3

BEGINNING DATA EXPLORATION WITH SELECT

For me, the best part of digging into data isn't the prerequisites of gathering, loading, or cleaning the data, but when I actually get to *interview* the data. Those are the moments when I discover whether the data is clean or dirty, whether it's complete, and, most of all, what story the data can tell. Think of interviewing data as a process akin to interviewing a person applying for a job. You want to ask questions that reveal whether the reality of their expertise matches their résumé.

Interviewing the data is exciting because you discover truths. For example, you might find that half the respondents forgot to fill out the email field in the questionnaire, or the mayor hasn't paid property taxes for the past five years. Or you might learn that your data is dirty: names are spelled inconsistently, dates are incorrect, or numbers don't jibe with your expectations. Your findings become part of the data's story.

In SQL, interviewing data starts with the SELECT keyword, which retrieves rows and columns from one or more of the tables in a database. A SELECT statement can be simple, retrieving everything in a single table, or it can be complex enough to link dozens of tables while handling multiple calculations and filtering by exact criteria.

We'll start with simple SELECT statements and then look into the more powerful things SELECT can do.

Basic SELECT Syntax

Here's a SELECT statement that fetches every row and column in a table called my_table:

```
SELECT * FROM my_table;
```

This single line of code shows the most basic form of a SQL query. The asterisk following the SELECT keyword is a *wildcard*, which is like a stand-in for a value: it doesn't represent anything in particular and instead represents everything that value could possibly be. Here, it's shorthand for "select all columns." If you had given a column name instead of the wildcard, this command would select the values in that column. The FROM keyword indicates you want the query to return data from a particular table. The semicolon after the table name tells PostgreSQL it's the end of the query statement.

Let's use this SELECT statement with the asterisk wildcard on the teachers table you created in Chapter 2. Once again, open pgAdmin, select the analysis database, and open the Query Tool. Then execute the statement shown in Listing 3-1. Remember, as an alternative to typing these statements into the Query Tool, you can also run the code by clicking **Open File** and navigating to the place where you saved the code you downloaded from GitHub. Always do this if you see the code is truncated with `--snip--`. For this chapter, you should open *Chapter_03.sql* and highlight each statement before clicking the **Execute/Refresh** icon.

```
SELECT * FROM teachers;
```

Listing 3-1: Querying all rows and columns from the teachers table

Once you execute the query, the result set in the Query Tool's output pane contains all the rows and columns you inserted into the teachers table in Chapter 2. The rows may not always appear in this order, but that's okay.

id	first_name	last_name	school	hire_date	salary
1	Janet	Smith	F.D. Roosevelt HS	2011-10-30	36200
2	Lee	Reynolds	F.D. Roosevelt HS	1993-05-22	65000
3	Samuel	Cole	Myers Middle School	2005-08-01	43500
4	Samantha	Bush	Myers Middle School	2011-10-30	36200
5	Betty	Diaz	Myers Middle School	2005-08-30	43500
6	Kathleen	Roush	F.D. Roosevelt HS	2010-10-22	38500

Note that the `id` column (of type `bigserial`) is automatically filled with sequential integers, even though you didn't explicitly insert them. Very handy. This auto-incrementing integer acts as a unique identifier, or key, that not only ensures each row in the table is unique, but also later gives us a way to connect this table to other tables in the database.

Before we move on, note that you have two other ways to view all rows in a table. Using pgAdmin, you can right-click the `teachers` table in the object tree and choose **View/Edit Data ▸ All Rows**. Or you can use a little-known bit of standard SQL:

```
TABLE teachers;
```

Both provide the same result as the code in Listing 3-1. Now, let's refine this query to make it more specific.

Querying a Subset of Columns

Often, it's more practical to limit the columns the query retrieves, especially with large databases, so you don't have to wade through excess information. You can do this by naming columns, separated by commas, right after the `SELECT` keyword. Here's an example:

```
SELECT some_column, another_column, amazing_column FROM table_name;
```

With that syntax, the query will retrieve all rows from just those three columns.

