SQL for Data Analytics

Early Access Publication: SQL for Data Analytics

Early Access Production Reference: B31925

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-83664-625-9

www.packt.com

Table of Contents

SQL for Data Analytics, Fourth Edition: Analyze data effectively, uncover insights and master advanced SQL for real-world applications

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1 Introduction to Data Management Systems

Join our book community on Discord


https://packt.link/EarlyAccessCommunity

Data collection and analysis is an old practice dating back to the beginning of civilization. Records from ancient Egyptian papyrus suggest that pharaohs collected census information from villages, possibly to determine the number of soldiers that could be enlisted for war. Local authorities would collect the census and send it to the pharaoh's court, where the data would be analyzed to determine the possible size of the army. The same process has been repeated thousands of times in the years since then. However, it was after the arrival of modern computers that the art of data analytics became a significant phenomenon that is changing people's lives every day.This book, as its name suggests, teaches you how to use **Structured Query Language** (**SQL**) for data analytics. SQL is the language that you will be focusing on in the rest of the book. But before diving into SQL, this chapter will provide an overview of data management. You will be introduced to fundamental concepts of a typical data management system, which will lay the foundation for the concepts that future chapters will be based on, define the purpose of the SQL operations that you will learn about, and set up the infrastructure for analytics on which the SQL operations will run.The following topics are covered in this chapter:

- Describing the world with data
- Understanding relational databases and SQL
- Setting up a PostgreSQL relational database
- With these topics, you will be able to analyze and interpret real-world objects using data effectively. You will also set up a PostgreSQL database on your machine to store, organize, and query data.

Technical requirements

In order to complete the exercises in this chapter, you need to have a computer that has internet connectivity, and you need to have the privilege to install software on the computer.

Describing the world with data

Start with a simple question: What is data? Data is the recorded description or measurements of something in the real world. A real-world object or event is a **unit of observation**. For example, an exam participation of one student is a unit of observation. Data is used to describe **observations**. In the case of these grades, a list of exam grades is data; that is, exam grades are a measure of the performance of students in exams.As you can imagine, there is a lot of data you can gather to describe a person. For a student, that can include their age, height, courses registered, exam grades, and more. One or more of these measurements used to describe a specific unit of observation is called a **data point**, and each measurement in a data point is called a **variable** (also referred to as a **feature**). When you have several data points together, you have a **dataset**. For example, you may have Student A, who is a 19-year-old male, and Student B, who is a 22-year-old female. Here, age is a variable. The age of student

A is one measurement, and that of student B is another. 19 and 22 are the values of measurement. A compilation of data points with measurements such as the ages, heights, and exam grades of various people is called a dataset.For a given object of observation, there can be many measurements. It is important to note that data is collected for analytics as well as functional purposes. Not all measurements are relevant to the questions that you want to ask. For example, height, weight, and blood pressure measurements are very important if you are analyzing the students' health conditions, but may not be relevant if you would like to evaluate their academic performances. In the latter case, exam grades are more relevant. Identifying relevant measurements is an important step to start the analytical process. You will do this in the process of data modeling.

Data modeling

Data modeling is the process of identifying the content of relevant data and its relationships. It helps in designing databases and structuring data in a way that supports business processes, ensures data integrity, and improves efficiency. Data modeling usually consists of three steps: conceptual, logical, and physical. It starts with the conceptual definition of concepts, and each following step is built on top of the step before.

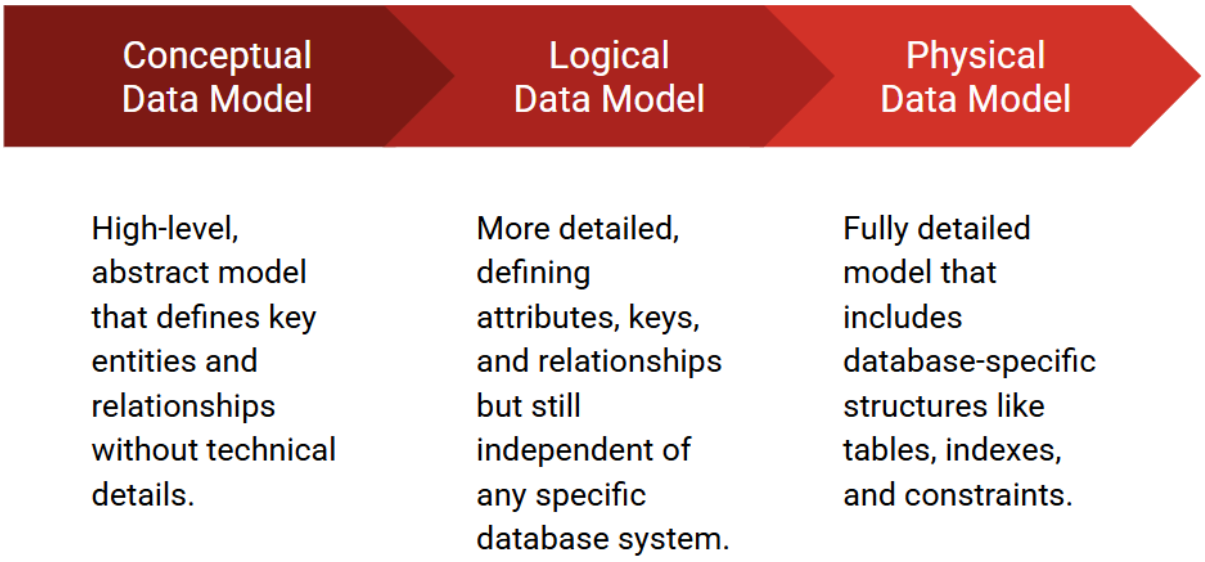

Figure 1.1: Data modeling process

The first step of the general modeling process is the **conceptual data model**. A conceptual data model is a high-level abstract of real-world concepts and scenarios. It defines what data should be considered in the system, without considering the implementation details. For example, when you study the data model for an e-commerce system, you would first identify independent parties such as customers and products. Then you would identify the relationship between parties (e.g., customers purchase products). When you dive deeper into the operational process of the e-commerce system, you will see that the process requires a customer to place an order, which includes one or more product items. This study drives the general concept of product further into quantifiable value of purchased item counts and introduces conceptual parties such as order and order date. Conceptual data models provide a simplified, high-level representation of all the concepts in the system and help users understand, communicate, and collaborate on complex ideas or systems by offering a clear and shared understanding of its fundamental principles and functionality.The most popular conceptual data model is the **entity-relationship model**, or **ER model**. In an ER model, real-world objects are defined as **entities** (such as a customer or product) and **relationships** between different entities, usually with the help of a diagram (**entity-relationship diagram**, or **ERD**). The end product is a data model containing a set of entities, interconnected by relationships.The following diagram shows such an entity-relational diagram, which depicts an e-commerce system. In this model, the real-world scenario of e-commerce sales is broken down into multiple entities such as **Customer**, **Salespeople**, and **Product**. These entities meet each other at the **Sales** event:

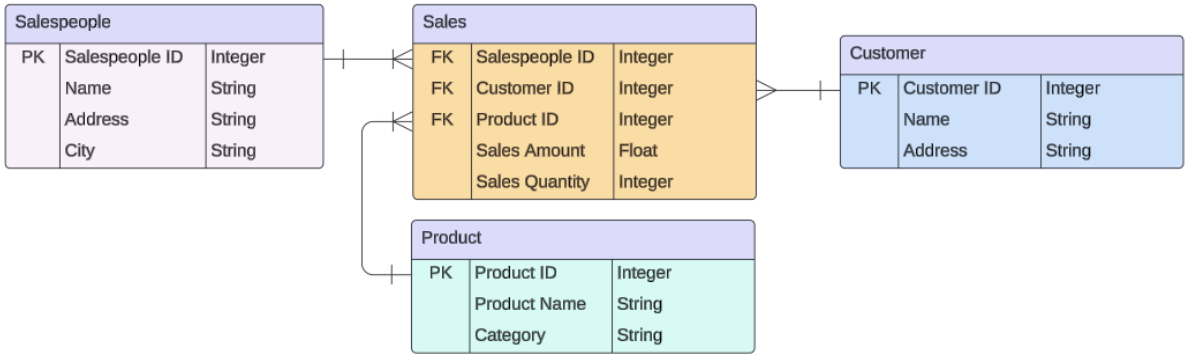

Figure 1.2: An example entity-relationship diagram (ERD)

When discussing the relationship between entities, one important aspect is the degree to which each entity relates to others. This is called cardinality (i.e., the number of entities that can be associated with each other within a relationship). A customer can place many orders, but each order is only associated with one customer. This type of cardinality, in which one entity (customer and product) can be mapped to many other entities (order), is called one-to-many cardinality, or a 1:M relationship. If an entity can only have one mapping for another entity, this is called one-to-one cardinality, or a 1:1 relationship. For example, in the United States, each state has only one capital city, and each capital city can only be the capital of one state. This is a 1:1 relationship.The third and last cardinality type is many-to-many, also called an M:M relationship. It occurs when each of two entities can be mapped to multiple instances of the other. For example, each movie has multiple actors/actresses. Most actors/actresses play in multiple movies. Note that in an M:M relationship, it is possible that some individual entity relationships are 1:M. For example, a particular actor may only play in one movie. But as long as there is an instance where the relationship is M:M, the whole relationship's cardinality is M:M.There are many ways of representing cardinality in an ER diagram. One of the most popular ways is **Crow's Foot Notation**. In this notation, the 1 side of the relationship is represented by a short vertical line, as depicted in the customer ID on the customer entity side in *Figure 1.2*. The M side of the relationship is represented by a three-pronged fork, resembling a crow's foot (hence the name), as depicted in the customer ID on the sales side in *Figure 1.2*. For M:M relationships, both sides will be depicted as the crow's foot. After you define what entities/relationships are needed and how they are related to each other in the data modeling process, you convert your conceptual model into a **logical data model**. A logical data model is a mid-level model that defines how the data will be structured mathematically and logically, typically using collections, tables, and attributes. Logical models are technology-independent. They follow the rules of a particular data organization (relational, document-oriented, etc.). For example, a relational logical model converted from the preceding conceptual model would have tables for customers, salespersons, products, and sales, with attributes such as **Customer_ID**, **Product_ID**, and **Sales_Transaction_Date**.A logical data model is about data organization. The data inside the model can be presented in many different forms. One of the most natural ways of recording data is to use the **JavaScript Object Notation** (**JSON**) document format. JSON represents data as key-value pairs. Keys are strings, and values can be strings, numbers, booleans, objects, or null. Some examples of key-value pairs include the following:

```
    "First Name": "John",
    "Last Name": "Smith",
```

When you have a collection of key-value pairs describing one single entity/relationship, you have a **document**. Each document is enclosed with a pair of curly braces `{}`. Key-value pairs within one document are separated by commas. For example, the following JSON document describes the relevant information of a student:

```
{
    "First Name": "John",
    "Last Name": "Smith",
    "Age": 27,
    "ID": 1
}
```

After data is defined, it must be collected and recorded. It must be properly received, stored within a location, retrieved as requested by the user, and sent out in the proper format. This must be done at the physical level and thus requires a **physical data model**.A physical data model is a low-level, technology-specific model. It defines how the data will be stored physically in a computer system. It includes table structures, indexes, partitions, storage details, access paths, and so on, and is optimized for performance by considering hardware, file organization, and storage methods. For example, a physical model built on top of the preceding logical model would specify data types (text, integer), indexes, tablespaces, and storage engine settings for each document collection.At the center of the physical model is the **database**. A database is an organized collection of data that is stored and managed electronically. It allows users to efficiently store, retrieve, update, and delete information. Database operations are usually handled by specific software, which is called the **database management system** (**DBMS**). If the DBMS handles the operations of JSON document databases, it is called a document-oriented database. There are many different types of DBMS, but the most important type is the relational DBMS, or RDBMS.A detailed explanation and implementation of data modeling is beyond the scope of this book. For beginner data analysts and data engineers, the key takeaway is to understand that before the collection and processing of data, efforts must be made to analyze the business questions and identify what data can/should be used. Equally important is how datasets are related to each other. For example, courses are taught by teachers and offered to students, sales orders are placed by customers for product items, and doctors' appointments are tightly associated with patients, insurance, and specific health conditions. All these are defined during the data modeling process, and the data analysts and data engineers will build their data analytical tasks on the data model that is the result of the modeling process.Now that you have learned about the general data modeling process, in the next section, you will dive deeper into a specific data model, the relational model.

Understanding relational databases and SQL

Now that you have learned how to describe a real-world entity/relationship using a document comprised of key-value pairs, it is time to consider the downside of it (i.e., the inherited complexity of this modeling approach).First of all, key-value pairs within a document do not require any order. For example, the **First Name** key-value pair can show up either before or after the **Last Name** key-value pair. Furthermore, documents describing the same type of observation units (such as customers) can have different keys in different documents. A customer may have key-value pairs such as pets or children, or both, but also may not have either. These two characteristics, the lack of order and the flexibility of definition, reflect the real-world scenario and provide flexibility to the usage. The downside of them is that the applications utilizing these databases must carry the burden of checking key existence and searching for key-value pairs in an unsorted list. This can increase code complexity and degrade performance.In most real-world scenarios, documents in a dataset do share the same set of keys, and the key-value pairs will benefit from a defined order. For this type of dataset, it is possible to apply a modeling technique called the **relational data model** to make data organization simpler.The relational data model, invented by Dr. Edgar F. Codd in 1970, organizes data as relations, or sets of **tuples**. Tuple is the mathematical term for a series of attributes grouped together in a particular order. A more common (and more practical) name for a tuple is **record**. Each record consists of the same series of attributes that generally describe the record. For instance, a fast-moving consumer goods company wants to track its customers and has the following documents:

```
[
  {
    “First Name”: “John”,
    “Last Name”: “Smith”,
    “Age”: 27,
    “ID”: 1
  },
  {
    “ID”: 2,
    “Age”: 53,
    “First Name”: “Mary”,
    “Last Name”: “Higgins”
  }
]
```

The attributes in each record include information such as the customer's last name, first name, age, date of signup, delivery address, and others. When converting this document array into a relational data model, you can save customer information in a relation called **customers**. Each record in this relation contains details about one

customer. While the key-value pairs can be of any order in the JSON document, all these attributes in a relational model are arranged in a predefined order. This relation and its first two records will look like this:

ID	Last Name	First Name	Age	…
1	Smith	John	27	…
2	Higgins	Mary	53	…

Table 1.1: An example relation – customersAs shown, each relation is indeed a two-dimensional table that looks like an Excel spreadsheet. This two-dimensional table consists of rows with predefined columns. A **row**, also called a tuple or a record, represents a single record in the table. Furthermore, it represents a unique instance of a real-world object or event (i.e., an entity or relationship). For example, in the **customers** table, each row represents a single customer. In the **products** table, each row represents a single product. A column represents a specific attribute or characteristic of the data, such as the first name and last name of a customer. Each column has a name (field name) and a specific data type (e.g., `VARCHAR`, `INT`, `DATE`). All rows must have values under the same set of columns, and each column holds the same type of information for all rows in the table. There cannot be duplicate rows; otherwise, you will have two rows representing the same real-world object. There cannot be duplicate columns either, as you should not have duplicate descriptions of the same attribute of the entity/relationship.Finally, when you place these relations into a database, you get a relational database. A database that utilizes the relational model of data for the bulk of its datasets is called a relational database. Relational databases, built on top of a solid mathematical foundation, can be manipulated using a tool called **relational algebra**. This tool can be further implemented as a descriptive computer language for interaction with databases. This standard language for relational database operation is the **Structured Query Language**, or **SQL**. SQL is the language utilized by users of a relational DBMS to access and interact with the database. With the popularity of relational databases, SQL is arguably one of the most utilized computer languages in the world, and definitely the number one choice for operations in the data management field.

Primary keys and foreign keys

The preceding discussion regarding uniqueness raises an interesting question: how do you identify each unique real-world instance? For example, how do you map the first row of the `customers` table to a real customer? In the real world, entity instances come with different attributes that can uniquely identify them. For a customer, this can be the social security number or any system ID, such as an insurance policy number. For a product, this can be a manufacturer's serial ID number. Such an attribute that can uniquely identify the instance is called a **primary key**. When defining your entity-relationship model, you must also define the primary key as an attribute of the entity/relationship and carry this key definition into the following logical/physical models.It is also possible that the primary key consists of multiple columns. This is common in relationships. For example, to identify a sales event, you need to identify the customer, the product, and the date/time that the sales occurred. So, the sales relationship's primary key will include the primary key of the customer, the primary key of the product, and the transaction date/time. This is called a compound key. Here, the keys of customer and product are primary keys from outside tables (`customers` and `products`). They are called foreign keys in the `sales` table. They are used to establish relationships and ensure data consistency between tables.So far, the keys you have seen are all natural keys (i.e., an existing real-world attribute such as a social security number or email address. One issue with natural keys is that they may change over time, causing referential integrity issues. A common practice in data modeling is to create a **surrogate key**. A surrogate key is an artificially generated unique identifier for a record in a table, rather than using a natural key. Surrogate keys are guaranteed to be unique and never change. Surrogate keys are usually also faster for indexing, searching, and joining tables compared to long text-based natural keys because numeric keys require less storage and perform better in queries. For these reasons, surrogate keys are widely used in data modeling. For example, the ID column in *Table 1.1* is a surrogate key that is automatically assigned to the customer and does not have business implications. Having a larger ID does not mean the customer is more important or less so. You can also create a surrogate key for sales events to simplify the data model.

Normalization

In practice, different data modelers can create different models for the same real-world usage scenario. All of them can help with the data analytics that you would like to perform. However, some models may contain redundancies,

be more prone to mistakes in operation, or have less-than-ideal performance.In practice, the most common issue for data models is that the relationships between attributes are too complex. Models with this issue can be further optimized by applying **normalization**. Normalization is the practice of reviewing the definition of existing data models and applying a series of **normal forms** to these models. These normal forms are design principles that, when applied, can reduce the complexity of the model and make the data operation simpler.There are several normal forms; the most notable ones are the **First Normal Form** (**1NF**), **Second Normal Form** (**2NF**), and **Third Normal Form** (**3NF**), each built on top of the previous. A brief explanation is as follows:

1. **First Normal Form**: Data with substructures, such as arrays, is broken into smaller pieces that are easily accessible
2. **Second Normal Form**: Data depending only on a portion of a compound key is moved into separate tables to avoid redundancy
3. **Third Normal Form**: Data that does not depend on the key is broken into separate tables to simplify the processing

The common trait, apparently, is to split overly complex tables into smaller ones. The process of applying these normal forms will generate a more optimized model.This book is a beginner's book about SQL. It teaches the usage of SQL statements and functions, as well as their application in the data analytical process. As such, it will not dive deep into data modeling theories and practices. However, it is very important for data analysts and engineers to be aware of the preceding concepts, including the entity-relationship model and the normalization process, and their importance in setting up the framework in which later analytics take place. Understanding these concepts will help you create more efficient work.

Advantages and disadvantages of SQL databases

Since the invention of the relational model, SQL has become an industry standard for data management in nearly all computer applications—and for good reasons. SQL databases provide a range of advantages that make them the de facto choice for many applications:

- **Intuitive**: Relations represented as tables serve as a common data structure that almost everyone understands. As such, working with and reasoning about relational databases is much easier than doing so with other models, such as graph models.
- **Efficient**: Relational databases allow the representation of data without unnecessarily repeating it with the help of normalization. As such, relational databases can represent large amounts of information while utilizing less space. This reduced storage footprint also allows the database to reduce operation costs, making well-designed relational databases quick to process.
- **Declarative**: SQL is a declarative language, meaning that when you write code, you only need to tell the computer what data you want, and the database takes care of determining how to execute the SQL code. You never have to worry about telling the computer how to access and pull data from the table.
- **Robust**: Most popular SQL databases have a property known as **atomicity**, **consistency**, **isolation**, and **durability** (**ACID**) compliance, which guarantees the validity of the data, even if the hardware fails.

