now your SQL analysis toolkit includes ways to find relationships among variables using statistical functions, create rankings from ordered data, smooth spiky data to find trends, and properly compare raw numbers by turning them into rates. That toolkit is starting to look impressive!

Next, we'll dive deeper into date and time data, using SQL functions to extract the information we need.

TRY IT YOURSELF

Test your new skills with the following questions:

In *Listing 11-2*, the correlation coefficient, or `r` value, of the variables `pct_bachelors_higher` and `median_hh_income` was about 0.70. Write a query using the same dataset to show the correlation between `pct_masters_higher` and `median_hh_income`. Is the `r` value higher or lower? What might explain the difference?

Using the exports data, create a 12-month rolling sum using the values in the column `soybeans_export_value` and the query pattern from *Listing 11-8*. Copy and paste the results from the pgAdmin output pane and graph the values using Excel. What trend do you see?

As a bonus challenge, revisit the libraries data in the table `pls_fy2018_libraries` in Chapter 9. Rank library agencies based on the rate of visits per 1,000 population (column `popu_lsa`), and limit the query to agencies serving 250,000 people or more.

12
WORKING WITH DATES AND TIMES

Columns filled with dates and times can indicate *when* events happened or *how long* they took, and that can lead to interesting lines of inquiry. What patterns exist in the moments on a timeline? Which events were shortest or longest? What relationships exist between a particular activity and the time of day or season in which it occurred?

In this chapter, we'll explore these kinds of questions using SQL data types for dates and times and their related functions. We'll start with a closer look at data types and functions related to dates and times. Then we'll explore a dataset on trips by New York City taxicabs to look for patterns and try to discover what, if any, story the data tells. We'll also explore time zones using Amtrak data to calculate the duration of train trips across the United States.

Understanding Data Types and Functions for Dates and Times

Chapter 4 explored primary SQL data types, but to review, here are the four data types related to dates and times:

`timestamp` Records date and time. You will almost always want to add the keywords `with time zone` to ensure that times stored include time zone information. Otherwise, times recorded around the globe become impossible to compare. The format `timestamp with time zone` is part of the SQL standard; with PostgreSQL you can specify the same data type using `timestamptz`. You can specify time zones in three different formats: its UTC offset, an area/location designator, or a standard abbreviation. If you supply a time without a time zone to a `timestamptz` column, the database will add time zone information using your server's default setting.

`date` Records only the date and is part of the SQL standard. PostgreSQL accepts several date formats. For example, valid formats for adding the 21st day of September 2022 are `September 21, 2022` or `9/21/2022`. I recommend using *`YYYY-MM-DD`* (or `2022-09-21`), which is the ISO 8601 international standard format and also the default PostgreSQL date output. Using the ISO format helps avoid confusion when sharing data internationally.

`time` Records only the time and is part of the SQL standard. Adding `with time zone` makes the column time zone aware, but without a date the time zone will be meaningless. Given that, using `time with time zone` and its PostgreSQL shortcut `timetz` is strongly discouraged. The ISO 8601 format is *`HH:MM:SS`*, where *`HH`* represents the hour, *`MM`* the minutes, and *`SS`* the seconds.

`interval` Holds a value that represents a unit of time expressed in the format *`quantity unit`*. It doesn't record the start or end of a period, only its duration. Examples include `12 days` or `8 hours`. It's also part of the SQL standard, although PostgreSQL-specific syntax offers more options.

The first three data types, `date`, `time`, and `timestamp with time zone` (or `timestamptz`), are known as *datetime types* whose values are called *datetimes*. The `interval` value is an *interval type* whose values are *intervals*. All four data types can track the system clock and the nuances of the calendar. For example, `date` and `timestamp with time zone` recognize that June has 30 days. If you try to use June 31, PostgreSQL will display an error: `date/time field value out of range`. Likewise, the date February 29 is valid only in a leap year, such as 2024.

Manipulating Dates and Times

We can use SQL functions to perform calculations on dates and times or extract their components. For example, we can retrieve the day of the week from a timestamp or extract just the month from a date. ANSI SQL outlines a handful of functions for this purpose, but many database managers (including MySQL and Microsoft SQL Server) deviate from the standard to implement their own date and time data types, syntax, and function names. If you're using a database other than PostgreSQL, check its documentation.

Let's review how to manipulate dates and times using PostgreSQL functions.

Extracting the Components of a timestamp Value

It's not unusual to need just one piece of a date or time value for analysis, particularly when you're aggregating results by month, year, or even minute. We can extract these components using the PostgreSQL `date_part()` function. Its format looks like this:

```
date_part(text, value)
```

The function takes two inputs. The first is a string in `text` format that represents the part of the date or time to extract, such as `hour`, `minute`, or `week`. The second is the `date`, `time`, or `timestamp` value. To see the `date_part()` function in action, we'll execute it multiple times on the same value using the code in Listing 12-1.

```
SELECT
  date_part('year', '2022-12-01 18:37:12 EST'::timestamptz)
AS year,
  date_part('month', '2022-12-01 18:37:12 EST'::timestamptz)
AS month,
  date_part('day', '2022-12-01 18:37:12 EST'::timestamptz) AS
day,
  date_part('hour', '2022-12-01 18:37:12 EST'::timestamptz)
AS hour,
  date_part('minute', '2022-12-01 18:37:12 EST'::timestamptz)
AS minute,
  date_part('seconds', '2022-12-01 18:37:12
EST'::timestamptz) AS seconds,
  date_part('timezone_hour', '2022-12-01 18:37:12
EST'::timestamptz) AS tz,
  date_part('week', '2022-12-01 18:37:12 EST'::timestamptz)
AS week,
  date_part('quarter', '2022-12-01 18:37:12
EST'::timestamptz) AS quarter,
  date_part('epoch', '2022-12-01 18:37:12 EST'::timestamptz)
AS epoch;
```

Listing 12-1: Extracting components of a `timestamp` value using `date_part()`

Each column statement in this `SELECT` query first uses a string to name the component we want to extract: `year`, `month`, `day`, and so on. The second input uses the string `2022-12-01 18:37:12 EST` cast as a `timestamp with time zone` with the PostgreSQL double-colon syntax and the `timestamptz` shorthand. We specify that this timestamp occurs in the Eastern time zone using the Eastern Standard Time (EST) designation.

Here's the output as shown on my computer. The database converts the values to reflect your PostgreSQL time zone setting, so your output might be different; for example, if it's set to the US Pacific time zone, the hour will show as `15`:

```
year    month    day    hour    minute    seconds    tz
week    quarter    epoch
----    -----    ---    ----    ------    -------    --    --
--    -------    ----------
2022       12      1      18        37         12    -5
48          4    1669937832
```

Each column contains a single component of the timestamp that represents 6:37:12 PM on December 1, 2022. The first six values are easy to recognize from the original timestamp, but the last four deserve an explanation.

In the `tz` column, PostgreSQL reports back the hours difference, or *offset*, from Coordinated Universal Time (UTC), the time standard for the world. The value of UTC is +/− 00:00, so `-5` specifies a time zone five hours behind UTC. From November through early March, UTC -5 represents the Eastern time zone. In March, when the Eastern time zone moves to daylight saving time and clocks "spring forward" an hour, its UTC offset changes to -4. (For a map of UTC time zones, see https://en.wikipedia.org/wiki/Coordinated_Universal_Time#/media/File:Standard_World_Time_Zones.tif.)

NOTE

You can derive the UTC offset from the time zone but not vice versa. Each UTC offset can refer to multiple named time zones plus standard and daylight saving time variants.

The `week` column shows that December 1, 2022, falls in the 48th week of the year. This number is determined by ISO 8601 standards, which start each week on a Monday. A week at the end of a year can extend from December into January of the following year.

The `quarter` column shows that our test date is part of the fourth quarter of the year. The `epoch` column shows a measurement, which is used in computer systems and programming languages, that represents the number of seconds elapsed before or after 12 AM, January 1, 1970, at UTC 0. A positive value designates a time since that point; a negative value designates a time before it. In this example, 1,669,937,832 seconds elapsed between January 1, 1970, and the timestamp. Epoch can be useful for comparing two timestamps mathematically on an absolute scale.

NOTE

Proceed with caution with epoch times. PostgreSQL's `date_part()` *returns epoch time as a double precision type, which is subject to floating-point computational errors (see Chapter 4). Epoch time also faces the so-called Year 2038 problem, when epoch values will grow too large for some computer systems to store.*

PostgreSQL also supports the SQL-standard `extract()` function, which parses datetimes in the same way as the `date_part()` function. I've featured `date_part()` here instead for two reasons. First, its name helpfully reminds us what it does. Second, `extract()` isn't widely supported by other database managers. Most notably, it's absent in Microsoft's SQL Server. Nevertheless, if you need to use `extract()`, the syntax takes this form:

```
extract(text from value)
```

To replicate the first `date_part()` example in *Listing 12-1* where we pull the year from the timestamp, we'd set up `extract()` like this (note that we don't need single quotes around the time unit, in this case `year`):

```
extract(year from '2022-12-01 18:37:12 EST'::timestamptz)
```

PostgreSQL provides additional components you can extract or calculate from dates and times. For the full list of functions, see the documentation at *https://www.postgresql.org/docs/current/functions-datetime.html*.

Creating Datetime Values from timestamp Components

It's not unusual to come across a dataset in which the year, month, and day exist in separate columns, and you might want to create a datetime value from these components. To perform calculations on a date, it's helpful to combine and format those pieces correctly into one column.

You can use the following PostgreSQL functions to make datetime objects:

`make_date(year, month, day)` Returns a value of type `date`.

`make_time(hour, minute, seconds)` Returns a value of type `time` without time zone.

`make_timestamptz(year, month, day, hour, minute, second, time zone)` Returns a timestamp with time zone.

The variables for these three functions take `integer` types as input, with two exceptions: seconds are of the type `double precision` because you can supply fractions of seconds, and time zones must be specified with a `text` string that names the time zone.

Listing 12-2 shows examples of the three functions in action using components of February 22, 2022, for the date, and 6:04:30.3 PM in

Lisbon, Portugal for the time.

```
SELECT make_date(2022, 2, 22);
SELECT make_time(18, 4, 30.3);
SELECT make_timestamptz(2022, 2, 22, 18, 4, 30.3,
'Europe/Lisbon');
```

Listing 12-2: Three functions for making datetimes from components

When I run each query in order, the output on my computer is as follows. Again, yours may differ depending on your PostgreSQL time zone setting:

```
2022-02-22
18:04:30.3
2022-02-22 13:04:30.3-05
```

Notice that on my computer the timestamp in the third line shows `13:04:30.3`, which is five hours behind the time input to the function: `18:04:30.3`. That output is appropriate because Lisbon's time zone is at UTC 0, and my PostgreSQL is set to the Eastern time zone, which is UTC – 5 in winter months. We'll explore working with time zones in more detail, and you'll learn to adjust its display, in the "Working with Time Zones" section.

Retrieving the Current Date and Time

If you need to record the current date or time as part of a query—when updating a row, for example—standard SQL provides functions for that too. The following functions record the time as of the start of the query:

`current_timestamp` Returns the current timestamp with time zone. A shorthand PostgreSQL-specific version is `now()`.

`localtimestamp` Returns the current timestamp without time zone. Avoid using `localtimestamp`, as a timestamp without a time zone can't be placed in a global location and is thus meaningless.

`current_date` Returns the date.

`current_time` Returns the current time with time zone. Remember, though, without a date, the time alone with a time zone is useless.

`localtime` Returns the current time without time zone.

Because these functions record the time at the start of the query (or a collection of queries grouped under a *transaction*—see Chapter 10), they'll provide that same time throughout the execution of a query regardless of how long the query runs. So, if your query updates 100,000 rows and takes 15 seconds to run, any timestamp recorded at the start of the query will be applied to each row, and so each row will receive the same timestamp.

If, instead, you want the date and time to reflect how the clock changes during the execution of the query, you can use the PostgreSQL-specific `clock_timestamp()` function to record the current time as it elapses. That way, if you're updating 100,000 rows and inserting a timestamp each time, each row gets the time the row updated rather than the time at the start of the query. Note that `clock_timestamp()` can slow large queries and may be subject to system limitations.

Listing 12-3 shows `current_timestamp` and `clock_timestamp()` in action when inserting a row in a table.

```
CREATE TABLE current_time_example (
    time_id integer GENERATED ALWAYS AS IDENTITY,
  1 current_timestamp_col timestamptz,
  2 clock_timestamp_col timestamptz
);

INSERT INTO current_time_example
            (current_timestamp_col, clock_timestamp_col)
  3 (SELECT current_timestamp,
            clock_timestamp()
     FROM generate_series(1,1000));

SELECT * FROM current_time_example;
```

Listing 12-3: Comparing `current_timestamp` and `clock_timestamp()` during row insert

The code creates a table that includes two `timestamptz` columns (the PostgreSQL shorthand for `timestamp with time zone`). The first holds the result of the `current_timestamp` function 1, which records the time at the start of the `INSERT` statement that adds 1,000 rows to the table. To do that, we use the `generate_series()` function, which returns a set of integers starting with 1 and ending with 1,000. The second column holds the result of the `clock_timestamp()` function 2, which records the time of insertion of each row. You call both functions as part of the `INSERT` statement 3. Run the query, and the result from the final `SELECT` statement should show that the time in the `current_timestamp_col` is the same for all rows, whereas the time in `clock_timestamp_col` increases with each row inserted.

Working with Time Zones

Recording a timestamp is most useful when you know where on the globe that time occurred—whether in Asia, Eastern Europe, or one of the 12 time zones of Antarctica.

Sometimes, however, datasets contain no time zone data in their datetime columns. This isn't always a deal-breaker in terms of analyzing the data. If you know that every event happened in the same location—for example, readings from a temperature sensor in Bar Harbor, Maine—you can factor that into your analysis. Better, though, during import is to set your session time zone to represent the time zone of the data and load the datetimes into a `timestamptz` column. That strategy helps ward off dangerous misinterpretation of the data later.

Let's look at some strategies for managing how we work with time zones.

Finding Your Time Zone Setting

When working with timestamps that contain time zones, it's important to know your current time zone setting. If you installed PostgreSQL on your own computer, the server's default will be your local time zone. If you're connecting to a PostgreSQL database elsewhere, perhaps on a cloud provider such as Amazon Web Services, its time zone setting may be different than your own. To help avoid confusion, database administrators often set a shared server's time zone to UTC.

Listing 12-4 shows two ways to view your current time zone setting: the `SHOW` command with `timezone` keyword and the `current_setting()` function with a `timezone` argument.

```
SHOW timezone;
SELECT current_setting('timezone');
```

Listing 12-4: Viewing your current time zone setting

Running either statement will display your time zone setting, which will vary according to your operating system and locale. Entering the statements in Listing 12-4 into pgAdmin and running both my macOS and Linux computers returns `America/New_York`, one of several location names that falls into the Eastern time zone, which encompasses eastern Canada and the United States, the Caribbean, and parts of Mexico. On my Windows machine, the setting shows as `US/Eastern`.

NOTE

You can use `SHOW ALL;` to see the settings of every parameter on your PostgreSQL server.

Though both statements provide the same information, you may find `current_setting()` extra handy as an input to another function such as `make_timestamptz()`:

```
 SELECT make_timestamptz(2022, 2, 22, 18, 4, 30.3,
current_setting('timezone'));
```

Listing 12-5 shows how to retrieve all time zone names, abbreviations, and their UTC offsets.

```
SELECT * FROM pg_timezone_abbrevs ORDER BY abbrev;
SELECT * FROM pg_timezone_names ORDER BY name;
```

Listing 12-5: Showing time zone abbreviations and names

You can easily filter either of these `SELECT` statements with a `WHERE` clause to look up specific location names or time zones:

```
SELECT * FROM pg_timezone_names
WHERE name LIKE 'Europe%'
```

```
ORDER BY name;
```

This code should return a table listing that includes the time zone name, abbreviation, UTC offset, and a `boolean` column `is_dst` that notes whether the time zone is currently observing daylight saving time:

```
name                abbrev    utc_offset    is_dst
-----------------   ------    ----------    ------
Europe/Amsterdam    CEST      02:00:00      true
Europe/Andorra      CEST      02:00:00      true
Europe/Astrakhan    +04       04:00:00      false
Europe/Athens       EEST      03:00:00      true
Europe/Belfast      BST       01:00:00      true
--snip--
```

This is a faster way of looking up time zones than using Wikipedia. Now let's look at how to set the time zone to a particular value.

Setting the Time Zone

When you installed PostgreSQL, the server's default time zone was set as a parameter in *postgresql.conf*, a file that contains dozens of values read by PostgreSQL each time it starts. The location of *postgresql.conf* in your file system varies depending on your operating system and sometimes on the way you installed PostgreSQL. To make permanent changes to *postgresql.conf*, such as changing your time zone, you need to edit the file and restart the server, which might be impossible if you're not the owner of the machine. Changes to configurations might also have unintended consequences for other users or applications. Instead, we'll look at setting the time zone on a per-session basis, which should last as long as you're connected to the server, and then I'll cover working with *postgresql.conf* in more depth in Chapter 19. This solution is handy when you want to specify how you view a particular table or handle timestamps in a query.

To set the time zone for the current session while using pgAdmin, we use the command `SET TIME ZONE`, as shown in Listing 12-6.

```
1 SET TIME ZONE 'US/Pacific';

2 CREATE TABLE time_zone_test (
        test_date timestamptz
   );
3 INSERT INTO time_zone_test VALUES ('2023-01-01 4:00');

4 SELECT test_date
   FROM time_zone_test;

5 SET TIME ZONE 'US/Eastern';

6 SELECT test_date
   FROM time_zone_test;

7 SELECT test_date AT TIME ZONE 'Asia/Seoul'
   FROM time_zone_test;
```

Listing 12-6: Setting the time zone for a client session

First, we set the time zone to `US/Pacific` 1, which designates the Pacific time zone that covers western Canada and the United States along with Baja California in Mexico. The syntax `SET TIME ZONE` is part of the ANSI SQL standard. PostgreSQL also supports the nonstandard syntax `SET timezone TO`.

Second, we create a one-column table 2 with a data type of `timestamptz` and insert a single row to display a test result. Notice that the value inserted, `2023-01-01 4:00`, is a timestamp with no time zone 3. You'll encounter timestamps with no time zone often, particularly when you acquire datasets restricted to a specific location.

When executed, the first `SELECT` statement 4 returns `2023-01-01 4:00` as a timestamp that now contains time zone data:

```
test_date
----------------------
2023-01-01 04:00:00-08
```

Here, the `-08` shows that the Pacific time zone is eight hours behind UTC in January, when standard time is in effect. Because we set the pgAdmin client's time zone to `US/Pacific` for this session, any value without a time zone entered into a column that is time zone-aware will be set to Pacific time. If we had entered a date that falls during daylight saving time, the UTC offset would be `-07`.

NOTE

On the server, the `timestamp with time zone` (or `timestamptz` shorthand) data type always stores data as UTC internally; the time zone setting governs how it's displayed.

Now comes some fun. We change the time zone for this session to the Eastern time zone using the `SET` command 5 and the `US/Eastern` designation. Then, when we execute the `SELECT` statement 6 again, the result should be as follows:

```
test_date
----------------------
2023-01-01 07:00:00-05
```

In this example, two components of the timestamp have changed: the time is now `07:00`, and the UTC offset is `-05` because we're viewing the timestamp from the perspective of the Eastern time zone: 4 AM Pacific is 7 AM Eastern. The database converts the original Pacific time value to whatever time zone we set at 5.

Even more convenient is that we can view a timestamp through the lens of any time zone without changing the session setting. The final `SELECT` statement uses the `AT TIME ZONE` keywords 7 to display the timestamp in our session as the Korea standard time (KST) zone by specifying `Asia/Seoul`:

```
timezone
-------------------
2023-01-01 21:00:00
```

Now we know that the value of 4 AM in `US/Pacific` on January 1, 2023, is equivalent to 9 PM that same day in `Asia/Seoul`. Again, this syntax changes the output data, but the data on the server remains unchanged. When using the `AT TIME ZONE` keywords, also note this quirk: if the original value is a `timestamp with time zone`, the output is a `timestamp` with no time zone. If the original value has no time zone, the output is `timestamp with time zone`.

The ability of databases to track time zones is extremely important for accurate calculations of intervals, as you'll see next.

Performing Calculations with Dates and Times

We can perform simple arithmetic on datetime and interval types the same way we can on numbers. Addition, subtraction, multiplication, and division are all possible in PostgreSQL using the math operators `+`, `-`, `*`, and `/`. For example, you can subtract one date from another date to get an integer that represents the difference in days between the two dates. The following code returns an integer of `3`:

```
SELECT '1929-09-30'::date - '1929-09-27'::date;
```

The result indicates that these two dates are exactly three days apart.

Likewise, you can use the following code to add a time interval to a date to return a new date:

```
SELECT '1929-09-30'::date + '5 years'::interval;
```

This code adds five years to the date `1929-09-30` to return a timestamp value of `1934-09-30`.

More examples of math functions you can use with dates and times are available in the PostgreSQL documentation at *https://www.postgresql.org/docs/current/functions-datetime.html*. Let's explore some more practical examples using actual transportation data.

Finding Patterns in New York City Taxi Data

When I visit New York City, I usually take at least one ride in one of the thousands of iconic yellow cars that ferry hundreds of thousands of people across the city's five boroughs each day. The New York City Taxi and Limousine Commission releases data on monthly yellow taxi trips plus other for-hire vehicles. We'll use this large, rich dataset to put date functions to practical use.

The *nyc_yellow_taxi_trips.csv* file available from the book's resources on GitHub (via the link at *https://nostarch.com/practical-sql-2nd-edition/*) holds one day of yellow taxi trip records from June 1, 2016. Save the file to your computer and execute the code in *Listing 12-7* to build the `nyc_yellow_taxi_trips` table. Remember to change the file path in the `COPY` command to the location where you've saved the file and adjust the path format to reflect whether you're using Windows, macOS, or Linux.

```
1 CREATE TABLE nyc_yellow_taxi_trips (
       trip_id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
       vendor_id text NOT NULL,
       tpep_pickup_datetime timestamptz NOT NULL,
       tpep_dropoff_datetime timestamptz NOT NULL,
       passenger_count integer NOT NULL,
       trip_distance numeric(8,2) NOT NULL,
       pickup_longitude numeric(18,15) NOT NULL,
       pickup_latitude numeric(18,15) NOT NULL,
       rate_code_id text NOT NULL,
       store_and_fwd_flag text NOT NULL,
       dropoff_longitude numeric(18,15) NOT NULL,
       dropoff_latitude numeric(18,15) NOT NULL,
       payment_type text NOT NULL,
       fare_amount numeric(9,2) NOT NULL,
       extra numeric(9,2) NOT NULL,
       mta_tax numeric(5,2) NOT NULL,
       tip_amount numeric(9,2) NOT NULL,
       tolls_amount numeric(9,2) NOT NULL,
       improvement_surcharge numeric(9,2) NOT NULL,
       total_amount numeric(9,2) NOT NULL
  );

2 COPY nyc_yellow_taxi_trips (
       vendor_id,
       tpep_pickup_datetime,
       tpep_dropoff_datetime,
       passenger_count,
       trip_distance,
       pickup_longitude,
       pickup_latitude,
       rate_code_id,
       store_and_fwd_flag,
       dropoff_longitude,
       dropoff_latitude,
       payment_type,
       fare_amount,
       extra,
       mta_tax,
       tip_amount,
       tolls_amount,
       improvement_surcharge,
       total_amount
      )
   FROM 'C:\YourDirectory\nyc_yellow_taxi_trips.csv'
   WITH (FORMAT CSV, HEADER);

3 CREATE INDEX tpep_pickup_idx
   ON nyc_yellow_taxi_trips (tpep_pickup_datetime);
```

Listing 12-7: Creating a table and importing NYC yellow taxi data

The code in Listing 12-7 builds the table 1, imports the rows 2, and creates an index 3. In the `COPY` statement, we provide the names of columns because the input CSV file doesn't include the `trip_id` column that exists in the target table. That column is of type `bigint` and set as an auto-incrementing surrogate primary key. After your import is complete, you should have 368,774 rows, one for each yellow cab ride on June 1, 2016. You can count the rows in your table using the following code:

```
SELECT count(*) FROM nyc_yellow_taxi_trips;
```

Each row includes data on the number of passengers, the location of pickup and drop-off in latitude and longitude, and the fare and tips in US dollars. The data dictionary that describes all columns and codes is available at https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf. For these exercises, we're most interested in the timestamp columns `tpep_pickup_datetime` and `tpep_dropoff_datetime`, which represent the start and end times of the ride. (The Technology Passenger Enhancements Project [TPEP] is a program that in part includes automated collection of data about taxi rides.)