Let's apply this to the `teachers` table. Perhaps in your analysis you want to focus on teachers' names and salaries, not the school where they work or when they were hired. In that case, you would select just the relevant columns. Enter the statement shown in Listing 3-2. Notice that the order of the columns in the query is different than the order in the table: you're able to retrieve columns in any order you'd like.

```
SELECT last_name, first_name, salary FROM teachers;
```

Listing 3-2: Querying a subset of columns

Now, in the result set, you've limited the columns to three:

```
last_name    first_name    salary
---------    ----------    ------
Smith        Janet         36200
Reynolds     Lee           65000
Cole         Samuel        43500
Bush         Samantha      36200
Diaz         Betty         43500
Roush        Kathleen      38500
```

Although these examples are basic, they illustrate a good strategy for beginning your interview of a dataset. Generally, it's wise to start your analysis by checking whether your data is present and in the format you

expect, which is a task well suited to SELECT. Are dates in a proper format complete with month, date, and year, or are they entered (as I once ruefully observed) as text with the month and year only? Does every row have values in all the columns? Are there mysteriously no last names starting with letters beyond *M*? All these issues indicate potential hazards ranging from missing data to shoddy record keeping somewhere in the workflow.

We're only working with a table of six rows, but when you're facing a table of thousands or even millions of rows, it's essential to get a quick read on your data quality and the range of values it contains. To do this, let's dig deeper and add several SQL keywords.

NOTE *pgAdmin allows you to drag and drop column names, table names, and other objects from the object browser into the Query Tool. This can be helpful if you're writing a new query and don't want to keep typing lengthy object names. Expand the object tree to find your tables or columns, as you did in Chapter 1, and click and drag them into the Query Tool.*

Sorting Data with ORDER BY

Data can make more sense, and may reveal patterns more readily, when it's arranged in order rather than jumbled randomly.

In SQL, we order the results of a query using a clause containing the keywords ORDER BY followed by the name of the column or columns to sort. Applying this clause doesn't change the original table, only the result of the query. Listing 3-3 shows an example using the teachers table:

```
SELECT first_name, last_name, salary
FROM teachers
ORDER BY salary DESC;
```

Listing 3-3: Sorting a column with ORDER BY

By default, ORDER BY sorts values in ascending order, but here I sort in descending order by adding the DESC keyword. (The optional ASC keyword specifies sorting in ascending order.) Now, by ordering the salary column from highest to lowest, I can determine which teachers earn the most:

```
first_name    last_name    salary
----------    ---------    ------
Lee           Reynolds     65000
Samuel        Cole         43500
Betty         Diaz         43500
Kathleen      Roush        38500
Janet         Smith        36200
Samantha      Bush         36200
```

The ORDER BY clause also accepts numbers instead of column names, with the number identifying the sort column according to its position in

the SELECT clause. Thus, you could rewrite Listing 3-3 this way, using 3 to refer to the third column in the SELECT clause, salary:

```
SELECT first_name, last_name, salary
FROM teachers
ORDER BY 3 DESC;
```

The ability to sort in our queries gives us great flexibility in how we view and present data. For example, we're not limited to sorting on just one column. Enter the statement in Listing 3-4:

```
SELECT last_name, school, hire_date
FROM teachers
❶ ORDER BY school ASC, hire_date DESC;
```

Listing 3-4: Sorting multiple columns with ORDER BY

In this case, we're retrieving the last names of teachers, their school, and the date they were hired. By sorting the school column in ascending order and hire_date in descending order ❶, we create a listing of teachers grouped by school with the most recently hired teachers listed first. This shows us who the newest teachers are at each school. The result set should look like this:

```
last_name    school               hire_date
---------    ------------------   ----------
Smith        F.D. Roosevelt HS    2011-10-30
Roush        F.D. Roosevelt HS    2010-10-22
Reynolds     F.D. Roosevelt HS    1993-05-22
Bush         Myers Middle School  2011-10-30
Diaz         Myers Middle School  2005-08-30
Cole         Myers Middle School  2005-08-01
```

You can use ORDER BY on more than two columns, but you'll soon reach a point of diminishing returns where the effect will be hardly noticeable. Imagine if you added columns about teachers' highest college degree attained, the grade level taught, and birthdate to the ORDER BY clause. It would be difficult to understand the various sort directions in the output all at once, much less communicate that to others. Digesting data happens most easily when the result focuses on answering a specific question; therefore, a better strategy is to limit the number of columns in your query to only the most important and then run several queries to answer each question you have.