That said, there are still some downsides to SQL databases, which are as follows:

- **Relatively lower specificity**: While SQL is declarative, its functionality can often be limited to what has already been programmed into it. Although most popular relational database software is updated constantly with new functionality being built all the time, it can be difficult to process and work with data structures and algorithms that are not programmed into a relational database.
- **Limited scalability**: SQL databases are incredibly robust, but this robustness comes at a cost. As the amount of information you have doubles, the cost of resources increases even more than double. When very large volumes of information are involved, other data stores such as NoSQL databases may be better.
- **Sacrificing performance for consistency**: Relational databases are generally designed for consistency, which means they will take extra steps to make sure multiple users will see the same data when they try to access/modify the data at the same time. To achieve this, relational databases implement some complex checking and data locking mechanisms into their operational logic. For usage scenarios that do not require

consistency, especially for high-performance operations such as search engines or social network sites, this is an unnecessary burden and will hurt the performance of the application.
- **Lack of semi-structured and unstructured data processing ability**: The fundamental theory that SQL is built on is relational theory, which, by definition, handles only pre-defined data. Relational databases can store and fetch semi-structured and unstructured data. But to process this data, it requires processing power and functionalities that are beyond standard SQL. Later chapters of this book will cover some examples of this type of processing.

Even with these disadvantages, relational databases are no doubt the dominant databases in the market. This book will help you understand the usage of relational databases, with the popular SQL tool. To begin this journey, you will set up a relational database on your own machine first, as detailed in the following section.

Setting up a PostgreSQL relational database

Relational DBMS (**RDBMS**) is the dominant DBMS in the current market. There are many different types of RDBMS. They can be loosely categorized into two groups, commercial and open source. Different RDBMSs differ slightly in the way they operate data and even in some minor parts of the SQL syntax. There is an **American National Standards Institute** (**ANSI**) standard for SQL, which is largely followed by all RDBMSs. But each RDBMS may also have its own interpretations and extensions of the standard.In this book, you will use one of the most popular open source RDBMSs, PostgreSQL. In this section, you will go through a series of exercises, install a copy of PostgreSQL on your local machine, launch its user interface, and import a sample database into your local PostgreSQL DBMS server. This PostgreSQL database will be used for the activities described in the rest of this book. After the installation, the PostgreSQL DBMS server will be running on the backend of your computer. Your data will be stored on the local machine's hard disk. Users will access the data by communicating with the server via a client tool. There are many popular client tools that you can choose from. PostgreSQL itself comes with two tools, a graphical user interface called **pgAdmin** (sometimes called **pgAdmin4**), and a command-line tool called **psql**. You will check out both in this exercise. For the rest of this book, you will use **psql** for SQL operations.

Exercise 1.1: Installing PostgreSQL on your local machine

In this exercise, you will install PostgreSQL on your local machine. The instructions provided here are for installing and setting up PostgreSQL 17 (the most current version) on Windows. However, the basic workflow is the same for Linux and macOS. At the end of this exercise, you will find links to instructions for installing and setting up PostgreSQL 17 on Linux and macOS.Perform the following steps to complete the exercise:

1. Visit the PostgreSQL official site at https://www.postgresql.org/ and click on **Download**. You will see the following web page:

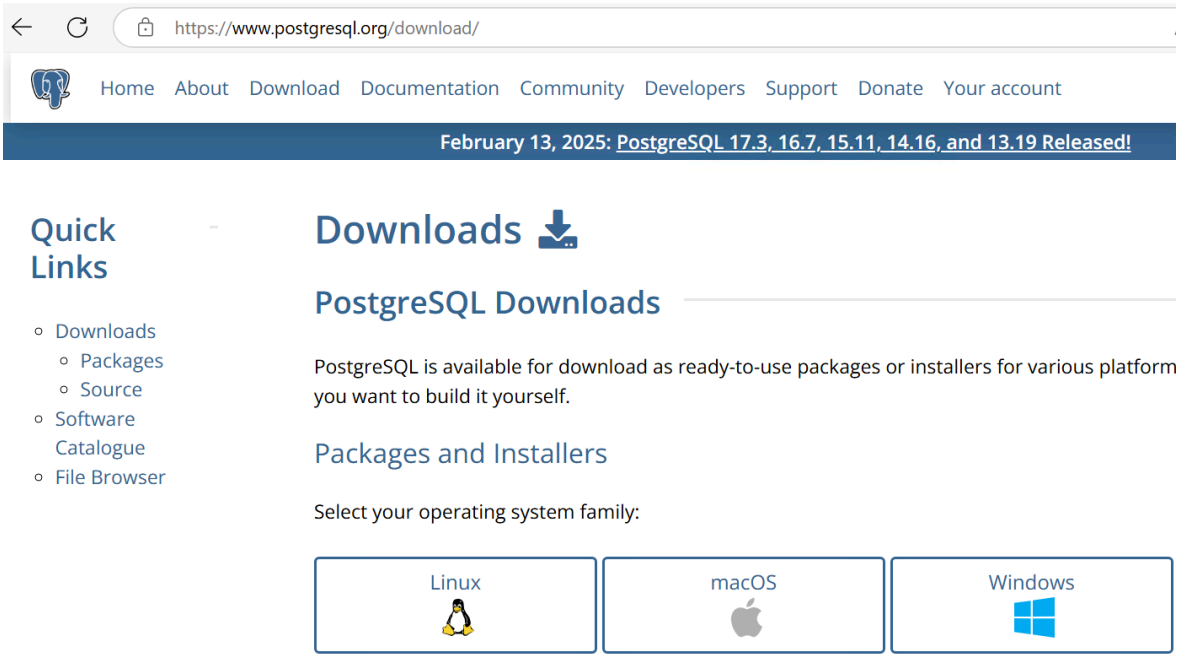

Figure 1.3: PostgreSQL download page

2. Click on the operating system that matches your local machine. Since you are using Windows for these instructions, the following page will appear:

Figure 1.4: PostgreSQL Windows installer page

3. Click on the **Download the installer** link. You will be provided with links for installation packages. For Windows, it is an EXE application. Note that the latest version that you see may be different from version 17.3, shown in *Figure 1.5*.

Figure 1.5: Current PostgreSQL version

4. Once downloaded, you can run the application to start the installation process. The installer's initial screen looks like the following:

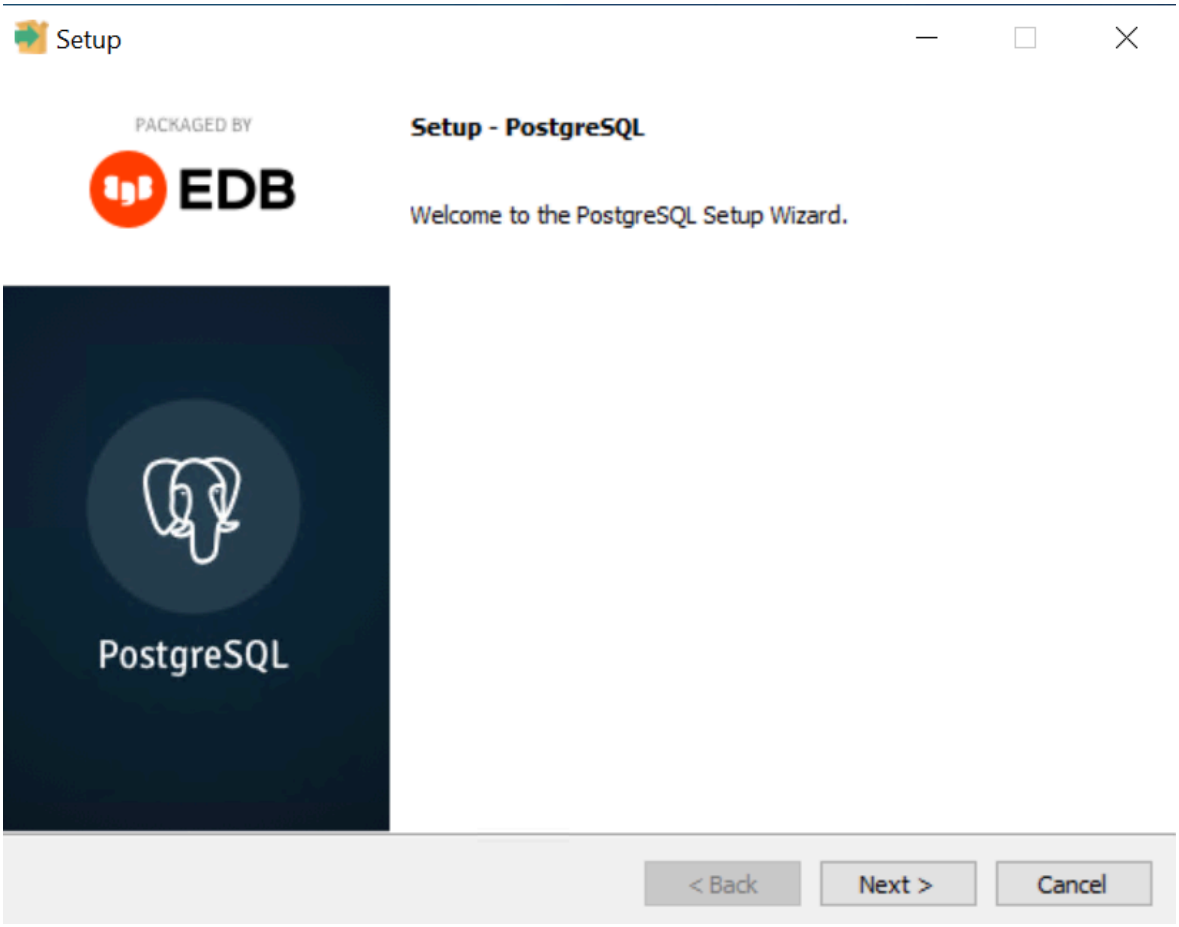

Figure 1.6: PostgreSQL installer interface

5. You need to record the installation path of PostgreSQL, as later you will need to access programs inside the PostgreSQL folder. You should choose all the default settings, except that, on the **Select Components** page, you should uncheck the **Stack Builder** component. **Stack Builder** is a graphical tool that helps you download and install additional PostgreSQL modules. It is out of the scope of this book.

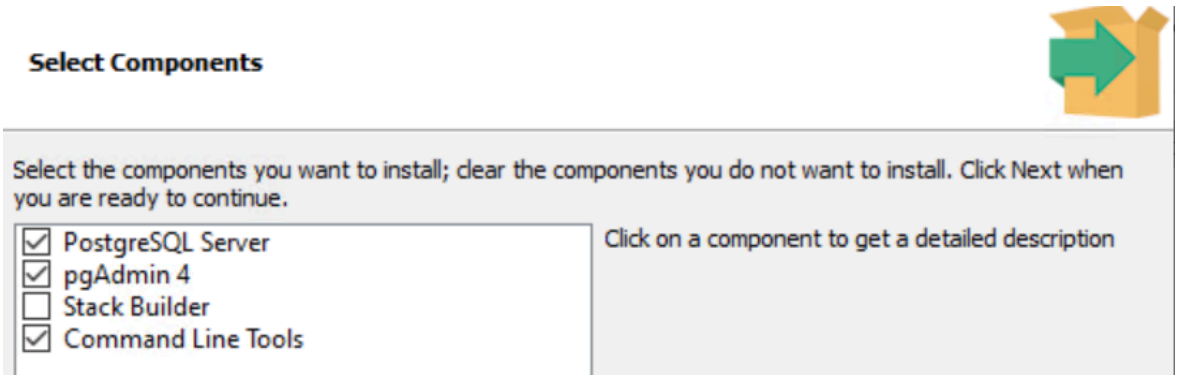

Figure 1.7: Select desired components

6. On the **Password** page, enter the superuser password. Each PostgreSQL installation comes with one superuser called **postgres**. You must set up its password in this step. You will need this password to log in to the PostgreSQL server and perform administrative work.

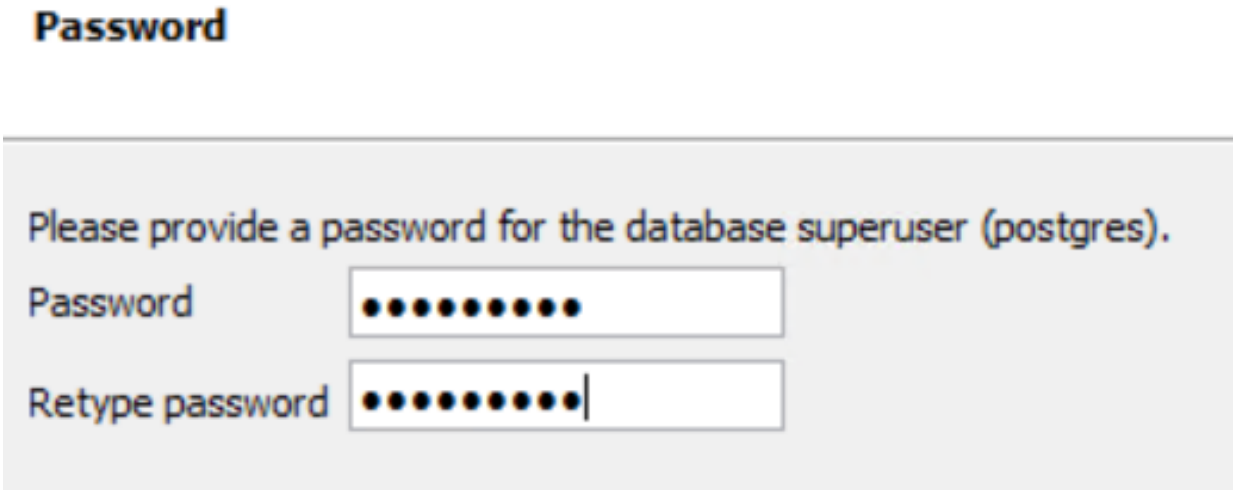

Figure 1.8: Enter superuser password

After the password page, you will be asked about a few setup options, such as the port number and locale. For all these settings, choose the default values. After all these steps are completed, PostgreSQL is successfully set up on your local machine. The PostgreSQL server now runs in the background.You have seen how to install PostgreSQL on Windows systems here. For instructions to install it on other operating systems, please refer to the PostgreSQL documentation for your OS. The PostgreSQL download page for Linux can be found at https://www.postgresql.org/download/. This page provides the links for installation on different Linux flavors. The PostgreSQL download page for macOS can be found at https://www.postgresql.org/download/macosx/.

Exercise 1.2: Accessing and understanding PostgreSQL

In the previous exercise, you installed PostgreSQL on your machine. In this exercise, you will use the client tools provided by PostgreSQL to connect to the local PostgreSQL server, look at different components in the system, and try some simple queries against the database.Perform the following steps to complete the exercise:

1. Run the **pgAdmin 4** tool provided by PostgreSQL.
2. **pgAdmin** is an open source management tool for PostgreSQL. PostgreSQL installation packages for Windows and macOS automatically install **pgAdmin** on your machine. For Linux installations, you need to check whether you need to install **pgAdmin** separately, depending on your operating system.
3. Open the **Servers** entry to access the local server in Object Explorer.

4. As **pgAdmin** opens, you will see Object Explorer on the left panel.

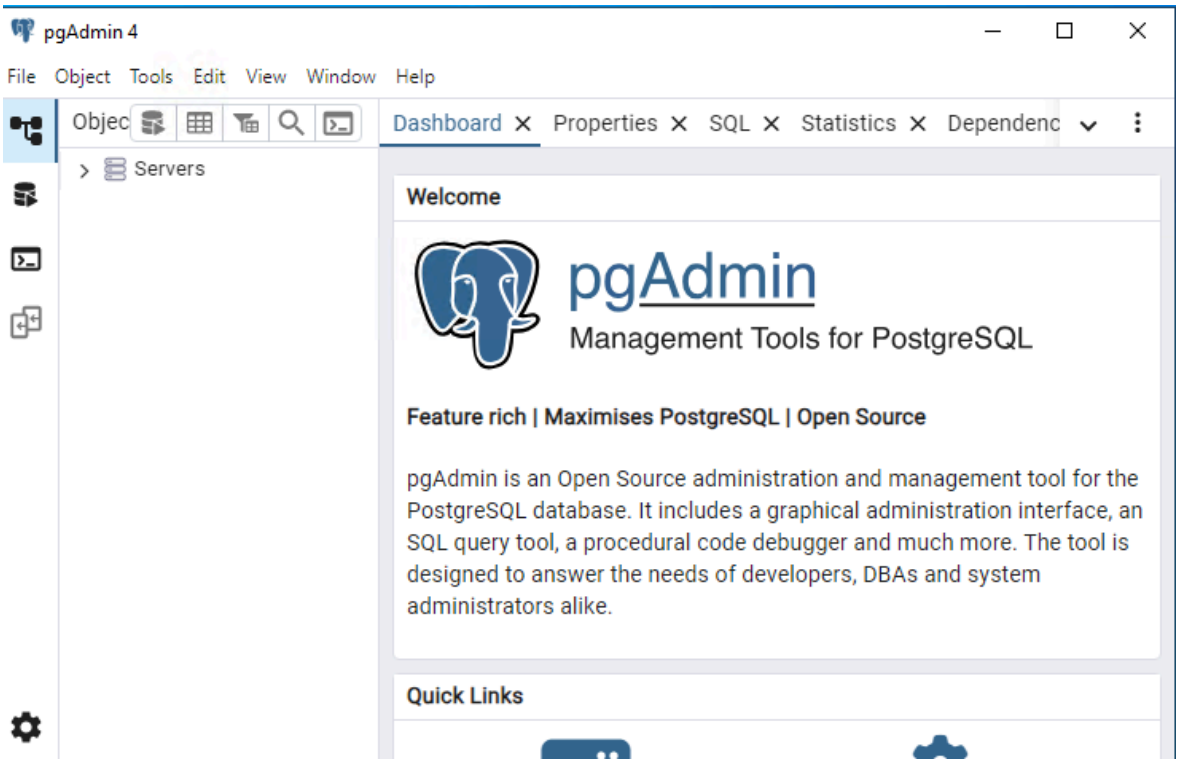

Figure 1.9: The pgAdmin interface

5. Object Explorer provides you with a consolidated view of PostgreSQL servers you have access to, as well as the object hierarchies within each server. By default, **pgAdmin** has a connection to the local server. You can also add more servers if you have their connectivity information.

6. When opening **pgAdmin**, you will be requested to enter the superuser password that you specified in *Step 6* of *Exercise 1.01*:

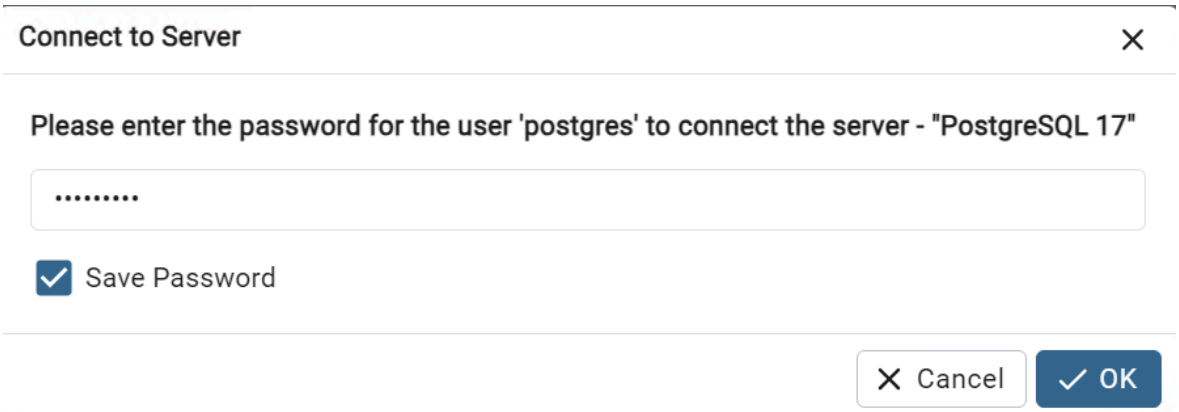

Figure 1.10: Enter superuser password to open pgAdmin

7. Inside the local server, browse through the object hierarchy.
8. As discussed, the fundamental data storage unit in a relational database is a table. In PostgreSQL, tables are collected in common collections called **schemas**, and schemas are stored inside databases. Schemas are like folders that organize tables and other objects inside databases. Each database contains one or several schemas. For example, a `products` table can be placed in the `analytics` schema, which is in the `postgres` database. Tables are usually referred to in the format `[schema].[table]`. For example, a `products` table in the

`analytics` schema would generally be referred to as `analytics.products`. There is also a special schema called `public`. This is a default schema. When you first enter the database querying interface, the database assumes the table exists in the `public` schema. For example, once you open pgAdmin, when you specify the `products` table without a schema name, the database will assume you are referring to the `public.products` table.