The values in both timestamp columns include the time zone: `-4`. That's the UTC offset for the Eastern time zone during summer, when daylight saving time is observed. If your PostgreSQL server isn't set to default to Eastern time, I suggest setting your time zone using the following code so your results will match mine:

```
SET TIME ZONE 'US/Eastern';
```

Now let's explore the patterns in these timestamps.

The Busiest Time of Day

One question you might ask of this data is when taxis provide the most rides. Is it morning or evening rush hour, or is there another time when ridership spikes? You can find the answer with a simple aggregation query that uses `date_part()`.

Listing 12-8 contains the query to count rides by hour using the pickup time as the input.

```
SELECT
   1 date_part('hour', tpep_pickup_datetime) AS trip_hour,
   2 count(*)
FROM nyc_yellow_taxi_trips
GROUP BY trip_hour
ORDER BY trip_hour;
```

Listing 12-8: Counting taxi trips by hour

In the query's first column 1, `date_part()` extracts the hour from `tpep_pickup_datetime` so we can group the number of rides by hour. Then we aggregate the number of rides in the second column via the `count()` function 2. The rest of the query follows the standard patterns for grouping and ordering the results, which should return 24 rows, one for each hour of the day:

```
trip_hour    count
---------    -----
        0     8182
        1     5003
```

```
         2     3070
         3     2275
         4     2229
         5     3925
         6    10825
         7    18287
         8    21062
         9    18975
        10    17367
        11    17383
        12    18031
        13    17998
        14    19125
        15    18053
        16    15069
        17    18513
        18    22689
        19    23190
        20    23098
        21    24106
        22    22554
        23    17765
```

Eyeballing the numbers, it's apparent that on June 1, 2016, New York City taxis had the most passengers between 6 PM and 10 PM, possibly reflecting commutes home plus the plethora of city activities on a summer evening. But to see the overall pattern, it's best to visualize the data. Let's do this next.

Exporting to CSV for Visualization in Excel

Charting data with a tool such as Microsoft Excel makes it easier to understand patterns, so I often export query results to a CSV file and work up a quick chart. *Listing 12-9* uses the query from the preceding example within a `COPY ... TO` statement, similar to *Listing 5-9* in Chapter 5.

```
COPY
    (SELECT
        date_part('hour', tpep_pickup_datetime) AS trip_hour,
        count(*)
    FROM nyc_yellow_taxi_trips
    GROUP BY trip_hour
    ORDER BY trip_hour
    )
```

```
TO 'C:\YourDirectory\hourly_taxi_pickups.csv'
WITH (FORMAT CSV, HEADER);
```

Listing 12-9: Exporting taxi pickups per hour to a CSV file

When I load the data into Excel and build a line graph, the day's pattern becomes more obvious and thought-provoking, as shown in Figure 12-1.

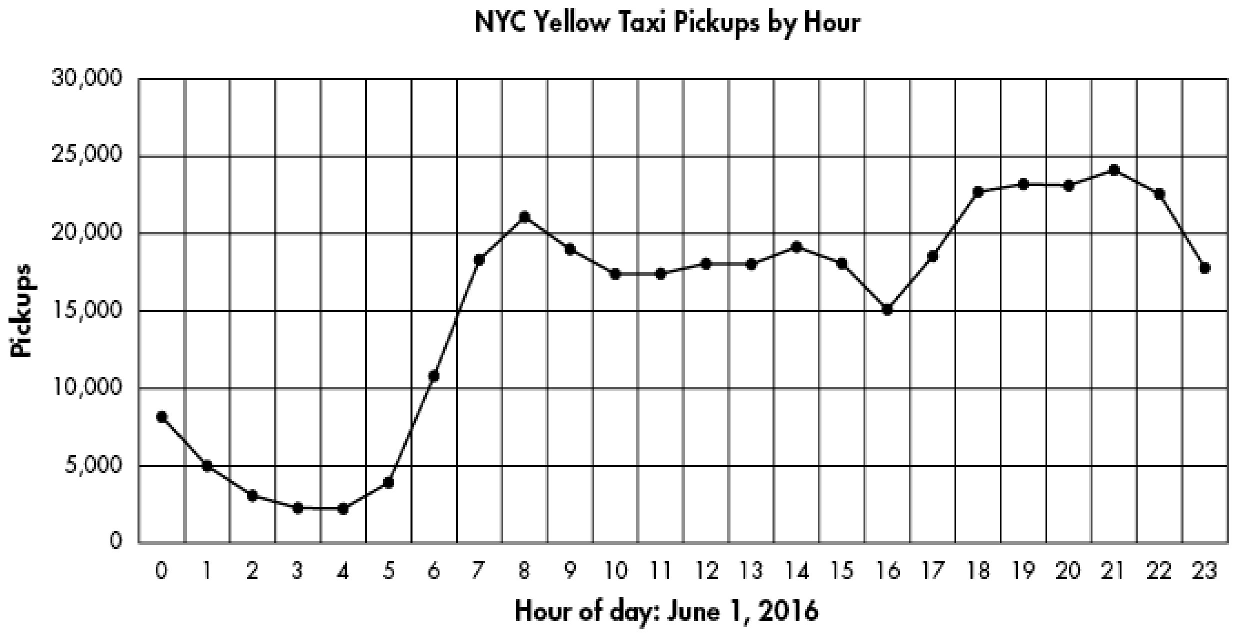


Figure 12-1: NYC yellow taxi pickups by hour

Rides bottomed out in the wee hours of the morning before rising sharply between 5 AM and 8 AM. Volume remained relatively steady throughout the day and increased again for evening rush hour after 5 PM. But there was a dip between 3 PM and 4 PM—why?

To answer that question, we would need to dig deeper to analyze data that spanned several days or even several months to see whether our data from June 1, 2016, is typical. We could use the `date_part()` function to compare trip volume on weekdays versus weekends by extracting the day of the week. To be even more ambitious, we could check weather reports and compare trips on rainy days versus sunny days. You can slice a dataset many ways to reach conclusions.

When Do Trips Take the Longest?

Let's investigate another interesting question: at which hour did taxi trips take the longest? One way to find an answer is to calculate the median trip time for each hour. The median is the middle value in an ordered set of values; it's often more accurate than an average for making comparisons because a few very small or very large values in the set won't skew the results as they would with the average.

In Chapter 6, we used the `percentile_cont()` function to find medians. We use it again in *Listing 12-10* to calculate median trip times.

```
SELECT
   1 date_part('hour', tpep_pickup_datetime) AS trip_hour,
   2 percentile_cont(.5)
      3 WITHIN GROUP (ORDER BY
              tpep_dropoff_datetime - tpep_pickup_datetime)
AS median_trip
FROM nyc_yellow_taxi_trips
GROUP BY trip_hour
ORDER BY trip_hour;
```

Listing 12-10: Calculating median trip time by hour

We're aggregating data by the hour portion of the timestamp column `tpep_pickup_datetime` again, which we extract using `date_part()` 1. For the input to the `percentile_cont()` function 2, we subtract the pickup time from the drop-off time in the `WITHIN GROUP` clause 3. The results show that the 1 PM hour has the highest median trip time of 15 minutes:

```
date_part    median_trip
---------    -----------
        0    00:10:04
        1    00:09:27
        2    00:08:59
        3    00:09:57
        4    00:10:06
        5    00:07:37
        6    00:07:54
        7    00:10:23
        8    00:12:28
        9    00:13:11
       10    00:13:46
       11    00:14:20
```

```
12    00:14:49
13    00:15:00
14    00:14:35
15    00:14:43
16    00:14:42
17    00:14:15
18    00:13:19
19    00:12:25
20    00:11:46
21    00:11:54
22    00:11:37
23    00:11:14
```

As we would expect, trip times are shortest in the early morning. This makes sense because less traffic early in the day means passengers are more likely to get to their destinations faster.

Now that we've explored ways to extract portions of the timestamp for analysis, let's dig deeper into analysis that involves intervals.

Finding Patterns in Amtrak Data

Amtrak, the nationwide rail service in America, offers several packaged trips across the United States. The All American, for example, is a train that departs from Chicago and stops in New York, New Orleans, Los Angeles, San Francisco, and Denver before returning to Chicago. Using data from the Amtrak website (*https://www.amtrak.com/*), we'll build a table with information for each segment of the trip. The trip spans four time zones, so we'll track the time zone with each arrival and departure. Then we'll calculate the duration of the journey at each segment and figure out the length of the entire trip.

Calculating the Duration of Train Trips

Using *Listing 12-11*, let's create a table that tracks the six segments of the All American route.

```
CREATE TABLE train_rides (
    trip_id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    segment text NOT NULL,
    departure timestamptz NOT NULL, 1
    arrival timestamptz NOT NULL
);

INSERT INTO train_rides (segment, departure, arrival) 2
VALUES
    ('Chicago to New York', '2020-11-13 21:30 CST', '2020-11-
14 18:23 EST'),
    ('New York to New Orleans', '2020-11-15 14:15 EST',
'2020-11-16 19:32 CST'),
    ('New Orleans to Los Angeles', '2020-11-17 13:45 CST',
'2020-11-18 9:00 PST'),
    ('Los Angeles to San Francisco', '2020-11-19 10:10 PST',
'2020-11-19 21:24 PST'),
    ('San Francisco to Denver', '2020-11-20 9:10 PST', '2020-
11-21 18:38 MST'),
    ('Denver to Chicago', '2020-11-22 19:10 MST', '2020-11-23
14:50 CST');

SET TIME ZONE 'US/Central'; 3

SELECT * FROM train_rides;
```

Listing 12-11: Creating a table to hold train trip data

First, we use the standard `CREATE TABLE` statement. Note that columns for departure and arrival times are set to `timestamptz` 1. Next, we insert rows that represent the six legs of the trip 2. Each timestamp input reflects the time zone of the city of departure or arrival. Specifying the city's time zone is the key to getting an accurate calculation of trip duration and accounting for time zone changes. It also accounts for annual changes to and from daylight saving time if they were to occur during the time span you're examining.

Next, we set the session to the Central time zone, the value for Chicago, using the `US/Central` designator 3. We'll use Central time as our reference when viewing the timestamps so that regardless of your and my machine's default time zones, we'll share the same view of the data.

The final `SELECT` statement should return the contents of the table like this:

```
trip_id  segment                         departure
arrival
-------  ----------------------------    --------------------
```

```
--    ----------------------
      1  Chicago to New York             2020-11-13 21:30:00-
06    2020-11-14 17:23:00-06
      2  New York to New Orleans         2020-11-15 13:15:00-
06    2020-11-16 19:32:00-06
      3  New Orleans to Los Angeles      2020-11-17 13:45:00-
06    2020-11-18 11:00:00-06
      4  Los Angeles to San Francisco    2020-11-19 12:10:00-
06    2020-11-19 23:24:00-06
      5  San Francisco to Denver         2020-11-20 11:10:00-
06    2020-11-21 19:38:00-06
      6  Denver to Chicago               2020-11-22 20:10:00-
06    2020-11-23 14:50:00-06
```

All timestamps should now carry a UTC offset of `-06`, reflecting the Central time zone in the United States during November, when standard time is in effect. All time values display in their Central time equivalents.

Now that we've created segments corresponding to each leg of the trip, we'll use Listing 12-12 to calculate the duration of each segment.

```
SELECT segment,
     1 to_char(departure, 'YYYY-MM-DD HH12:MI a.m. TZ') AS
departure,
     2 arrival - departure AS segment_duration
FROM train_rides;
```

Listing 12-12: Calculating the length of each trip segment

This query lists the trip segment, the departure time, and the duration of the segment journey. Before we look at the calculation, notice the additional code around the `departure` column 1. These are PostgreSQL-specific formatting functions that specify how to format different components of the timestamp. In this case, the `to_char()` function turns the `departure` timestamp column into a string of characters formatted as `YYYY-MM-DD HH12:MI a.m. TZ`. The `YYYY-MM-DD` portion specifies the ISO format for the date, and the `HH12:MI a.m.` portion presents the time in hours and minutes. The `HH12` portion specifies the use of a 12-hour clock rather than 24-hour military time. The `a.m.` portion specifies that we want to show morning or night times using lowercase characters separated by periods, and the `TZ` portion denotes the time zone.

For a complete list of formatting functions, check out the PostgreSQL documentation at *https://www.postgresql.org/docs/current/functions-formatting.html*.

Last, we subtract `departure` from `arrival` to determine the `segment_duration` 2. When you run the query, the output should look like this:

```
segment                         departure
segment_duration
----------------------------    -------------------------
----------------
Chicago to New York             2020-11-13 09:30 p.m. CST
19:53:00
New York to New Orleans         2020-11-15 01:15 p.m. CST
1 day 06:17:00
New Orleans to Los Angeles      2020-11-17 01:45 p.m. CST
21:15:00
Los Angeles to San Francisco    2020-11-19 12:10 p.m. CST
11:14:00
San Francisco to Denver         2020-11-20 11:10 a.m. CST
1 day 08:28:00
Denver to Chicago               2020-11-22 08:10 p.m. CST
18:40:00
```

Subtracting one timestamp from another produces an `interval` data type, which was introduced in Chapter 4. As long as the value is less than 24 hours, PostgreSQL presents the interval in the *`HH:MM:SS`* format. For values greater than 24 hours, it returns the format `1 day 08:28:00`, as shown in the San Francisco to Denver segment.

In each calculation, PostgreSQL accounts for the changes in time zones so we don't inadvertently add or lose hours when subtracting. If we used a `timestamp without time zone` data type, we would end up with an incorrect trip length if a segment spanned multiple time zones.

Calculating Cumulative Trip Time

As it turns out, San Francisco to Denver is the longest leg of the All American train trip. But how long does the entire trip take? To answer this question, we'll revisit window functions, which you first learned about in "Ranking with rank() and dense_rank()" in Chapter 11.

Our prior query produced an interval, which we labeled `segment_duration`. The next natural next step would be to write a query to add those values, creating a cumulative interval after each segment. And indeed, we can use `sum()` as a window function, combined with the `OVER` clause used in Chapter 11, to create running totals. But when we do, the resulting values are odd. To see what I mean, run the code in Listing 12-13.

```
SELECT segment,
       arrival - departure AS segment_duration,
       sum(arrival - departure) OVER (ORDER BY trip_id) AS
cume_duration
FROM train_rides;
```

Listing 12-13: Calculating cumulative intervals using `OVER`

In the third column, we sum the intervals generated when we subtract `departure` from `arrival`. The resulting running total in the `cume_duration` column is accurate but formatted in an unhelpful way:

```
segment                         segment_duration
cume_duration
----------------------------    ----------------  -----------
----
Chicago to New York             19:53:00          19:53:00
New York to New Orleans         1 day 06:17:00    1 day
26:10:00
New Orleans to Los Angeles      21:15:00          1 day
47:25:00
Los Angeles to San Francisco    11:14:00          1 day
58:39:00
San Francisco to Denver         1 day 08:28:00    2 days
67:07:00
Denver to Chicago               18:40:00          2 days
85:47:00
```

PostgreSQL creates one sum for the day portion of the interval and another for the hours and minutes. So, instead of a more understandable cumulative time of `5 days 13:47:00`, the database reports `2 days 85:47:00`. Both results amount to the same length of time, but `2 days 85:47:00` is harder to decipher. This is an unfortunate limitation of summing the database intervals using this syntax.

To get around the limitation, we'll wrap the window function calculation for the cumulative duration inside the `justify_interval()` function, shown in *Listing 12-14*.

```
SELECT segment,
       arrival - departure AS segment_duration,
     1 justify_interval(sum(arrival - departure)
                        OVER (ORDER BY trip_id)) AS
cume_duration
FROM train_rides;
```

Listing 12-14: Using `justify_interval()` to better format cumulative trip duration

The `justify_interval()` function 1 standardizes output of interval calculations so that groups of 24 hours are rolled up to days, and groups of 30 days are rolled up to months. So, instead of returning a cumulative duration of `2 days 85:47:00`, as in the previous listing, `justify_interval()` converts 72 of those 85 hours to three days and adds them to the `days` value. The output is easier to understand:

```
          segment            segment_duration  cume_duration
---------------------------- ---------------- ---------------
Chicago to New York          19:53:00         19:53:00
New York to New Orleans      1 day 06:17:00   2 days 02:10:00
New Orleans to Los Angeles   21:15:00         2 days 23:25:00
Los Angeles to San Francisco 11:14:00         3 days 10:39:00
San Francisco to Denver      1 day 08:28:00   4 days 19:07:00
Denver to Chicago            18:40:00         5 days 13:47:00
```

The final `cume_duration` adds all the segments to return the total trip duration of `5 days 13:47:00`. That's a long time to spend on a train, but I'm sure the scenery is well worth the ride.

Wrapping Up

Handling times and dates in SQL databases adds an intriguing dimension to your analysis, letting you answer questions about when an event occurred along with other temporal concerns in your data. With a solid grasp of time and date formats, time zones, and functions to dissect the components of a timestamp, you can analyze just about any dataset you come across.

Next, we'll look at advanced query techniques that help answer more complex questions.

TRY IT YOURSELF

Try these exercises to test your skills on dates and times:

Using the New York City taxi data, calculate the length of each ride using the pickup and drop-off timestamps. Sort the query results from the longest ride to the shortest. Do you notice anything about the longest or shortest trips that you might want to ask city officials about?

Using the `AT TIME ZONE` keywords, write a query that displays the date and time for London, Johannesburg, Moscow, and Melbourne the moment January 1, 2100, arrives in New York City. Use the code in *Listing 12-5* to find time zone names.

As a bonus challenge, use the statistics functions in Chapter 11 to calculate the correlation coefficient and *r*-squared values using trip time and the `total_amount` column in the New York City taxi data, which represents the total amount charged to passengers. Do the same with the `trip_distance` and `total_amount` columns. Limit the query to rides that last three hours or less.

13
ADVANCED QUERY TECHNIQUES

Sometimes data analysis requires advanced SQL techniques that go beyond a table join or basic `SELECT` query. In this chapter, we'll cover techniques that include writing a query that uses the results of other queries as inputs and reclassifying numerical values into categories before counting them.

For the exercises, I'll introduce a dataset of temperatures recorded in select US cities, and we'll revisit datasets you've created in previous chapters. The code for the exercises is available, along with all the book's resources, at *https://nostarch.com/practical-sql-2nd-edition/*. You'll continue to use the `analysis` database you've already built. Let's get started.

Using Subqueries

A *subquery* is a query nested inside another query. Typically, it performs a calculation or a logical test or generates rows to be passed into the main outer query. Subqueries are part of standard ANSI SQL, and the syntax is not unusual: we just enclose a query in parentheses. For example, we can write a subquery that returns multiple rows and treat those results as a table in the `FROM` clause of the main outer query. Or we can create a *scalar subquery* that returns a single value and use it as part of an *expression* to filter rows via `WHERE`, `IN`, and `HAVING` clauses. A *correlated subquery* is one that depends on a value or table name from the outer query to execute. Conversely, an *uncorrelated subquery* has no reference to objects in the main query.

It's easier to understand these concepts by working with data, so let's revisit several datasets from earlier chapters, including the census county-level population estimates table `us_counties_pop_est_2019` and the business patterns table `cbp_naics_72_establishments`.

Filtering with Subqueries in a WHERE Clause

A `WHERE` clause lets you filter query results based on criteria you provide, using an expression such as `WHERE quantity > 1000`. But this requires that you already know the value to use for comparison. What if you don't? That's one way a subquery comes in handy: it lets you write a query that generates one or more values to use as part of an expression in a `WHERE` clause.

Generating Values for a Query Expression

Say you wanted to write a query to show which US counties are at or above the 90th percentile, or top 10 percent, for population. Rather than writing two separate queries—one to calculate the 90th percentile and another to find counties with populations at or higher—you can do both at once using a subquery as part of a `WHERE` clause, as shown in [Listing 13-1](#).

```
SELECT county_name,
       state_name,
       pop_est_2019
FROM us_counties_pop_est_2019
1 WHERE pop_est_2019 >= (
    SELECT percentile_cont(.9) WITHIN GROUP (ORDER BY
pop_est_2019)
    FROM us_counties_pop_est_2019
    )
ORDER BY pop_est_2019 DESC;
```

Listing 13-1: Using a subquery in a `WHERE` clause

The `WHERE` clause 1, which filters by the total population column `pop_est_2019`, doesn't include a value as it normally would. Instead, after the `>=` comparison operators, we provide a subquery in parentheses. This subquery uses the `percentile_cont()` function to generate one value: the 90th percentile cutoff point in the `pop_est_2019` column.

NOTE

Using `percentile_cont()` to filter with a subquery works only if you pass in a single input, as shown. If you pass in an array, as in Listing 6-12 on page 90, `percentile_cont()` returns an array, and the query will fail to evaluate the `>=` against an array type.

This is an example of an uncorrelated subquery. It does not depend on any values in the outer query, and it will be executed just once to generate the requested value. If you run the subquery portion only, by highlighting it in pgAdmin, it will execute, and you should see a result of `213707.3`. But you won't see that number when you run the entire query in Listing 13-1, because the subquery result is passed directly to the outer query's `WHERE` clause.

The entire query should return 315 rows, or about 10 percent of the 3,142 rows in `us_counties_pop_est_2019`.

```
      county_name              state_name       pop_est_2019
----------------------- -------------------- ------------
Los Angeles County      California               10039107
Cook County             Illinois                  5150233
Harris County           Texas                     4713325
Maricopa County         Arizona                   4485414
San Diego County        California                3338330
--snip--
Cabarrus County         North Carolina             216453
Yuma County             Arizona                    213787
```

The result includes all counties with a population greater than or equal to `213707.3`, the value the subquery generated.

Using a Subquery to Identify Rows to Delete

We can use the same subquery in a `DELETE` statement to specify what to remove from a table. In Listing 13-2, we make a copy of the census table using the method you learned in Chapter 10 and then delete everything from that backup except the 315 counties in the top 10 percent of population.

```
CREATE TABLE us_counties_2019_top10 AS
SELECT * FROM us_counties_pop_est_2019;

DELETE FROM us_counties_2019_top10
WHERE pop_est_2019 < (
    SELECT percentile_cont(.9) WITHIN GROUP (ORDER BY
pop_est_2019)
    FROM us_counties_2019_top10
    );
```

Listing 13-2: Using a subquery in a `WHERE` clause with `DELETE`

Run the code in Listing 13-2, and then execute `SELECT count(*) FROM us_counties_2019_top10;` to count the remaining rows. The result should be 315 rows, which is the original 3,142 minus the 2,827 below the value identified by the subquery.

Creating Derived Tables with Subqueries

If your subquery returns rows and columns, you can place it in a `FROM` clause to create a new table known as a *derived table* that you can query or join with other tables, just as you would a regular table. It's another example of an uncorrelated subquery.

Let's look at a simple example. In Chapter 6, you learned the difference between average and median. A median often better indicates a dataset's central value because a few very large or small values (or outliers) can skew an average. For that reason, I often compare the two. If they're close, the data more likely falls in a *normal distribution* (the familiar bell curve), and the average is a good representation of the central value. If the average and median are far apart, some outliers might be having an effect or the distribution is skewed, not normal.

Finding the average and median population of US counties as well as the difference between them is a two-step process. We need to calculate the average and the median and then subtract the two. We can do both operations in one fell swoop with a subquery in the `FROM` clause, as shown in *Listing 13-3*.

```
SELECT round(calcs.average, 0) AS average,
       calcs.median,
       round(calcs.average - calcs.median, 0) AS
median_average_diff
FROM (
    1 SELECT avg(pop_est_2019) AS average,
             percentile_cont(.5)
                 WITHIN GROUP (ORDER BY pop_est_2019)::numeric
AS median
     FROM us_counties_pop_est_2019
     )
2 AS calcs;
```

Listing 13-3: Subquery as a derived table in a `FROM` clause

The subquery 1 that produces a derived table is straightforward. We use the `avg()` and `percentile_cont()` functions to find the average and median of the census table's `pop_est_2019` column and name each column with an alias. Then we name the derived table `calcs` 2 so we can reference it in the main query.

In the main query, we subtract the `median` from the `average`, both of which are returned by the subquery. The result is rounded and labeled with the alias `median_average_diff`. Run the query, and the result should be the following:

```
average    median     median_average_diff
-------    -------    -------------------
 104468      25726                  78742
```

The difference between the median and average, 78,742, is nearly three times the size of the median. That indicates we have some high-population counties inflating the average.