Using DISTINCT to Find Unique Values

In a table, it's not unusual for a column to contain rows with duplicate values. In the teachers table, for example, the school column lists the same school names multiple times because each school employs many teachers.

To understand the range of values in a column, we can use the DISTINCT keyword as part of a query that eliminates duplicates and shows only unique values. Use DISTINCT immediately after SELECT, as shown in Listing 3-5:

```
SELECT DISTINCT school
FROM teachers
ORDER BY school;
```

Listing 3-5: Querying distinct values in the school column

The result is as follows:

```
school
-------------------
F.D. Roosevelt HS
Myers Middle School
```

Even though six rows are in the table, the output shows just the two unique school names in the school column. This is a helpful first step toward assessing data quality. For example, if a school name is spelled more than one way, those spelling variations will be easy to spot and correct, especially if you sort the output.

When you're working with dates or numbers, DISTINCT will help highlight inconsistent or broken formatting. For example, you might inherit a dataset in which dates were entered in a column formatted with a text data type. That practice (which you should avoid) allows malformed dates to exist:

```
date
---------
5/30/2023
6//2023
6/1/2023
6/2/2023
```

The DISTINCT keyword also works on more than one column at a time. If we add a column, the query returns each unique pair of values. Run the code in Listing 3-6:

```
SELECT DISTINCT school, salary
FROM teachers
ORDER BY school, salary;
```

Listing 3-6: Querying distinct pairs of values in the school and salary columns

Now the query returns each unique (or distinct) salary earned at each school. Because two teachers at Myers Middle School earn $43,500, that pair is listed in just one row, and the query returns five rows rather than all six in the table:

```
school                salary
-------------------   ------
F.D. Roosevelt HS     36200
F.D. Roosevelt HS     38500
F.D. Roosevelt HS     65000
Myers Middle School   36200
Myers Middle School   43500
```

This technique gives us the ability to ask, "For each *x* in the table, what are all the *y* values?" For each factory, what are all the chemicals it produces? For each election district, who are all the candidates running for office? For each concert hall, who are the artists playing this month?

SQL offers more sophisticated techniques with aggregate functions that let us count, sum, and find minimum and maximum values. I'll cover those in detail in Chapter 6 and Chapter 9.

Filtering Rows with WHERE

Sometimes, you'll want to limit the rows a query returns to only those in which one or more columns meet certain criteria. Using `teachers` as an example, you might want to find all teachers hired before a particular year or all teachers making more than $75,000 at elementary schools. For these tasks, we use the `WHERE` clause.

The `WHERE` clause allows you to find rows that match a specific value, a range of values, or multiple values based on criteria supplied via an *operator*—a keyword that let us perform math, comparison, and logical operations. You also can use criteria to exclude rows.

Listing 3-7 shows a basic example. Note that in standard SQL syntax, the `WHERE` clause follows the `FROM` keyword and the name of the table or tables being queried:

```
SELECT last_name, school, hire_date
FROM teachers
WHERE school = 'Myers Middle School';
```

Listing 3-7: Filtering rows using `WHERE`

The result set shows just the teachers assigned to Myers Middle School:

```
last_name    school                hire_date
---------    -------------------   ----------
Cole         Myers Middle School   2005-08-01
Bush         Myers Middle School   2011-10-30
Diaz         Myers Middle School   2005-08-30
```

Here, I'm using the equals comparison operator to find rows that exactly match a value, but of course you can use other operators with `WHERE` to customize your filter criteria. Table 3-1 summarizes the most commonly used comparison operators. Depending on your database system, many more might be available.