9. Inside Object Explorer, you can open up the object hierarchy under the local server. By default, there is only one database called `postgres`, and one schema, `public`, under the `postgres` database. When you scroll down the object hierarchy, you will also see a `Tables` collection under the `public` schema, which is empty for now.

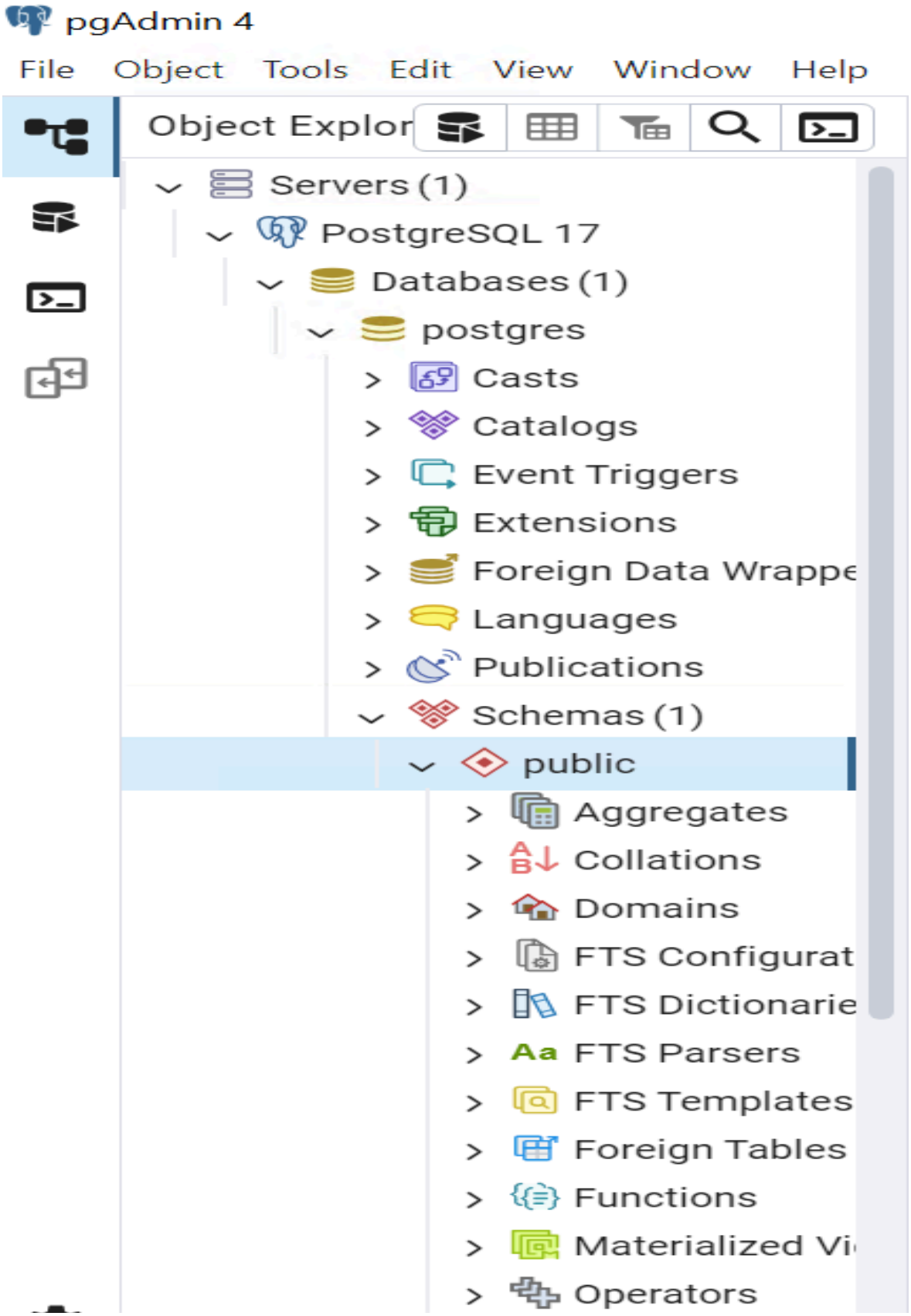
pgAdmin 4
File
Object
Tools
Edit
View
Window
Help
Object Explor
Servers (1)
PostgreSQL 17
Databases (1)
postgres
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappe
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurat
FTS Dictionarie
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Vi
Operators

Figure 1.11: PostgreSQL Object Explorer

10. In the next exercises, you will learn how to run commands against PostgreSQL databases and how to import a database into the local server, which comes with a pre-defined schema and tables.

Exercise 1.3: Utilizing PostgreSQL query tools

In this exercise, you will learn how to run a simple SQL query via pgAdmin and psql.Perform the following steps to complete the exercise:

1. Inside pgAdmin, expand the `postgres` database and then expand the `public` shema. Right-click on the `Tables` entry of the pgAdmin Object Explorer and choose **Query Tool**.

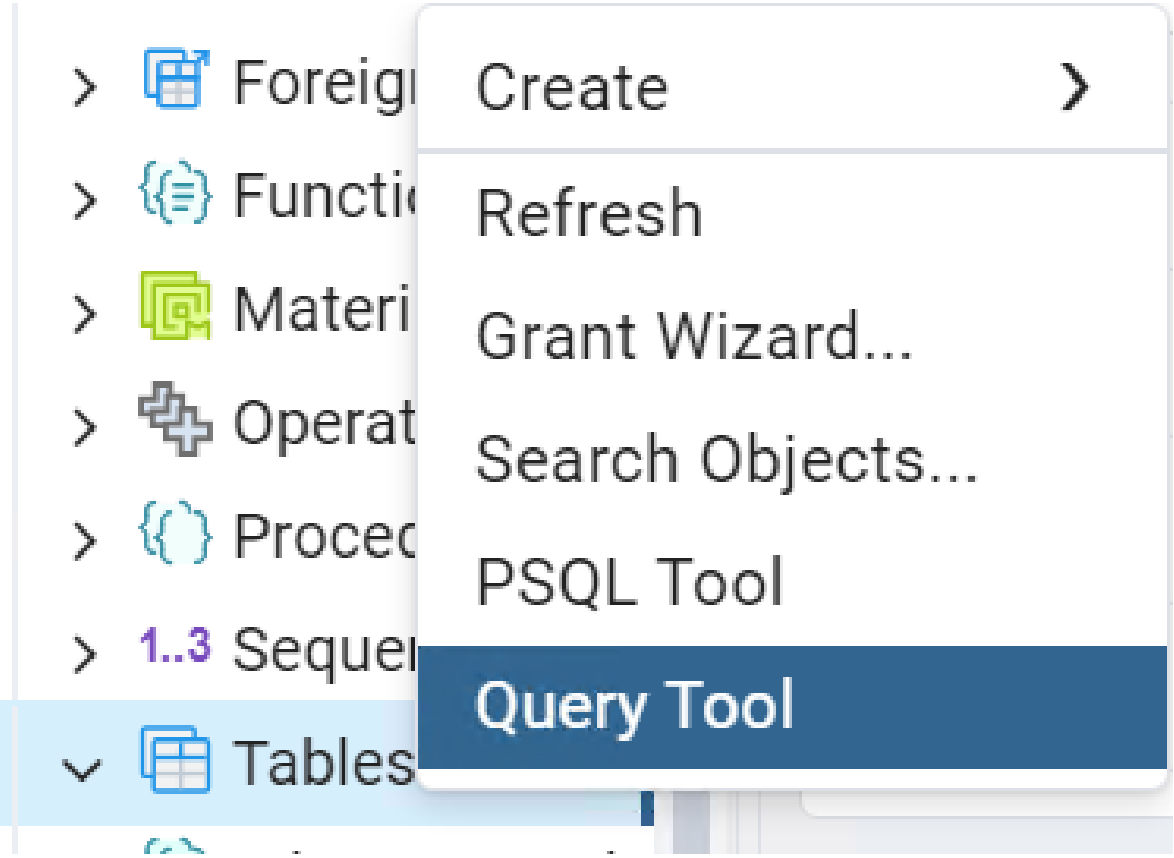

Figure 1.12: Open SQL query window in pgAdmin

2. You will be brought to the query interface of pgAdmin. You can run SQL in the query window of pgAdmin. For example, you can type the following SQL statement in the query editor and click the **Execute Script** button to run the statement:

```
select current_timestamp;
```

This statement returns the current time:

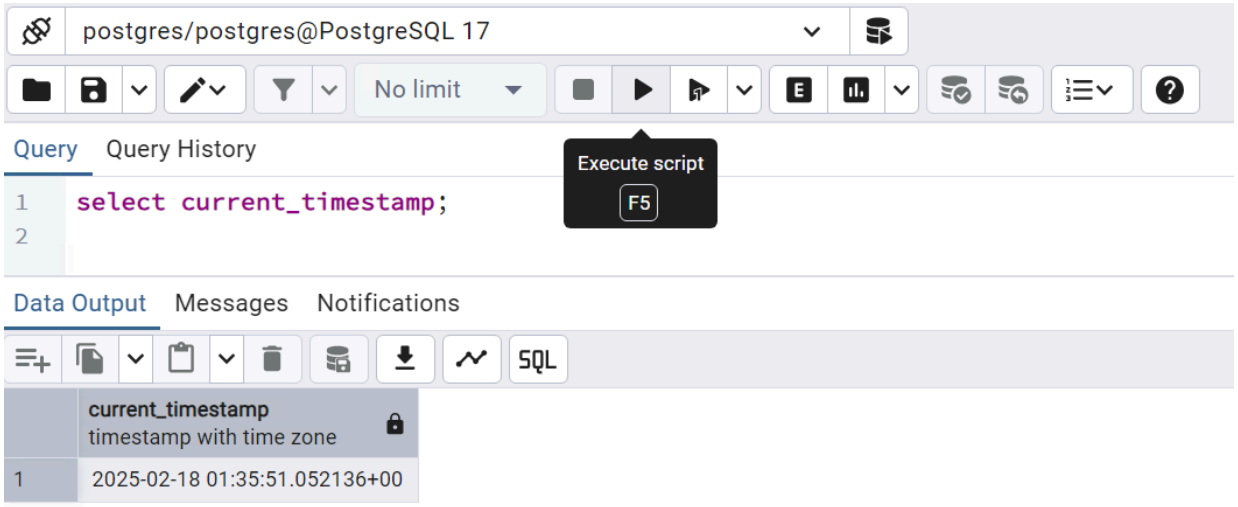

Figure 1.13: Selecting the current timestamp from PostgreSQL

3. Run the psql tool provided by PostgreSQL.

psql is the interactive command-line tool provided by PostgreSQL for interacting with a PostgreSQL database. It allows users to connect to an existing server, execute SQL queries, manage database objects, and perform administrative tasks. By default, PostgreSQL installation packages will install psql in your local environment regardless of what operating system you use.For Windows and macOS systems, psql is directly offered as an application. To run psql in Linux, however, you need to get into your local PostgreSQL folder, which was specified during your installation (*Step 5* of *Exercise 1.01*). You may need to check the subfolders to identify the exact location of the `psql` command. Then, you will launch the psql application by typing `psql`.Once launched, psql will ask you for the server connectivity information.In the following screenshot, the `localhost` server, `postgres` database, and superuser `postgres` credential are used.Once connected, you can run the same SQL as in *Step 2* and get the following result.

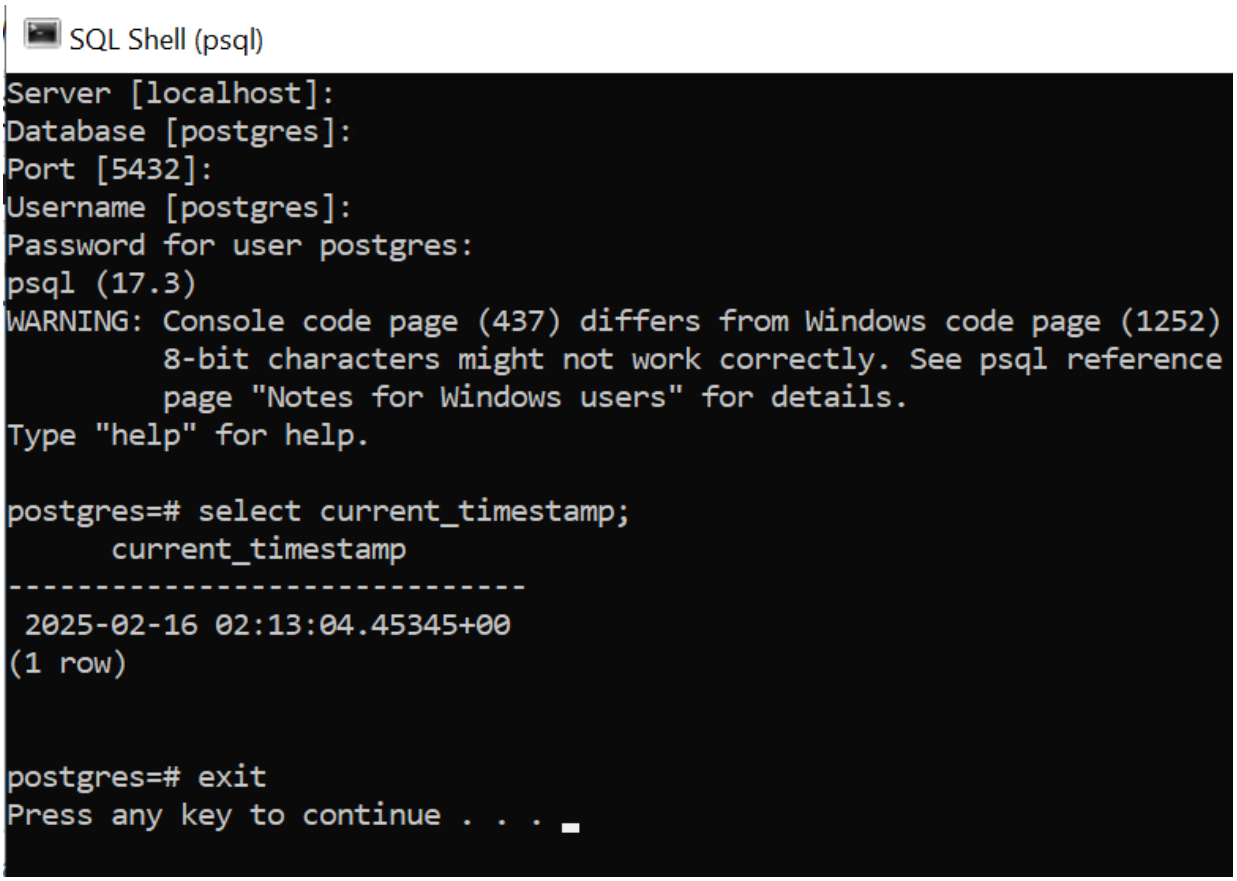

Figure 1.14: psql command interface

4. To exit psql, type `exit`:

Figure 1.15: Exiting psql

Exercise 1.4: Import a sample sqlda database

In this exercise, you will use psql to import a sample database that is provided in this book into your local server. The sample database is based on a fictitious company called **ZoomZoom**, whose database name is `sqlda`. It will be used as the base of the code in the following chapters.Perform the following steps to complete the exercise:

1. Download the `data.dump` file from the `Datasets` folder in the GitHub repository of this course by clicking this link: https://github.com/PacktPublishing/SQL-for-Data-Analytics-Fourth-Edition/tree/main/Datasets.
2. Launch your psql application as instructed in *Step 3* of *Exercise 1.03*.
3. Inside psql, run the following scripts to create and use a new database.

The first statement creates a new database called `sqlda`:

```
CREATE DATABASE sqlda;
```

The second statement switches the current database to the new database.

```
\c sqlda;
```

1. Run the following command to import the ZoomZoom database into PostgreSQL:

```
\i '<local path of the data.dump file>'
```

The local path should be the path of the data dump file that you downloaded in *Step 1*. You can use both the relative path (the path from your current `psql` folder to the file) or the absolute path (the path from the root of your local file system). It is generally recommended that you use the absolute path for easier path identification.In a Windows system, you also must replace the `\` character in the path with double `\` (`\\`). For example, if the file sits in the C drive and the path is `\Users\sqluser\Downloads\data.dump`, the path that the `\i` command uses must be `C:\\Users\\sqluser\\Downloads\\data.dump`.

1. Within the psql application, type `\l` (that's a backslash and a lowercase L) and then press the *Enter* key to check whether the database is created. The `sqlda` database should appear along with a list of the default databases. You can also open `pgAdmin` to confirm the existence of the `sqlda` database and see the database object hierarchy. If you do not see the `sqlda` database, try to right-click on the database in Object Explorer and click refresh.
2. Use a `SELECT` statement to check the data inside a table.

The `SELECT` statement retrieves data from database tables and is one of the most important statements in SQL. You will see a large portion of this book dedicated to its usage. For now, running the following query statement returns the first 5 rows of the `products` table.

```
SELECT * FROM public.products LIMIT 5;
```

SQL statements in both pgAdmin and psql can span multiple lines as long as you don’t break the words (such as the table name, `SELECT`, and `FROM`). You can try to enter this statement in multi-line format by pressing *Enter* after each word, and PostgreSQL will be able to parse the statement correctly. In psql, you need to use the `;` sign to indicate the end of the SQL statement. After you enter the `;` sign and press the *Enter* key, PostgreSQL will start executing the full statement. Also note that in this statement, SQL keywords such as `SELECT` and `FROM` are in uppercase, while the names of tables and schema are in lowercase. SQL statements (and keywords) are indeed case-insensitive. However, when you write your own SQL, it is generally recommended to follow certain conventions on the usage of case and indentation. It will help the user to easily understand the structure and purpose of this statement.The result of the query is shown here.

```
 product_id |         model         | year | product_type | base_msrp | production_start_date | p
------------+-----------------------+------+--------------+-----------+-----------------------+--
          1 | Lemon                 | 2016 | scooter      |    399.99 | 2015-10-28 00:00:00   | 2
          2 | Lemon Limited Edition | 2017 | scooter      |    799.99 | 2016-08-30 00:00:00   | 2
          3 | Lemon                 | 2019 | scooter      |    499.99 | 2018-12-27 00:00:00   | 2
          5 | Blade                 | 2020 | scooter      |    699.99 | 2020-02-17 00:00:00   | 2
          7 | Bat                   | 2022 | scooter      |    599.99 | 2022-06-07 00:00:00   |
(5 rows)
```

Depending on how your table was populated, you may get other products instead of those here. However, you should get the same columns, and you should see meaningful values in each row.After confirming that the `sqlda` database has been successfully created, you can type the `exit` command to exit psql and wrap up this exercise.In the preceding exercise, you set up the `sqlda` database. Now it is time to get yourself more familiar with the contents of this database. The following section provides a description of this database’s contents.