Joining Derived Tables

Joining multiple derived tables lets you perform several preprocessing steps before final calculations in a main query. For example, in Chapter 11, we calculated the rate of tourism-related businesses per 1,000 population in each county. Let's say we want to do the same at the state level. Before we can calculate that rate, we need to know the number of tourism businesses in each state and the population of each state. Listing 13-4 shows how to write subqueries for both tasks and join them to calculate the overall rate.

```
SELECT census.state_name AS st,
       census.pop_est_2018,
       est.establishment_count,
     1
round((est.establishment_count/census.pop_est_2018::numeric)
* 1000, 1)
           AS estabs_per_thousand
FROM
    (
      2 SELECT st,
               sum(establishments) AS establishment_count
        FROM cbp_naics_72_establishments
        GROUP BY st
    )
    AS est
JOIN
    (
      3 SELECT state_name,
               sum(pop_est_2018) AS pop_est_2018
        FROM us_counties_pop_est_2019
        GROUP BY state_name
    )
    AS census
4 ON est.st = census.state_name
  ORDER BY estabs_per_thousand DESC;
```

Listing 13-4: Joining two derived tables

You learned how to calculate rates in Chapter 11, so the math and syntax in the outer query for finding `estabs_per_thousand` 1 should be familiar. We divide the number of establishments by the population and then multiply that quotient by a thousand. For the inputs, we use the values generated from two derived tables.

The first 2 finds the number of establishments in each state using the `sum()` aggregate function. We give this derived table the alias `est` for reference in the main part of the query. The second 3 finds the 2018 estimated population by state by using `sum()` on the `pop_est_2018` column. We alias this derived table as `census`.

Next, we join the derived tables 4 by linking the `st` column in `est` to the `state_name` column in `census`. We then list the results in descending order based on the rate. Here's a sample of the 51 rows showing the highest and lowest rates:

```
         st          pop_est_2018 establishment_count
estabs_per_thousand
-------------------- ------------ ------------------- -------
------------
District of Columbia       701547                2754
3.9
Montana                   1060665                3569
3.4
Vermont                    624358                1991
3.2
Maine                     1339057                4282
3.2
Wyoming                    577601                1808
3.1
--snip--
Arizona                   7158024               13288
1.9
Alabama                   4887681                9140
1.9
Utah                      3153550                6062
1.9
Mississippi               2981020                5645
1.9
Kentucky                  4461153                8251
1.8
```

At the top is Washington, DC, unsurprising given the tourist activity generated by the museums, monuments, and other attractions in the nation's capital. Montana may seem like a surprise in second place, but it's a low-population state with major tourist destinations including Glacier and Yellowstone national parks. Mississippi and Kentucky are among those states with the fewest tourism-related businesses per 1,000 population.

Generating Columns with Subqueries

You can also place a subquery in the column list after `SELECT` to generate a value for that column in the query result. The subquery must generate only a single row. For example, the query in *Listing 13-5* selects the geography and population information from `us_counties_pop_est_2019` and then adds an uncorrelated subquery to add the median of all counties to each row in the new column `us_median`.

```
SELECT county_name,
       state_name AS st,
       pop_est_2019,
       (SELECT percentile_cont(.5) WITHIN GROUP (ORDER BY
pop_est_2019)
        FROM us_counties_pop_est_2019) AS us_median
FROM us_counties_pop_est_2019;
```

Listing 13-5: Adding a subquery to a column list

The first rows of the result set should look like this:

```
           county_name                     st
pop_est_2019 us_median
--------------------------------- -------------------- ------
------ ---------
Autauga County                    Alabama
55869     25726
Baldwin County                    Alabama
223234     25726
Barbour County                    Alabama
24686     25726
Bibb County                       Alabama
22394     25726
Blount County                     Alabama
57826     25726
--snip--
```

On its own, that repeating `us_median` value isn't very helpful. It would be more interesting and useful to generate values that indicate how much each county's population deviates from the median value. Let's look at how we can use the same subquery technique to do that. *Listing 13-6* builds on *Listing 13-5* by substituting a subquery after `SELECT` that calculates the difference between the population and the median for each county.

```
SELECT county_name,
       state_name AS st,
       pop_est_2019,
       pop_est_2019 - (SELECT percentile_cont(.5) WITHIN
GROUP (ORDER BY pop_est_2019) 1
                       FROM us_counties_pop_est_2019) AS
diff_from_median
FROM us_counties_pop_est_2019
WHERE (pop_est_2019 - (SELECT percentile_cont(.5) WITHIN
GROUP (ORDER BY pop_est_2019) 2
                       FROM us_counties_pop_est_2019))
       BETWEEN -1000 AND 1000;
```

Listing 13-6: Using a subquery in a calculation

The subquery 1 is now part of a calculation that subtracts the subquery's result from `pop_est_2019`, the total population, giving the column an alias of `diff_from_median`. To make this query even more useful, we can filter results to show counties whose population is close to the median. To do this, we repeat the calculation with the subquery in the `WHERE` clause 2 and filter results using the `BETWEEN -1000 AND 1000` expression.

The outcome should reveal 78 counties. Here are the first five rows:

```
      county_name            st        pop_est_2019
diff_from_median
----------------------- -------------- ------------ ---------
-------
Cherokee County         Alabama               26196
470
Geneva County           Alabama               26271
545
Cleburne County         Arkansas              24919
-807
Johnson County          Arkansas              26578
```

```
852
St. Francis County      Arkansas              24994
-732
--snip--
```

Bear in mind that subqueries can add to overall query execution time. In Listing 13-6, I removed the subquery from Listing 13-5 that displays the column `us_median` to avoid repeating the subquery a third time. With our data set, the impact is minimal; if we were working with millions of rows, winnowing some unneeded subqueries might provide a significant speed boost.

Understanding Subquery Expressions

You can also use subqueries to filter rows by evaluating whether a condition evaluates as `true` or `false`. For this, we can use *subquery expressions*, which are a combination of a keyword with a subquery and are generally used in `WHERE` clauses to filter rows based on the existence of values in another table.

The PostgreSQL documentation at *https://www.postgresql.org/docs/current/functions-subquery.html* lists available subquery expressions, but here we'll examine the syntax for two that tend to be used most often: `IN` and `EXISTS`. To prep, run the code in Listing 13-7 to create a small table called `retirees` that we'll query along with the `employees` table you built in Chapter 7. We'll imagine that we've received this data from a vendor listing people who've applied for retirement benefits.

```
CREATE TABLE retirees (
    id int,
    first_name text,
    last_name text
);

INSERT INTO retirees
VALUES (2, 'Janet', 'King'),
       (4, 'Michael', 'Taylor');
```

Listing 13-7: *Creating and filling a* `retirees` *table*

Now let's use this table in some subquery expressions.

Generating Values for the IN Operator

The subquery expression `IN (subquery)` works like the `IN` operator example in Chapter 3 except we employ a subquery to provide the list of values to check against rather than manually entering one. In Listing 13-8, we use an uncorrelated subquery, which will be executed one time, to generate `id` values from the `retirees` table. The values it returns become the list for the `IN` operator in the `WHERE` clause. This lets us find employees who are also present in the table of retirees.

```
SELECT first_name, last_name
FROM employees
WHERE emp_id IN (
    SELECT id
    FROM retirees)
ORDER BY emp_id;
```

Listing 13-8: Generating values for the `IN` operator

Run the query, and the output shows the two people in `employees` whose `emp_id` have a matching `id` in the `retirees` table:

```
first_name last_name
---------- ---------
Janet      King
Michael    Taylor
```

NOTE

Avoid using `NOT IN`. The presence of `NULL` values in a subquery result set will cause a query with a `NOT IN` expression to return no rows. The PostgreSQL wiki recommends using `NOT EXISTS` instead, described in the next section.

Checking Whether Values Exist

The subquery expression `EXISTS (subquery)` returns a value of `true` if the subquery in parentheses returns at least one row. If it returns no rows, `EXISTS` evaluates to `false`.

The `EXISTS` subquery expression in Listing 13-9 shows an example of a correlated subquery—it includes an expression in its `WHERE` clause that requires data from the outer query. Also, because the subquery is correlated, it will execute once for each row returned by the outer query, each time checking whether there's an `id` in `retirees` that matches `emp_id` in `employees`. If there is a match, the `EXISTS` expression returns `true`.

```
SELECT first_name, last_name
FROM employees
WHERE EXISTS (
    SELECT id
    FROM retirees
    WHERE id = employees.emp_id);
```

Listing 13-9: Using a correlated subquery with `WHERE EXISTS`

When you run the code, it should return the same result as it did in Listing 13-8. Using this approach is particularly helpful if you need to join on more than one column, which you can't do with the `IN` expression. You also can add the `NOT` keyword with `EXISTS` to perform the opposite function and find rows in the employees table with no corresponding record in `retirees`, as in Listing 13-10.

```
SELECT first_name, last_name
FROM employees
WHERE NOT EXISTS (
    SELECT id
    FROM retirees
    WHERE id = employees.emp_id);
```

Listing 13-10: Using a correlated subquery with `WHERE NOT EXISTS`

That should produce these results:

```
first_name last_name
---------- ---------
```

```
Julia      Reyes
Arthur     Pappas
```

The technique of using `NOT` with `EXISTS` is helpful for finding missing values or assessing whether a dataset is complete.

Using Subqueries with LATERAL

Placing the keyword `LATERAL` before subqueries in a `FROM` clause adds several bits of functionality that help simplify otherwise complicated queries.

LATERAL with FROM

First, a subquery preceded by `LATERAL` can reference tables and other subqueries that appear before it in the `FROM` clause, which can reduce redundant code by making it easy to reuse calculations.

Listing 13-11 calculates the change in county population from 2018 to 2019 two ways: raw change in numbers and percent change.

```
SELECT county_name,
       state_name,
       pop_est_2018,
       pop_est_2019,
       raw_chg,
       round(pct_chg * 100, 2) AS pct_chg
FROM us_counties_pop_est_2019,
     1 LATERAL (SELECT pop_est_2019 - pop_est_2018 AS
raw_chg) rc,
     2 LATERAL (SELECT raw_chg / pop_est_2018::numeric AS
pct_chg) pc
ORDER BY pct_chg DESC;
```

Listing 13-11: Using `LATERAL` subqueries in the `FROM` clause

In the `FROM` clause, after naming the `us_counties_pop_est_2019` table, we add the first `LATERAL` subquery 1. In parentheses, we place a query that subtracts the 2018 population estimate from the 2019 estimate and alias the result as `raw_chg`. Because a `LATERAL` subquery can reference a table listed before it in the `FROM` clause without needing to specify its name, we can omit the `us_counties_pop_est_2019` table from the subquery. Subqueries in `FROM` must have an alias, so we label this one `rc`.

The second `LATERAL` subquery 2 calculates the percent change in population from 2018 to 2019. To find percent change, we must know the raw change. Rather than re-calculate it, we can reference the `raw_chg` value from the previous subquery. That helps make our code shorter and easier to read.

The query results should look like this:

```
  county_name      state_name   pop_est_2018 pop_est_2019
raw_chg pct_chg
---------------- ------------ ------------ ------------ -----
-- -------
Loving County    Texas                 148          169
21   14.19
McKenzie County  North Dakota        13594        15024
1430   10.52
Loup County      Nebraska              617          664
47    7.62
Kaufman County   Texas              128279       136154
7875    6.14
Williams County  North Dakota        35469        37589
2120    5.98
--snip--
```

LATERAL with JOIN

Combining `LATERAL` with `JOIN` creates functionality similar to a *for loop* in a programming language: for each row generated by the query in front of the `LATERAL` join, a subquery or function after the `LATERAL` join will be evaluated once.

We'll reuse the `teachers` table from Chapter 2 and create a new table to record each time a teacher swipes a badge to unlock a lab door. Our task is to find the two most recent times a teacher accessed a lab. *Listing 13-12* shows the code.

```
1 ALTER TABLE teachers ADD CONSTRAINT id_key PRIMARY KEY (id);

2 CREATE TABLE teachers_lab_access (
    access_id bigint PRIMARY KEY GENERATED ALWAYS AS
IDENTITY,
    access_time timestamp with time zone,
    lab_name text,
    teacher_id bigint REFERENCES teachers (id)
);

3 INSERT INTO teachers_lab_access (access_time, lab_name,
teacher_id)
VALUES ('2022-11-30 08:59:00-05', 'Science A', 2),
       ('2022-12-01 08:58:00-05', 'Chemistry B', 2),
       ('2022-12-21 09:01:00-05', 'Chemistry A', 2),
       ('2022-12-02 11:01:00-05', 'Science B', 6),
       ('2022-12-07 10:02:00-05', 'Science A', 6),
       ('2022-12-17 16:00:00-05', 'Science B', 6);

SELECT t.first_name, t.last_name, a.access_time, a.lab_name
FROM teachers t
4 LEFT JOIN LATERAL (SELECT *
                   FROM teachers_lab_access
                 5 WHERE teacher_id = t.id
                   ORDER BY access_time DESC
                   LIMIT 2)6 a
7 ON true
ORDER BY t.id;
```

Listing 13-12: Using a subquery with a `LATERAL` join

First, we add a primary key 1 to the `teachers` table using `ALTER TABLE` (we didn't place a constraint on this table in Chapter 2 because we were just covering the basics about creating tables). Next, we make a simple `teachers_lab_access` table 2 with columns to record the lab name and access timestamp. The table has a surrogate primary key `access_id` and a foreign key `teacher_id` that references `id` in `teachers`. Finally, we add six rows to the table using an `INSERT` 3 statement.

Now we're ready to query the data. In our `SELECT` statement, we join `teachers` to a subquery using `LEFT JOIN`. We add the `LATERAL` 4 keyword, which means for each row returned from `teachers`, the subquery will execute, returning the two most recent labs accessed by that particular teacher and the times they were accessed. Using `LEFT JOIN` will return all rows from `teachers` regardless of whether the subquery finds a matching teacher in `teachers_lab_access`.

In the `WHERE` 5 clause, the subquery references the outer query using the foreign key of `teacher_lab_access`. This `LATERAL` join syntax requires that the subquery have an alias 6, which here is `a`, and the value `true` in the `ON` portion 7 of the `JOIN` clause. In this case, `true` lets us create the join without naming specific columns to join upon.

Run the query, and the results should look like this:

```
first_name last_name      access_time          lab_name
---------- --------- ---------------------- ------------
Janet      Smith
Lee        Reynolds  2022-12-21 09:01:00-05  Chemistry A
Lee        Reynolds  2022-12-01 08:58:00-05  Chemistry B
Samuel     Cole
Samantha   Bush
Betty      Diaz
Kathleen   Roush     2022-12-17 16:00:00-05  Science B
Kathleen   Roush     2022-12-07 10:02:00-05  Science A
```

The two teachers with IDs in the access table have their two most recent lab access times shown. Teachers who didn’t access a lab display `NULL` values; if we want to remove those from the results, we could substitute `INNER JOIN` (or just `JOIN`) for `LEFT JOIN`.

Next, let’s explore another syntax for working with subqueries.

Using Common Table Expressions

The *common table expression (CTE)*, a relatively recent addition to standard SQL, allows you to use one or more `SELECT` queries to predefine temporary tables that you can reference as often as needed in your main query. CTEs are informally called `WITH` queries because you define them using a `WITH ... AS` statement. The following examples show some advantages of using them, including cleaner code and less redundancy.

Listing 13-13 shows a simple CTE based on our census estimates data. The code determines how many counties in each state have 100,000 people or more. Let’s walk through the example.

```
1 WITH large_counties (county_name, state_name, pop_est_2019)
   AS (
     2 SELECT county_name, state_name, pop_est_2019
       FROM us_counties_pop_est_2019
       WHERE pop_est_2019 >= 100000
      )
3 SELECT state_name, count(*)
   FROM large_counties
   GROUP BY state_name
   ORDER BY count(*) DESC;
```

Listing 13-13: Using a simple CTE to count large counties

The `WITH ... AS` statement 1 defines the temporary table `large_counties`. After `WITH`, we name the table and list its column names in parentheses. Unlike column definitions in a `CREATE TABLE` statement, we don't need to provide data types, because the temporary table inherits those from the subquery 2, which is enclosed in parentheses after `AS`. The subquery must return the same number of columns as defined in the temporary table, but the column names don't need to match. The column list is optional if you're not renaming columns; I've included it here so you can see the syntax.

The main query 3 counts and groups the rows in `large_counties` by `state_name` and then orders by the count in descending order. The top six rows of the results should look like this:

```
     state_name      count
-------------------- -----
Texas                   40
Florida                 36
California              35
Pennsylvania            31
New York                28
North Carolina          28
--snip--
```

Texas, Florida, and California are among the states that had the most counties with a 2019 population of 100,000 or more.

Listing 13-14 uses a CTE to rewrite the join of derived tables in Listing 13-4 (finding the rate of tourism-related businesses per 1,000 population in each state) into a more readable format.

```
WITH
  1 counties (st, pop_est_2018) AS
    (SELECT state_name, sum(pop_est_2018)
     FROM us_counties_pop_est_2019
     GROUP BY state_name),

  2 establishments (st, establishment_count) AS
    (SELECT st, sum(establishments) AS establishment_count
     FROM cbp_naics_72_establishments
     GROUP BY st)

SELECT counties.st,
       pop_est_2018,
       establishment_count,
       round((establishments.establishment_count /
              counties.pop_est_2018::numeric(10,1)) * 1000,
1)
           AS estabs_per_thousand
3 FROM counties JOIN establishments
ON counties.st = establishments.st
ORDER BY estabs_per_thousand DESC;
```

Listing 13-14: Using CTEs in a table join

Following the `WITH` keyword, we define two tables using subqueries. The first subquery, `counties` 1, returns the 2018 population of each state. The second, `establishments` 2, returns the number of tourism-related businesses per state. With those tables defined, we join them 3 on the `st` column in each table and calculate the rate per thousand. The results are identical to the joined derived tables in Listing 13-4, but Listing 13-14 is easier to comprehend.

As another example, you can use a CTE to simplify queries that have redundant code. For example, in Listing 13-6, we used a subquery with the `percentile_cont()` function in two locations to find median county population. In Listing 13-15, we can write that subquery just once as a CTE.

```
1 WITH us_median AS
      (SELECT percentile_cont(.5)
       WITHIN GROUP (ORDER BY pop_est_2019) AS us_median_pop
       FROM us_counties_pop_est_2019)

  SELECT county_name,
         state_name AS st,
         pop_est_2019,
       2 us_median_pop,
       3 pop_est_2019 - us_median_pop AS diff_from_median
4 FROM us_counties_pop_est_2019 CROSS JOIN us_median
5 WHERE (pop_est_2019 - us_median_pop)
         BETWEEN -1000 AND 1000;
```

Listing 13-15: Using CTEs to minimize redundant code

After the `WITH` keyword, we define `us_median` 1 as the result of the same subquery used in *Listing 13-6*, which finds the median population using `percentile_cont()`. Then we reference the `us_median_pop` column on its own 2, as part of a calculated column 3, and in a `WHERE` clause 5. To make the value available to every row in the `us_counties_pop_est_2019` table during `SELECT`, we use the `CROSS JOIN` 4 you learned in Chapter 7.

This query provides identical results to those in *Listing 13-6*, but we had to write the subquery that finds the median only once. Another bonus is that you can more easily revise the query. For example, to find counties whose population is close to the 90th percentile, you need to substitute `.9` for `.5` as input to `percentile_cont()` in only one place.

Readable code, less redundancy, and easier modifications are often-cited reasons for using CTEs. Another, beyond the scope of this book, is the ability to add a `RECURSIVE` keyword that lets a CTE loop through query results within the CTE itself—a task useful when dealing with data organized in a hierarchy. An example is a company's personnel listing, where you might want to find all the people who report to a particular executive. The recursive CTE will start with the executive and then loop down through rows finding her direct reports and then the people who report to those people. You can learn more about recursive query syntax via the PostgreSQL documentation at https://www.postgresql.org/docs/current/queries-with.html.

Performing Cross Tabulations

Cross tabulations provide a simple way to summarize and compare variables by displaying them in a table layout, or matrix. Rows in the matrix represent one variable, columns represent another variable, and each cell where a row and column intersect holds a value, such as a count or percentage.

You'll often see cross tabulations, also called *pivot tables* or *crosstabs*, used to report summaries of survey results or to compare pairs of variables. A frequent example happens during elections when candidates' votes are tallied by geography:

```
candidate     ward 1    ward 2    ward 3
---------     ------    ------    ------
Collins          602     1,799     2,112
Banks            599     1,398     1,616
Rutherford       911       902     1,114
```

In this case, the candidates' names are one variable, the wards (or city districts) are another variable, and the cells at the intersection of the two hold the vote totals for that candidate in that ward. Let's look at how to generate cross tabulations.

Installing the crosstab() Function

Standard ANSI SQL doesn't have a crosstab function, but PostgreSQL does as part of a *module* you can install easily. Modules are PostgreSQL extras that aren't part of the core application; they include functions related to security, text search, and more. You can find a list of PostgreSQL modules at https://www.postgresql.org/docs/current/contrib.html.

PostgreSQL's `crosstab()` function is part of the `tablefunc` module. To install `tablefunc`, execute this command in pgAdmin:

```
CREATE EXTENSION tablefunc;
```

PostgreSQL should return the message `CREATE EXTENSION`. (If you're working with another database management system, check its documentation for a similar functionality. For example, Microsoft SQL Server has the `PIVOT` command.)

Next, we'll create a basic crosstab so you can learn the syntax, and then we'll handle a more complex case.

Tabulating Survey Results

Let's say your company needs a fun employee activity so you coordinate an ice cream social at each of your three offices. The trouble is that people are particular about ice cream flavors. To choose flavors people will like in each office, you decide to conduct a survey.

The CSV file *ice_cream_survey.csv* contains 200 responses to your survey. You can download this file, along with all the book's resources, at *https://nostarch.com/practical-sql-2nd-edition/*. Each row includes a `response_id`, `office`, and `flavor`. You'll need to count how many people chose each flavor at each office and share the results in a readable way.

In your `analysis` database, use the code in *Listing 13-16* to create a table and load the data. Make sure you change the file path to the location on your computer where you saved the CSV file.

```
CREATE TABLE ice_cream_survey (
    response_id integer PRIMARY KEY,
    office text,
    flavor text
);

COPY ice_cream_survey
FROM 'C:\YourDirectory\ice_cream_survey.csv'
WITH (FORMAT CSV, HEADER);
```

Listing 13-16: Creating and filling the `ice_cream_survey` table

If you want to inspect the data, run the following to view the first five rows:

```
SELECT *
FROM ice_cream_survey
```

```
ORDER BY response_id
LIMIT 5;
```

The data should look like this:

```
response_id    office      flavor
-----------    ---------   -----------
          1    Uptown      Chocolate
          2    Midtown     Chocolate
          3    Downtown    Strawberry
          4    Uptown      Chocolate
          5    Midtown     Chocolate
```

It looks like chocolate is in the lead! But let's confirm this choice by using the code in Listing 13-17 to generate a crosstab.

```
   SELECT *
1 FROM crosstab('SELECT 2 office,
                         3 flavor,
                         4 count(*)
                  FROM ice_cream_survey
                  GROUP BY office, flavor
                  ORDER BY office',

               5 'SELECT flavor
                  FROM ice_cream_survey
                  GROUP BY flavor
                  ORDER BY flavor')

6 AS (office text,
       chocolate bigint,
       strawberry bigint,
       vanilla bigint);
```

Listing 13-17: Generating the ice cream survey crosstab

The query begins with a `SELECT *` statement that selects everything from the contents of the `crosstab()` function 1. We supply two queries as parameters to the `crosstab()` function; note that because these queries are parameters, we place them inside single quotes. The first query generates the data for the crosstab and has three required columns. The first column, `office` 2, supplies the row names for the crosstab. The second column, `flavor` 3, supplies the category (or column) name to be associated with the value provided in the third column. Those values will display in each cell where a row and a column intersect in the table. In this case, we want the intersecting cells to show a `count()` 4 of each flavor selected at each office. This first query on its own creates a simple aggregated list.

The second query parameter 5 produces the category names for the columns. The `crosstab()` function requires that the second subquery returns only one column, so we use `SELECT` to retrieve `flavor` and `GROUP BY` to return that column's unique values.

Then we specify the names and data types of the crosstab's output columns following the `AS` keyword 6. The list must match the row and column names in the order the queries generate them. For example, because the second query that supplies the category columns orders the flavors alphabetically, the output column list must as well.