Table 3-1: Comparison and Matching Operators in PostgreSQL

Operator	Function	Example
=	Equal to	`WHERE school = 'Baker Middle'`
<> or !=	Not equal to*	`WHERE school <> 'Baker Middle'`
>	Greater than	`WHERE salary > 20000`
<	Less than	`WHERE salary < 60500`
>=	Greater than or equal to	`WHERE salary >= 20000`
<=	Less than or equal to	`WHERE salary <= 60500`
BETWEEN	Within a range	`WHERE salary BETWEEN 20000 AND 40000`
IN	Match one of a set of values	`WHERE last_name IN ('Bush', 'Roush')`
LIKE	Match a pattern (case sensitive)	`WHERE first_name LIKE 'Sam%'`
ILIKE	Match a pattern (case insensitive)	`WHERE first_name ILIKE 'sam%'`
NOT	Negates a condition	`WHERE first_name NOT ILIKE 'sam%'`

* The != operator is not part of standard ANSI SQL but is available in PostgreSQL and several other database systems.

The following examples show comparison operators in action. First, we use the equal operator to find teachers whose first name is Janet:

```
SELECT first_name, last_name, school
FROM teachers
WHERE first_name = 'Janet';
```

Next, we list all school names in the table but exclude F.D. Roosevelt HS using the not equal operator:

```
SELECT school
FROM teachers
WHERE school <> 'F.D. Roosevelt HS';
```

Here we use the less than operator to list teachers hired before January 1, 2000 (using the date format *YYYY-MM-DD*):

```
SELECT first_name, last_name, hire_date
FROM teachers
WHERE hire_date < '2000-01-01';
```

Then we find teachers who earn $43,500 or more using the >= operator:

```
SELECT first_name, last_name, salary
FROM teachers
WHERE salary >= 43500;
```

The next query uses the `BETWEEN` operator to find teachers who earn from $40,000 to $65,000. Note that `BETWEEN` is *inclusive*, meaning the result will include values matching the start and end ranges specified.

```
SELECT first_name, last_name, school, salary
FROM teachers
WHERE salary BETWEEN 40000 AND 65000;
```

Use caution with BETWEEN, because its inclusive nature can lead to inadvertent double-counting of values. For example, if you filter for values with BETWEEN 10 AND 20 and run a second query using BETWEEN 20 AND 30, a row with the value of 20 will appear in both query results. You can avoid this by using the more explicit greater than and less than operators to define ranges. For example, this query returns the same result as the previous one but more obviously specifies the range:

```
SELECT first_name, last_name, school, salary
FROM teachers
WHERE salary >= 40000 AND salary <= 65000;
```

We'll return to these operators throughout the book, because they'll play a key role in helping us ferret out the data and answers we want to find.

Using LIKE and ILIKE with WHERE:

Comparison operators are fairly straightforward, but the matching operators LIKE and ILIKE deserve additional explanation. Both let you find a variety of values that include characters matching a specified pattern, which is handy if you don't know exactly what you're searching for or if you're rooting out misspelled words. To use LIKE and ILIKE, you specify a pattern to match using one or both of these symbols:

Percent sign (%) A wildcard matching one or more characters

Underscore (_) A wildcard matching just one character

For example, if you're trying to find the word baker, the following LIKE patterns will match it:

```
LIKE 'b%'
LIKE '%ak%'
LIKE '_aker'
LIKE 'ba_er'
```

The difference? The LIKE operator, which is part of the ANSI SQL standard, is case sensitive. The ILIKE operator, which is a PostgreSQL-only implementation, is case insensitive. Listing 3-8 shows how the two keywords give you different results. The first WHERE clause uses LIKE ❶ to find names that start with the characters sam, and because it's case sensitive, it will return zero results. The second, using the case-insensitive ILIKE ❷, will return Samuel and Samantha from the table:

```
SELECT first_name
FROM teachers
❶ WHERE first_name LIKE 'sam%';
```

```
SELECT first_name
FROM teachers
❷ WHERE first_name ILIKE 'sam%';
```

Listing 3-8: Filtering with `LIKE` and `ILIKE`

Over the years, I've gravitated toward using `ILIKE` and wildcard operators to make sure I'm not inadvertently excluding results from searches, particularly when vetting data. I don't assume that whoever typed the names of people, places, products, or other proper nouns always remembered to capitalize them. And if one of the goals of interviewing data is to understand its quality, using a case-insensitive search will help you find variations.