Introducing the sqlda database

Inside the `sqlda` database, there are many tables that have been created and populated with data. You can use a `SELECT` statement to check the data inside each table. For your reference, here is the list of the tables in this `sqlda` database, as well as a brief description for each table:

List of tables	**Description of the table**
countries	An empty table with columns describing countries

customer_sales	Raw data in the semi-structured format of some sales records
customer_survey	Feedback with ratings from the customers
customers	Detailed information for all customers
dealerships	Detailed information for all dealerships
emails	Details of emails sent to each customer
products	Products sold by ZoomZoom
public_transportation_by_zip	Availability measure of public transportation in different zip codes in the United States
sales	Sales records of ZoomZoom on a per customer per product basis
salespeople	Details of salespeople in all the dealerships

Table 1.2: The list of sqlda database tables

Activity 1

You have set up your working environment. You have got PostgreSQL up and running. You have even installed the `sqlda` database. Now it is time to take a look at the data you have in hand and identify its characteristics.Please create SQL statements to fetch the first 20 rows of each table in your `sqlda` database. Please answer the following two questions: Which table is empty? Which table has fewer than 20 records?

Summary

The chapter provided an introduction to data management systems, focusing on SQL and its applications in data analytics. It covered topics such as data modeling, relational databases, and the installation and use of PostgreSQL. These topics lay the foundation for the remaining chapters of this book and ensure you have an established environment for all the exercises and activities in the coming chapters.In the rest of this book, you will learn about SQL statements for different tasks involved in managing and analyzing data. The starting point is table creation, which is covered in the next chapter.

2 Creating Tables with Solid Structures

Join our book community on Discord


https://packt.link/EarlyAccessCommunity

Most of the operations in a relational database are organized around tables and the data inside them. The operations in the lifecycle of data generally can be categorized into four groups—create, read, update, and delete, or CRUD. To utilize any data, you must create the definition of the table first, then populate the table with data records. Once a dataset is created, you can read all aspects of information from it. If there is any need to change the data, you need to update the affected records. After the update operation, you may need to read the updated data again. And finally, when you do not need the data anymore, you will want to delete the records to save storage costs and increase performance. If you do not need this dataset, you can even delete the whole dataset by removing its definition from the database. These CRUD operations are the most common data management activities, and are demonstrated in the following figure:

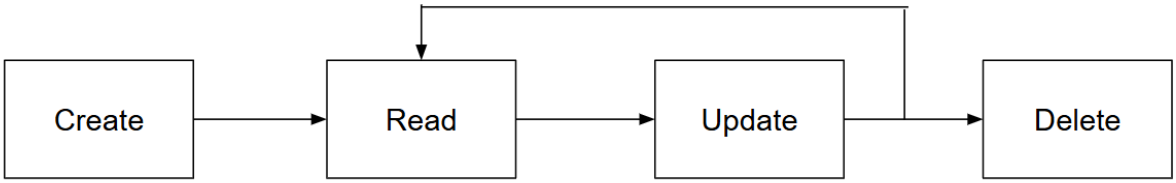

Figure 2.1: CRUD operations

You will learn about CRUD-related SQL statements in this and the upcoming chapters. In this chapter, you will start by diving into data creation and population operations, together with the removal of tables.The following topics are covered in this chapter:

- Running CRUD with SQL
- Creating a table from an existing dataset
- Learning about basic data types of SQL
- Creating a table with an explicit definition
- Inserting data into a table
- Deleting/dropping tables

With these topics, you will be able to handle the creation and deletion of relational datasets, the beginning and end of the data CRUD lifecycle.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in *Chapter 1*.

Running CRUD with SQL

As the most important data manipulation technology for relational databases, SQL provides full support for CRUD operations. Here is a breakdown of the different SQL statements involved in the CRUD process.

CREATE

In any system, there are two ways to create datasets. You can create a table structure first and insert data into this table at a later phase, or you can create the table by designating what data will be stored in this table, and the table will inherit the structure of the incoming data. SQL provides statements for both approaches. The `SQL CREATE` statement allows you to create a table definition and run a separate `INSERT` statement or `COPY` command to populate the table, and you can also run the `CREATE` statement with a `SELECT` clause, which reads data from other datasets and creates the table directly based on the data. You will go through both approaches in this chapter. Then, you will go through the `COPY` command in *Chapter 3*.

READ

SQL uses the `SELECT` statement to retrieve data from one or more tables in a database. `SELECT` is no doubt the most versatile statement in the SQL world. It allows users to specify which columns to fetch and apply sorting, grouping, and filtering to refine the results. It can also serve as a part in other statements, such as offering data to create a new table in the `CREATE` statement. This book will cover `SELECT` statements in detail, starting in *Chapter 5*, all the way to the end.

UPDATE

SQL uses the `UPDATE` statement to perform simple updates to existing data in tables. `UPDATE` can also be used to update a table based on data from multiple tables. Further, SQL provides the `ALTER` statement, which allows you to change the table structure, such as adding or removing columns. All these will be covered in *Chapter 6*.

DELETE

There are two levels of the delete operation. You can delete the data – either a part of it or the entirety – but keep the table structure, or you can remove the table completely, thus removing all data inside it. SQL uses the `DELETE` statement for the former operation, which can remove part of the data or the entire dataset, but keep the table, then uses the `DROP` statement for the latter, which drops both the table and the data inside. The `DROP` operation will be covered in this chapter, and the `DELETE` operation will be discussed in *Chapter 6*.In this chapter, you will learn about the first and last operations of CRUD, `CREATE` and `DROP` . You will start by creating a table.

Creating a table from an existing dataset

As discussed in *Chapter 1*, a relational table organizes data into rows, which are an ordered collection of columns, also known as attributes. The table also contains other properties, such as primary key/foreign key. To create a table, you can either start by filling rows in or by defining its column definitions (also called the table schema) and filling in data later.In the first approach, based on an existing dataset, you can create a table using the `CREATE…AS` statement. This involves running a `SELECT` statement to get existing data and adding a `CREATE` wrapper around the data.

```
CREATE TABLE {table_name} AS (
    {select_query}
);
```

Here, `{select_query}` is any `SELECT` query that can be run in your database. For example, you have already executed the following `SELECT` statement in *Chapter 1*:

```
SELECT * FROM products LIMIT 5;
```

This `SELECT` statement will return the following result:

```
 product_id |         model         | year | product_type | base_msrp | production_start_date | p
------------+-----------------------+------+--------------+-----------+-----------------------+--
          1 | Lemon                 | 2016 | scooter      |    399.99 | 2015-10-28 00:00:00   | 2
          2 | Lemon Limited Edition | 2017 | scooter      |    799.99 | 2016-08-30 00:00:00   | 2
          3 | Lemon                 | 2019 | scooter      |    499.99 | 2018-12-27 00:00:00   | 2
          5 | Blade                 | 2020 | scooter      |    699.99 | 2020-02-17 00:00:00   | 2
          7 | Bat                   | 2022 | scooter      |    599.99 | 2022-06-07 00:00:00   |
(5 rows)
```

Based on the preceding dataset, the following statement will create a new table called `first_products_sample`, which contains the same dataset, including its structure.

```
CREATE TABLE first_products_sample AS (
    SELECT * FROM products LIMIT 5
);
```

This statement only creates a table and does not have any system output. PostgreSQL will only display the following notice:

```
SELECT 5
```

But you can run a query on the newly created table to confirm the `CREATE` statement has been executed properly, with data populated:

```
SELECT * FROM first_products_sample;
```

The result is as follows:

```
 product_id |         model         | year | product_type | base_msrp | production_start_date | p
------------+-----------------------+------+--------------+-----------+-----------------------+--
          1 | Lemon                 | 2016 | scooter      |    399.99 | 2015-10-28 00:00:00   | 2
          2 | Lemon Limited Edition | 2017 | scooter      |    799.99 | 2016-08-30 00:00:00   | 2
          3 | Lemon                 | 2019 | scooter      |    499.99 | 2018-12-27 00:00:00   | 2
          5 | Blade                 | 2020 | scooter      |    699.99 | 2020-02-17 00:00:00   | 2
          7 | Bat                   | 2022 | scooter      |    599.99 | 2022-06-07 00:00:00   |
(5 rows
```

This `SELECT` statement does not have any limit on the number of rows returned, but still you only see 5 rows, because this is the size of the data returned from the previous `SELECT` clause in the `CREATE` statement with the `LIMIT 5` clause, which only brings back the first 5 rows of the entire result dataset. The result is identical to the result of the previous `SELECT` statement. PostgreSQL puts the dataset from the original `SELECT` statement into the new table.The `SELECT` statement used here is very simple. But there are many more functionalities that the `SELECT` statement can provide. You will learn about these functionalities in depth in future chapters. For now, just keep in mind that every functionality of the `SELECT` statement is applicable to the table creation process here.PostgreSQL also provides another way to create tables from a query, which utilizes a `SELECT … INTO …` syntax. An example of this syntax is shown here:

```
SELECT *
INTO second_products_sample
FROM products LIMIT 5;
```

This query achieves the same result as the `CREATE … AS` statement. In this book, you will use the `CREATE … AS` statement because the syntax inside the parentheses is a complete `SELECT` statement; thus, it is easier to create and modify the query without changing the structure of the statement. You can choose either based on your personal preference.The `CREATE … AS` statement and the `SELECT … INTO` statement are simple and straightforward. However, one issue in creating a table with a query is that the definition of the data is not explicitly specified and

can be confusing. For example, you created the `first_products_sample` table, but you don't really know what columns are in the table and what their definitions are. Fortunately, you can get the list of columns and their respective data types with certain PostgreSQL commands.

Describing columns

You can get the list of columns and their respective data types by running the following command in psql:

```
\d first_products_sample
```

The result of this query is this:

```
             Table "public.first_products_sample"
        Column         |            Type             | Collation |  Nullable  | Default
-----------------------+-----------------------------+-----------+----------+---------
 product_id            | bigint        |           |          |
 model                 | text          |           |          |
 year                  | bigint        |           |          |
 product_type          | text          |           |          |
 base_msrp             | numeric       |           |          |
 production_start_date | timestamp without time zone
         |           |          |
 production_end_date   | timestamp without time zone
     |           |          |
```

The result contains the list of columns and their respective data types for this table. PostgreSQL also stores the table definitions in a set of system tables, and you can read the table definition from the system tables. For example, you can run the following SQL to check the column definitions of the `first_products_sample` table:

```
SELECT COLUMN_NAME, DATA_TYPE, ORDINAL_POSITION
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = 'first_products_sample';
```

From the result, you can identify all the columns, their data types, and their position in the table. The difference between this approach and the `\d` command is that the column definitions in the system tables are not displayed in a particular order, while the `\d` command's result follows the order in which columns are defined. For the former to be displayed in order, you must sort the output by the `ORDINAL_POSITION` column by adding an `ORDER BY` clause to the `SELECT` statement, which will be discussed in *Chapter 5*.As shown in the column descriptions, each column belongs to a specific data type. You will learn more about these data types in the next section.

Learning about basic data types of SQL

SQL data types define the type of value that can be stored in a table column, as well as the operations that can be performed on the column. For example, in the customers table, possible columns could be the following:

- `customer_id` (stores unique identifiers for customers)
- `first_name` and `last_name` (stores customer names)
- `total_purchase` (stores total purchase values)
- `birth_date` (stores the birthdate of customers)

Here, you can filter the customers by birthdate to get those who are turning 18, you can add recent purchases to the `total_purchase` field to have the latest purchasing history, and you can also search for customers with a specific name, such as Tom or Jerry. But you cannot add `first_name` and `birth_date` together because one is a string and the other a date, not numbers. SQL data types define the type of value that can be stored in a table column, as well as the operations that can and cannot be performed on the column. The basic data types include numeric, string, and datetime, but there are many more defined in PostgreSQL.

Numeric and monetary

Numeric data types represent numbers. The following figure provides an overview of some of the major types:

Name	Storage Size	Description	Range
smallint	2 bytes	Small-range integer	-32.768 to +32.767
integer	4 bytes	Typical choice for integer	-2,147,483,648 to +2,147,483,647
bigint	8 bytes	Large-range integer	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
decimal	variable	User-specified precision, exact	Up to 131,072 digits before the decimal point; up to 16,383 digits after the decimal point
numeric	variable	User-specified precision, exact	Up to 131,072 digits before the decimal point; up to 16,383 digits after the decimal point
real	4 bytes	Variable precision, inexact	6 decimal digits precision
double precision	8 bytes	Variable precision, inexact	15 decimal digits precision
smallserial	2 bytes	Small autoincrementing integer	1 to 32,767
serial	4 bytes	Autoincrementing integer	1 to 2,147,483,647
bigserial	8 bytes	Large autoincrementing integer	1 to 9,223,372,036,854,775,807
money	8 bytes	currency amount	-92233720368547758.08 to +92233720368547758.07

Table 2.1: Major numeric data typesNumeric data types can be loosely divided into two categories: integer (`smallint` , `integer` , and `bigint`) and numeric (all the remaining data types). Integer numbers can be precisely stored and processed in computer systems. However, since computer systems use binary format for storage, numbers with decimal fractions cannot be stored and represented exactly as decimal numbers. That's why the real and double precision numbers are not exact values. They may contain a certain degree of rounding errors during conversion between decimal format and binary format. To resolve this issue, PostgreSQL introduces the `DECIMAL` and `NUMERIC` data types. These two data types are exactly equivalent and are defined interchangeably for compatibility with other RDBMS. For these two data types, you will define both a precision and a scale in the following format:

```
NUMERIC(precision, scale)
```

The precision of `NUMERIC` is the total count of significant digits in the whole number, that is, the number of digits on both sides of the decimal point. The scale of `NUMERIC` is the count of decimal digits in the fractional part, to the right of the decimal point. Within this scale, the calculation is precise. For example, `NUMERIC(20, 5)` numbers will be accurate to the fifth digit to the right of the decimal point, and can have up to 15 (20-5) digits of value of the whole number part. Let's look at some scenarios:

- The number 987654321012345.12345 can be stored exactly in this data type
- The number 987654321012345.123456 will lose the last decimal digit when stored in this data type
- The number 1987654321012345.12345 cannot be stored in this data type because it has 16 digits of whole numbers

The `money` data type is specifically designed for monetary values. It follows all the arithmetic rules of the `NUMERIC` type, with the ability to allow different money formats. For example, the value `$1,234.56` is a valid value for the money data type but not a valid numeric value because the `$` sign is not a valid digit. However, not all external tools support this data type. So, many data modelers still use `NUMERIC` fields to represent currency.

Character

Character data types, sometimes called text data types or string data types, store text information. The following figure summarizes character datatypes:

Name	Description
character varying(n), varchar(n)	Variable-length with limit. The maximum of N is 10,485,760
character(n), char(n)	Fixed-length, blank-padded.
bpcha	Variable-length up to 10,485,760, blank-trimmed
text	Variable-length up to 10,485,760

Table 2.2: Major character data typesThe most common character data type is `varchar(n)`, which allows you to store a variable-length string up to N characters. For example, a `varchar(100)` column can store any string between the length of 0 and 100. The `varchar` character data type is widely used for natural text, such as names and addresses. While varchar is the most popular option for text data, there are specific usage scenarios where you can use other character types. First of all, if you are not sure about the upper limit of your text field and would like to reserve the maximum capacity, you can also use the text data type, which is the equivalent of `varchar(10485760)`. Here, `10485760` is the maximum length of the `varchar` data type. In another scenario, many texts from data sources may contain leading or trailing blank spaces. You can use the `bpchar` data type, which automatically trims the leading or trailing blank spaces upon receiving the data. Finally, some text fields, such as car plate numbers and driver's license numbers, have a fixed length. You can use `char(n)` for them. The `char(n)` data type applies a fixed length to the data inside and helps ensure data quality.The text of character data types must be enclosed in single quotes when used in a SQL statement. For example, the name Tom must be written as `'Tom'` in a SQL statement. These single quotes are used for SQL to properly delimit the text. They are not a part of the text string. Furthermore, an empty string is also a string, which can be written as `''` (i.e., an empty string enclosed by a single-quote pair).

Boolean

Booleans are a data type used to represent `True` or `False`. The following table summarizes values that are represented as Boolean when used in a query with a data column type of Boolean:

Boolean Value	Accepted Values
True	t, true, y, yes, on, 1
False	f, false, n, no, off, 0

Table 2.3: Accepted Boolean valuesNote that the values are not case sensitive. While all these values are accepted, the values of `True` and `False` are compliant with best practice. Booleans can also take on `NULL` values.

Datetime

The **datetime** data type is used to store time-based information, such as dates and times. The following are some examples of datetime data types:

Name	Storage size	Description
timestamp without time zone	8 bytes	Both date and time (no time zone). For example, 2026-01-23 04:05:06.
timestamp with time zone	8 bytes	Both date and time, with time zone. For example, 2026-01-23 04:05:06+02. Here, +02 indicates Eastern European Time (UTC+2).
date	4 bytes	Date (no time of day). For example, 2026-01-23.
time without time zone	8 bytes	Time of day (no date). For example, 04:05:06.
time with time zone	12 bytes	Time of day (no date), with time zone. For example, 04:05:06+02.
interval	16 bytes	Time interval. For example, 100 days.

Table 2.4: Popular datetime data typesSimilar to strings, the text of date/time data types must be enclosed in single quotes when used in a SQL statement. For example, the date January 23, 2026 must be written as `'2026-01-23'`

in a SQL statement.You will explore this data type further in *Chapter 12*.

More data types

In *Chapter 1*, you explored the definition of JSON documents. You also learned that JSON is a natural way of describing real-world usage scenarios, but due to performance and ease of operation concerns, the relational data model is the dominant force in the database market. However, there are situations when the relational data model does not fit well. For example, when you evaluate customer information, a small but considerable portion of customers may have pets, and the number of pets owned by a customer may vary. This varying data may not fit into the fixed columns of a relational table well.PostgreSQL, as well as many versions of modern DBMS, also support complex data structures, such as JSON and arrays. Arrays are simply lists of data, usually written as members enclosed in square brackets. For example, `['cat', 'dog', 'horse']` is an array. Arrays can also contain a series of JSON objects. These data structures can be used in certain usage scenarios, and being able to use them in a database makes it easier to perform many kinds of analysis work. You will explore complex data structures in more detail in *Chapter 11*.In addition to the preceding data types, PostgreSQL also supports many other data types, usually for specific purposes, such as geometrics and network addresses. You can visit the PostgreSQL official website at https://www.postgresql.org/docs/current/datatype.html for further information.

Note

> Although the ANSI SQL standard defines a list of data types, different RDBMSs may have their own interpretation and extension. The data types discussed in this book are based on the PostgreSQL definition. If you use a different RDBMS, you may see some differences in implementation. Furthermore, all RDBMSs, including PostgreSQL, are actively evolving. They constantly add support for new data types and adjust data type implementations if necessary. So, it is always prudent to use the data type definitions in this book as general guidance and double-check your RDBMS for the exact data type definitions it has.

With an understanding of column data types, now you can use the combination of multiple columns to define a table, which will be covered in the next section.

Creating a table with an explicit definition

To create a new blank table, you use a different flavor of the `CREATE TABLE` statement. This statement takes the following structure:

```
CREATE TABLE {table_name} (
    {column_name_1} {data_type_1} {column_constraint_1},
    {column_name_2} {data_type_2} {column_constraint_2},
    {column_name_3} {data_type_3} {column_constraint_3},
    …
);
```

Here, `{table_name}` is the name of the table, `{column_name}` is the name of the column, `{data_type}` is the data type of the column, and `{column_constraint}` is one or more options giving special properties to the column. Column constraints will be discussed in the subsection that follows. You can start creating your first table by specifying column names and data types. Suppose you want to create a table called `employees` with columns for the name and salary of employees. The query would look as follows:

```
CREATE TABLE employees (
    employee_name VARCHAR(100),
    salary NUMERIC(12,2)
);
```

Once you execute this statement, you can run a simple `SELECT` statement to verify that the table is created:

```
SELECT * FROM employees;
```

There are no rows inside as you have not run any statements to populate it.In the next subsection, you will learn more about the table/column definition by applying table/column constraints.