When we run the code, our data displays in a clean, readable crosstab:

```
office      chocolate    strawberry    vanilla
---------   ----------   -----------   --------
Downtown            23            32         19
Midtown             41                       23
Uptown              22            17         23
```

It's easy to see at a glance that the Midtown office favors chocolate but has no interest in strawberry, which is represented by a `NULL` value showing that strawberry received no votes. But strawberry is the top choice Downtown, and the Uptown office is more evenly split among the three flavors.

Tabulating City Temperature Readings

Let's create another crosstab, but this time we'll use real data. The *temperature_readings.csv* file, also available with all the book's resources at *https://nostarch.com/practical-sql-2nd-edition/*, contains a year's worth of daily temperature readings from three observation stations around the United States: Chicago, Seattle, and Waikiki, a neighborhood on the south shore of the city of Honolulu. The data come from the US National Oceanic

and Atmospheric Administration (NOAA) at
https://www.ncdc.noaa.gov/cdo-web/datatools/findstation/.

Each row in the CSV file contains four values: the station name, the date, and the day's maximum and minimum temperatures. All temperatures are in Fahrenheit. For each month in each city, we want to compare climates using the median high temperature. Listing 13-18 has the code to create the `temperature_readings` table and import the CSV file.

```
CREATE TABLE temperature_readings (
    station_name text,
    observation_date date,
    max_temp integer,
    min_temp integer,
    CONSTRAINT temp_key PRIMARY KEY (station_name,
observation_date)
);

COPY temperature_readings
FROM 'C:\YourDirectory\temperature_readings.csv'
WITH (FORMAT CSV, HEADER);
```

Listing 13-18: Creating and filling a `temperature_readings` table

The table contains the four columns from the CSV file; we add a natural primary key using the station name and observation date. A quick count should return 1,077 rows. Now, let's see what cross tabulating the data does using Listing 13-19.

```
SELECT *
FROM crosstab('SELECT
                1 station_name,
                2 date_part(''month'', observation_date),
                3 percentile_cont(.5)
                      WITHIN GROUP (ORDER BY max_temp)
               FROM temperature_readings
               GROUP BY station_name,
                        date_part(''month'',
observation_date)
               ORDER BY station_name',

              'SELECT month
```

```
                    FROM 4 generate_series(1,12) month')

AS (station text,
    jan numeric(3,0),
    feb numeric(3,0),
    mar numeric(3,0),
    apr numeric(3,0),
    may numeric(3,0),
    jun numeric(3,0),
    jul numeric(3,0),
    aug numeric(3,0),
    sep numeric(3,0),
    oct numeric(3,0),
    nov numeric(3,0),
    dec numeric(3,0)
);
```

Listing 13-19: Generating the temperature readings crosstab

The crosstab structure is the same as in *Listing 13-18*. The first subquery inside `crosstab()` generates the data for the crosstab, finding the median maximum temperature for each month. It supplies three required columns. The first, `station_name` 1, names the rows. The second column uses the `date_part()` function 2 from Chapter 12 to extract the month from `observation_date`, which provides the crosstab columns. Then we use `percentile_cont(.5)` 3 to find the 50th percentile, or the median, of the `max_temp`. We group by station name and month so we have a median `max_temp` for each month at each station.

As in *Listing 13-18*, the second subquery produces the set of category names for the columns. I'm using a function called `generate_series()` 4 in a manner noted in the official PostgreSQL documentation to create a list of numbers from 1 to 12 that match the month numbers `date_part()` extracts from `observation_date`.

Following `AS`, we provide the names and data types for the crosstab's output columns. Each is a `numeric` type, matching the output of the percentile function. The following output is practically poetry:

```
station                           jan  feb  mar  apr  may  jun
jul  aug  sep  oct  nov  dec
------------------------------   ---  ---  ---  ---  ---  ---
```

```
---  ---  ---  ---  ---  ---
CHICAGO NORTHERLY ISLAND IL US    34   36   46   50   66   77
81   80   77   65   57   35
SEATTLE BOEING FIELD WA US        50   54   56   64   66   71
76   77   69   62   55   42
WAIKIKI 717.2 HI US               83   84   84   86   87   87
88   87   87   86   84   82
```

We've transformed a raw set of daily readings into a compact table showing the median maximum temperature each month for each station. At a glance, we can see that the temperature in Waikiki is consistently balmy, whereas Chicago's median high temperatures vary from just above freezing to downright pleasant. Seattle falls between the two.

Crosstabs do take time to set up, but viewing datasets in a matrix often makes comparisons easier than viewing the same data in a vertical list. Keep in mind that the `crosstab()` function is resource-intensive, so tread carefully when querying sets that have millions or billions of rows.

Reclassifying Values with CASE

The ANSI Standard SQL `CASE` statement is a *conditional expression*, meaning it lets you add some "if this, then . . ." logic to a query. You can use `CASE` in multiple ways, but for data analysis, it's handy for reclassifying values into categories. You can create categories based on ranges in your data and classify values according to those categories.

The `CASE` syntax follows this pattern:

```
1 CASE WHEN condition THEN result
      2 WHEN another_condition THEN result
      3 ELSE result
4 END
```

We give the `CASE` keyword 1 and then provide at least one `WHEN` *`condition`* `THEN` *`result`* clause, where *`condition`* is any expression the database can evaluate as `true` or `false`, such as `county = 'Dutchess County'` or `date > '1995-08-09'`. If the condition is `true`, the `CASE` statement returns the *`result`* and stops checking any further conditions. The result can be any valid data type. If the condition is `false`, the database moves on to evaluate the next condition.

To evaluate more conditions, we can add optional `WHEN ... THEN` clauses 2. We can also provide an optional `ELSE` clause 3 to return a result in case no condition evaluates as `true`. Without an `ELSE` clause, the statement would return a `NULL` when no conditions are `true`. The statement finishes with an `END` keyword 4.

Listing 13-20 shows how to use the `CASE` statement to reclassify the temperature readings into descriptive groups (named according to my own bias against cold weather).

```
SELECT max_temp,
       CASE WHEN max_temp >= 90 THEN 'Hot'
            WHEN max_temp >= 70 AND max_temp < 90 THEN 'Warm'
            WHEN max_temp >= 50 AND max_temp < 70 THEN
'Pleasant'
            WHEN max_temp >= 33 AND max_temp < 50 THEN 'Cold'
            WHEN max_temp >= 20 AND max_temp < 33 THEN
'Frigid'
            WHEN max_temp < 20 THEN 'Inhumane'
            ELSE 'No reading'
        END AS temperature_group
FROM temperature_readings
ORDER BY station_name, observation_date;
```

Listing 13-20: *Reclassifying temperature data with* `CASE`

We create six ranges for the `max_temp` column in `temperature_readings`, which we define using comparison operators. The `CASE` statement evaluates each value to find whether any of the six expressions are `true`. If so, the statement outputs the appropriate text. Note that the ranges account for all possible values in the column, leaving no gaps. If none of the statements is `true`, then the `ELSE` clause assigns the value to the category `No reading`.

Run the code; the first five rows of output should look like this:

```
max_temp    temperature_group
--------    -----------------
      31    Frigid
```

```
34     Cold
32     Frigid
32     Frigid
34     Cold
--snip--
```

Now that we’ve collapsed the dataset into six categories, let’s use those categories to compare climate among the three cities in the table.

Using CASE in a Common Table Expression

The operation we performed with `CASE` on the temperature data in the previous section is a good example of a preprocessing step you could use in a CTE. Now that we’ve grouped the temperatures in categories, let’s count the groups by city in a CTE to see how many days of the year fall into each temperature category.

Listing 13-21 shows the code for reclassifying the daily maximum temperatures recast to generate a `temps_collapsed` CTE and then use it for an analysis.

```
1 WITH temps_collapsed (station_name, max_temperature_group) AS
      (SELECT station_name,
             CASE WHEN max_temp >= 90 THEN 'Hot'
                  WHEN max_temp >= 70 AND max_temp < 90 THEN
  'Warm'
                  WHEN max_temp >= 50 AND max_temp < 70 THEN
  'Pleasant'
                  WHEN max_temp >= 33 AND max_temp < 50 THEN
  'Cold'
                  WHEN max_temp >= 20 AND max_temp < 33 THEN
  'Frigid'
                  WHEN max_temp < 20 THEN 'Inhumane'
                  ELSE 'No reading'
              END
       FROM temperature_readings)

2 SELECT station_name, max_temperature_group, count(*)
  FROM temps_collapsed
  GROUP BY station_name, max_temperature_group
  ORDER BY station_name, count(*) DESC;
```

Listing 13-21: Using `CASE` in a CTE

This code reclassifies the temperatures and then counts and groups by station name to find general climate classifications of each city. The `WITH` keyword defines the CTE of `temps_collapsed` 1, which has two columns: `station_name` and `max_temperature_group`. We then run a `SELECT` query on the CTE 2, performing straightforward `count(*)` and `GROUP BY` operations on both columns. The results should look like this:

```
station_name                       max_temperature_group
count
------------------------------     ---------------------    --
---
CHICAGO NORTHERLY ISLAND IL US     Warm
133
CHICAGO NORTHERLY ISLAND IL US     Cold
92
CHICAGO NORTHERLY ISLAND IL US     Pleasant
91
CHICAGO NORTHERLY ISLAND IL US     Frigid
30
CHICAGO NORTHERLY ISLAND IL US     Inhumane
8
CHICAGO NORTHERLY ISLAND IL US     Hot
8
SEATTLE BOEING FIELD WA US         Pleasant
198
SEATTLE BOEING FIELD WA US         Warm
98
SEATTLE BOEING FIELD WA US         Cold
50
SEATTLE BOEING FIELD WA US         Hot
3
WAIKIKI 717.2 HI US                Warm
361
WAIKIKI 717.2 HI US                Hot
5
```

Using this classification scheme, the amazingly consistent Waikiki weather, with `Warm` maximum temperatures 361 days of the year, confirms its appeal as a vacation destination. From a temperature standpoint, Seattle looks good too, with nearly 300 days of `Pleasant` or `Warm` high temps (although this belies Seattle's legendary rainfall). Chicago, with 30 days of `Frigid` max temps and 8 days `Inhumane`, probably isn't for me.

Wrapping Up

In this chapter, you learned to make queries work harder for you. You can now add subqueries in multiple locations to provide finer control over filtering or preprocessing data before analyzing it in a main query. You also can visualize data in a matrix using cross tabulations and reclassify data into groups; both techniques give you more ways to find and tell stories using your data. Great work!

Throughout the next chapters, we'll dive into SQL techniques that are more specific to PostgreSQL. We'll begin by working with and searching text and strings.

TRY IT YOURSELF

Perform the following two tasks to help you become more familiar with the concepts introduced in the chapter:

Revise the code in Listing 13-21 to dig deeper into the nuances of Waikiki's high temperatures. Limit the `temps_collapsed` table to the Waikiki maximum daily temperature observations. Then use the `WHEN` clauses in the `CASE` statement to reclassify the temperatures into seven groups that would result in the following text output:

```
'90 or more'
'88-89'
'86-87'
'84-85'
'82-83'
'80-81'
'79 or less'
```

In which of those groups does Waikiki's daily maximum temperature fall most often?

Revise the ice cream survey crosstab in Listing 13-17 to flip the table. In other words, make `flavor` the rows and `office` the columns. Which elements of the query do you need to change? Are the counts different?

14
MINING TEXT TO FIND MEANINGFUL DATA

Next, you'll learn how to use SQL to transform, search, and analyze text. You'll start with simple text wrangling using string formatting and pattern matching before moving on to more advanced analysis. We'll use two data-sets: a small collection of crime reports from a sheriff's department near Washington, DC, and a set of speeches delivered by US presidents.

Text offers plenty of possibilities for analysis. You can extract meaning from *unstructured data*—paragraphs of text in speeches, reports, press releases, and other documents—by transforming it into *structured data*, in rows and columns in a table. Or you can use advanced text analysis features, such as PostgreSQL's full-text search. With these techniques, ordinary text can reveal facts or trends that might otherwise remain hidden.

Formatting Text Using String Functions

PostgreSQL has more than 50 built-in string functions that handle routine but necessary tasks, such as capitalizing letters, combining strings, and removing unwanted spaces. Some are part of the ANSI SQL standard, and others are specific to PostgreSQL. You'll find a complete list of string functions at *https://www.postgresql.org/docs/current/functions-string.html*, but in this section we'll examine several you might use often.

You can try each of these in a simple query that places a function after `SELECT`, like this: `SELECT upper('hello');`. Examples of each function plus code for all the listings in this chapter are available at *https://nostarch.com/practical-sql-2nd-edition/*.

Case Formatting

The capitalization functions format the text's case. The `upper(`*`string`*`)` function capitalizes all alphabetical characters of a string passed to it. Nonalphabetic characters, such as numbers, remain unchanged. For example, `upper('Neal7')` returns `NEAL7`. The `lower(`*`string`*`)` function lowercases all alphabetical characters while keeping nonalphabetic characters unchanged. For example, `lower('Randy')` returns `randy`.

The `initcap(`*`string`*`)` function capitalizes the first letter of each word. For example, `initcap('at the end of the day')` returns `At The End Of The Day`. This function can be handy for formatting titles of books or movies, but because it doesn't recognize acronyms, it's not always the perfect solution. For example, `initcap('Practical SQL')` returns `Practical Sql`, because it doesn't recognize SQL as an acronym.

The `upper()` and `lower()` functions are ANSI SQL standard commands, but `initcap()` is PostgreSQL-specific. These three functions give you enough options to rework a column of text into the case you prefer. Note that capitalization does not work with all locales or languages.

Character Information

Several functions return data about the string and are helpful on their own or combined with other functions. The `char_length(`*`string`*`)` function returns the number of characters in a string, including any spaces. For example, `char_length(' Pat ')` returns a value of `5`, because the three letters in `Pat` and the spaces on either end total five characters. You can also use the non-ANSI SQL function `length(`*`string`*`)` to count strings, which has a variant that lets you count the length of binary strings.

NOTE

The `length()` *function can return a different value than* `char_length()` *when used with multibyte encodings, such as character sets covering the Chinese, Japanese, or Korean languages.*

The `position(`*`substring`*` in `*`string`*`)` function returns the location of the substring characters in the string. For example, `position(', ' in 'Tan, Bella')` returns `4`, because the comma and space characters (`, `) specified in the substring passed as the first parameter starting at the fourth index position in the main string `Tan, Bella`.

Both `char_length()` and `position()` are in the ANSI SQL standard.

Removing Characters

The `trim(`*`characters`*` from `*`string`*`)` function removes characters from the beginning and end of a string. To declare one or more characters to remove, add them to the function followed by the keyword `from` and the string you want to change. Options to remove `leading` characters (at the front of the string), `trailing` characters (at the end of the string), or `both` make this function super flexible.

For example, `trim('s' from 'socks')` removes `s` characters at the beginning and end, returning `ock`. To remove only the `s` at the end of the string, add the `trailing` keyword before the character to trim: `trim(trailing 's' from 'socks')` returns `sock`.

If you don't specify any characters to remove, `trim()` removes spaces at either end of the string by default. For example, `trim(' Pat ')` returns `Pat` without the leading or trailing spaces. To confirm the length of the trimmed string, we can nest `trim()` inside `char_length()` like this:

```
SELECT char_length(trim(' Pat '));
```

This query should return `3`, the number of letters in `Pat`, which is the result of `trim(' Pat ')`.

The `ltrim(`*`string`*`, `*`characters`*`)` and `rtrim(`*`string`*`, `*`characters`*`)` functions are PostgreSQL-specific variations of the `trim()` function. They remove characters from the left or right ends of a string. For example, `rtrim('socks', 's')` returns `sock` by removing only the `s` on the right end of the string.

Extracting and Replacing Characters

The `left(`*`string`*`, `*`number`*`)` and `right(`*`string`*`, `*`number`*`)` functions, both ANSI SQL standard, extract and return selected characters from a string. For example, to get just the `703` area code from the phone number `703-555-1212`, use `left('703-555-1212', 3)` to specify that you want the first three characters of the string starting from the left. Likewise, `right('703-555-1212', 8)` returns eight characters from the right: `555-1212`.

To substitute characters in a string, use the `replace(`*`string`*`, `*`from`*`, `*`to`*`)` function. To change `bat` to `cat`, for example, you would use `replace('bat', 'b', 'c')` to specify that you want to replace the `b` in `bat` with a `c`.

Now that you know basic functions for manipulating strings, let's look at how to match more complex patterns in text and turn those patterns into data we can analyze.

Matching Text Patterns with Regular Expressions

Regular expressions (or *regex*) are a type of notational language that describes text patterns. If you have a string with a noticeable pattern (say, four digits followed by a hyphen and then two more digits), you can write a regular expression that matches the pattern. You can then use the notation in a `WHERE` clause to filter rows by the pattern or use regular expression functions to extract and wrangle text that contains the same pattern.

Regular expressions can seem inscrutable to beginning programmers; they take practice to comprehend because they use single-character symbols that aren't intuitive. Getting an expression to match a pattern can involve trial and error, and each programming language has subtle differences in the way it handles regular expressions. Still, learning regular expressions is a good investment because you gain superpower-like abilities to search text using many programming languages, text editors, and other applications.

In this section, I'll provide enough regular expression basics to work through the exercises. To learn more, I recommend interactive online code testers, such as *https://regexr.com/* or *https://www.regexpal.com/*, which have notation references.

Regular Expression Notation

Matching letters and numbers using regular expression notation is straightforward because letters and numbers (and certain symbols) are literals that indicate the same characters. For example, `Al` matches the first two characters in `Alicia`.

For more complex patterns, you'll use combinations of the regular expression elements in *Table 14-1*.

Table 14-1: *Regular Expression Notation Basics*

Expression	Description
`.`	A dot is a wildcard that finds any character except a newline.
`[FGz]`	Any character in the square brackets. Here, *F*, *G*, or *z*.
`[a-z]`	A range of characters. Here, lowercase *a* to *z*.
`[^a-z]`	The caret negates the match. Here, not lowercase *a* to *z*.
`\w`	Any word character or underscore. Same as `[A-Za-z0-9_]`.
`\d`	Any digit.
`\s`	A space.
`\t`	Tab character.
`\n`	Newline character.
`\r`	Carriage return character.
`^`	Match at the start of a string.
`$`	Match at the end of a string.
`?`	Get the preceding match zero or one time.
`*`	Get the preceding match zero or more times.
`+`	Get the preceding match one or more times.
`{m}`	Get the preceding match exactly *m* times.
`{m,n}`	Get the preceding match between *m* and *n* times.
`a|b`	The pipe denotes alternation. Find either *a* or *b*.
`( )`	Create and report a capture group or set precedence.
`(?: )`	Negate the reporting of a capture group.

Using these regular expressions, you can match various characters and indicate how many times and where to match them. For example, placing characters inside square brackets (`[]`) lets you match any single character or a range. So, `[FGz]` matches a single `F`, `G`, or `z`, whereas `[A-Za-z]` will match any uppercase or lowercase letter.

The backslash (`\`) precedes a designator for special characters, such as a tab (`\t`), digit (`\d`), or newline (`\n`), which is a line ending character in text files.

There are several ways to indicate how many times to match a character. Placing a number inside curly brackets indicates you want to match it that many times. For example, `\d{4}` matches four digits in a row, and `\d{1,4}` matches one to four digits.

The `?`, `*`, and `+` characters provide a useful shorthand notation for the number of matches you want. The plus sign (`+`) after a character indicates to match it one or more times, for example. So, the expression `a+` would find the `aa` characters in the string `aardvark`.

Additionally, parentheses indicate a *capture group*, which you can use to identify a portion of the entire matched expression. For example, if you were hunting for an *`HH`*`:`*`MM`*`:`*`SS`* time format in text and wanted to report only the hour, you could use an expression such as `(\d{2}):\d{2}:\d{2}`. This looks for two digits (`\d{2}`) of the hour followed by a colon, another two digits for the minutes and a colon, and then the two-digit seconds. By placing the first `\d{2}` inside parentheses, you can extract only those two digits, even though the entire expression matches the full time.

Table 14-2 shows examples of combining regular expressions to capture different portions of the sentence "The game starts at 7 p.m. on May 2, 2024."

Table 14-2: *Regular Expression Matching Examples*

Expression	What it matches	Result
`.+`	Any character one or more times	`The game starts at 7 p.m. on May 2, 2024.`
`\d{1,2} (?:a.m.|p.m.)`	One or two digits followed by a space and *a.m.* or *p.m.* in a noncapture group	`7 p.m.`
`^\w+`	One or more word characters at the start	`The`
`\w+.$`	One or more word characters followed by any character at the end	`2024.`
`May|June`	Either of the words *May* or *June*	`May`
`\d{4}`	Four digits	`2024`
`May \d, \d{4}`	*May* followed by a space, digit, comma, space, and four digits	`May 2, 2024`

These results show the usefulness of regular expressions for matching the parts of the string that interest us. For example, to find the time, we use the expression `\d{1,2} (?:a.m.|p.m.)` to look for either one or two digits because the time could be a single or double digit followed by a space. Then we look for either `a.m.` or `p.m.`; the pipe symbol separating the terms indicates the either-or condition, and placing them in parentheses separates the logic from the rest of the expression. We need the `?:` symbol to indicate that we don't want to treat the terms inside the parentheses as a capture group, which would report `a.m.` or `p.m.` only. The `?:` ensures that the full match will be returned.

You can use any of these regular expressions by placing the text and regular expression inside the `substring(`*`string`*` from `*`pattern`*`)` function to return the matched text. For example, to find the four-digit year, use the following query:

```
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '\d{4}');
```

This query should return `2024`, because we specified that the pattern should look for four digits in a row, and 2024 is the only portion of this string that matches these criteria. You can check out sample `substring()` queries for all the examples in Table 14-2 in the book's code resources at https://nostarch.com/practical-sql-2nd-edition/.

Using Regular Expressions with WHERE

You've filtered queries using `LIKE` and `ILIKE` in `WHERE` clauses. In this section, you'll learn to use regular expressions in `WHERE` clauses so you can perform more complex matches.

We use a tilde (`~`) to make a case-sensitive match on a regular expression and a tilde-asterisk (`~*`) to perform a case-insensitive match. You can negate either expression by adding an exclamation point in front. For example, `!~*` indicates to *not* match a regular expression that is case-insensitive. Listing 14-1 shows how this works using the 2019 US Census estimates `us_counties_pop_est_2019` table from previous exercises.

```
 SELECT county_name
 FROM us_counties_pop_est_2019
1 WHERE county_name ~* '(lade|lare)'
 ORDER BY county_name;

 SELECT county_name
 FROM us_counties_pop_est_2019
2 WHERE county_name ~* 'ash' AND county_name !~ 'Wash'
 ORDER BY county_name;
```

Listing 14-1: Using regular expressions in a `WHERE` clause

The first `WHERE` clause 1 uses the tilde-asterisk (`~*`) to perform a case-insensitive match on the regular expression `(lade|lare)` to find any county names that contain either the letters `lade` or `lare`. The results should show eight rows:

```
county_name
-------------------
Bladen County
Clare County
Clarendon County
Glades County
Langlade County
Philadelphia County
Talladega County
Tulare County
```

As you can see, each county name includes the letters `lade` or `lare`.

The second `WHERE` clause 2 uses the tilde-asterisk (`~*`) as well as a negated tilde (`!~`) to find county names containing the letters `ash` but excluding those that include `Wash`. This query should return the following:

```
county_name
--------------
Ashe County
Ashland County
Ashland County
Ashley County
Ashtabula County
Nash County
```

```
Wabash County
Wabash County
Wabasha County
```

All nine counties in this output have names that contain the letters `ash`, but none have `Wash`.

These are fairly simple examples, but you can do more complex matches using regular expressions that you wouldn't be able to perform with the wildcards available with just `LIKE` and `ILIKE`.

Regular Expression Functions to Replace or Split Text

Listing 14-2 shows three regular expression functions that replace and split text.

```
1 SELECT regexp_replace('05/12/2024', '\d{4}', '2023');

2 SELECT regexp_split_to_table('Four,score,and,seven,years,ago', ',');

3 SELECT regexp_split_to_array('Phil Mike Tony Steve', ' ');
```

Listing 14-2: Regular expression functions to replace and split text

The `regexp_replace(`*`string`*`, `*`pattern`*`, `*`replacement text`*`)` function lets you substitute a matched pattern with replacement text. In the example at 1, we're searching the date string `05/12/2024` for any set of four digits in a row using `\d{4}`. When found, we replace them with the replacement text `2023`. The result of that query is `05/12/2023` returned as text.