Because `LIKE` and `ILIKE` search for patterns, performance on large databases can be slow. We can improve performance using indexes, which I'll cover in "Speeding Up Queries with Indexes" in Chapter 8.

Combining Operators with AND and OR

Comparison operators become even more useful when we combine them. To do this, we connect them using the logical operators `AND` and `OR` along with, if needed, parentheses.

The statements in Listing 3-9 show three examples that combine operators this way:

```
SELECT *
FROM teachers
❶ WHERE school = 'Myers Middle School'
      AND salary < 40000;

SELECT *
FROM teachers
❷ WHERE last_name = 'Cole'
      OR last_name = 'Bush';

SELECT *
FROM teachers
❸ WHERE school = 'F.D. Roosevelt HS'
      AND (salary < 38000 OR salary > 40000);
```

Listing 3-9: Combining operators using `AND` and `OR`

The first query uses `AND` in the `WHERE` clause ❶ to find teachers who work at Myers Middle School and have a salary less than $40,000. Because we connect the two conditions using `AND`, both must be true for a row to meet the criteria in the `WHERE` clause and be returned in the query results.

The second example uses `OR` ❷ to search for any teacher whose last name matches Cole or Bush. When we connect conditions using `OR`, only one of the conditions must be true for a row to meet the criteria of the `WHERE` clause.

The final example looks for teachers at Roosevelt whose salaries are either less than $38,000 or greater than $40,000 ❸. When we place statements inside parentheses, those are evaluated as a group before being combined with other criteria. In this case, the school name must be

exactly `F.D. Roosevelt HS`, and the salary must be either less or higher than specified for a row to meet the criteria of the `WHERE` clause.

If we use both `AND` with `OR` in a clause but don't use any parentheses, the database will evaluate the `AND` condition first and then the `OR` condition. In the final example, that means we'd see a different result if we omitted parentheses—the database would look for rows where the school name is `F.D. Roosevelt HS` and the salary is less than $38,000 or rows for any school where the salary is more than $40,000. Give it a try in the Query Tool to see.

Putting It All Together

You can begin to see how even the previous simple queries allow us to delve into our data with flexibility and precision to find what we're looking for. You can combine comparison operator statements using the `AND` and `OR` keywords to provide multiple criteria for filtering, and you can include an `ORDER BY` clause to rank the results.

With the preceding information in mind, let's combine the concepts in this chapter into one statement to show how they fit together. SQL is particular about the order of keywords, so follow this convention:

```
SELECT column_names
FROM table_name
WHERE criteria
ORDER BY column_names;
```

Listing 3-10 shows a query against the `teachers` table that includes all the aforementioned pieces:

```
SELECT first_name, last_name, school, hire_date, salary
FROM teachers
WHERE school LIKE '%Roos%'
ORDER BY hire_date DESC;
```

Listing 3-10: A `SELECT` statement including `WHERE` and `ORDER BY`

This listing returns teachers at Roosevelt High School, ordered from newest hire to earliest. We can see a clear correlation between a teacher's hire date at the school and their current salary level:

```
first_name  last_name  school           hire_date  salary
----------  ---------  ---------------  ----------  ------
Janet       Smith      F.D. Roosevelt HS  2011-10-30  36200
Kathleen    Roush      F.D. Roosevelt HS  2010-10-22  38500
Lee         Reynolds   F.D. Roosevelt HS  1993-05-22  65000
```

Wrapping Up

Now that you've learned the basic structure of a few different SQL queries, you've acquired the foundation for many of the additional skills I'll cover in

later chapters. Sorting, filtering, and choosing only the most important columns from a table can yield a surprising amount of information from your data and help you find the story it tells.

In the next chapter, you'll learn about another foundational aspect of SQL: data types.

TRY IT YOURSELF

Explore basic queries with these exercises:

1. The school district superintendent asks for a list of teachers in each school. Write a query that lists the schools in alphabetical order along with teachers ordered by last name A–Z.
2. Write a query that finds the one teacher whose first name starts with the letter *S* and who earns more than $40,000.
3. Rank teachers hired since January 1, 2010, ordered by highest paid to lowest.