Column constraints and table constraints

Columns may have constraints that determine the valid values of the columns. Constraints are rules applied to table attributes to ensure data accuracy and integrity. For example, salary cannot be lower than 0, and hire date cannot be earlier than the company's starting date. Common constraints include the following:

- `NOT NULL`: Ensures a column cannot have `NULL` values. `NULL` represents a missing or unknown value. The `NOT NULL` constraint indicates that the values in this column can not be empty.
- `UNIQUE`: Ensures all values in a column are distinct.
- `CHECK`: Enforces specific conditions on column values, such as salary must be greater than 0 (`salary > 0`).
- `DEFAULT`: Specifies a default value for a column when no value is provided.

For example, the following statement utilizes all four constraints:

```
CREATE TABLE employees (
    employee_name VARCHAR(100) NOT NULL,
    employee_ssn CHAR(9) UNIQUE,
    salary NUMERIC(12,2)CHECK (salary > 0),
    employee_vacation INTEGER DEFAULT 20
);
```

Similarly, tables have constraints too. The table `CHECK` constraint can be applied to individual columns, in which case it is the same as the column `CHECK` constraints. But typically, table `CHECK` constraints are applied to the calculation between multiple columns. The following statement explains this:

```
CREATE TABLE employees (
    employee_name VARCHAR(100) NOT NULL,
    employee_ssn CHAR(9) UNIQUE,
    salary NUMERIC(12,2),
    bonus NUMERIC(12,2),
    employee_vacation INTEGER DEFAULT 20,
    CHECK (salary + bonus > 0),
    CHECK (employee_vacation > 0)
);
```

The `employee_vacation > 0` check can also be applied on the `employee_vacation` column, but the `salary + bonus > 0` check must be applied at the table level.The most common table constraint is the primary key constraint. As discussed in *Chapter 1*, primary key is a special constraint that is unique for each row so that each row is a unique mapping of a real-world object. That means a primary key must be unique in value and cannot contain NULLs. If the primary key of this table contains only one column, you can add this `PRIMARY KEY` constraint to the column. If the primary key of this table consists of multiple columns, you need to use a table constraint to define the key in the `CREATE` statement. For example, **social security numbers** (**SSNs**) are unique for each employee and can be used to identify employees. Employees can have the same names, but SSNs must be unique. So, you can define it as a primary key for the `employees` table. Since only one column is involved, you can directly define it on the column.

```
CREATE TABLE employees (
    employee_name VARCHAR(100),
    employee_ssn CHAR(9) PRIMARY KEY,
    salary NUMERIC(12,2)
);
```

However, for an `employee_dependents` table, each employee can have multiple dependents. So, you will need to use both the SSN and the dependent's name for identification purposes. In this case, the primary key is a combination of two columns and must be defined as a table constraint.

```
CREATE TABLE employee_dependents (
    employee_ssn CHAR(9),
    employee_dependent_name VARCHAR(100),
```

```
    PRIMARY KEY (employee_ssn, employee_dependent_name)
);
```

The last table constraint we will cover is the foreign key constraint. As discussed in *Chapter 1*, a column (or a group of columns) is a foreign key when its values come from the column(s) of another table. A foreign key is usually the primary key of the source table. In the following example, the `product_id` column and `customer_id` column are the foreign keys of table sales, coming from the `products` table and the `customers` table, respectively. They are defined by using the keyword `REFERENCES` followed by the source table and the source column's names. Have a look at the following code:

```
CREATE TABLE sales (
    sales_order_id integer,
    product_id integer
        REFERENCES products (product_id),
    customer_id integer
        REFERENCES customers (customer_id),
    sales_amount money
);
```

Exercise 2.1: Creating and populating tables

In this exercise, you will create two tables using the two flavors of the `CREATE TABLE` statement. The marketing team at ZoomZoom would like to perform some analytics on the `products` table. But their analysis will write back into the table, thus changing the data. To minimize the impact, they would like to duplicate the `products` table into two new tables called `products_new` and `products_new_2` .Perform the following steps to complete the exercise:

1. Open `psql` . When asked, connect to the `sqlda` database.

Note

You will use the `sqlda` database for the rest of the book. Please choose the `sqlda` database for all future exercises.

1. Execute the following command to see the definition of the `products` table.

```
\d products
```

1. Execute the following query in `psql` to create the `products_new` table. The table definition is based on the table definition from *Step 2*:

```
CREATE TABLE products_new (
    product_id            bigint,
    model                 text,
    year                  bigint,
    product_type          text,
    base_msrp             numeric,
    production_start_date timestamp,
    production_end_date   timestamp
);
```

1. Run the following query to create the `products_new_2` table:

```
CREATE TABLE products_new_2 as
SELECT * FROM products;
```

You should now be able to run a `SELECT` statement against each table to get the table content. The `products_new_2` table has been populated with all 12 rows from the `products` table, and the `products_new` table is still empty.So far, you have learned how to define empty tables with columns of different data types and constraints. The next step is to populate the empty table with the `INSERT` statement, which you will learn about in the next section.

Inserting data into a table

Once a table is created, you can add new data to it using several methods. The simplest method is to insert values using the `INSERT INTO… VALUES` statement to insert one row at a time. It has the following structure:

```
INSERT INTO {table_name} ({column_1], {column_2}, …)
VALUES ({column_value_1}, {column_value_2}, … );
```

Here, `{table_name}` is the name of the table you want to insert your data into, `{column_value_1}, {column_value_2}, …` is the list of values you want to insert into the new row, and `{column_1}, {column_2}, …` is a list of the columns that will receive those values.For example, if you want to insert a new entry for a scooter into the `products_new` table, this can be done with the following query:

```
INSERT INTO products_new (
    product_id, model, year,
    product_type, base_msrp,
    production_start_date,
    production_end_date
) VALUES (
    13, 'Nimbus 5000', 2017,
    'scooter', 500.00,
    '2017-03-03', '2023-03-03'
);
```

This query adds a new row to the `products_new` table accordingly. You can run a `SELECT` query to see this new row in the table.You can have multiple value lists, one for each row, separated by a comma. For example, the following statement will insert two rows into the table:

```
INSERT INTO products_new (
    product_id, model, year,
    product_type, base_msrp,
    production_start_date, production_end_date
) VALUES(
    14, 'Nimbus 6000', 2017,
    'scooter', 600.00,
    '2017-04-03', '2023-04-03'
),
(
    15, 'Nimbus 7000', 2017,
    'scooter', 700.00,
    '2017-05-03', '2023-05-03'
);
```

Another way to insert data into a table is to use the `INSERT` statement with a `SELECT` query using the following syntax:

```
INSERT INTO {table_name} ({column_1], {column_2}, …}
{select_query};
```

Here, `{table_name}` is the name of the table into which you want to insert the data, `{column_1}, {column_2}, …` is a list of the columns whose values you want to insert, and `{select query}` is a query with the same structure as the values you want to insert into the table.Take the example of the `products_new` table. You have created it and inserted some rows into it. If you also want to insert all the existing products, you could use the following query, which inserts one more row into the table:

```
INSERT INTO products_new (
    product_id, model, year, product_type, base_msrp,
    production_start_date, production_end_date
)
SELECT *
FROM products;
```

You will practice both approaches in the following exercise.

Exercise 2.2: Populating a table

The marketing team, upon receiving the tables you created, would like to perform their analysis on some fictitious data, as well as the original data inside the product table. In this exercise, you will use two flavors of the `INSERT` statement to populate the empty `products_new` table.Perform the following steps to complete the exercise:

1. Open `psql` and connect to the `sqlda` database.
2. Execute the following command to insert three new rows into the `products_new` table.

```
INSERT INTO products_new (
    product_id, model, year,
    product_type,base_msrp,
    production_start_date,
    production_end_date
) VALUES(
    13, 'Nimbus 5000', 2017,
    'scooter', 500.00,
    '2017-03-03', '2023-03-03'
),(
    14, 'Nimbus 6000', 2017,
    'scooter', 600.00,
    '2017-04-03', '2023-04-03'
),(
    15, 'Nimbus 7000', 2017,
    'scooter', 700.00,
    '2017-05-03', '2023-05-03'
);
```

1. Execute the following query to insert the rows from the table into the `products_new` table:

```
INSERT INTO products_new (
    product_id, model, year, product_type,base_msrp,
    production_start_date, production_end_date
)
SELECT * FROM products;
```

1. Run the following query to verify the data in the `products_new` table:

```
SELECT * FROM products_new;
```

You should see 15 rows inside the `products_new` table, 3 new from the `INSERT` statement in *Step 2* and 12 from the `products` table in *Step 3*.With the introduction of the `INSERT` statement, you can now use multiple ways to create tables, either from an existing dataset or with an explicit definition of columns. At the other end of the CRUD process, you will also need the ability to drop tables, which will be covered in the next section.

Deleting/dropping tables

To delete all the data in a table and the table itself, you can just use the `DROP TABLE` statement with the following syntax:

```
DROP TABLE {table_name};
```

Here, `{table_name}` is the name of the table you want to delete. If you wanted to delete all the data in the `products_new` table along with the table itself, you would write the following:

```
DROP TABLE products_new;
```

Once the table is dropped, all aspects of this table are gone, and you cannot perform any operations on it. If you want to read from this table, you will receive an error message from PostgreSQL telling you that the table does not exist. A PostgreSQL enhancement of the `DROP` statement is `DROP TABLE IF EXISTS`. This statement will check the existence of the table. If the table is not in the database, PostgreSQL will skip this statement with a notification, but without reporting an error, as shown here.

```
DROP TABLE IF EXISTS products_new;
```

`DROP TABLE IF EXISTS` is helpful if you want to automate SQL script execution. One common usage scenario is to use it before the `CREATE TABLE` statement. If the table already exists, your `CREATE TABLE` statement will fail and raise an error. But if you have a `DROP TABLE IF EXISTS` statement before the `CREATE TABLE` statement, pre-existing tables will have been dropped before you try to recreate them. This is useful in automated computing operations where you constantly create temporary tables that you will not need after the computing job is completed. The catch is that you must pay extreme attention to make sure that the table is truly temporary and is not used by anyone else. Otherwise, you may accidentally drop tables that are used by some other users. For this reason, the `DROP TABLE IF EXISTS` statement is usually used on tables designated for automated data processing.Now test what you have learned by performing an exercise to drop the table using the `DROP TABLE` statement.

Exercise 2.03: Deleting an unnecessary table

In this exercise, you will learn how to drop a table. For instance, the marketing team has finished analyzing the products, and they no longer need the `products_new` and `products_new_2` tables. To save space on the database, they want to remove the tables completely.Perform the following steps to complete the exercise:

1. Open psql and connect to the `sqlda` database.
2. Run the following query to drop the `products_new` table:

```
DROP TABLE products_new;
DROP TABLE IF EXISTS products_new_2;
```

1. Now, try to access the `products_new` table again:

```
SELECT * FROM products_new;
```

You will get the following error:

```
ERROR:  relation "products_new" does not exist
LINE 1: SELECT * FROM products_new;
```

You can see that this table no longer exists, and its data has been completely wiped out.In this chapter, you have learned how to create and populate tables. You will apply this knowledge in the next activity to enhance your understanding.

Activity 2

Your business team is interested in a side project that studies customer behavior. To ensure that this study will not impact the everyday activities in the production system, they would like to use a new table, `customers_copy`, which is a complete copy of the existing `customers` table, instead of directly running queries against the production customers table. They also want to create a new table called `customer_segment`, which contains the following three fields:

- `customer_segment_id`: An integer number that comes from the source system
- `customer_segment_desc`: A text field that contains up to 100 characters of the segment name, such as `50 year old male with wife, kids, and pets`
- `segment_creation_date`: A date field indicating when the segment was created

Please create this table.For now, the business user has only one sample segment with ID 1. The segment name is `50 year old male with wife, kids, and pets`, and the creation date was 01/01/2025. Please insert the sample record into the table.After you finish all these tasks, please drop both tables to save database storage.

Summary

The chapter provided a comprehensive guide on creating, populating, and dropping tables in SQL. It introduced the CRUD process, the lifecycle of data creation and population. It also discussed different data types used in PostgreSQL. With knowledge of these topics, you will be able to handle the creation and deletion of relational datasets, the beginning and end of the data CRUD lifecycle. `CREATE`, `INSERT`, and `DROP` are useful to manage the data lifecycle inside a DBMS. However, the DBMS often needs to get the data from outside sources and eventually needs to output the data to outside locations. The PostgreSQL tool used for this purpose is `COPY`. You will learn more about `COPY` in the next chapter.

3 Exchanging Data Using COPY

Join our book community on Discord

 https://packt.link/EarlyAccessCommunity

In the previous chapter, you learned about the SQL `CREATE`, `INSERT`, and `DROP` operations in the CRUD process. You also learned how to populate the created tables with manually written values or values from other tables. These operations are good enough to work on small datasets inside a database. However, in real-world scenarios, the bulk of the data comes from systems outside of the database. For example, different systems within the same company need to exchange data, and business analysts need to utilize data from external vendors for analytics. This data is usually sent in the form of files. In addition, once you finish your analytical tasks, you also need to share your results with internal teams or external customers. This data sharing is often done using files, too. While it is possible to use SQL statements to read from or write to these files, they are generally slow and are limited in file processing power.Typically, data transfer is done with applications outside of the DBMS. As the following figure shows, applications running from a workstation machine can connect to the DBMS and instruct the DBMS to perform the import/export tasks.

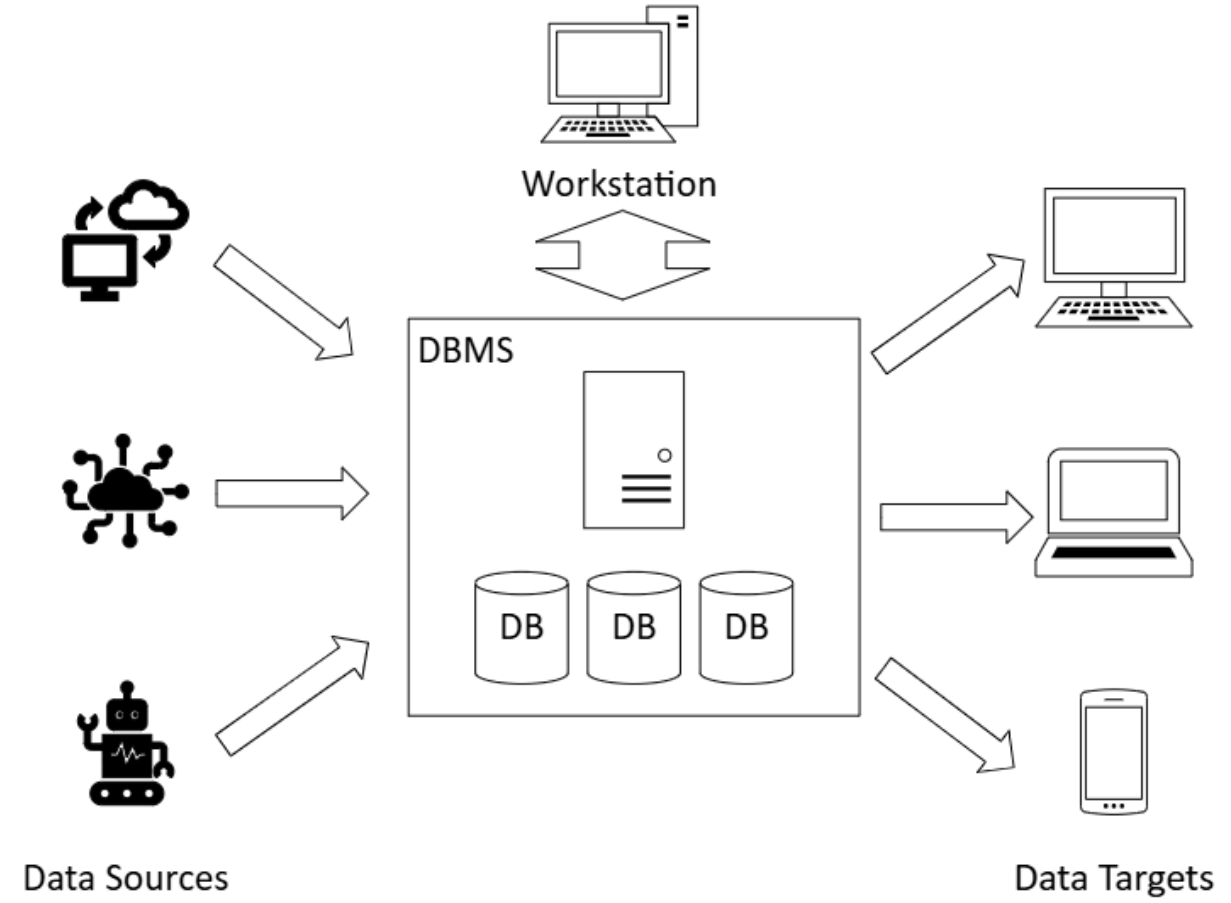

Figure 3.1: The DBMS dataflow

PostgreSQL provides the general-purpose psql application for easier data management. Inside the psql application, there is a `COPY` command that can be used to efficiently transfer data between a table and a file. The `COPY` command allows bulk data loading from files into tables and exporting data from tables to files, faster than individual `INSERT` or `SELECT` statements. You will learn how to use the `COPY` command in this chapter.The following topics are covered in this chapter:

- Exporting data from a PostgreSQL database
- Importing data into a PostgreSQL database

With knowledge of these topics, you will be able to transport data between the database management system and external file systems.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in *Chapter 1*, *Introduction to Data Management Systems*.

Exporting data from a PostgreSQL database

One of the functionalities of the `COPY` command is to retrieve data from your database and dump it into the file location and format that you choose. For example, consider the following statement:

```
COPY (SELECT * FROM products LIMIT 5)
TO STDOUT
WITH CSV HEADER;
```

The following is the output of the code:

```
product_id,model,year,product_type,base_msrp,production_start_date,production_end_date
1,Lemon,2016,scooter,399.99,2015-10-28 00:00:00,2018-02-03 00:00:00
2,Lemon Limited Edition,2017,scooter,799.99,2016-08-30 00:00:00,2016-11-24 00:00:00
3,Lemon,2019,scooter,499.99,2018-12-27 00:00:00,2024-08-24 00:00:00
5,Blade,2020,scooter,699.99,2020-02-17 00:00:00,2020-09-23 00:00:00
7,Bat,2022,scooter,599.99,2022-06-07 00:00:00,
```

Note that `COPY` is a psql command. You need to run it inside the psql prompt. This command returns five rows from the `products` table, with each record on a new line, and each value separated by a comma, in a typical `.csv` file format. The header is also included at the top.Here is a breakdown of this command and the parameters that were passed in:

- `COPY` is simply the command used to transfer data to a file format.
- `(SELECT * FROM products LIMIT 5)` is the query that you want to copy the result from.
- `TO` indicates that the results should be printed to the designated device. Normally, the device is a file on the hard drive.
- `STDOUT` indicates that the output will be directed to the standard output rather than a file. "Standard output" is the common term for displaying output in a command-line terminal environment, which is often shortened to `STDOUT`. Since the target of the `COPY` command is specified as `STDOUT` in this example, the results will only be copied into the command-line interface and not into a file.
- `WITH` is an optional keyword used to separate the parameters that you will use in the database-to-file transfer. Within `WITH`, you can specify multiple parameters, such as the following:
  - `CSV` indicates that you will use the CSV file format. You can also specify `BINARY`, or you can leave this out and receive the output in text format.
  - `HEADER` indicates that you want the header printed as well. For CSV files, you can optionally choose whether you want to list the names of the columns as the first row. This helps the users of the files understand the meaning of each column and is commonly considered a best practice.