The `regexp_split_to_table(`*`string`*`, `*`pattern`*`)` function splits delimited text into rows. Listing 14-2 uses this function to split the string `'Four,score,and,seven,years,ago'` on commas 2, resulting in a set of rows that has one word in each row:

```
regexp_split_to_table
---------------------
Four
score
and
seven
years
ago
```

Keep this function in mind as you tackle the "Try It Yourself" exercises at the end of the chapter.

The `regexp_split_to_array(`*`string`*`, `*`pattern`*`)` function splits delimited text into an array. The example splits the string `Phil Mike Tony Steve` on spaces 3, returning a text array that should look like this in pgAdmin:

```
regexp_split_to_array
----------------------
{Phil,Mike,Tony,Steve}
```

The `text[]` notation in pgAdmin's column header along with curly brackets around the results confirms that this is indeed an array type, which provides another means of analysis. For example, you could then use a function such as `array_length()` to count the number of words, as shown in Listing 14-3.

```
SELECT array_length(regexp_split_to_array('Phil Mike Tony Steve', ' '), 1
1);
```

Listing 14-3: Finding an array length

The array that `regexp_split_to_array()` produces is one-dimensional; that is, the result contains one list of names. Arrays can have additional dimensions—for example, a two-dimensional array can represent a matrix with rows and columns. Thus, here we pass `1` as a second argument 1 to `array_length()`, indicating we want the length of the first (and only) dimension of the array. The query should return `4` because the array has four elements. You can read more about `array_length()` and other array functions at *https://www.postgresql.org/docs/current/functions-array.html*.

If you can identify a pattern in the text, you can use a combination of regular expression symbols to locate it. This technique is particularly useful when you have repeating patterns in text that you want to turn into a set of data to analyze. Let's practice how to use regular expression functions using a real-world example.

Turning Text to Data with Regular Expression Functions

A sheriff's department in one of the Washington, DC, suburbs publishes daily reports that detail the date, time, location, and description of incidents the department investigates. These reports would be great to analyze, except they post the information in Microsoft Word documents saved as PDF files, which is not the friendliest format for importing into a database.

If I copy and paste incidents from the PDF into a text editor, the result is blocks of text that look something like Listing 14-4.

```
1 4/16/17-4/17/17
2 2100-0900 hrs.
3 46000 Block Ashmere Sq.
4 Sterling
5 Larceny: 6 The victim reported that a
  bicycle was stolen from their opened
  garage door during the overnight hours.
7 C0170006614

  04/10/17
```

```
1605 hrs.
21800 block Newlin Mill Rd.
Middleburg
Larceny: A license plate was reported
stolen from a vehicle.
SO170006250
```

Listing 14-4: Crime reports text

Each block of text includes dates 1, times 2, a street address 3, city or town 4, the type of crime 5, and a description of the incident 6. The last piece of information is a code 7 that might be a unique ID for the incident, although we'd have to check with the sheriff's department to be sure. There are slight inconsistencies. For example, the first block of text has two dates (`4/16/17-4/17/17`) and two times (`2100-0900 hrs.`), meaning the exact time of the incident is unknown and likely occurred within that time span. The second block has one date and time.

If you compile these reports regularly, you can expect to find some good insights that could answer important questions: Where do crimes tend to occur? Which crime types occur most frequently? Do they happen more often on weekends or weekdays? Before you can start answering these questions, you'll need to extract the text into table columns using regular expressions.

NOTE

Extracting elements from text is labor-intensive, so it's a good idea to ask the owner of the data whether the text was produced from a database. If it was and you can obtain a structured export such as a CSV file from that database, you'll save considerable time.

Creating a Table for Crime Reports

I've collected five of the crime incidents into a file named *crime_reports.csv* that you can download via the link to the book's resources at https://nostarch.com/practical-sql-2nd-edition/. Download the file and save it on your computer. Then use the code in Listing 14-5 to build a table that has a column for each data element you can parse from the text using a regular expression.

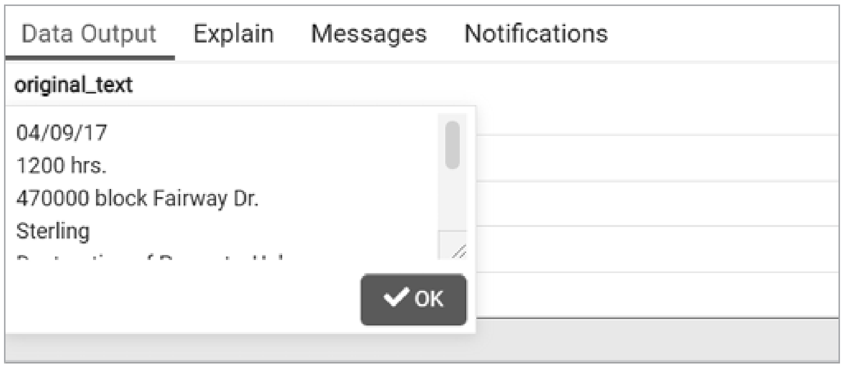
```
CREATE TABLE crime_reports (
    crime_id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    case_number text,
    date_1 timestamptz,
    date_2 timestamptz,
    street text,
    city text,
    crime_type text,
    description text,
    original_text text NOT NULL
);
```

```
COPY crime_reports (original_text)
FROM 'C:\YourDirectory\crime_reports.csv'
WITH (FORMAT CSV, HEADER OFF, QUOTE '"');
```

Listing 14-5: Creating and loading the `crime_reports` *table*

Run the `CREATE TABLE` statement in Listing 14-5 and then use `COPY` to load the text into the column `original_text`. The rest of the columns will be `NULL` until we fill them.

When you run `SELECT original_text FROM crime_reports;` in pgAdmin, the results grid should display five rows and the first several words of each report. When you double-click any cell, pgAdmin shows all the text in that row, as shown in Figure 14-1.



```
04/09/17
1200 hrs.
470000 block Fairway Dr.
Sterling
```

Figure 14-1: Displaying additional text in the pgAdmin results grid

Now that you've loaded the text you'll be parsing, let's explore this data using PostgreSQL regular expression functions.

Matching Crime Report Date Patterns

The first piece of data we want to extract from `original_text` is the date or dates of the crime. Most reports have one date, although one has two. The reports also have associated times, and we'll combine the extracted date and time into a timestamp. We'll fill `date_1` with each report's first (or only) date and time. If a second date or second time exists, we'll add it to `date_2`.

We'll use the `regexp_match(`*`string`*`, `*`pattern`*`)` function, which is similar to `substring()` with a few exceptions. One is that it returns each match as text in an array. Also, if there are no matches, it returns `NULL`. As you might recall from Chapter 6, you use an array to pass a list of values into the `percentile_cont()` function to calculate quartiles. I'll show you how to work with results that come back as an array when we parse the crime reports.

NOTE

The `regexp_match()` function was introduced in PostgreSQL 10 and is not available in earlier versions.

To start, let's use `regexp_match()` to find dates in each of the five incidents. The general pattern to match is *`MM/DD/YY`*, although there may be one or two digits for both the month and date. Here's a regular expression that matches the pattern:

```
\d{1,2}\/\d{1,2}\/\d{2}
```

In this expression, the first `\d{1,2}` indicates the month. The numbers inside the curly brackets specify that you want at least one digit and at most two digits. Next, you want to look for a forward slash (`/`), but because a forward slash can have special meaning in regular expressions, you must *escape* that character by placing a backslash (`\`) in front of it, like this `\/`. Escaping a character in this context simply means we want to treat it as a literal rather than letting it take on special meaning. So, the combination of the backslash and forward slash (`\/`) indicates you want a forward slash.

Another `\d{1,2}` follows for a single- or double-digit day of the month. The expression ends with a second escaped forward slash and `\d{2}` to indicate the two-digit year. Let's pass the expression `\d{1,2}\/\d{1,2}\/\d{2}` to `regexp_match()`, as shown in Listing 14-6.

```
SELECT crime_id,
       regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}')
FROM crime_reports
ORDER BY crime_id;
```

Listing 14-6: Using `regexp_match()` to find the first date

Run that code in pgAdmin, and the results should look like this:

```
crime_id    regexp_match
--------    ------------
       1    {4/16/17}
       2    {4/8/17}
       3    {4/4/17}
       4    {04/10/17}
       5    {04/09/17}
```

Note that each row shows the first date listed for the incident, because `regexp_match()` returns the first match it finds by default. Also note that each date is enclosed in curly brackets. That's PostgreSQL indicating that `regexp_match()` returns each result as an array type, or list of elements. Later, in the "Extracting Text from the regexp_match() Result" section, I'll show you how to access elements in the array. You also can read more about arrays in PostgreSQL at https://www.postgresql.org/docs/current/arrays.html.

Matching the Second Date When Present

We’ve successfully matched the first date from each report. But recall that one of the five incidents has a second date. To find and display all the dates in the text, you must use the related `regexp_matches()` function and pass in an option in the form of the flag `g`, as shown in Listing 14-7.

```
SELECT crime_id,
       regexp_matches(original_text, '\d{1,2}\/\d{1,2}\/\d{2}', 'g'1)
FROM crime_reports
ORDER BY crime_id;
```

Listing 14-7: *Using the* `regexp_matches()` *function with the* `g` *flag*

The `regexp_matches()` function, when supplied the `g` flag 1, differs from `regexp_match()` by returning each match the expression finds as a row in the results rather than returning just the first match.

Run the code again with this revision; you should now see two dates for the incident that has a `crime_id` of `1`, like this:

```
crime_id    regexp_matches
--------    --------------
       1    {4/16/17}
       1    {4/17/17}
       2    {4/8/17}
       3    {4/4/17}
       4    {04/10/17}
       5    {04/09/17}
```

Any time a crime report has a second date, we want to load it and the associated time into the `date_2` column. Although adding the `g` flag shows us all the dates, to extract just the second date in a report, we can use the pattern we always see when two dates exist. In Listing 14-4, the first block of text showed the two dates separated by a hyphen, like this:

```
4/16/17-4/17/17
```

This means you can switch back to `regexp_match()` and write a regular expression to look for a hyphen followed by a date, as shown in Listing 14-8.

```
SELECT crime_id,
       regexp_match(original_text, '-\d{1,2}\/\d{1,2}\/\d{2}')
FROM crime_reports
ORDER BY crime_id;
```

Listing 14-8: *Using* `regexp_match()` *to find the second date*

Although this query finds the second date in the first item (and returns a `NULL` for the rest), there’s an unintended consequence: it displays the hyphen along with it.

```
crime_id    regexp_match
--------    ------------
       1    {-4/17/17}
       2
       3
       4
       5
```

You don't want to include the hyphen, because it's an invalid format for the `timestamp` data type. Fortunately, you can specify the exact part of the regular expression you want to return by placing parentheses around it to create a capture group, like this:

```
-(\d{1,2}/\d{1,2}/\d{1,2})
```

This notation returns only the part of the regular expression you want. Run the modified query in Listing 14-9 to report only the data in parentheses.

```
SELECT crime_id,
       regexp_match(original_text, '-(\d{1,2}\/\d{1,2}\/\d{2})')
FROM crime_reports
ORDER BY crime_id;
```

Listing 14-9: Using a capture group to return only the date

The query in Listing 14-9 should return just the second date without the leading hyphen, as shown here:

```
crime_id    regexp_match
--------    ------------
       1    {4/17/17}
       2
       3
       4
       5
```

The process you've just completed is typical. You start with text to analyze and then write and refine the regular expression until it finds the data you want. So far, we've created regular expressions to match the first date and a second date, if it exists. Now, let's use regular expressions to extract additional data elements.

Matching Additional Crime Report Elements

In this section, we'll capture times, addresses, crime type, description, and case number from the crime reports. Here are the expressions for capturing this information:

First hour `\/\d{2}\n(\d{4})`

The first hour, which is the hour the crime was committed or the start of the time range, always follows the date in each crime report, like this:

```
4/16/17-4/17/17
2100-0900 hrs.
```

To find the first hour, we start with an escaped forward slash and `\d{2}`, which represents the two-digit year preceding the first date (`17`). The `\n` character indicates the newline because the hour always starts on a new line, and `\d{4}` represents the four-digit hour (`2100`). Because we just want to return the four digits, we put `\d{4}` inside parentheses as a capture group.

Second hour `\/\d{2}\n\d{4}-(\d{4})`

If the second hour exists, it will follow a hyphen, so we add a hyphen and another `\d{4}` to the expression we just created for the first hour. Again, the second `\d{4}` goes inside a capture group, because `0900` is the only hour we want to return.

Street `hrs.\n(\d+ .+(?:Sq.|Plz.|Dr.|Ter.|Rd.))`

In this data, the street always follows the time's `hrs.` designation and a newline (`\n`), like this:

```
04/10/17
1605 hrs.
21800 block Newlin Mill Rd.
```

The street address always starts with some number that varies in length and ends with an abbreviated suffix of some kind. To describe this pattern, we use `\d+` to match any digit that appears one or more times. Then we specify a space and look for any character one or more times using the dot wildcard and plus sign (`.+`) notation. The expression ends with a series of terms separated by the alternation pipe symbol that looks like this: `(?:Sq.|Plz.|Dr.|Ter.|Rd.)`. The terms are inside parentheses, so the expression will match one or another of those terms. When we group terms like this, if we don't want the parentheses to act as a capture group, we need to add `?:` to negate that effect.

NOTE

In a large dataset, it's likely roadway names would end with suffixes beyond the five in our regular expression. After making an initial pass at extracting the street, you can run a query to check for unmatched rows to find additional suffixes to match.

City `(?:Sq.|Plz.|Dr.|Ter.|Rd.)\n(\w+ \w+|\w+)\n`

Because the city always follows the street suffix, we reuse the terms separated by the alternation symbol we just created for the street. We follow that with a newline (`\n`) and then use a capture group to look for two words or one word `(\w+ \w+|\w+)` before a final newline, because a town or city name can be more than a single word.

Crime type `\n(?:\w+ \w+|\w+)\n(.*):`

The type of crime always precedes a colon (the only time a colon is used in each report) and might consist of one or more words, like this:

```
--snip--
Middleburg
Larceny: A license plate was reported
stolen from a vehicle.
SO170006250
--snip--
```

To create an expression that matches this pattern, we follow a newline with a nonreporting capture group that looks for the one- or two-word city. Then we add another newline and match any character that occurs zero or more times before a colon using `(.*):`.

Description `:\s(.+)(?:C0|SO)`

The crime description always comes between the colon after the crime type and the case number. The expression starts with the colon, a space character (`\s`), and then a capture group to find any character that appears one or more times using the `.+` notation. The nonreporting capture group `(?:C0|SO)` tells the program to stop looking when it encounters either `C0` or `SO`, the two character pairs that start each case number (a *C* followed by a zero, and an *S* followed by a capital *O*). We have to do this because the description might have one or more line breaks.

Case number `(?:C0|SO)[0-9]+`

The case number starts with either `C0` or `SO`, followed by a set of digits. To match this pattern, the expression looks for either `C0` or `SO` in a nonreporting capture group followed by any digit from 0 to 9 that occurs one or more times using the `[0-9]` range notation.

Now let's pass some of these regular expressions to `regexp_match()` to see them in action. *Listing 14-10* shows a sample `regexp_match()` query that retrieves the case number, first date, crime type, and city.

```
SELECT
    regexp_match(original_text, '(?:C0|SO)[0-9]+') AS case_number,
    regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}') AS date_1,
    regexp_match(original_text, '\n(?:\w+ \w+|\w+)\n(.*):') AS crime_type,
    regexp_match(original_text, '(?:Sq.|Plz.|Dr.|Ter.|Rd.)\n(\w+
\w+|\w+)\n')
        AS city
FROM crime_reports
ORDER BY crime_id;
```

Listing 14-10: Matching case number, date, crime type, and city

Run the code, and the results should look like this:

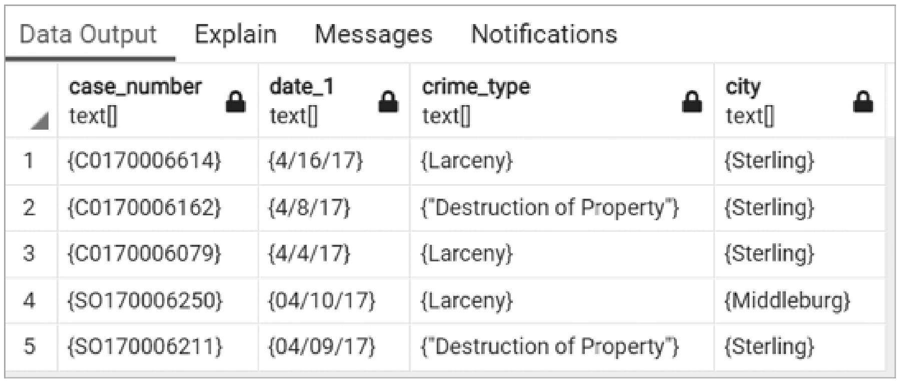
```
 case_number      date_1        crime_type                     city
-------------    ----------    -------------------------      ------------
{C0170006614}    {4/16/17}     {Larceny}                      {Sterling}
{C0170006162}    {4/8/17}      {"Destruction of Property"}    {Sterling}
{C0170006079}    {4/4/17}      {Larceny}                      {Sterling}
{SO170006250}    {04/10/17}    {Larceny}                      {Middleburg}
{SO170006211}    {04/09/17}    {"Destruction of Property"}    {Sterling}
```

After all that wrangling, we've transformed the text into a structure that is more suitable for analysis. Of course, you would have to include many more incidents to count the frequency of crime type by city or by the number of crimes per month to identify any trends.

To load each parsed element into the table's columns, we'll create an `UPDATE` query. But before you can insert the text into a column, you'll need to learn how to extract the text from the array that `regexp_match()` returns.

Extracting Text from the regexp_match() Result

In "Matching Crime Report Date Patterns," I mentioned that `regexp_match()` returns data in an array type containing text. Two clues reveal that these are array types. The first is that the data type designation in the column header shows `text[]` instead of `text`. The second is that each result is surrounded by curly brackets. Figure 14-2 shows how pgAdmin displays the results of the query in Listing 14-10.



	case_number text[]	date_1 text[]	crime_type text[]	city text[]
1	{C0170006614}	{4/16/17}	{Larceny}	{Sterling}
2	{C0170006162}	{4/8/17}	{"Destruction of Property"}	{Sterling}
3	{C0170006079}	{4/4/17}	{Larceny}	{Sterling}
4	{SO170006250}	{04/10/17}	{Larceny}	{Middleburg}
5	{SO170006211}	{04/09/17}	{"Destruction of Property"}	{Sterling}

Figure 14-2: Array values in the pgAdmin results grid

The `crime_reports` columns we want to update are not array types, so rather than passing in the array values returned by `regexp_match()`, we need to extract the values from the array first. We do this by using array notation, as shown in Listing 14-11.

```
SELECT
    crime_id,
  1 (regexp_match(original_text, '(?:C0|SO)[0-9]+'))[1] 2
        AS case_number
FROM crime_reports
ORDER BY crime_id;
```

Listing 14-11: Retrieving a value from within an array

First, we wrap the `regexp_match()` function 1 in parentheses. Then, at the end, we provide a value of `1`, which represents the first element in the array, enclosed in square brackets 2. The query should produce the following results:

```
crime_id    case_number
--------    -----------
       1    C0170006614
       2    C0170006162
       3    C0170006079
       4    SO170006250
       5    SO170006211
```

Now the data type designation in the pgAdmin column header should show `text` instead of `text[]`, and the values are no longer enclosed in curly brackets. We can now insert these values into `crime_reports` using an `UPDATE` query.

Updating the crime_reports Table with Extracted Data

To start updating columns in `crime_reports`, *Listing 14-12* combines the extracted first date and time into a single `timestamp` value for the column `date_1`.

```
  UPDATE crime_reports
1 SET date_1 =
  (
    2 (regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}'))[1]
        3 || ' ' ||
    4 (regexp_match(original_text, '\/\d{2}\n(\d{4})'))[1]
        5 ||' US/Eastern'
6 )::timestamptz
  RETURNING crime_id, date_1, original_text;
```

Listing 14-12: Updating the `crime_reports` `date_1` column

Because the `date_1` column is of type `timestamp`, we must provide an input in that data type. To do that, we'll use the PostgreSQL double-pipe (`||`) concatenation operator to combine the extracted date and time in a format that's acceptable for `timestamp with time zone` input. In the `SET` clause 1, we start with the regex pattern that matches the first date 2. Next, we concatenate the date with a space using two single quotes 3 and repeat the concatenation operator. This step combines the date with a space before connecting it to the regex pattern that matches the time 4. Then we include the time zone for the Washington, DC, area by concatenating that at the end of the string 5 using the `US/Eastern` designation. Concatenating these elements creates a string in the pattern of *`MM/DD/YY HH:MM TIMEZONE`*, which is acceptable as a `timestamp` input. We cast the string to a `timestamp with time zone` data type 6 using the PostgreSQL double-colon shorthand and the `timestamptz` abbreviation.

When you run the `UPDATE`, the `RETURNING` clause will display the columns we specify from the updated rows, including the now-filled `date_1` column alongside a portion of the original_text column, like this:

```
crime_id         date_1                            original_text
-------- ---------------------- -------------------------------------------
-
    1    2017-04-16 21:00:00-04  4/16/17-4/17/17
                                 2100-0900 hrs.
                                 46000 Block Ashmere Sq.
                                 Sterling
                                 Larceny: The victim reported that a
                                 bicycle was stolen from their opened
                                 garage door during the overnight hours.
                                 C0170006614
    2    2017-04-08 16:00:00-04  4/8/17
                                 1600 hrs.
                                 46000 Block Potomac Run Plz.
                                 Sterling
                                 Destruction of Property: The victim
                                 reported that their vehicle was spray
                                 painted and the trim was ripped off while
                                 it was parked at this location.
                                 C0170006162
--snip--
```

At a glance, you can see that `date_1` accurately captures the first date and time that appears in the original text and puts it into a format that we can analyze—quantifying, for example, which times of day crimes most often occur. Note that if you're not in the Eastern time zone, the timestamps will instead reflect your pgAdmin client's time zone. Also, in pgAdmin, you may need to double-click a cell in the `original_text` column to see the full text.

Using CASE to Handle Special Instances

You could write an `UPDATE` statement for each remaining data element, but combining those statements into one would be more efficient. *Listing 14-13* updates all the `crime_reports` columns using a single statement while handling inconsistent values in the data.

```
UPDATE crime_reports
SET date_11 =
    (
      (regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}'))[1]
          || ' ' ||
      (regexp_match(original_text, '\/\d{2}\n(\d{4})'))[1]
          ||' US/Eastern'
    )::timestamptz,

    date_22 =
    CASE3
        WHEN4 (SELECT regexp_match(original_text, '-
(\d{1,2}\/\d{1,2}\/\d{2})') IS NULL5)
                AND (SELECT regexp_match(original_text, '\/\d{2}\n\d{4}-
(\d{4})') IS NOT NULL6)
        THEN7
          ((regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}'))[1]
              || ' ' ||
          (regexp_match(original_text, '\/\d{2}\n\d{4}-(\d{4})'))[1]
              ||' US/Eastern'
          )::timestamptz

        WHEN8 (SELECT regexp_match(original_text, '-
(\d{1,2}\/\d{1,2}\/\d{2})') IS NOT NULL)
                AND (SELECT regexp_match(original_text, '\/\d{2}\n\d{4}-
(\d{4})') IS NOT NULL)
        THEN
          ((regexp_match(original_text, '-(\d{1,2}\/\d{1,2}\/\d{1,2})'))[1]
              || ' ' ||
          (regexp_match(original_text, '\/\d{2}\n\d{4}-(\d{4})'))[1]
              ||' US/Eastern'
          )::timestamptz
    END,
    street = (regexp_match(original_text, 'hrs.\n(\d+ .+
(?:Sq.|Plz.|Dr.|Ter.|Rd.))'))[1],
    city = (regexp_match(original_text,
                        '(?:Sq.|Plz.|Dr.|Ter.|Rd.)\n(\w+ \w+|\w+)\n'))
[1],
    crime_type = (regexp_match(original_text, '\n(?:\w+ \w+|\w+)\n(.*):'))
[1],
    description = (regexp_match(original_text, ':\s(.+)(?:C0|SO)'))[1],
    case_number = (regexp_match(original_text, '(?:C0|SO)[0-9]+'))[1];
```

Listing 14-13: Updating all `crime_reports` columns

This `UPDATE` statement might look intimidating, but it's not if we break it down by column. First, we use the same code from *Listing 14-9* to update the `date_1` column 1. But to update `date_2` 2, we need to account for the inconsistent presence of a second date and time. In our limited dataset, there are three possibilities:

A second hour exists but not a second date. This occurs when a report covers a range of hours on one date.