While the `STDOUT` option is useful, often, you will want to save data to a file. The `COPY` command offers the functionality to do this. The resulting data file is saved on the PostgreSQL server machine. You must specify the full file path (relative file paths are not permitted). If your PostgreSQL database runs on a Windows computer, you can test this out using the following command in `psql`:

```
COPY (SELECT * FROM products LIMIT 5)
TO 'c:\Users\Public\my_file.csv'
WITH CSV HEADER;
```

The output will be the following:

```
COPY 5
```

The preceding code saves the file to the `Public` user's personal folder. You can then use a normal text editor to open the file, which has now been saved in CSV format at the location you specified:

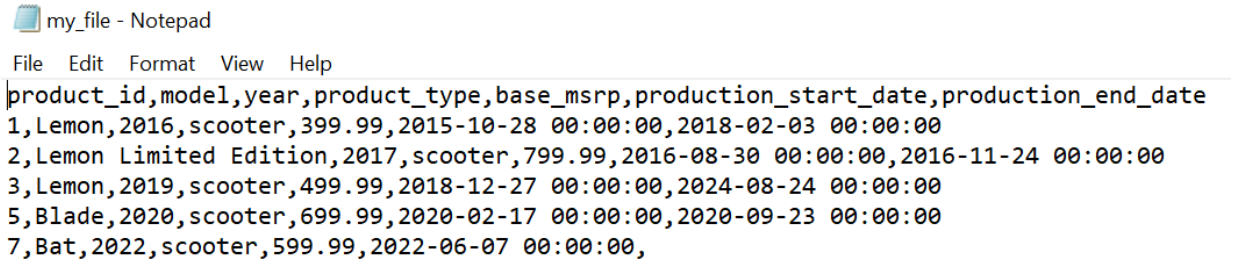

Figure 3.2: The output file of the COPY command

Since this example is executed on a PostgreSQL server that is hosted on a Windows machine, the full file path is in the Windows file path format. If you are running the command on any other operating system, you'll need to adjust the file path accordingly. Also, you must use a folder that you have permission to work on. On any operating system, you may be restricted on which folder you can write to. In this example, `Public` is a reserved user in the Windows system, which has the least restrictive security measures. Most of you should have permission to perform this file saving. Otherwise, you will receive a permission error. In this chapter, all the files will be placed into the `c:\Users\Public` folder because the Windows system usually allows users to read/write in this folder, so it can be used for your exercise if you can't find a better folder.The value in single quotes that follows the `TO` keyword is the absolute path to the output file. The format of the path will depend on the operating system you are using. On Linux and Mac, the directory separator would be a forward-slash (`/`) character, and the root of the main drive would be `/`. On Windows, however, the directory separator would be a backslash (`\`) character, and the path would start with the drive letter.The `COPY` command works on the database server. However, a common scenario is that users would like to download files to their workstations. For this situation, you would use the `\COPY` command, which will be covered in the next subsection.

\COPY in psql

The PostgreSQL server in this book is installed on your local machine. So, your local machine is both the DBMS server and the client workstation. Your psql application will send the `COPY` command to the PostgreSQL server and the `COPY` command will run on the server. As such, it saves the file to the server file paths, which happen to be paths on your local machine. However, in a real-world setup, servers and workstations are often separate machines. Servers are highly protected, while workstations belong to the user. Users usually do not have access to the file system of the server machines and need to save the files to their local workstation paths.psql has a different flavor of the `COPY` command called `\COPY`. The `\COPY` command is similar in syntax to the `COPY` command but runs on the local machine and saves the file to the local machine. Once you have connected to your database using psql, you can test out the `\COPY` instruction by using the following command:

```
\COPY (SELECT * FROM products LIMIT 5) TO 'c:\Users\Public\my_file_local.csv' WITH CSV HEADER;
```

Note that the `\COPY` instruction does not allow multiple lines. All this code must be typed in one line. The following is the output of the code:

```
COPY 5
```

Here is a breakdown of this command and the parameters that were passed in:

- `\COPY` invokes the PostgreSQL `COPY` command to output the data.
- `(SELECT * FROM products LIMIT 5)` is the query that you want to copy the result from.
- `TO 'c:\Users\Public\my_file_local.csv'` indicates the file path that psql should save the output into. Note that the `\COPY` command runs on the client workstation and works with files on local disks, so it allows both absolute paths and relative paths. However, this chapter will only use the absolute path `c:\Users\Public` for clarity.
- The `WITH CSV HEADER` parameters operate in the same way as before.

You can now find the `my_file_local.csv` file in the file folder, which you can open with the text editor of your choice, such as Notepad. However, it is worth noting that although this file is saved to the same folder as the file in the previous example, the logic behind the two files is different. The `COPY` command runs on the DBMS server. DBMS saves the file to the server file path. The `\COPY` command runs on the workstation. It saves the file to the local workstation's file path. In your system, these two paths happen to be overlapping. But if you are running psql from a different machine, you will see that the machine each file gets saved to is different.

Note

While you can split the text of the `COPY` command into multiple lines, the `\COPY` command does not allow the query to contain multiple lines. A simple way to leverage multiline queries is to create a view containing your data before the `\COPY` command and drop the view after your `\COPY` command has finished. You will learn how to create a view in *Chapter 7*, *Defining Datasets from Existing Datasets*.

There are more options that you can utilize in the `COPY` and `\COPY` commands. You will learn about these options in the next subsection.

Configuring COPY and \COPY

A simplified syntax of the `COPY` and `\COPY` command looks like this:

```
COPY { table_name | ( query ) }
    TO { 'filename' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]
```

You can not only save the content of a `table_name` table but also specify the data you want to save by using a `SELECT` query. Also, there are several options under the `WITH` option that you can use to configure the `COPY` and `\COPY` commands. Some of them include the following:

- `FORMAT: format_name` can be used to specify the format. The options for `format_name` are `csv`, `text`, or `binary`. Alternatively, you can simply specify `CSV` or `BINARY` without the `FORMAT` keyword or not specify the format at all and let the output default to a text file format.
- `DELIMITER`: Inside a file row, you need a character to indicate the beginning and end of each value. This character is called a delimiter. The most common delimiter for CSV files is a comma (,), but you can use the `delimiter_character` option to specify the delimiter character for CSV or text files (for example, | for pipe-separated files).
- `NULL: null_string` can be used to specify how `NULL` values should be represented (for example, whether to use blanks or the string `NULL` to represent `NULL` values if that is how missing values should be represented in the data).
- `HEADER`: This specifies that the header should be output.
- `QUOTE: quote_character` can be used to specify how fields with special characters (for example, a comma in a text value within a CSV file) can be wrapped in quotes so that they are ignored by `COPY`.
- `ESCAPE: escape_character` specifies the character that can be used to escape special characters.
- `ENCODING: encoding_name` allows the specification of the encoding, which is particularly useful when you are dealing with foreign languages that contain special characters or user input.

For example, running from psql, the following would create a pipe-separated file, with a header, an empty (0 length) string to represent a missing (`NULL`) value, and the double quote (`"`) character to represent the quote character:

```
\COPY products TO 'c:\Users\Public\my_file.csv' WITH CSV HEADER DELIMITER '|' NULL '' QUOTE '"'
```

The following is the output of the code:

```
COPY 12
```

The file generated will have the designated delimiter and header:

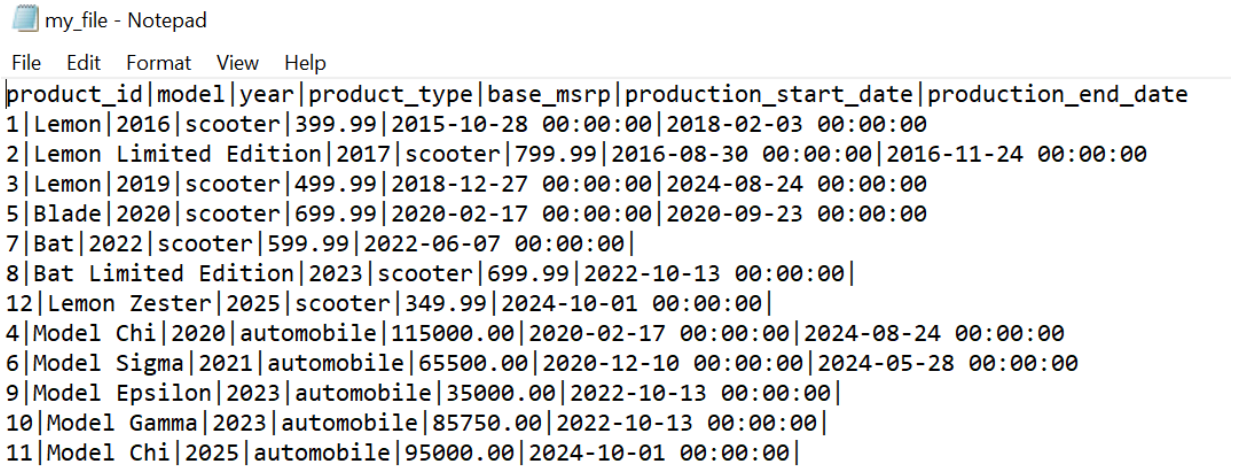

Figure 3.3: File output with OPTIONS specified

In this section, you use the `COPY` and `\COPY` commands to download the database data into files. In the next section, you will learn how to use the `COPY` and `\COPY` commands to upload large amounts of data from files into a database.

Importing data into a PostgreSQL database

The `COPY` and `\COPY` commands can not only be used to efficiently download data from tables to external files but can also be used to upload data from external files to tables. They are far more efficient at uploading data than an `INSERT` statement. There are a few reasons for this:

- When using `COPY`, there is only one push of a data block, which occurs after all the rows have been properly allocated
- There is less communication between the database and the client, so there is less network latency
- PostgreSQL includes optimizations for `COPY` that would not be available through `INSERT`

Similar to the data exporting process discussed in the previous section, for data importing, the `COPY` command runs on the server and the `\COPY` command runs on the user workstation.Here is an example of using the `\COPY` command to copy rows into a table from a file on the local machine:

1. First, run the following SQL to create a new table for `\COPY` command testing:

```
CREATE TABLE products_csv (
  product_id            bigint,
  model                 text,
  year                  bigint,
  product_type          text,
  base_msrp             numeric,
  production_start_date timestamp,
  production_end_date   timestamp
);
```

1. Then run the following `\COPY` command to test its data loading functionality:

```
\COPY products_csv FROM 'c:\Users\Public\my_file.csv' CSV HEADER DELIMITER '|'
```

Note that the command uses the `FROM` keyword to indicate that the data will come into the database instead of the `TO` keyword, which indicates the data will go from the database table to an external file. This outputs the following:

```
COPY 12
```

If you run a `SELECT` query against this table, you will see the 12 rows from the file you saved before. You can even manually edit the file using Notepad and change some of the values before running this `\COPY` command, and verify the loaded table containing the same change you made to the file.Here is a breakdown of this command and the parameters that were passed in:

- `\COPY` loads the external file's data into the database table.
- `products_csv` is the name of the table that you want to append to.
- `FROM 'c:\Users\Public\my_file.csv'` specifies that you are uploading records from `c:\Users\Public\my_file.csv`. The `FROM` keyword specifies that you are uploading records, as opposed to the `TO` keyword, which you use to download records.
- The `(WITH) CSV HEADER` parameters operate the same as before. Note that when you indicate the file format is CSV, the `WITH` keyword is optional, as shown in the preceding statement.
- `DELIMITER '|'` specifies what the delimiter is in the file. For a CSV file, this is assumed to be a comma, but other delimiter characters can be used.

Note

While `COPY` and `\COPY` are great for exporting data to other tools, there is additional functionality in PostgreSQL for exporting a database backup.

For these maintenance tasks, you can use `pg_dump` for a specific table and `pg_dumpall` for an entire database or schema. These commands even let you save data in a compressed (TAR) format, which saves space. Unfortunately, the output format from these commands cannot be readily consumed outside of PostgreSQL. Therefore, it does not help you with importing or exporting data to and from other analytics tools, such as Python.

Since you have now learned how to import and export data, you will implement an exercise to export data to a file and process it in Excel.

Exercise 3.1: Exporting data to a file for further processing in Excel

The **ZoomZoom** executive committee is busy scouting for new locations to open their next dealership. Since a presentation needs to be made in PowerPoint, you can use Microsoft Excel to analyze customer distributions based on the `.csv` file. Then, you can simply copy the analysis to your slide. As a data analyst, you will be helping them make this decision by presenting the data about the customers in the CSV file format. The data will need to be retrieved from the `customers` table of the `sqlda` database. The `psql` and `COPY` commands you learned about will come in handy. This analysis will help the **ZoomZoom** executive committee decide where they might want to open the next dealership.

1. Open `psql`. When asked, connect to the `sqlda` database.
2. Save the data in the `customers` table from your `sqlda` database to a local file in the `.csv` format. Please note that the OS-specific path needs to be prepended to the `customer_details.csv` filename to specify the location to save the file in. Here, in a Windows environment, you would use `c:\Users\Public` as the folder:

```
COPY
(SELECT * FROM customers)
TO 'c:\Users\Public\customer_details.csv'
WITH CSV HEADER DELIMITER ',';
```

`psql` will display the following output:

```
COPY 50000
```

If you open the `customer_details.csv` text file, you should see the following output:

Figure 3.4: Output from the COPY command

1. Using Microsoft Excel or your favorite spreadsheet software or text editor, open the `customer_details.csv` file. Now that you have the output from your database in CSV file format, you can do whatever analytics you would like to perform. This book is an introductory SQL book. Excel is not the focus here. But you will learn some possible analysis approaches in the next few steps.
2. In Microsoft Excel, select any data cell and press *Ctrl + A*. All data will be selected. Right-click on the data and you will see the following action menu:

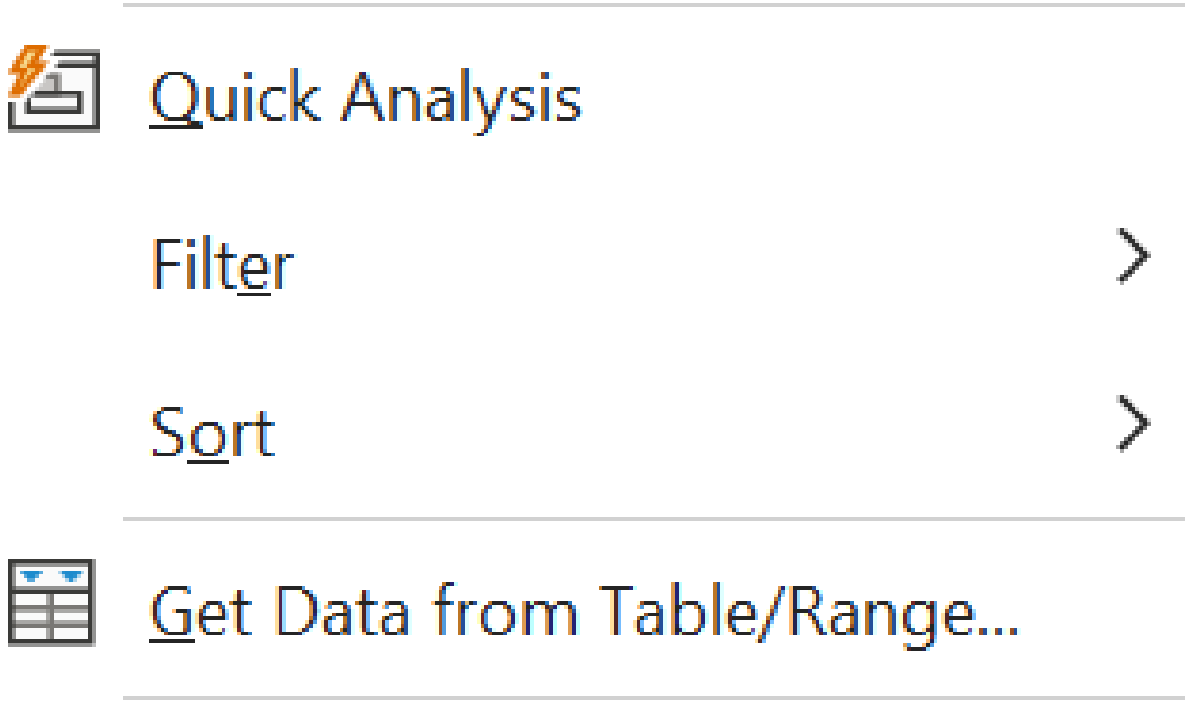

Figure 3.5: Different ways of doing Excel data analytics

You can start by choosing **Quick Analysis**, and a list of action items will show up for you to choose:

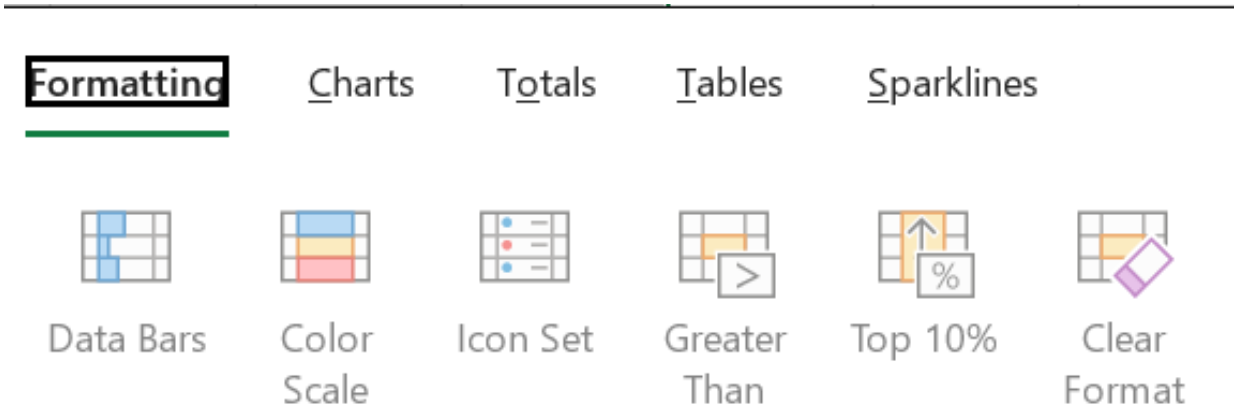

Figure 3.6: Excel Quick Analysis options

The **Quick Analysis** tool allows you to check the data characteristics easily. For example, you can choose the **Totals** tab and select **Sum**. A row of totals will show up at the bottom of the data.

1. For more in-depth analysis, you can also choose **Get Data from Table/Range…** from the menu in *Figure 3.5*. Once chosen, you will confirm the data selection:

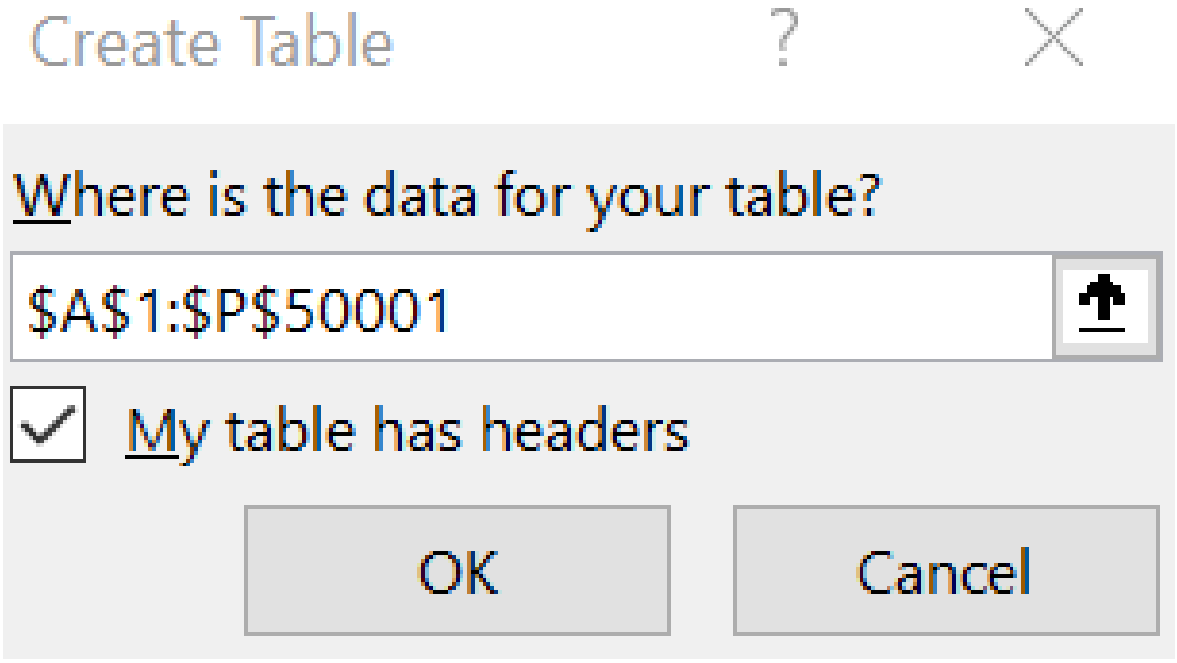

Figure 3.7: Exccel data selection confirmation

2. The Excel Power Query Editor will show up once you click the **OK** button. Excel Power Query is a built-in data transformation tool within Microsoft Excel that allows users to easily import, clean, and reshape data. It is beyond the scope of this book to discuss it in detail. This exercise will simply provide you with one example of what it can do for you. You will learn this by highlighting the **state** column and clicking on the **Group By** button on the top menu:

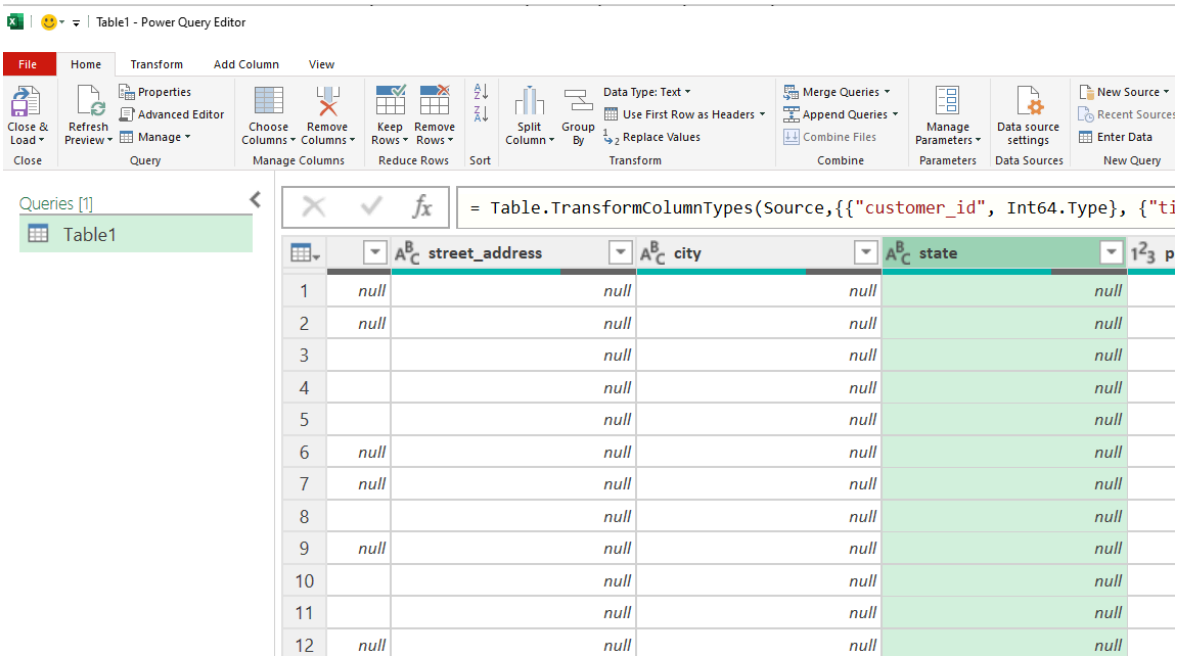

Figure 3.8: Group By State column in Excel Power Query Editor

3. Click on **OK** in the following option dialog box and you will get the `customer count by state` table. Click the **Close & Load** button in the upper-left corner. The table will be saved as a new sheet in the workbook.

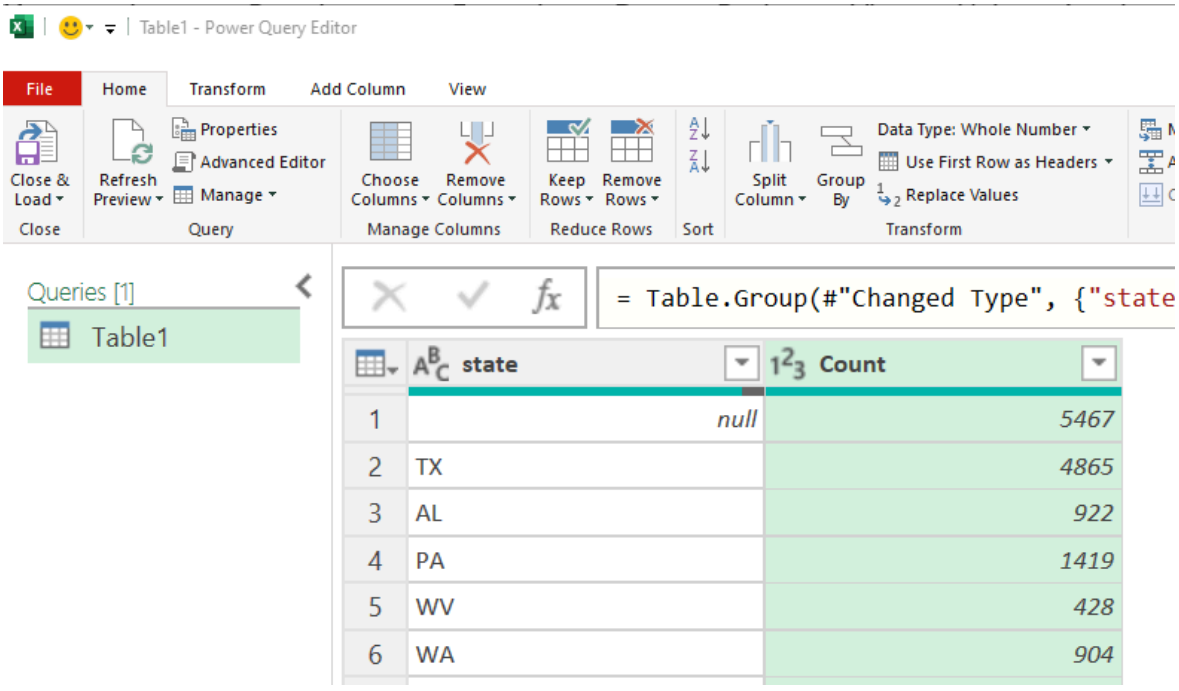

Figure 3.9: Aggregation table created by Excel Power Query

4. To further explore this dataset, you can highlight the dataset and choose the **Insert** menu. Click on the **2-D Column** button.

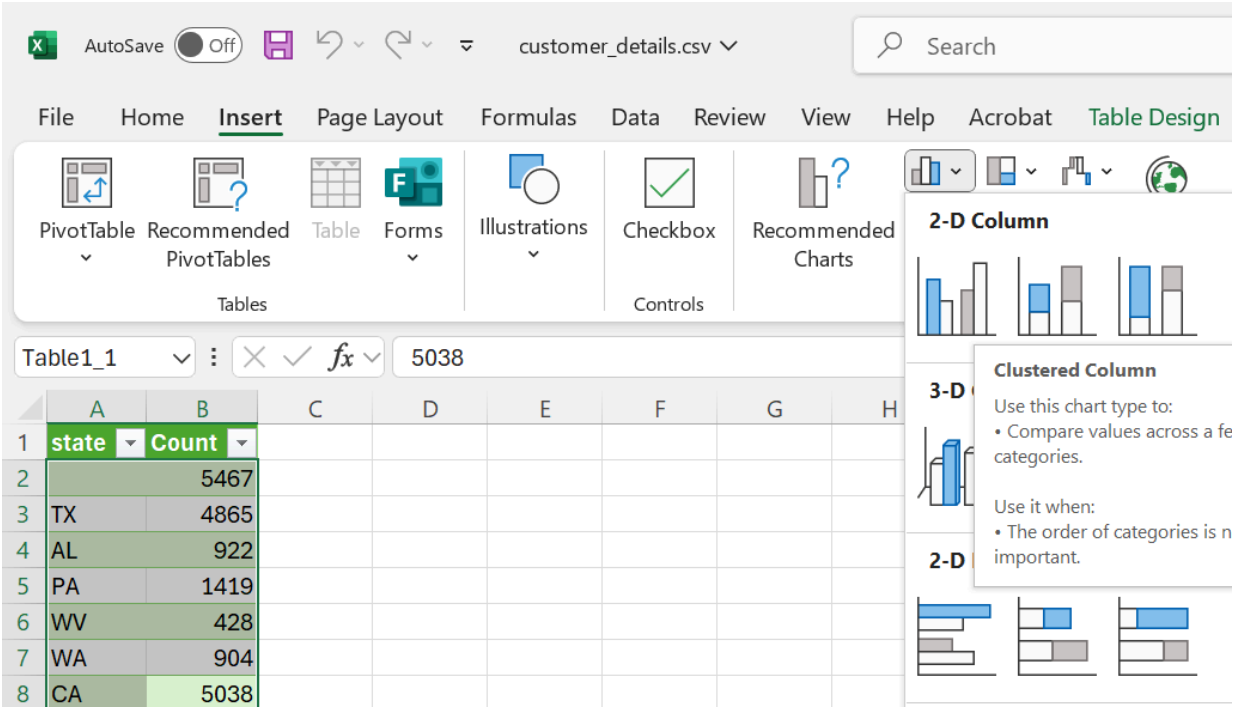

Figure 3.10: Column chart visualization for Excel data

Excel will provide you with a column chart showing the customer counts per state:

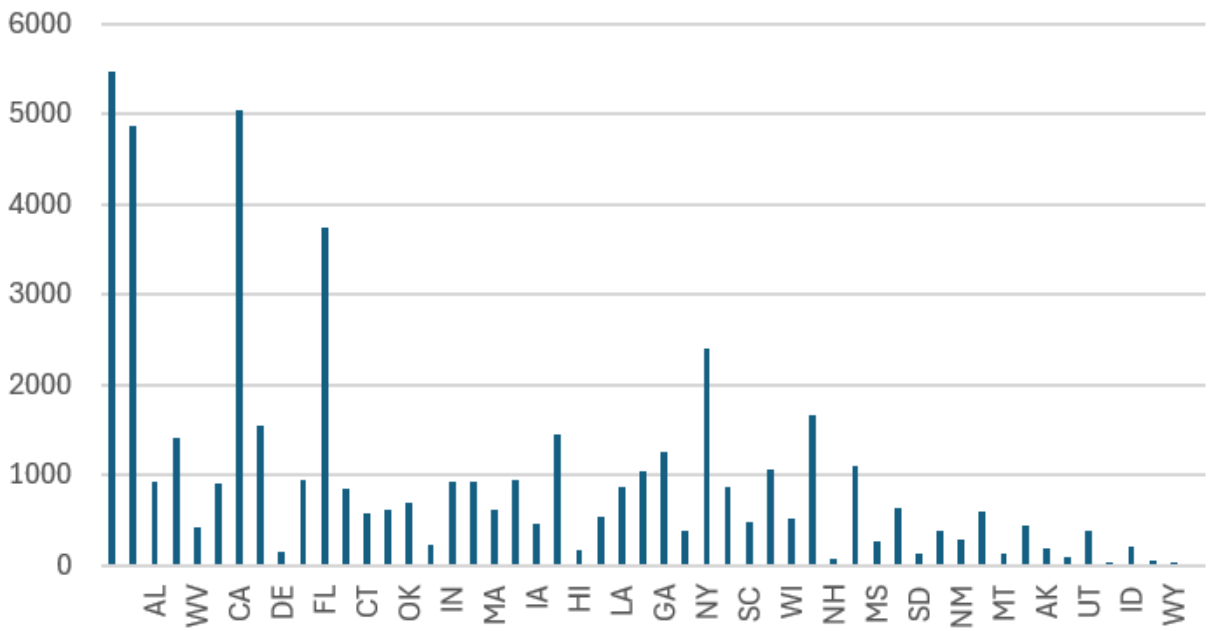

Figure 3.11: Customer counts per state chart

As you go through the importing and exporting processes in this chapter, you will see that they are very manual and require a lot of detailed specifications, such as file location, delimiter, and, implicitly, access to folders. In most enterprise environments, these tasks are completed by automated tools. Without these tools, the analytics

would not function smoothly. As a data scientist, normally, you will not be required to configure and maintain these tools. However, it is important to be aware of the sources and targets of your analysis.With everything you have learned in this chapter, you now have the power to communicate with external file systems from your database. In the following activity, you will use this power to help your business team perform data export/import tasks.

Activity 3

Your HR team is trying to conduct some analysis of salespersons. Per legal requirements, they can only use a small amount of the actual salespersons' profiles, and the names of the sample salespersons must be hashed (turning them into some unrecognized string, e.g., Tom to #!$, Jerry to *&@!@, etc.). They would like to save the data in the `salespeople` table in a CSV file, keep the first 10 rows, delete the rest of `salespeople`, and change the names of these 10 salespeople to hash code. Once this is done, load the data in the new CSV file into a new table, `salespeople_sample`. After you finish all these tasks, please confirm the changes to names have been applied and drop the table to save database storage.

Summary

In this chapter, you learned how to import/export your data in and out of a DBMS. You did this by transferring the data using psql and the `COPY` and `\COPY` commands. You then performed some data analytics against the files in Excel. This analysis could be useful for helping an executive to make a data-driven decision regarding where they should open their next retail location.With this knowledge, you will be able to utilize data from external sources by importing files into your database system and share your data with your customers by saving your data into files.But these are not the only things you can do to communicate with external tools. In the next chapter, you will look at how you can use external programming tools to manage your data.

4 Manipulating Data with Python

Join our book community on Discord


https://packt.link/EarlyAccessCommunity

So far, all the SQL statements and commands in this book have been executed through the client tools provided by PostgreSQL, either `pgAdmin` or `psql`. The user types the statements into the client tools, the tools forward the statements to the **Database Management System** (**DBMS**), the DBMS executes the statements and passes the result back to the tools, and the tools either display the data on the screen or save it as a file for further analytics.While `pgAdmin` and `psql` offer a breadth of functionalities, many data scientists and data analysts are starting to use Python as a helpful client tool, too. Python is a high-level language that can be easily used to process data. Just like `pgAdmin` and `psql`, companies can use Python to connect to the database, execute SQL statements for different stages of the CRUD lifecycle process, and get results back. In this usage scenario, Python serves a similar role to `psql`, in that it carries the SQL to the database from the user, retrieves data back from the database, and saves the data within Python objects. With the data stored in Python objects, you can also perform data manipulations and analytics, even data visualizations, inside its runtime, adding valuable analytical capacities to SQL. For these reasons, the usage of Python in the data analytics field is growing fast and has generally become one of the most important data analytics tools in recent polls. The other large advantage that Python has is that it is versatile. While SQL is generally only used in the data science and statistical analysis communities, Python can be used to do anything from statistical analysis to building a web application. As a result, it is very easy to expand the analytics into any practical usage field. As a result of these advantages, it is quite useful to learn Python as a data scientist.The following topics are covered in this chapter:

- Getting started with Python
- Managing data with Python

After you complete these topics, you will be able to set up a Python environment, communicate with the relational database, and manipulate data within that database.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in *Chapter 1*, *Introduction to Data Management Systems*. You should also have a basic understanding of the Python programming language and be able to set up a Python Anaconda environment on your local machine.

Getting started with Python

Python is usually not installed on the database server. Instead, it runs within the Python runtime on your workstation or on an application server. When you use Python for data analytics in PostgreSQL, you need to install

a specific library called `psycopg2` on your workstation or application server. This library, when called from the Python runtime environment, will connect to the PostgreSQL server and handle traffic between your Python script and the database server. In its simplest form, once you connect to the PostgreSQL server using `psycopg2`, you can submit SQL to the database using Python scripts, in the same way that you would with `psql`.While there are many ways to get access to Python, the Anaconda distribution of Python makes it particularly easy to obtain and install it along with other analytical tools, as it comes with many commonly used analytics packages preinstalled alongside a great package manager. For that reason, we will be using the Anaconda distribution in this book. In the following exercise, you will set up Anaconda on your machine.

Exercise 4.1: Setting up Python on your machine

You can take the following steps to get the Anaconda distribution set up and to connect to Postgres:

1. Download and install Anaconda from the official Anaconda site: https://www.anaconda.com/download. The Anaconda version for this book is 2024-06, which uses Python 3.13.5. There are different download options. It is generally recommended to use the **Distribution** download. The installation is quite straightforward. You can simply accept all the default settings.

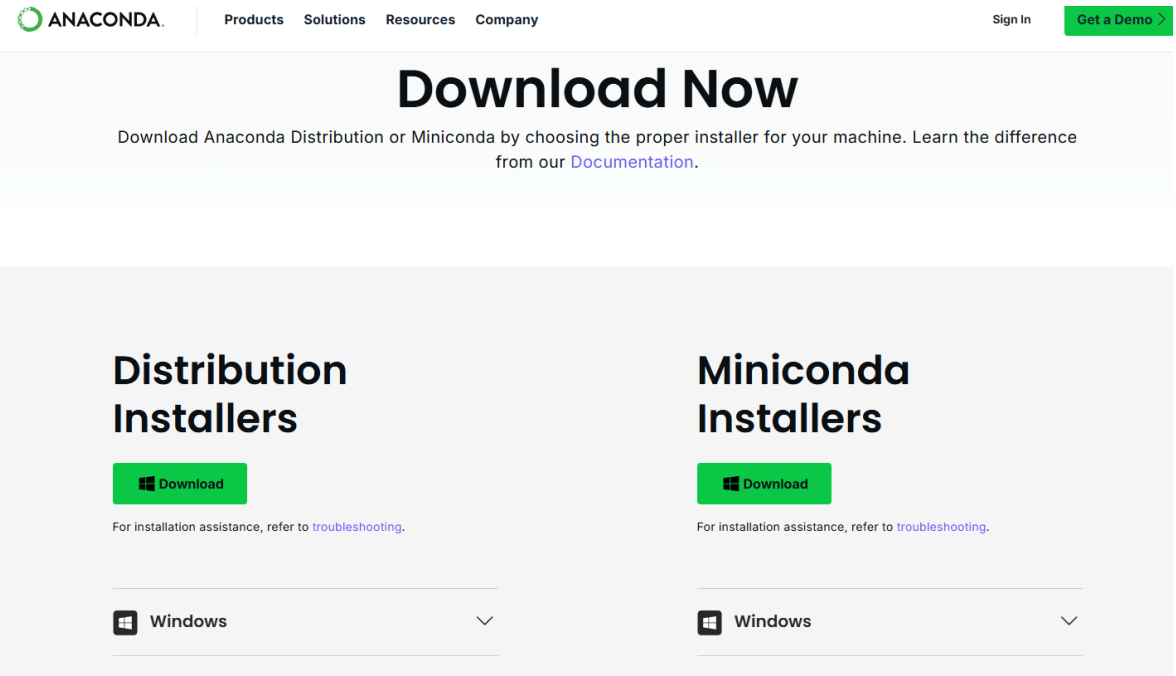

Figure 4.1: Anaconda Download page

2. Download and install the PostgreSQL database client for Python, `psycopg2`, using the Anaconda package manager:

    I. Open Anaconda Navigator. In the left panel, choose the **Environments** tab. Then, in the first drop-down box of the right panel, choose **All**.