A second date and second hour exist. This occurs when a report covers more than one date.

Neither a second date nor a second hour exists.

To insert the correct value in `date_2` for each scenario, we use a `CASE` statement to test for each possibility. After the `CASE` keyword 3, we use a series of `WHEN ... THEN` statements to check for the first two conditions and provide the value to insert; if neither condition exists, the `CASE` statement will by default return a `NULL`.

The first `WHEN` statement 4 checks whether `regexp_match()` returns a `NULL` 5 for the second date and a value for the second hour (using `IS NOT NULL` 6). If that condition evaluates as `true`, the `THEN` statement 7 concatenates the first date with the second hour to create a timestamp for the update.

The second WHEN statement 8 checks that regexp_match() returns a value for the second hour and second date. If true, the THEN statement concatenates the second date with the second hour to create a timestamp.

If neither of the two WHEN statements returns true, the CASE statement will return a NULL because there is only a first date and first time.

NOTE

The WHEN statements handle the possibilities that exist in our small sample dataset. If you are working with more data, you might need to handle additional variations, such as a second date but not a second time.

When we run the full query in Listing 14-13, PostgreSQL should report UPDATE 5. Success! Now that we've updated all the columns with the appropriate data while accounting for elements that have additional data, we can examine all the columns of the table and find the parsed elements from original_text. Listing 14-14 queries four of the columns.

```
SELECT date_1,
       street,
       city,
       crime_type
FROM crime_reports
ORDER BY crime_id;
```

Listing 14-14: Viewing selected crime data

The results of the query should show a nicely organized set of data that looks something like this:

```
date_1                  street                            city         
crime_type
----------------------  --------------------------------  ----------    --
----------------
2017-04-16 21:00:00-04  46000 Block Ashmere Sq.           Sterling
Larceny
2017-04-08 16:00:00-04  46000 Block Potomac Run Plz.      Sterling
Destruction of ...
2017-04-04 14:00:00-04  24000 Block Hawthorn Thicket Ter. Sterling
Larceny
2017-04-10 16:05:00-04  21800 block Newlin Mill Rd.       Middleburg
Larceny
2017-04-09 12:00:00-04  470000 block Fairway Dr.          Sterling
Destruction of ...
```

You've successfully transformed raw text into a table that can answer questions and reveal storylines about crime in this area.

The Value of the Process

Writing regular expressions and coding a query to update a table can take time, but there is value to identifying and collecting data this way. In fact, some of the best datasets you'll encounter are those you build yourself. Everyone can download the same datasets, but the ones you build are yours alone. You get to be first person to find and tell the story behind the data.

Also, after you set up your database and queries, you can use them again and again. In this example, you could collect crime reports every day (either by hand or by automating downloads using a programming language such as Python) for an ongoing dataset that you can mine continually for trends.

In the next section, we'll continue our exploration of text by implementing a search engine using PostgreSQL.

Full-Text Search in PostgreSQL

PostgreSQL comes with a powerful full-text search engine that adds capabilities for searching large amounts of text, similar to online search tools and technology that powers search on research databases, such as Factiva. Let's walk through a simple example of setting up a table for text search and associated search functions.

For this example, I assembled 79 speeches by US presidents since World War II. Consisting mostly of State of the Union addresses, these public texts are available through the Internet Archive at *https://archive.org/* and the University of California's American Presidency Project at *https://www.presidency.ucsb.edu/*. You can find the data in the *president_speeches.csv* file along with the book's resources at *https://nostarch.com/practical-sql-2nd-edition/*.

Let's start with the data types unique to full-text search.

Text Search Data Types

PostgreSQL's implementation of text search includes two data types. The `tsvector` data type represents the text to be searched and to be stored in a normalized form. The `tsquery` data type represents the search query terms and operators. Let's look at the details of both.

Storing Text as Lexemes with tsvector

The `tsvector` data type reduces text to a sorted list of *lexemes*, which are linguistic units in a given language. It's helpful to think of lexemes as word roots without the variations created by suffixes. For example, a `tsvector` type column would store the words *washes*, *washed*, and *washing* as the lexeme *wash* while noting each word's position in the original text. Converting text to `tsvector` also removes small *stop words* that usually don't play a role in search, such as *the* or *it*.

To see how this data type works, let's convert a string to `tsvector` format. *Listing 14-15* uses the PostgreSQL search function `to_tsvector()`, which normalizes the text "I am walking across the sitting room to sit with you" to lexemes using the `english` language search configuration.

```
SELECT to_tsvector('english', 'I am walking across the sitting room to sit
with you.');
```

Listing 14-15: Converting text to `tsvector` *data*

Execute the code, and it should return the following output in the `tsvector` data type:

```
'across':4 'room':7 'sit':6,9 'walk':3
```

The `to_tsvector()` function reduces the number of words from eleven to four, eliminating words such as *I*, *am*, and *the*, which are not helpful search terms. The function removes suffixes, changing *walking* to *walk* and *sitting* to *sit*. It orders the words alphabetically, and the number following each colon indicates its position in the original string, taking stop words into account. Note that *sit* is recognized as being in two positions, one for *sitting* and one for *sit*.

NOTE

To see additional search language configurations installed with PostgreSQL, you can run the query `SELECT cfgname FROM pg_ts_config;`.

Creating the Search Terms with tsquery

The `tsquery` data type represents the full-text search query, again optimized as lexemes. It also provides operators for controlling the search. Examples of operators include the ampersand (`&`) for AND, the pipe symbol (`|`) for OR, and the exclamation point (`!`) for NOT. The `<->` followed by operator lets you search for adjacent words or words a certain distance apart.

Listing 14-16 shows how the `to_tsquery()` function converts search terms to the `tsquery` data type.

```
SELECT to_tsquery('english', 'walking & sitting');
```

Listing 14-16: Converting search terms to `tsquery` *data*

After running the code, you should see that the resulting `tsquery` data type has normalized the terms into lexemes, which match the format of the data to search:

```
'walk' & 'sit'
```

Now you can use terms stored as `tsquery` to search text optimized as `tsvector`.

Using the @@ Match Operator for Searching

With the text and search terms converted to the full-text search data types, you can use the double at sign (`@@`) match operator to check whether a query matches text. The first query in Listing 14-17 uses `to_tsquery()` to evaluate whether the text contains both *walking* and *sitting*, which we combine with the `&` operator. It returns a Boolean value of `true` because the lexemes of both *walking* and *sitting* are present in the text converted by `to_tsvector()`.

```
SELECT to_tsvector('english', 'I am walking across the sitting room') @@
       to_tsquery('english', 'walking & sitting');

SELECT to_tsvector('english', 'I am walking across the sitting room') @@
       to_tsquery('english', 'walking & running');
```

Listing 14-17: Querying a `tsvector` type with a `tsquery`

However, the second query returns `false` because both *walking* and *running* are not present in the text. Now let's build a table for searching the speeches.

Creating a Table for Full-Text Search

The code in Listing 14-18 creates and fills `president_speeches` with a column for the original text as well as a column of type `tsvector`. After the import, we'll convert the speech text to the `tsvector` data type. Note that to accommodate how I set up the CSV file, the `WITH` clause in `COPY` has a different set of parameters than what we've generally used. It's pipe-delimited and uses an ampersand for quoting.

```
CREATE TABLE president_speeches (
    president text NOT NULL,
    title text NOT NULL,
    speech_date date NOT NULL,
    speech_text text NOT NULL,
    search_speech_text tsvector,
    CONSTRAINT speech_key PRIMARY KEY (president, speech_date)
);

COPY president_speeches (president, title, speech_date, speech_text)
FROM 'C:\YourDirectory\president_speeches.csv'
WITH (FORMAT CSV, DELIMITER '|', HEADER OFF, QUOTE '@');
```

Listing 14-18: Creating and filling the `president_speeches` table

After executing the query, run `SELECT * FROM president_speeches;` to see the data. In pgAdmin, double-click any cell to see extra words not visible in the results grid. You should see a sizable amount of text in each row of the `speech_text` column.

Next, we use the `UPDATE` query in Listing 14-19 to copy the contents of `speech_text` to the `tsvector` column `search_speech_text` and transform it to that data type at the same time:

```
  UPDATE president_speeches
1 SET search_speech_text = to_tsvector('english', speech_text);
```

Listing 14-19: Converting speeches to `tsvector` *in the* `search_speech_text` *column*

The `SET` clause 1 fills `search_speech_text` with the output of `to_tsvector()`. The first argument in the function specifies the language for parsing the lexemes. We're using `english` here, but you can substitute `spanish`, `german`, `french`, and other languages (some languages may require you to find and install additional dictionaries). Using `simple` for the language will remove stop words but not reduce words to lexemes. The second argument is the name of the input column. Run the code to fill the column.

Finally, we want to index the `search_speech_text` column to speed up searches. You learned about indexing in Chapter 8, which focused on PostgreSQL's default index type, B-tree. For full-text search, the PostgreSQL documentation recommends using the *generalized inverted index* (*GIN*). A GIN index, according to the documentation, contains "an index entry for each word (lexeme), with a compressed list of matching locations." See *https://www.postgresql.org/docs/current/textsearch-indexes.html* for details.

You can add a GIN index using `CREATE INDEX` in Listing 14-20.

```
CREATE INDEX search_idx ON president_speeches USING
gin(search_speech_text);
```

Listing 14-20: Creating a GIN index for text search

Now you're ready to use search functions.

NOTE

Another way to set up a column for search is to create an index on a text column using the `to_tsvector()` *function. See https://www.postgresql.org/docs/current/textsearch-tables.html for details.*

Searching Speech Text

Nearly 80 years' worth of presidential speeches is fertile ground for exploring history. For example, the query in Listing 14-21 lists the speeches in which the president discussed Vietnam.

```
  SELECT president, speech_date
  FROM president_speeches
1 WHERE search_speech_text @@ to_tsquery('english', 'Vietnam')
  ORDER BY speech_date;
```

Listing 14-21: Finding speeches containing the word Vietnam

In the WHERE clause, the query uses the double at sign (@@) match operator 1 between the search_speech_text column (of data type tsvector) and the query term *Vietnam*, which to_tsquery() transforms into tsquery data. The results should list 19 speeches, showing that the first mention of Vietnam came up in a 1961 special message to Congress by John F. Kennedy and became a recurring topic starting in 1966 as America's involvement in the Vietnam War escalated.

```
president            speech_date
-----------------    -----------
John F. Kennedy      1961-05-25
Lyndon B. Johnson    1966-01-12
Lyndon B. Johnson    1967-01-10
Lyndon B. Johnson    1968-01-17
Lyndon B. Johnson    1969-01-14
Richard M. Nixon     1970-01-22
Richard M. Nixon     1972-01-20
Richard M. Nixon     1973-02-02
Gerald R. Ford       1975-01-15
--snip--
```

Before we try more searches, let's add a method for showing the location of our search term in the text.

Showing Search Result Locations

To see where our search terms appear in text, we can use the ts_headline() function. It displays one or more highlighted search terms surrounded by adjacent words with options to format the display, the number of words to show around the matched search term, and how many matched results to show from each row of text. Listing 14-22 highlights how to display a search for a specific instance of the word *tax* using ts_headline().

```
SELECT president,
       speech_date,
     1 ts_headline(speech_text, to_tsquery('english', 'tax'),
                 2 'StartSel = <,
                    StopSel = >,
                    MinWords=5,
                    MaxWords=7,
                    MaxFragments=1')
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'tax')
ORDER BY speech_date;
```

Listing 14-22: Displaying search results with ts_headline()

To declare ts_headline() 1, we pass the original speech_text column rather than the tsvector column we used in the search function as the first argument. Then, as the second argument, we pass a to_tsquery() function that specifies the word to highlight. We follow this with a third argument that lists optional formatting parameters 2 separated by commas. Here, we specify characters that will identify the start and end of the matched search term or terms (`StartSel` and `StopSel`). We also set the minimum and maximum number of total words to display, including the matched terms (`MinWords` and `MaxWords`), plus the maximum number of fragments (or instances of a match) to show using `MaxFragments`. These settings are optional, and you can adjust them according to your needs.

The results of this query should show at most seven words per speech, highlighting words in which *tax* is the root:

```
     president       speech_date                    ts_headline
-------------------- ----------- -----------------------------------------
---------------
Harry S. Truman      1946-01-21  price controls, increased <taxes>, savings
bond campaigns
Harry S. Truman      1947-01-06  excise <tax> rates which, under the
present
Harry S. Truman      1948-01-07  increased-after <taxes>-by more than
Harry S. Truman      1949-01-05  Congress enact new <tax> legislation to
bring
Harry S. Truman      1950-01-04  considered <tax> reduction of the 80th
Congress
Harry S. Truman      1951-01-08  major increase in <taxes> to meet
Harry S. Truman      1952-01-09  This means high <taxes> over the next
Dwight D. Eisenhower 1953-02-02  reduction of the <tax> burden;
Dwight D. Eisenhower 1954-01-07  brought under control. <Taxes> have begun
Dwight D. Eisenhower 1955-01-06  prices and materials. <Tax> revisions
encouraged increased
--snip--
```

Now, we can quickly see the context of the term we searched. You might also use this function to provide flexible display options for a search feature on a web application. And notice that we didn't just find exact matches. The search engine identified `tax` along with `taxes`, `Tax`, and `Taxes`—words with *tax* as the root and regardless of case.

Let's continue trying forms of searches.

Using Multiple Search Terms

As another example, we could look for speeches in which a president mentioned the word *transportation* but didn't discuss *roads*. We might want to do this to find speeches that focused on broader policy rather than a specific roads program. To do this, we use the syntax in Listing 14-23.

```
SELECT president,
       speech_date,
     1 ts_headline(speech_text,
                   to_tsquery('english', 'transportation & !roads'),
                   'StartSel = <,
                    StopSel = >,
                    MinWords=5,
                    MaxWords=7,
                    MaxFragments=1')
FROM president_speeches
2 WHERE search_speech_text @@
```

```
      to_tsquery('english', 'transportation & !roads')
ORDER BY speech_date;
```

Listing 14-23: Finding speeches with the word transportation *but not* roads

Again, we use `ts_headline()` 1 to highlight the terms our search finds. In the `to_tsquery()` function in the `WHERE` clause 2, we pass `transportation` and `roads`, combining them with the ampersand (`&`) operator. We use the exclamation point (`!`) in front of `roads` to indicate that we want speeches that do not contain this word. This query should find 15 speeches that fit the criteria. Here are the first four rows:

```
president          speech_date ts_headline
------------------ ----------- ---------------------------------------------
--------------
Harry S. Truman    1947-01-06  such industries as <transportation>, coal,
oil, steel
Harry S. Truman    1949-01-05  field of <transportation>.
John F. Kennedy    1961-01-30  Obtaining additional air <transport>
mobility--and obtaining
Lyndon B. Johnson 1964-01-08  reformed our tangled <transportation> and
transit policies
--snip--
```

Notice that the highlighted words in the `ts_headline` column include `transportation` and `transport`. Again, `to_tsquery()` converted `transportation` to the lexeme `transport` for the search term. This database behavior is extremely useful in helping to find relevant related words.

Searching for Adjacent Words

Finally, we'll use the distance (`<->`) operator, which consists of a hyphen between the less-than and greater-than signs, to find adjacent words. Alternatively, you can place a number between the signs to find terms that many words apart. For example, Listing 14-24 searches for any speeches that include the word *military* immediately followed by *defense*.

```
SELECT president,
       speech_date,
       ts_headline(speech_text,
                   to_tsquery('english', 'military <-> defense'),
                   'StartSel = <,
                    StopSel = >,
                    MinWords=5,
                    MaxWords=7,
                    MaxFragments=1')
FROM president_speeches
WHERE search_speech_text @@
      to_tsquery('english', 'military <-> defense')
ORDER BY speech_date;
```

Listing 14-24: Finding speeches where defense *follows* military

This query should find five speeches, and because `to_tsquery()` converts the search terms to lexemes, the words identified in the speeches should include plurals, such as *military defenses*. The following shows the speeches that have the adjacent terms:

```
president                speech_date    ts_headline
--------------------     -----------    ------------------------------------
---------------
Dwight D. Eisenhower     1956-01-05     system our <military> <defenses> are
designed
Dwight D. Eisenhower     1958-01-09     direct <military> <defense> efforts,
but likewise
Dwight D. Eisenhower     1959-01-09     survival--the <military> <defense>
of national life
Richard M. Nixon         1972-01-20     <defense> spending. Strong
<military> <defenses>
Jimmy Carter             1979-01-23     secure. Our <military> <defenses>
are strong
```

If you changed the query terms to `military <2> defense`, the database would return matches where the terms are exactly two words apart, as in the phrase "our military and defense commitments."

Ranking Query Matches by Relevance

You can also rank search results by relevance using two of PostgreSQL's full-text search functions. These functions are helpful when you're trying to understand which piece of text, or speech in this case, is most relevant to your particular search terms.

One function, `ts_rank()`, generates a rank value (returned as a variable-precision `real` data type) based on how often the lexemes you're searching for appear in the text. The other function, `ts_rank_cd()`, considers how close the lexemes searched are to each other. Both functions can take optional arguments to consider document length and other factors. The rank value they generate is an arbitrary decimal that's useful for sorting but doesn't have any inherent meaning. For example, a value of `0.375` generated during one query isn't directly comparable to the same value generated during a different query.

As an example, Listing 14-25 uses `ts_rank()` to rank speeches containing all the words *war*, *security*, *threat*, and *enemy*.

```
SELECT president,
       speech_date,
     1 ts_rank(search_speech_text,
               to_tsquery('english', 'war & security & threat & enemy'))
               AS score
FROM president_speeches
2 WHERE search_speech_text @@
       to_tsquery('english', 'war & security & threat & enemy')
ORDER BY score DESC
LIMIT 5;
```

Listing 14-25: Scoring relevance with `ts_rank()`

In this query, the `ts_rank()` function 1 takes two arguments: the `search_speech_text` column and the output of a `to_tsquery()` function containing the search terms. The output of the function receives the alias `score`. In the `WHERE` clause 2 we filter the results to only those speeches that contain the search terms specified. Then we order the results in `score` in descending order and return just five of the highest-ranking speeches. The results should be as follows:

```
    president       speech_date   score
------------------ ----------- ----------
William J. Clinton 1997-02-04  0.35810584
George W. Bush     2004-01-20  0.29587495
George W. Bush     2003-01-28  0.28381455
Harry S. Truman    1946-01-21  0.25752166
William J. Clinton 2000-01-27  0.22214262
```

Bill Clinton's 1997 State of the Union message contains the words *war*, *security*, *threat*, and *enemy* more often than the other speeches, as he discussed the Cold War and other topics. However, it also happens to be one of the longer speeches in the table (which you can determine by using `char_length()`, as you learned earlier in the chapter). The lengths of speeches influences these rankings because `ts_rank()` factors in the number of matching terms in a given text. Two speeches by George W. Bush, delivered in the years before and after the start of the Iraq War, rank next.

It would be ideal to compare frequencies between speeches of identical lengths to get a more accurate ranking, but this isn't always possible. However, we can factor in the length of each speech by adding a normalization code as a third parameter of the `ts_rank()` function, as shown in Listing 14-26.

```
SELECT president,
       speech_date,
       ts_rank(search_speech_text,
               to_tsquery('english', 'war & security & threat & enemy'),
21)::numeric
               AS score
FROM president_speeches
WHERE search_speech_text @@
      to_tsquery('english', 'war & security & threat & enemy')
ORDER BY score DESC
LIMIT 5;
```

Listing 14-26: Normalizing `ts_rank()` by speech length

Adding the optional code `2` 1 instructs the function to divide the `score` by the length of the data in the `search_speech_text` column. This quotient then represents a score normalized by the document length, giving an apples-to-apples comparison among the speeches. The PostgreSQL documentation at https://www.postgresql.org/docs/current/textsearch-controls.html lists all the options available for text search, including using the document length and dividing by the number of unique words.

After running the code in Listing 14-26, the rankings should change:

```
    president       speech_date   score
------------------- ----------- ----------
George W. Bush      2004-01-20  0.0001028060
William J. Clinton 1997-02-04  0.0000982188
George W. Bush      2003-01-28  0.0000957216
Jimmy Carter        1979-01-23  0.0000898701
Lyndon B. Johnson   1968-01-17  0.0000728288
```

In contrast to the ranking results in *Listing 14-25*, George W. Bush's 2004 speech now tops the rankings, and Truman's 1946 message falls out of the top five. This might be a more meaningful ranking than the first sample output, because we normalized it by length. But three of the five top-ranked speeches are the same between the two sets, and you can be reasonably certain that each of these three is worthy of closer examination to understand more about presidential speeches that include wartime terminology.

Wrapping Up

Far from being boring, text offers abundant opportunities for data analysis. In this chapter, you've learned techniques for turning ordinary text into data you can extract, quantify, search, and rank. In your work or studies, keep an eye out for routine reports that have facts buried inside chunks of text. You can use regular expressions to dig them out, turn them into structured data, and analyze them to find trends. You can also use search functions to analyze the text.

In the next chapter, you'll learn how PostgreSQL can help you analyze geographic information.

TRY IT YOURSELF

Use your new text-wrangling skills to tackle these tasks:

The style guide of a publishing company you're writing for wants you to avoid commas before suffixes in names. But there are several names like `Alvarez, Jr.` and `Williams, Sr.` in your database. Which functions can you use to remove the comma? Would a regular expression function help? How would you capture just the suffixes to place them into a separate column?

Using any one of the presidents' speeches, count the number of unique words that are five characters or more. (Hint: You can use `regexp_split_to_table()` in a subquery to create a table of words to count.) Bonus: Remove commas and periods at the end of each word.

Rewrite the query in *Listing 14-25* using the `ts_rank_cd()` function instead of `ts_rank()`. According to the PostgreSQL documentation, `ts_rank_cd()` computes cover density, which takes into account how close the lexeme search terms are to each other. Does using the `ts_rank_cd()` function significantly change the results?

15
ANALYZING SPATIAL DATA WITH POSTGIS

We now turn to *spatial data*, defined as information about the location, shape, and attributes of objects—points, lines, or polygons, for example—within a geographical space. In this chapter, you'll learn how to construct and query spatial data using SQL, and you'll be introduced to the PostGIS extension for PostgreSQL that enables support for spatial data types and functions.

Spatial data has become a critical piece of our world's data ecosystem. A phone app can find nearby coffee shops because it queries a spatial database, asking it to return a list of shops within a certain distance of your location. Governments use spatial data to track the footprints of residential and business parcels; epidemiologists use it to visualize the spread of diseases.

For our exercises, we'll analyze the location of farmers' markets across the United States as well as roads and waterways in Santa Fe, New Mexico. You'll learn how to construct and query spatial data types and incorporate map projections and grid systems. You'll receive tools to glean information from spatial data, similar to how you've analyzed numbers and text.

We'll start by setting up PostGIS. All code and data for the exercises are available with the book's resources at *https://nostarch.com/practical-sql-2nd-edition/*.

Enabling PostGIS and Creating a Spatial Database

PostGIS is a free, open source project created by the Canadian geospatial company Refractions Research and maintained by an international team of developers under the Open Source Geospatial Foundation (OSGeo). The GIS portion of its name refers to *geographic information system*, defined as a system that allows for storing, editing, analyzing, and displaying spatial data. You'll find documentation and updates at *https://postgis.net/*.

If you installed PostgreSQL following the steps for Windows, macOS, or the Ubuntu flavor of Linux in Chapter 1, PostGIS should be on your machine. If you installed PostgreSQL some other way on Windows or macOS or if you're on another Linux distribution, follow the installation instructions at *https://postgis.net/install/*.

To enable PostGIS on your `analysis` database, open pgAdmin's Query Tool and run the statement in *Listing 15-1*.

```
CREATE EXTENSION postgis;
```

Listing 15-1: Loading the PostGIS extension

You'll see the message `CREATE EXTENSION`, advising that your database has been updated to include spatial data types and analysis functions. Run **`SELECT postgis_full_version();`** to display the version number of PostGIS along with the versions of its installed components. The version won't match your installed PostgreSQL version, and that's okay.

Understanding the Building Blocks of Spatial Data

Before you learn to query spatial data, let's look at how it's described in GIS and related data formats. This is important background, but if you want to dive straight into queries, you can skip to "Understanding PostGIS Data Types" later in the chapter and return here afterward.

A point on a grid is the smallest building block of spatial data. The grid might be marked with x- and y-axes, or longitude and latitude if we're using a map. A grid could be flat with two dimensions, or it could describe a three-dimensional space such as a cube. In some data formats, such as the JavaScript-based *GeoJSON*, a point may have attributes in addition to its location. We could describe a grocery store with a point containing its longitude and latitude as well as attributes for the store's name and hours of operation.

Understanding Two-Dimensional Geometries

The Open Geospatial Consortium (OGC) and International Organization for Standardization (ISO) have created a *simple features access* model that describes standards for building and querying two- and three-dimensional shapes, sometimes referred to as *geometries*. PostGIS supports the standard.

The following are the more common features, starting with points and building in complexity:

Point

A single location in a two- or three-dimensional plane. On maps, a Point is usually a dot marking a longitude and latitude.

LineString

Two or more Points, each connected by straight lines. A LineString can represent features such as a road, biking trail, or stream.

Polygon

A two-dimensional shape with three or more straight sides, each constructed from a LineString. On maps, Polygons represent objects such as nations, states, buildings, and bodies of water. A Polygon can have one or more interior Polygons that act as holes inside the larger Polygon.

MultiPoint

A set of Points. A single MultiPoint object could represent multiple locations of a retailer with each store's latitude and longitude.

MultiLineString

A set of LineStrings. An example is a road that has several noncontinuous segments.

MultiPolygon

A set of Polygons. A parcel of land that's divided into parts by a road could be grouped in one MultiPolygon object instead of separate polygons.

Figure 15-1 shows an example of each feature. PostGIS enables functions to build, edit, and analyze these objects. These functions take a variety of inputs depending on their purpose, including latitude and longitude, specialized text and binary formats, and simple features. Some functions also take an optional *spatial reference system identifier (SRID)* that specifies the grid on which to place the objects.

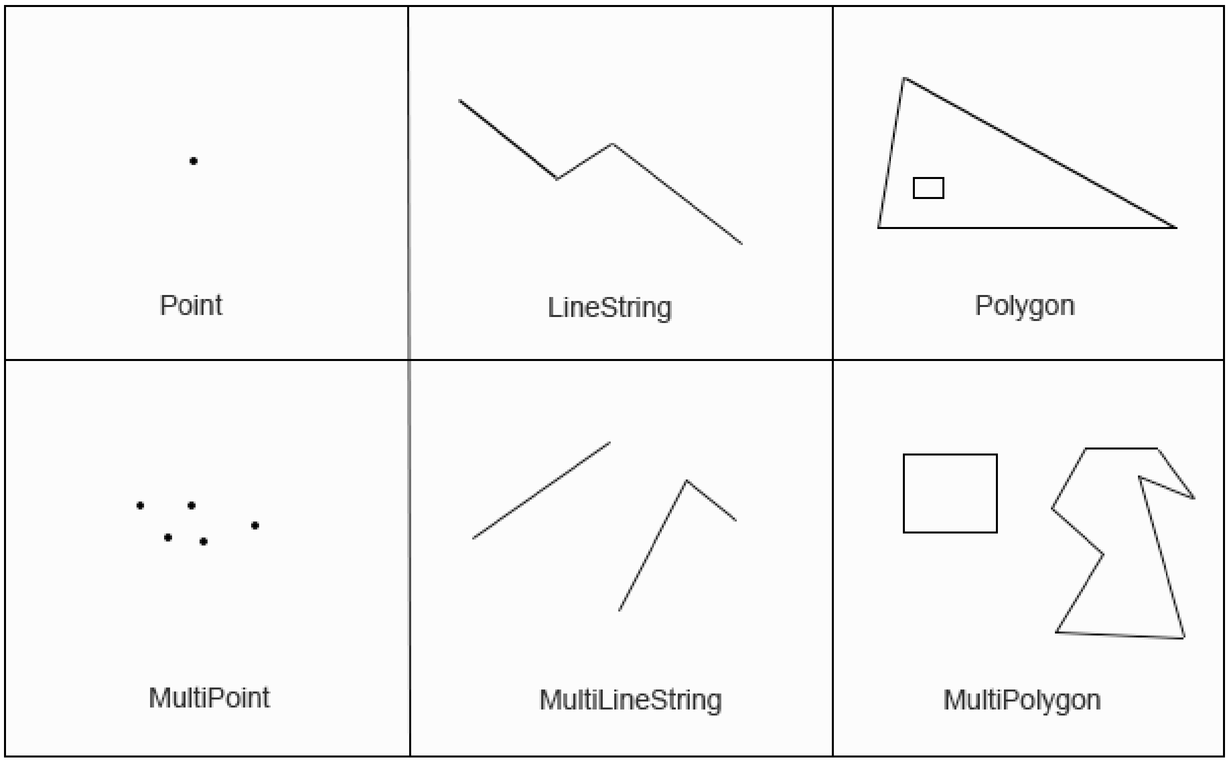


Figure 15-1: Visual examples of geometries

I'll explain the SRID shortly, but first, let's look at examples of an input used by PostGIS functions called *well-known text (WKT)*—a text-based format that represents a geometry.

Well-Known Text Formats

The OGC standard's WKT format specifies a geometry type and its coordinates inside one or more sets of parentheses. The number of coordinates and parentheses varies depending on the type of geometry. *Table 15-1* shows examples of frequently used geometry types and their WKT formats. Longitude/latitude pairs are shown for the coordinates, but you might encounter grid systems that use other measures.

NOTE

WKT accepts coordinates in the order of (longitude, latitude), which is backward from Google Maps and some other software. Tom MacWright, formerly of the Mapbox software company, notes at https://macwright.com/lonlat/ that neither order is "right" and catalogs the "frustrating inconsistency" in which mapping-related code handles the order of coordinates.

Table 15-1: *Well-Known Text Formats for Geometries*

Geometry	Format	Notes
Point	`POINT (-74.9 42.7)`	A coordinate pair marking a point at −74.9 longitude and 42.7 latitude.
LineString	`LINESTRING (-74.9 42.7, -75.1 42.7)`	A straight line with endpoints marked by two coordinate pairs.
Polygon	`POLYGON ((-74.9 42.7, -75.1 42.7, -75.1 42.6, -74.9 42.7))`	A triangle outlined by three different pairs of coordinates. Although listed twice, the first and last pair are the same coordinates where we close the shape.
MultiPoint	`MULTIPOINT (-74.9 42.7, -75.1 42.7)`	Two Points, one for each pair of coordinates.
MultiLineString	`MULTILINESTRING ((-76.27 43.1, -76.06 43.08), (-76.2 43.3, -76.2 43.4, -76.4 43.1))`	Two LineStrings. The first has two points; the second has three.
MultiPolygon	`MULTIPOLYGON (((-74.92 42.7, -75.06 42.71, -75.07 42.64, -74.92 42.7), (-75.0 42.66, -75.0 42.64, -74.98 42.64, -74.98 42.66, -75.0 42.66)))`	Two Polygons. The first is a triangle, and the second is a rectangle.

These examples create simple shapes, as you'll see when we construct them using PostGIS later in the chapter. In practice, complex geometries will comprise thousands of coordinates.

Projections and Coordinate Systems

Representing Earth's spherical surface on a two-dimensional map is not easy. Imagine peeling the outer layer of Earth from the globe and trying to spread it on a table while keeping all pieces of the continents and oceans connected. Inevitably, you'd have to stretch some parts of the map. That's what happens when cartographers create a map *projection* with its own *projected coordinate system*. A projection is simply a flattened representation of the globe with its own two-dimensional coordinate system.

Some projections represent the entire world; others are specific to regions or purposes. The *Mercator projection* has properties useful for navigation; Google Maps and other online maps use a variant of called *Web Mercator*. The math behind its transformation distorts land areas close to the North and South Poles, making them appear much larger than reality. The US Census Bureau uses the *Albers projection*, which minimizes distortion and is the one you see on TV in the United States as votes are tallied on election night.

Projections are derived from *geographic coordinate systems*, which define the grid of latitude, longitude, and height of any point on the globe along with factors including Earth's shape. Whenever you obtain geographic data, it's critical to know the coordinate systems it references so you provide the correct information when writing queries. Often, user documentation will name the coordinate system. Next, let's look at how to specify the coordinate system in PostGIS.

Spatial Reference System Identifier

When using PostGIS (and many GIS applications), you specify the coordinate system via its unique SRID. When you enabled the PostGIS extension at the beginning of this chapter, the process created the table spatial_ref_sys, which contains SRIDs as its primary key. The table also contains the column `srtext`, which includes a WKT representation of the spatial reference system plus other metadata.

In this chapter, we'll frequently use SRID `4326`, the ID for the geographic coordinate system WGS 84. That's the most recent World Geodetic System (WGS) standard used by GPS, and you'll encounter it often in spatial data. You can see the WKT representation for WGS 84 by running the code in Listing 15-2 that looks for its SRID, `4326`:

```
SELECT srtext
FROM spatial_ref_sys
WHERE srid = 4326;
```

Listing 15-2: Retrieving the WKT for SRID `4326`

Run the query and you should get the following result, indented for readability:

```
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563,
            AUTHORITY["EPSG","7030"]],
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0,
        AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.0174532925199433,
        AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4326"]]
```

You don't need to use this information for any of this chapter's exercises, but it's helpful to know some of the variables and how they define the projection. The `GEOGCS` keyword provides the geographic coordinate system in use. Keyword `PRIMEM` specifies the location of the *prime meridian*, or longitude 0. To see definitions of all the variables, check the reference at *https://docs.geotools.org/stable/javadocs/org/opengis/referencing/doc-files/WKT.html*.

Conversely, if you ever need to find the SRID associated with a coordinate system, you can query the `srtext` column in `spatial_ref_sys` to find it.

Understanding PostGIS Data Types

Installing PostGIS adds several data types to your database. We'll use two: `geography` and `geometry`. Both types can store spatial data, such as the points, lines, polygons, and SRIDs you just learned about, but they have important distinctions:

`geography` A data type based on a sphere, using the round-Earth coordinate system (longitude and latitude). All calculations occur on the globe, taking its curvature into account. This makes the math complex and limits the number of functions available to work with the `geography` type. But because Earth's curvature is factored in, calculations for distance are more precise; you should use the `geography` data type when handling data that spans large areas. The results from calculations on the `geography` type will be expressed in meters.

`geometry` A data type based on a plane, using the Euclidean coordinate system. Calculations occur on straight lines as opposed to along the curvature of a sphere, making calculations for geographical distance less precise than with the `geography` data type; the results of calculations are expressed in units of whichever coordinate system you've designated.

The PostGIS documentation at *https://postgis.net/docs/using_postgis_dbmanagement.html* offers guidance on when to use one or the other type. In short, if you're working strictly with longitude/latitude data or if your data covers a large area, such as a continent or the globe, use the `geography` type, even though it limits the functions you can use. If your data covers a smaller area, the `geometry` type provides more functions and better performance. You can also convert one type to the other using `CAST`.

With the background you have now, we can start working with spatial objects.

Creating Spatial Objects with PostGIS Functions

PostGIS has more than three dozen constructor functions that build spatial objects using WKT or coordinates. You can find a list at *https://postgis.net/docs/reference.html#Geometry_Constructors*, but the following sections explain several that you'll use in the exercises. Most PostGIS functions begin with the letters *ST*, which is an ISO naming standard that means *spatial type*.

Creating a Geometry Type from Well-Known Text

The `ST_GeomFromText(`*`WKT, SRID`*`)` function creates a `geometry` data type from an input of a WKT string and an optional SRID. *Listing 15-3* shows simple `SELECT` statements that generate `geometry` data types for each of the simple features described in *Table 15-1*.

```
SELECT ST_GeomFromText(1'POINT(-74.9233606 42.699992)',
24326);

SELECT ST_GeomFromText('LINESTRING(-74.9 42.7, -75.1 42.7)',
4326);

SELECT ST_GeomFromText('POLYGON((-74.9 42.7, -75.1 42.7,
                                 -75.1 42.6, -74.9 42.7))',
4326);

SELECT ST_GeomFromText('MULTIPOINT (-74.9 42.7, -75.1 42.7)',
4326);

SELECT ST_GeomFromText('MULTILINESTRING((-76.27 43.1, -76.06
43.08),
                                         (-76.2 43.3, -76.2
43.4,
                                          -76.4 43.1))',
4326);

SELECT ST_GeomFromText('MULTIPOLYGON3((
                                        (-74.92 42.7, -75.06
42.71,
                                         -75.07 42.64, -74.92
42.7)4,
                                        (-75.0 42.66, -75.0
42.64,
                                         -74.98 42.64, -74.98
42.66,
                                         -75.0 42.66)))',
4326);
```

Listing 15-3: Using `ST_GeomFromText()` to create spatial objects

For each example, we give a WKT string as the first input and the SRID `4326` as the second. In the first example, we create a Point by inserting the WKT `POINT` string 1 as the first argument to `ST_GeomFromText()` with the SRID 2 as the optional second argument. We use the same format in the rest of the examples. Note that we don't have to indent the coordinates. I do so here only to make the coordinate pairs more readable.

Be sure to mind the number of parentheses that segregate objects, particularly in complex structures such as the MultiPolygon. For example, we need to use two opening parentheses 3 and enclose each polygon's coordinates within another set of parentheses 4.

If you run each statement separately in pgAdmin, you can view both its data output and visual representation. Upon execution, each statement should return a single column of the `geometry` data type displayed as a string of characters that looks something like this truncated example:

```
0101000020E61000008EDA0E5718BB52C017BB7D5699594540 ...
```

The string is of the format *extended well-known binary (EWKB)*, which you typically won't need to interpret directly. Instead, you'll use columns of geometry (or geography) data as inputs to other functions. To see the visual representation, click the eye icon in the pgAdmin result column header. That should open a Geometry Viewer pane in pgAdmin that displays the geometry atop a map that uses OpenStreetMap as the base layer. For example, the `MULTIPOLYGON` example in *Listing 15-3* should look like *Figure 15-2*, with a triangle and a rectangle.

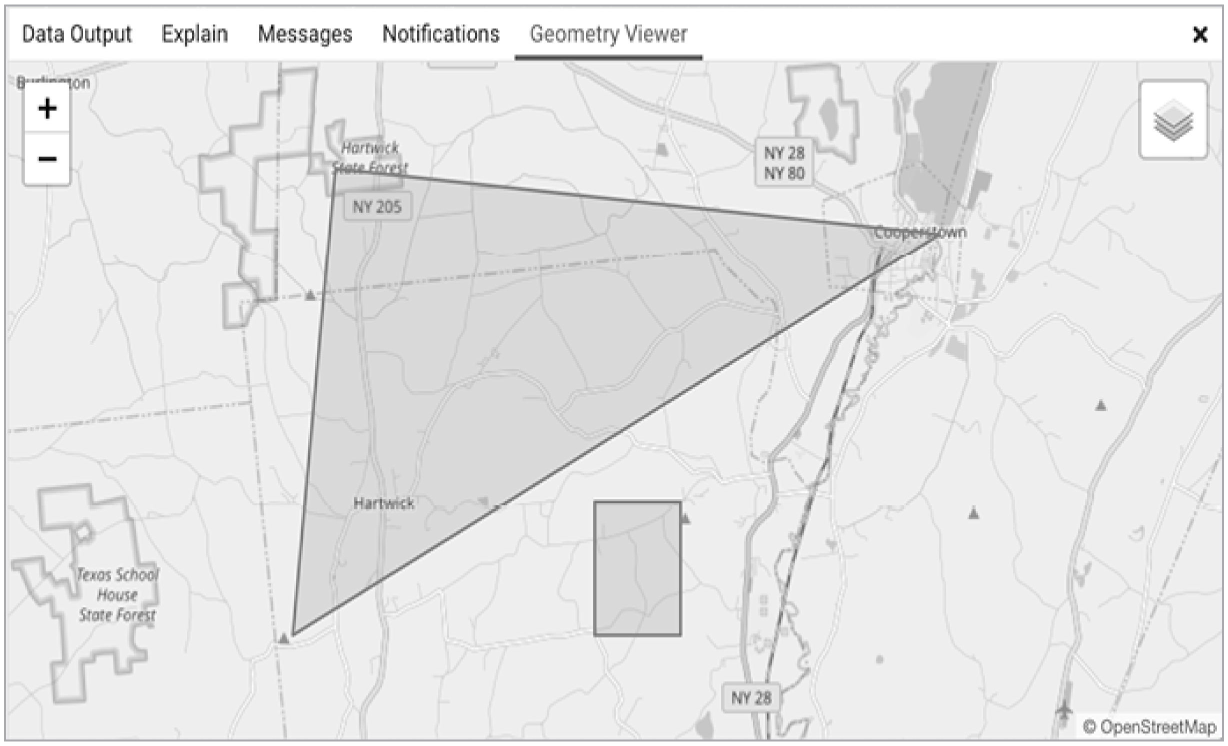


Figure 15-2: Viewing geometries in pgAdmin

Try viewing each example in Listing 15-3 to get to know the differences between objects.

Creating a Geography Type from Well-Known Text

To create a `geography` data type, you can use `ST_GeogFromText(`*`WKT`*`)` to convert a WKT or `ST_GeogFromText(`*`EWKT`*`)` to convert a PostGIS-specific variation called *extended WKT* that includes the SRID. Listing 15-4 shows how to pass in the SRID as part of the extended WKT string to create a MultiPoint `geography` object with three points.

```
SELECT
ST_GeogFromText('SRID=4326;MULTIPOINT(-74.9 42.7, -75.1 42.7,
-74.924 42.6)')
```

Listing 15-4: Using `ST_GeogFromText()` to create spatial objects

Again, you can view the Points on a map by clicking the eye icon in the geography column in the pgAdmin results grid.

Along with the all-purpose `ST_GeomFromText()` and `ST_GeogFromText()` functions, PostGIS includes several that are specific to creating certain spatial objects. I'll cover those briefly next.

Using Point Functions

The `ST_PointFromText()` and `ST_MakePoint()` functions will turn a WKT `POINT` or a collection of coordinates, respectively, into a `geometry` data type. Points mark coordinates, such as longitude and latitude, which you would use to identify locations or use as building blocks of other objects, such as LineStrings.

Listing 15-5 shows how these functions work.

```
SELECT 1ST_PointFromText('POINT(-74.9233606 42.699992)',
4326);

SELECT 2ST_MakePoint(-74.9233606, 42.699992);
SELECT 3ST_SetSRID(ST_MakePoint(-74.9233606, 42.699992),
4326);
```

Listing 15-5: Functions specific to making Points

The `ST_PointFromText(`*`WKT, SRID`*`)` 1 function creates a point `geometry` type from a WKT `POINT` and an optional SRID as the second input. The PostGIS docs note that the function includes validation of coordinates that makes it slower than the `ST_GeomFromText()` function.

The `ST_MakePoint(`*`x, y, z, m`*`)` 2 function creates a point `geometry` type on a two-, three-, and four-dimensional grid. The first two parameters, *`x`* and *`y`* in the example, represent longitude and latitude coordinates. You can use the optional *`z`* to represent altitude and *`m`* to represent a measure. That would allow you, for example, to mark a water fountain on a bike trail at a certain altitude and certain distance from the start of the trail. The `ST_MakePoint()` function is faster than `ST_GeomFromText()` and `ST_PointFromText()`, but if you want to specify an SRID, you'll need to designate one by wrapping it inside the `ST_SetSRID()` 3 function.

Using LineString Functions

Now let's examine some functions we use specifically for creating LineString `geometry` data types. *Listing 15-6* shows how they work.

```
SELECT 1ST_LineFromText('LINESTRING(-105.90 35.67,-105.91
35.67)', 4326);
SELECT 2ST_MakeLine(ST_MakePoint(-74.9, 42.7),
ST_MakePoint(-74.1, 42.4));
```

Listing 15-6: Functions specific to making LineStrings

The `ST_LineFromText(WKT, SRID)` 1 function creates a LineString from a WKT `LINESTRING` and an optional SRID as its second input. Like `ST_PointFromText()` earlier, this function includes validation of coordinates that makes it slower than `ST_GeomFromText()`.

The `ST_MakeLine(geom, geom)` 2 function creates a LineString from inputs that must be of the `geometry` data type. In *Listing 15-6*, the example uses two `ST_MakePoint()` functions as inputs to create the start and endpoint of the line. You can also pass in an `ARRAY` object with multiple points, perhaps generated by a subquery, to generate a more complex line.

Using Polygon Functions

Let's look at three Polygon functions: `ST_PolygonFromText()`, `ST_MakePolygon()`, and `ST_MPolyFromText()`. All create `geometry` data types. *Listing 15-7* shows how you can create Polygons with each.

```
SELECT 1ST_PolygonFromText('POLYGON((-74.9 42.7, -75.1 42.7,
                                     -75.1 42.6, -74.9
42.7))', 4326);

SELECT 2ST_MakePolygon(
             ST_GeomFromText('LINESTRING(-74.92 42.7, -75.06
42.71,
                                        -75.07 42.64, -74.92
42.7)', 4326));

SELECT 3ST_MPolyFromText('MULTIPOLYGON((
                                        (-74.92 42.7, -75.06
42.71,
                                         -75.07 42.64,
-74.92 42.7),
                                           (-75.0 42.66, -75.0
42.64,
                                            -74.98 42.64,
-74.98 42.66,
                                            -75.0 42.66)
                                          ))', 4326);
```

Listing 15-7: Functions specific to making Polygons

The `ST_PolygonFromText(WKT, SRID)` 1 function creates a Polygon from a WKT `POLYGON` and an optional SRID. As with the similarly named functions for creating points and lines, it includes a validation step that makes it slower than `ST_GeomFromText()`.

The `ST_MakePolygon(linestring)` 2 function creates a Polygon from a LineString that must open and close with the same coordinates, ensuring the object is closed. This example uses `ST_GeomFromText()` to create the LineString geometry using a WKT `LINESTRING`.

The `ST_MPolyFromText(WKT, SRID)` 3 function creates a MultiPolygon from a WKT and an optional SRID.

Now you have the building blocks to analyze spatial data. Next, we'll use them to explore a set of data.

Analyzing Farmers' Markets Data

The National Farmers' Market Directory from the US Department of Agriculture catalogs the location and offerings of more than 8,600 "markets that feature two or more farm vendors selling agricultural products directly to customers at a common, recurrent physical location," according to the update page linked from the main directory site at https://www.ams.usda.gov/local-food-directories/farmersmarkets/. Attending these markets is a fun weekend activity, so let's use SQL spatial queries to find the closest markets.

The *farmers_markets.csv* file contains a portion of the USDA data on each market, and it's available along with the book's resources at https://nostarch.com/practical-sql-2nd-edition/. Save the file to your computer and run the code in *Listing 15-8* to create and load a `farmers_markets` table.

```
CREATE TABLE farmers_markets (
    fmid bigint PRIMARY KEY,
    market_name text NOT NULL,
    street text,
    city text,
    county text,
    st text NOT NULL,
    zip text,
    longitude numeric(10,7),
    latitude numeric(10,7),
    organic text NOT NULL
);

COPY farmers_markets
FROM 'C:\YourDirectory\farmers_markets.csv'
WITH (FORMAT CSV, HEADER);
```

Listing 15-8: Creating and loading the `farmers_markets` table

The table contains routine address data plus the `longitude` and `latitude` for most markets. Twenty-nine of the markets were missing those values when I downloaded the file from the USDA. An `organic` column indicates whether the market offers organic products; a hyphen (`-`) in that column indicates an unknown value. After you import the data, count the rows using `SELECT count(*) FROM farmers_markets;`. If everything imported correctly, you should have 8,681 rows.

Creating and Filling a Geography Column

To perform spatial queries on the markets' longitude and latitude, we need to convert those coordinates into a single column with a spatial data type. Because we're working with locations spanning the entire United States and an accurate measurement of a large spherical distance is important, we'll use the `geography` type. After creating the column, we can update it using Points derived from the coordinates and then apply an index to speed up queries. *Listing 15-9* contains the statements for doing these tasks.

```
ALTER TABLE farmers_markets ADD COLUMN geog_point
geography(POINT,4326); 1

UPDATE farmers_markets
SET geog_point =
   2 ST_SetSRID(
             3
ST_MakePoint(longitude,latitude)4::geography,4326
              );

CREATE INDEX market_pts_idx ON farmers_markets USING GIST
(geog_point); 5

SELECT longitude,
       latitude,
       geog_point,
     6 ST_AsEWKT(geog_point)
FROM farmers_markets
WHERE longitude IS NOT NULL
LIMIT 5;
```

Listing 15-9: Creating and indexing a `geography` column

The `ALTER TABLE` statement 1 you learned in Chapter 10 with the `ADD COLUMN` option creates a column of the `geography` type called `geog_point` that will hold points and reference the WGS 84 coordinate system, which we denote using SRID `4326`.

Next, we run a standard `UPDATE` statement to fill the `geog_point` column. Nested inside an `ST_SetSRID()` 2 function, the `ST_MakePoint()` 3 function takes as input the `longitude` and `latitude` columns from the table. The output, which is the `geometry` type by default, must be cast to `geography` to match the `geog_point` column type. To do this, we add the PostgreSQL-specific double-colon syntax (`::`) 4 to the output of `ST_MakePoint()`.

Adding a Spatial Index

Before you start analysis, it's wise to add an index to the new column to speed up queries. In Chapter 8, you learned about PostgreSQL's default index, the B-tree. A B-tree index is useful for data that you can order and search using equality and range operators, but it's less useful for spatial objects. The reason is that you cannot easily sort GIS data along one axis. For example, the application has no way to determine which of these coordinate pairs is greatest: (0,0), (0,1), or (1,0).

Instead, the makers of PostGIS include support for an index designed for spatial data called *R-tree*. In an R-tree index, each spatial item is represented in the index as a rectangle that surrounds its boundaries, and the index itself is a hierarchy of rectangles. (Find a good overview at *https://postgis.net/workshops/postgis-intro/indexing.html*.)

We add a spatial index to the `geog_point` column by including the keywords `USING GIST` in the `CREATE INDEX` statement 5 in *Listing 15-9*. `GIST` refers to a generalized search tree (GiST), an interface to facilitate incorporating specialized indexes to the database. PostgreSQL core team member Bruce Momjian describes GiST as "a general indexing framework designed to allow indexing of complex data types."

With the index in place, we use the `SELECT` statement to view the geography data to show the newly encoded `geog_points` column. To view the extended WKT version of `geog_point`, we wrap it in a `ST_AsEWKT()` function 6 to show the extended well-known text coordinates and SRID. The results should look similar to this, with `geog_point` truncated for brevity:

```
 longitude     latitude    geog_point             st_asewkt
------------ ---------- ------------ ------------------------
-----------
-105.5890000 47.4154000 01010000...  SRID=4326;POINT(-105.589
47.4154)
 -98.9530000 40.4998000 01010000...  SRID=4326;POINT(-98.953
40.4998)
-119.4280000 35.7610000 01010000...  SRID=4326;POINT(-119.428
35.761)
 -92.3063000 42.1718000 01010000...  SRID=4326;POINT(-92.3063
42.1718)
 -70.6868160 44.1129600 01010000...
SRID=4326;POINT(-70.686816 44.11296))
```

Now we're ready to perform calculations on the points.

Finding Geographies Within a Given Distance

Several years ago, while reporting a story on farming in Iowa, I visited the massive Downtown Farmers' Market in Des Moines. With hundreds of vendors, the market spanned several city blocks in the Iowa capital. Farming is big business there, and even though the downtown market is huge, it's not the only one in the area. Let's use PostGIS to find more farmers' markets near downtown Des Moines.

The PostGIS function `ST_DWithin()` returns a Boolean value of `true` if one spatial object is within a specified distance of another object. If you're working with the `geography` data type, as we are here, you need to use meters as the distance unit. If you're using the `geometry` type, use the distance unit specified by the SRID.

NOTE

PostGIS distance measurements are on a straight line for geometry data, and on a sphere for geography data. Be careful not to confuse either with driving distance along roads, which is usually farther from point to point. To perform calculations related to driving distances, check out the extension pgRouting at https://pgrouting.org/.

[Listing 15-10]() uses the `ST_DWithin()` function to filter `farmers_markets` to show markets within 10 kilometers of the Downtown Farmers' Market in Des Moines.

```
SELECT market_name,
       city,
       st,
       geog_point
FROM farmers_markets
WHERE ST_DWithin(1 geog_point,
                 2 ST_GeogFromText('POINT(-93.6204386
41.5853202)'),
                 3 10000)
ORDER BY market_name;
```

Listing 15-10: Using `ST_DWithin()` to locate farmers' markets within 10 km of a point

The first input for `ST_DWithin()` is `geog_point` 1, which holds the location of each row's market in the `geography` data type. The second input is the `ST_GeogFromText()` function 2 that returns a Point geography from WKT. The coordinates `-93.6204386` and `41.5853202` represent the longitude and latitude of the Downtown Farmers' Market. The final input is `10000` 3, which is the number of meters in 10 kilometers. The database calculates the distance between each market in the table and the downtown market. If a market is within 10 kilometers, it is included in the results.

We're using Points here, but this function works with any geography or geometry type. If you're working with objects such as polygons, you can use the related `ST_DFullyWithin()` function to find objects that are completely within a specified distance.

Run the query; it should return nine rows (I've omitted the `geog_point` column for brevity):

```
market_name                                  city
st
---------------------------------------      ---------------
----
Beaverdale Farmers Market                    Des Moines
Iowa
Capitol Hill Farmers Market                  Des Moines
Iowa
Downtown Farmers' Market - Des Moines        Des Moines
Iowa
Drake Neighborhood Farmers Market            Des Moines
Iowa
Eastside Farmers Market                      Des Moines
Iowa
Highland Park Farmers Market                 Des Moines
Iowa
Historic Valley Junction Farmers Market      West Des Moines
Iowa
LSI Global Greens Farmers' Market            Des Moines
Iowa
Valley Junction Farmers Market               West Des Moines
Iowa
```

One of these nine markets is the Downtown Farmers' Market in Des Moines, which makes sense because its location is at the point used for comparison. The rest are other markets in Des Moines or in nearby West Des Moines.

To see these points on a map, in pgAdmin's results grid, click the eye icon in the `geog_point` column header. The geography viewer should display a map as shown in Figure 15-3.

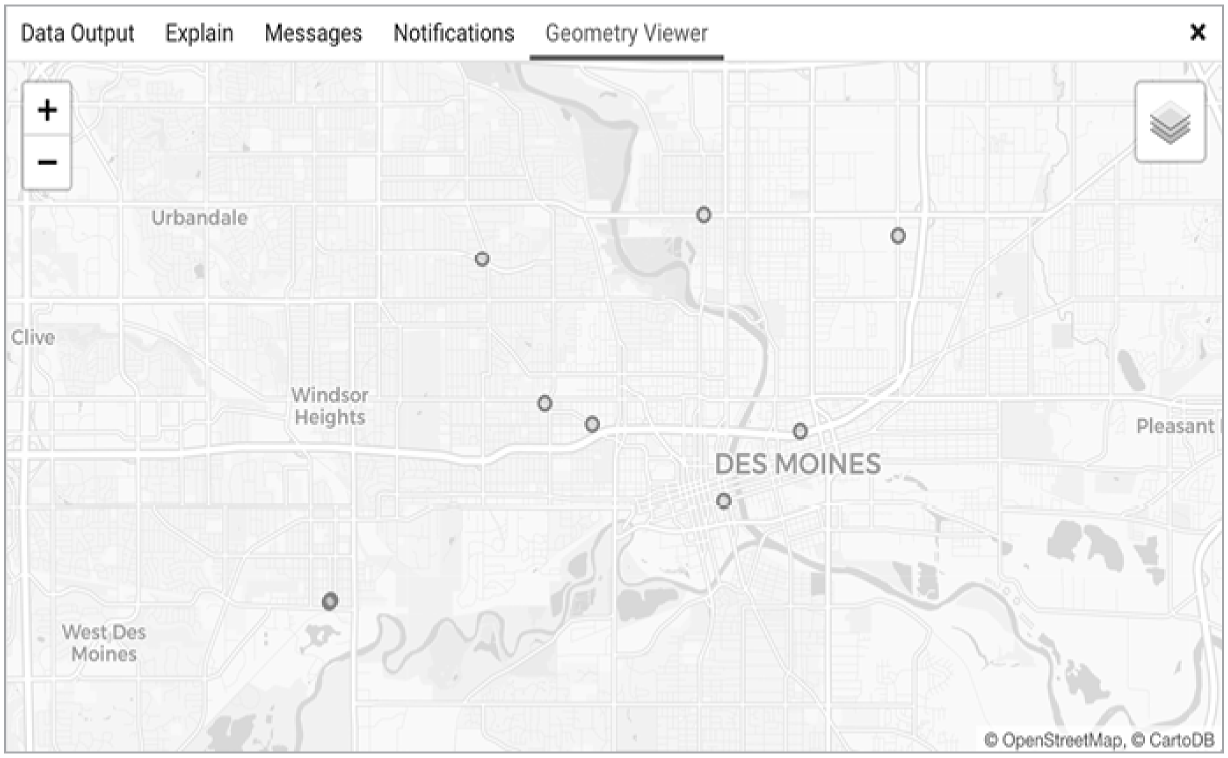


Figure 15-3: Farmers' markets near downtown Des Moines, Iowa

This operation should be familiar: it's a standard feature on many online maps and product apps that let you locate stores or points of interest near you.

Although this list of nearby markets is helpful, it would be even better to know the exact distance of markets from downtown. We'll use another function to report that.

Finding the Distance Between Geographies

The `ST_Distance()` function returns the minimum distance between two geometries, providing meters for geographies and SRID units for geometries. For example, Listing 15-11 finds the distance in miles from Yankee Stadium in New York City's Bronx borough to Citi Field in Queens, home of the New York Mets.

```
SELECT ST_Distance(
                   ST_GeogFromText('POINT(-73.9283685
40.8296466)'),
                   ST_GeogFromText('POINT(-73.8480153
40.7570917)')
                   ) / 1609.344 AS mets_to_yanks;
```

Listing 15-11: Using `ST_Distance()` to calculate the miles between Yankee Stadium and Citi Field

To convert the distance units from meters to miles, we divide the result of `ST_Distance()` by 1609.344 (the number of meters in a mile) The result is about 6.5 miles.

```
mets_to_yanks
-----------------
6.543861827875209
```

Let's apply this technique to the farmers' market data using the code in Listing 15-12. We'll again find all farmers' markets within 10 kilometers of the Downtown Farmers' Market in Des Moines and show the distance in miles.

```
SELECT market_name,
       city,
     1 round(
            (ST_Distance(geog_point,
                         ST_GeogFromText('POINT(-93.6204386
41.5853202)')
                         ) / 1609.344)2::numeric, 2
            ) AS miles_from_dt
FROM farmers_markets
WHERE3 ST_DWithin(geog_point,
                 ST_GeogFromText('POINT(-93.6204386
41.5853202)'),
                  10000)
ORDER BY miles_from_dt ASC;
```

Listing 15-12: Using `ST_Distance()` for each row in `farmers_markets`

The query is similar to *Listing 15-10*, which used `ST_DWithin()` to find markets 10 kilometers or closer to downtown, but adds the `ST_Distance()` function as a column to calculate and display the distance from downtown. I've wrapped the function inside `round()` 1 to trim the output.

We provide `ST_Distance()` with the same two inputs we gave `ST_DWithin()` in *Listing 15-10*: `geog_point` and the `ST_GeogFromText()` function. The `ST_Distance()` function then calculates the distance between the points specified by both inputs, returning the result in meters. To convert to miles, we divide by `1609.344` 2, the approximate number of meters in a mile. Then, to provide the `round()` function with the correct input data type, we cast the column result to type `numeric`.

The `WHERE` clause 3 uses the same `ST_DWithin()` function and inputs as in *Listing 15-10*. You should see the following results, ordered by distance in ascending order:

```
market_name                           city
miles_from_dt
-------------------------------------   ---------------    --
-----------
Downtown Farmers' Market - Des Moines   Des Moines
0.00
Capitol Hill Farmers Market             Des Moines
1.15
Drake Neighborhood Farmers Market       Des Moines
1.70
LSI Global Greens Farmers' Market       Des Moines
2.30
Highland Park Farmers Market            Des Moines
2.93
Eastside Farmers Market                 Des Moines
3.40
Beaverdale Farmers Market               Des Moines
3.74
Historic Valley Junction Farmers Market West Des Moines
4.68
```

```
Valley Junction Farmers Market          West Des Moines
4.70
```

Again, you see this type of result often when you're searching online for a store or address. You might also find the technique helpful for other analysis scenarios, such as finding all the schools within a certain distance of a known source of pollution or all the homes within five miles of an airport.

Finding the Nearest Geographies

Sometimes it's helpful to have the database simply return the spatial objects that are in closest proximity to another object without specifying some arbitrary distance in which to search. For example, we may want to find the closest farmers' market regardless of whether it's 10 kilometers away or 100. To do that, we can instruct PostGIS to implement a *K-nearest neighbors* search algorithm by using the `<->` distance operator in the `ORDER BY` clause of a query. Nearest neighbors algorithms solve a range of classification problems by identifying similar items—text recognition is an example. In this case, PostGIS will identify some number of spatial objects, represented by `K`, nearest to an object we specify.

For example, let's say we're planning to visit the vacation spot of Bar Harbor, Maine, and want to find the three farmer's markets closest to town. We can use the code in Listing 15-13.

```
SELECT market_name,
       city,
       st,
       round(
           (ST_Distance(geog_point,
                        ST_GeogFromText('POINT(-68.2041607
44.3876414)')
                        ) / 1609.344)::numeric, 2
            ) AS miles_from_bh
FROM farmers_markets
ORDER BY geog_point <->1 ST_GeogFromText('POINT(-68.2041607
44.3876414)')
LIMIT 3;
```

Listing 15-13: Using the `<->` *distance operator for a nearest neighbors search*

The query is similar to *Listing 15-12*, but instead of using a `WHERE` clause with `ST_DWithin()`, we provide an `ORDER BY` clause that contains the `<->` 1 distance operator. To the left of the operator, we place the `geog_point` column; to the right we supply the WKT for the Point locating downtown Bar Harbor inside `ST_GeogFromText()`. In effect, this syntax says, "Order the results by the distance from the geography to the Point."

Adding `LIMIT 3` restricts the results to the three closest markets (the three nearest neighbors):

```
          market_name                         city          st
miles_from_bh
--------------------------------  ----------------  ----- ---
----------
Bar Harbor Eden Farmers' Market   Bar Harbor        Maine
0.32
Northeast Harbor Farmers' Market  Northeast Harbor  Maine
7.65
Southwest Harbor Farmers' Market  Southwest Harbor  Maine
9.56
```

You can, of course, change the number in the `LIMIT` clause to return more or fewer results. Using `LIMIT 1`, for example, will return only the closest market.

So far, you've learned how to work with spatial objects constructed from WKT. Next, I'll show you a common data format used in GIS called the *shapefile* and how to bring it into PostGIS for analysis.

Working with Census Shapefiles

A *shapefile* is a GIS data file format developed by Esri, a US company known for its ArcGIS mapping visualization and analysis platform. Shapefiles are a standard file format for GIS platforms—such as ArcGIS and the open source QGIS—and are used by governments, corporations, nonprofits, and technical organizations to display, analyze, and distribute data with geographic features.

Shapefiles hold information describing the shape of a feature (such as a county, a road, or a lake) plus a database with each feature's attributes. Those attributes might include their name and other demographic descriptors. A single shapefile can contain only one type of shape, such as polygons or points, and when you load a shapefile into a GIS platform that supports visualization, you can view the shapes and query their attributes. PostgreSQL, with the PostGIS extension, lets you query the spatial data in the shapefile, which we'll do in "Exploring the Census 2019 Counties Shapefile" and "Performing Spatial Joins" later in the chapter.

First, let's examine the structure and contents of shapefiles.

Understanding the Contents of a Shapefile

A shapefile comprises a collection of files with different extensions, each with a different purpose. Often, when you download a shapefile, it comes in a compressed archive, such as *.zip*. You'll need to unzip it to access the individual files.

Per ArcGIS documentation, these are the most common extensions you'll encounter:

`.shp` Main file that stores the feature geometry.

`.shx` Index file that stores the index of the feature geometry.

`.dbf` Database table (in dBASE format) that stores the attribute information of features.

`.xml` XML-format file that stores metadata about the shapefile.

`.prj` Projection file that stores the coordinate system information. You can open this file with a text editor to view the geographic coordinate system and projection.

According to the documentation, files with the first three extensions include necessary data required for working with a shapefile. The other file types are optional. You can load a shapefile into PostGIS to access its spatial objects and the attributes for each. Let's do that next and explore some additional analysis functions.

I've included several shapefiles with the resources for this chapter at *https://nostarch.com/practical-sql-2nd-edition/*. We'll start with TIGER/Line Shapefiles from the US Census that contain the boundaries for each county or county equivalent, such as parish or borough, as of 2019. You can learn more about this series of shapefiles at *https://www.census.gov/geographies/mapping-files/time-series/geo/tiger-line-file.html*.

NOTE

Many organizations provide data in shapefile format. Start with your national or local government agencies or check the Wikipedia entry "List of GIS data sources."

Save *tl_2019_us_county.zip* from the book's resources for this chapter to your computer and unzip it; the archive should contain files including those with the extensions I listed earlier.

Loading Shapefiles

If you're using Windows, the PostGIS suite includes a Shapefile Import/Export Manager with a simple *graphical user interface (GUI)*. In recent years, builds of that GUI have become harder to find on macOS and Linux distributions, so for those operating systems we'll instead use the command line application `shp2pgsql`.

We'll start with the Windows GUI. If you're on macOS or Linux, skip ahead to "Importing Shapefiles Using shp2pgsql."

Windows Shapefile Importer/Exporter

On Windows, if you followed the installation steps in Chapter 1, you should find the Shapefile Import/Export Manager by selecting **Start▶PostGIS Bundle** *`x.y`* **for PostgreSQL x64** *`x.y`***▶ PostGIS Bundle** *`x.y`* **for PostgreSQL x64** *`x.y`* **Shapefile and DBF Loader Exporter**.

Whatever you see in place of *`x.y`* should match your PostgreSQL and PostGIS versions. Click to launch the application.

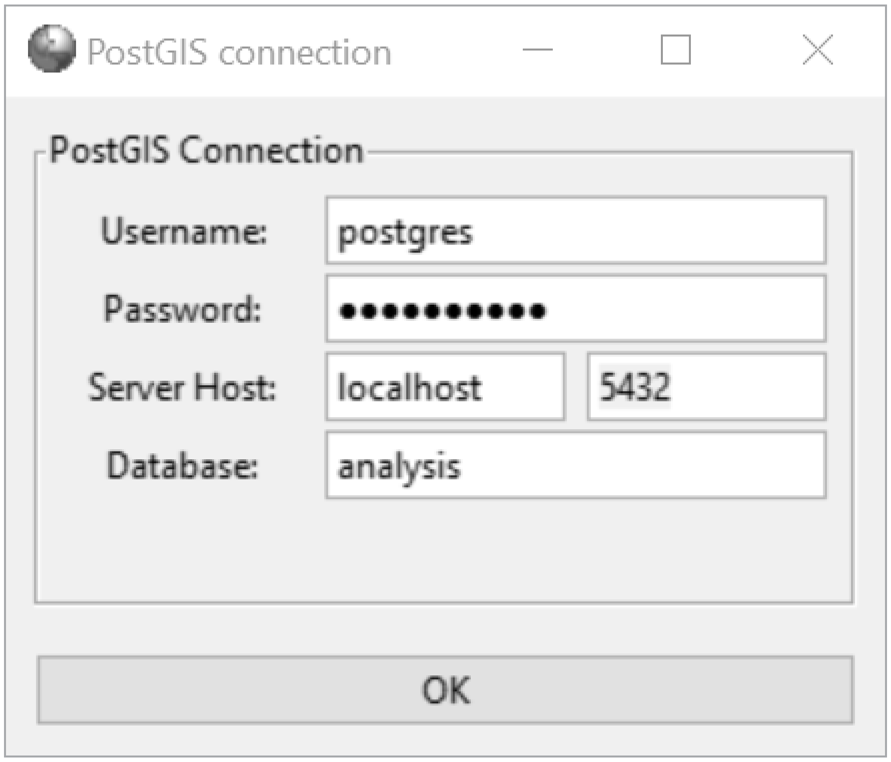
To establish a connection between the app and your `analysis` database, follow these steps:

. Click **View connection details**.

. In the dialog that opens, enter **postgres** for the username, and enter a password if you added one for the server during initial setup.

. Ensure that Server Host has `localhost` and `5432` by default. Leave those as is unless you're connecting to a different server or port.

. Enter **analysis** for the database name. *Figure 15-4* shows a screenshot of what the connection should look like.



PostGIS connection

PostGIS Connection

Username: postgres

Password: ●●●●●●●●●●●

Server Host: localhost 5432

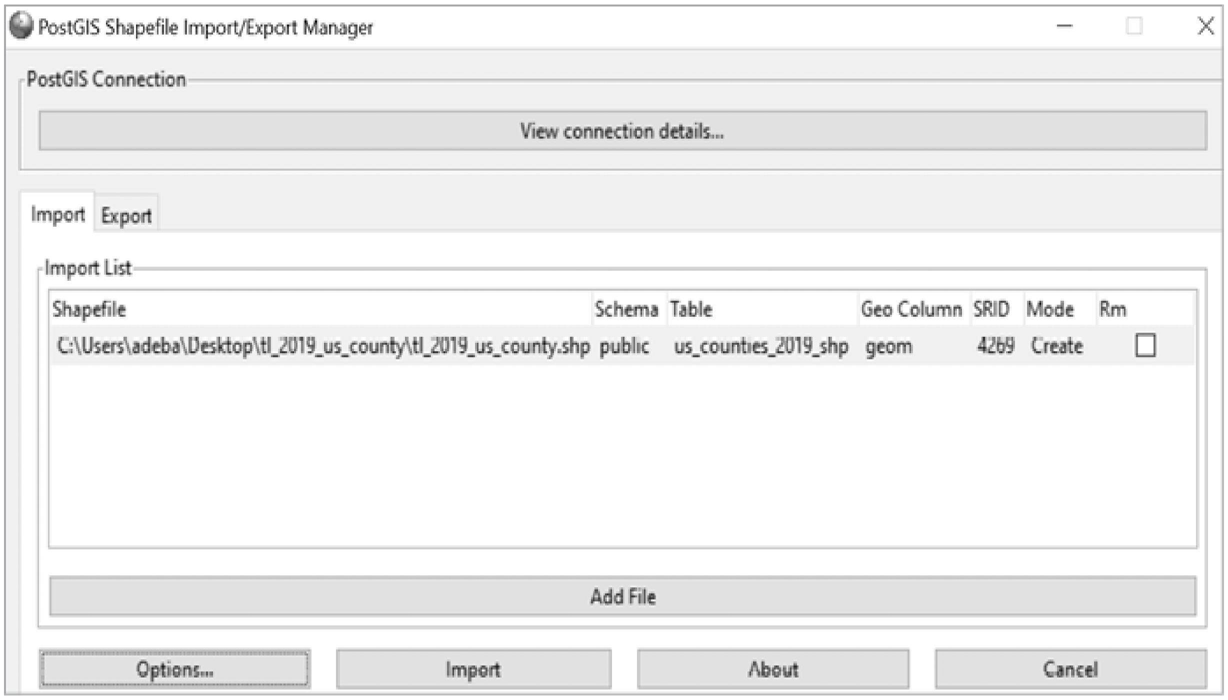
Database: analysis

Figure 15-4: Establishing the PostGIS connection in the shapefile loader

. Click **OK**. You should see the message `Connection Succeeded` in the log window. Now that you've successfully established the PostGIS connection, you can load your shapefile.

. Under Options, change `DBF file character encoding` to **Latin1**—we do this because the shapefile attributes include county names with characters that require this encoding. Keep the default checked boxes, including the one to create an index on the spatial column. Click **OK**.

Click **Add File** and select *tl_2019_us_county.shp* from the location you saved it. Click **Open**. The file should appear in the Shapefile list in the loader, as shown in *Figure 15-5*.



PostGIS Shapefile Import/Export Manager

PostGIS Connection

View connection details...

Import | Export

Import List

Shapefile	Schema	Table	Geo Column	SRID	Mode	Rm
C:\Users\adeba\Desktop\tl_2019_us_county\tl_2019_us_county.shp	public	us_counties_2019_shp	geom	4269	Create	☐

Add File

Options... | Import | About | Cancel

Figure 15-5: Specifying upload details in the shapefile loader

In the Table column, double-click to select the table name. Replace it with **us_counties_2019_shp**. Press ENTER to accept the value.

In the SRID column, double-click and enter **4269**. That's the ID for the North American Datum 1983 coordinate system, which is often used by US federal agencies including the US Census Bureau. Again, press ENTER to accept the value.

Click **Import**.

In the log window, you should see a message that ends with the following message:

```
Shapefile type: Polygon
PostGIS type: MULTIPOLYGON[2]
```