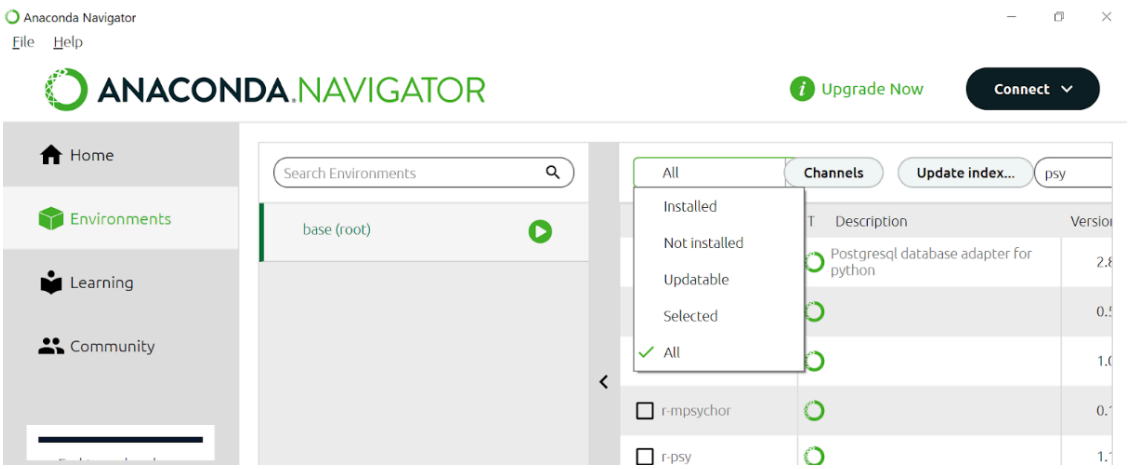

Figure 4.2: Accessing the Anaconda environment

II. Type `psy` in the search box. You will see a list of libraries, including `psycopg2`. By default, it is not installed, and the checkbox in front of its name is unchecked. Check the box and click on the **Apply** button in the bottom-right corner, as shown here:

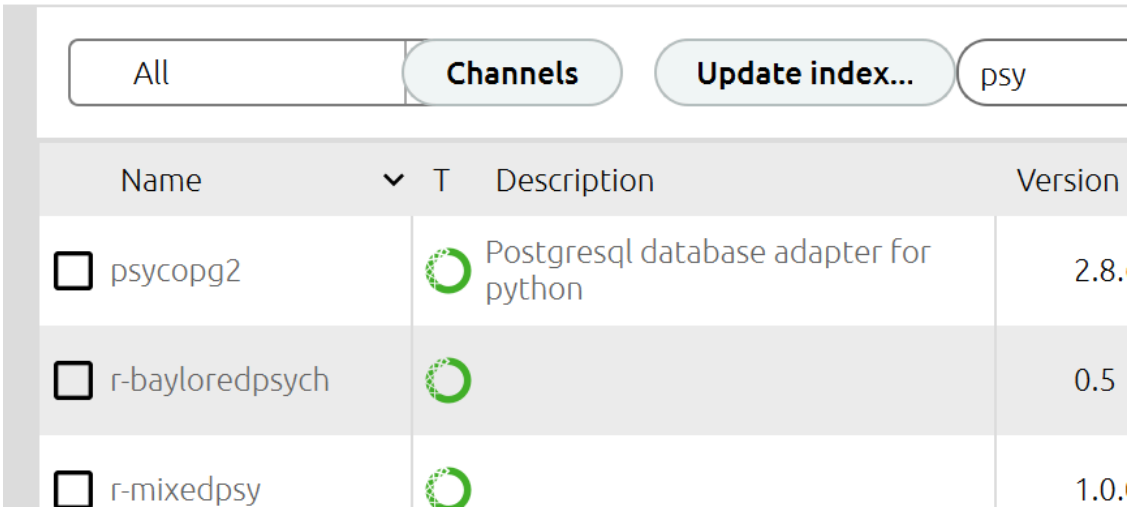

Figure 4.3: Installing the psycopg2 library

III. Follow the instructions for installation.

3. In the data analytics field, the most common way to run a Python program is to use Jupyter Notebook, which has been installed as part of the Anaconda installation. From the **Home** tab of Anaconda Navigator, find **Jupyter Notebook** and click on **Launch**. You should see the Jupyter Notebook environment pop up in your default browser:

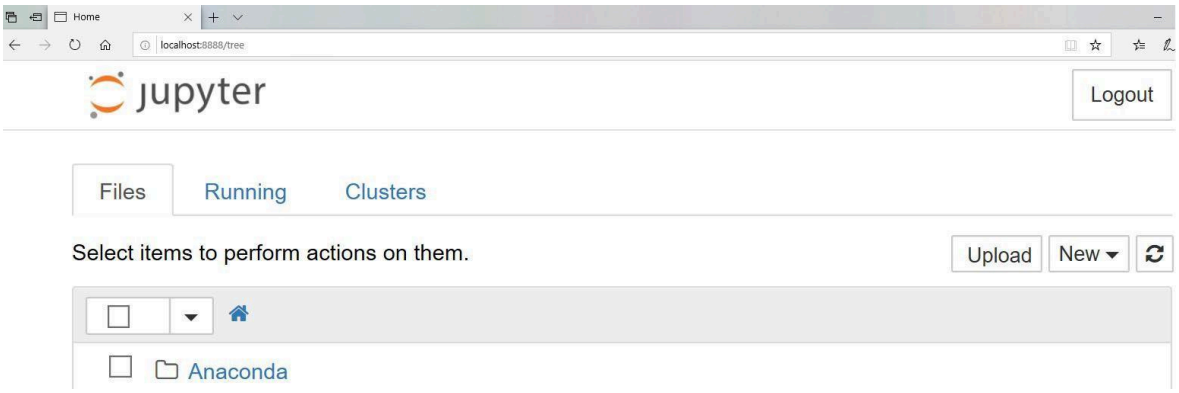

Figure 4.4: Starting the Jupyter Notebook environment

4. Create a new notebook by clicking the **New** drop-down button and choosing **Python** (it may come with a suffix for runtime version/environment). The notebook will be launched in another browser tab. Each notebook consists of multiple cells. Each cell contains some Python statements that will be executed, one cell at a time.

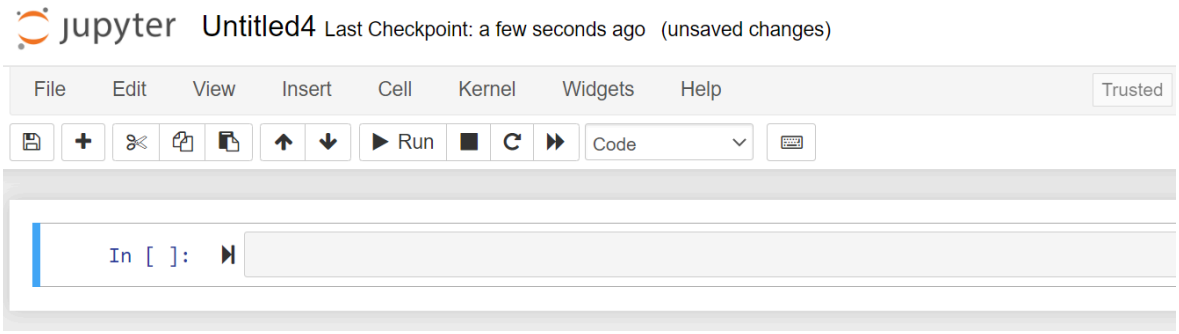

Figure 4.5: Jupyter Notebook user interface

5. The following code imports the `psycopg2` library and establishes the connection from your Python program (which is a client) to the database server specified in the parameters. Type the following Python script in the cell and click the **Run** button:

```
import psycopg2
conn = psycopg2.connect(
    host="localhost",
    user="postgres",
    password="<your superuser password>",
    dbname="sqlda",
    port=5432
)
```

Note

Python has strict rules on code format, especially code indentation. However, this indentation is not preserved in a PDF book. So if you want to get the code in this chapter, please copy it from the GitHub repository instead of from the book PDF.

1. As you finish running one cell, a new cell is created. Type the following code in the new cell and click on the **Run** button. This code creates a Python cursor that can send SQL statements to the database server and retrieve results:

```
cur = conn.cursor()
cur.execute("SELECT * FROM customers LIMIT 2")
```

```
records = cur.fetchall()
print(records)
```

You can see that this Python code executes the SQL inside the `execute` function and brings back the result. The output of this command is a Python list containing rows from your database as tuples. Note that the exact customers you get may vary:

```
[(716, None, 'Jarred', 'Bester', None, 'jbesterjv@nih.gov', 'M', '216.51.110.28', None, None, Nor
```

You may wonder how this is different from running the same SQL from `psql`. The power of Python in data analytics, as well as other programming languages, lies in the fact that inside a Python program, you can build the data operation process based on object models and organize the process into modules, which generally have more functionalities.In the next section, you will learn how to use some of the other packages in Python to facilitate interfacing with the database.

Managing data with Python

While `psycopg2` is a powerful database client for accessing PostgreSQL from Python, it is just a connector. It does nothing more than pass the SQL and the resulting data between your program and the database server. If you only rely on `psycopg2` for connection, you will very soon get lost in the swamp of SQL statements embedded in Python scripts. Fortunately, you can enhance the code by using a couple of other packages—namely, pandas and SQLAlchemy. While both packages are powerful, it is worth noting that they still use the `psycopg2` package to connect to the database and execute queries. The big advantage that these packages provide is that they abstract some of the complexities of connecting to and working with the database, while dedicating the connectivity handling to the `psycopg2` package. By abstracting these complexities, you can connect to the database without worrying that you might forget to close a connection or remove a Python object such as a cursor.In the next section, you will learn about SQLAlchemy, a Python SQL toolkit that maps representations of objects to database tables. You'll get familiar with the SQLAlchemy database engine and some of the advantages that it offers. This will enable you to access a database seamlessly without worrying about connections and Python objects.

What is SQLAlchemy?

SQLAlchemy is a Python SQL toolkit and **object-relational mapper** (**ORM**) that maps representations of objects to database tables. An ORM builds up mappings between SQL tables and programming language objects: in this case, Python objects. SQLAlchemy by itself is not SQL; rather, it is a toolkit helping SQL communicate between your Python code and the database.For example, in the following figure, there is a `Customer` table in the database. The Python ORM will thus create a class called `customer` and keep the content in the object synchronized with the data in the table. For each row in the `Customer` table, a `customer` object will be created inside the Python runtime. When there are changes (inserts, updates, and/or deletes), the ORM can initialize a sync and make the two sides consistent.

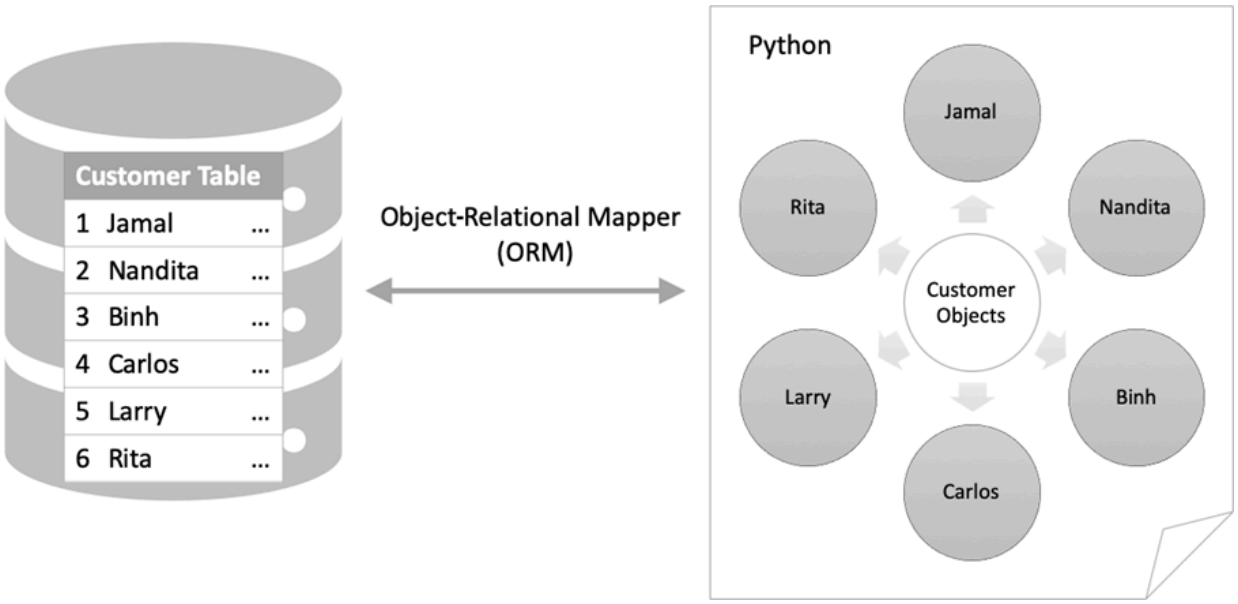

Figure 4.6: An ORM maps rows in a database to objects in memory

While the SQLAlchemy ORM offers many great functionalities, its key benefit is the engine object. A SQLAlchemy engine object contains information about the type of database (in your case, PostgreSQL) and a connection pool. The connection pool allows multiple connections to the database that operate simultaneously. The connection pool is also beneficial because it does not create a connection until a query is sent to be executed. Because these connections are not formed until the query is executed, the engine object is said to exhibit lazy initialization. The term "lazy" is used to indicate that nothing happens (the connection is not formed) until a request is made. This is advantageous because it saves the resources that Python needs to establish and maintain the connection and reduces the load on the database.Another advantage of the SQLAlchemy engine object is that it automatically commits changes to the database due to `CREATE TABLE`, `UPDATE`, `INSERT`, and other statements that modify a database. This will help the data in the database and the data in Python to be synchronized all the time.In your case, you will want to use SQLAlchemy because it provides a robust engine object to access databases. If the connection is dropped, the SQLAlchemy engine object can instantiate that connection because it has a connection pool. It also provides a nice interface that works well with other packages (such as pandas).Next, you will learn about pandas—a Python package that can perform data manipulation and facilitate data analysis. The pandas package will help you represent your data table structure (called a DataFrame) in memory. pandas also has high-level APIs that will enable you to read data from a database in just a few lines of code.

Using Python with SQLAlchemy and pandas

Normally, SQLAlchemy and pandas come together with Anaconda. When you install Anaconda on your machine, you have already set them up. However, if you are not sure about the installation, you can open Anaconda Navigator and go to the **Environments** tab. From there, you can verify the installed packages and install them if they are not there. From the same location, you can also install the packages referenced later in this book, such as `matplotlib`, if necessary.Now, in Anaconda, create a new notebook. Enter the following `import` statements in the first cell:

```
from sqlalchemy import create_engine, text
import pandas as pd
```

You are importing three packages here. The first two are the `create_engine` module and the `text` module within the `sqlalchemy` package, which correspond to the functionalities related to database connectivity and statement execution, and the third is pandas, which handles the analytics functionalities. You will rename pandas to `pd` following the standard convention. Using these three packages, you will be able to read and write data to and from your database and perform analytical tasks.Once you run the `import` code, Jupyter Notebook will move into the

next cell. In the next cell, configure your notebook to display plots and visualizations inline. You can do this with the following command:

```
%matplotlib inline
```

This tells the `matplotlib` package (which is a dependency of pandas) to create plots and visualizations inline in your notebook. Run this cell to move into the next cell. In the new cell, you will define your connection string:

```
cnxn_string = (
    "postgresql+psycopg2://{username}:{pswd}@{host}:{port}/{database}"
)
print(cnxn_string)
```

Run this cell to see how this connection string was printed. This is a generic connection string for `psycopg2`. You need to fill in your parameters to create the database engine object. You can replace the parameters using the parameters that are specific to your connection. The particular parameters corresponding to the setup of this book are as follows:

```
engine = create_engine(
    cnxn_string.format(
        username="postgres",
        pswd="your_password",
        host="localhost",
        port=5432,
        database="sqlda"
    )
)
```

Once the engine is created, you can run SQL statements against the database through the engine object:

```
with engine.connect() as conn:
  result = conn.execute(
    text("SELECT * FROM customers LIMIT 2")
  ).fetchall()
print(result)
```

The preceding code first connects to the database engine, then executes a `text` packaged SQL statement. Compared to how SQL was executed via `psycopg2` in earlier sections, you can see that SQLAlchemy adds a level of isolation around connectivity and execution. It may seem redundant in a small setup, but it will definitely make the system architecture simpler when it comes to large production implementations.The output of this command is a Python list containing rows from your database as tuples. While you have successfully read data from your database, you will probably find it more practical to read your data into a pandas data frame in the next section.

Reading and writing to a database with pandas

Python comes with great data structures, including lists, dictionaries, and tuples. While these are useful, it is usually more useful to represent the data in table form, with rows and columns, like how you would store data in your database. For this purpose, the DataFrame object in pandas can be particularly useful. In addition to providing powerful data structures, pandas also offers the following:

- Functionality to read data directly from a database
- Data visualization
- Data analysis tools

If you continue from where you left off with your Jupyter notebook, you can use the SQLAlchemy engine object to read data into a pandas DataFrame:

```
customers_data = pd.read_sql_table('customers', engine)
```

You have now stored your entire `customers` table as a pandas DataFrame in the `customers_data` variable. The pandas `read_sql_table` function requires two parameters: the name of a table and the connectable database (in this case, the SQLAlchemy engine object). Alternatively, you can use the `read_sql_query` function, which takes a

query string instead of a table name.Now that you know how to read data from the database, you can start to do some basic analysis and visualization. In essence, a pandas DataFrame is a relational table with enhanced information. You can apply similar operations in SQL on the DataFrame, such as querying, inserting, filtering, and deleting. For example, the `head()` method returns the first few (default of `5`) rows of the DataFrame, much like the `LIMIT` clause in SQL.Running the following Python code will yield the first five rows of the `customers_data` DataFrame:

```
customers_data.head()
```

This will result in the following output (note that your result may vary, but should have the same columns):

```
customers_data.head()
```

	customer_id	title	first_name	last_name	suffix	email	gender
0	716	None	Jarred	Bester	None	jbesterjv@nih.gov	M
1	1228	None	Ag	Smerdon	None	asmerdony3@house.gov	F
2	1876	None	Giuditta	Eim	None	geim1g3@typepad.com	F
3	1991	None	Nichole	Rosle	None	nrosle1ja@ning.com	M
4	2275	None	Chic	Bryning	None	cbryning1r6@pcworld.com	M

Figure 4.7: pandas DataFrame

You can also perform many more operations, such as pivoting, multi-column search and replacement, and semi-structured data parsing, which are not possible or are very difficult to achieve using SQL. The full functionalities of pandas are beyond the scope of this book, but we will demonstrate some basic ones in the upcoming exercises.As you complete your data manipulation and analysis, eventually, you would like to save the data to a relational table or file. pandas also provides this functionality, which will be discussed in the next subsection.

Writing data to the database from Python

The last step in any production data usage scenario is to use Python to write data back to the database. Luckily, pandas and SQLAlchemy make this a relatively easy task.If you have your data in a pandas DataFrame, you can write data back to a database table using the pandas `to_sql(…)` function, which requires two parameters: the name of the table to write to and the connection. Best of all, the `to_sql(…)` function can also create the target table for you by inferring column types using a DataFrame's data types. In the coming exercise, *Exercise 4.02: Reading, visualizing, and saving data in Python*, you will test out this functionality using the `products_new` DataFrame that you created.

Exercise 4.2: Reading, visualizing, and saving data in Python

In the previous chapter's exercise, you executed a SQL query in `psql` to get a list of customers. Then, you dumped the result into a CSV file using the `COPY` command and sent the file to the business department. Upon

receiving the file, they created a visualization on top of the CSV file in Excel and copied and pasted the visualization into a Microsoft PowerPoint slide file for presentation.Looking at this process, you can see that there is a lot of manual work and coordination between different applications. The whole process involves three applications: `psql`, Excel, and PowerPoint, and the copy and paste activities between Excel and PowerPoint must be carried out by a human being.In this exercise, you will analyze the same demographic information of customers by their state to help the business analysts by reading data from the database output and visualizing the results using Python with Jupyter Notebook, SQLAlchemy, and pandas. You will run the SQL inside Python, retrieve data in a pandas DataFrame, and create a visualization inside Jupyter Notebook. All this is automated. There will not be a need to pass files and clipboards between applications. The following steps are involved:

1. Open Anaconda Navigator and launch Jupyter Notebook. Create a new notebook.
2. Run the following code in the first cell. The code imports the required libraries into the Python runtime:

```
from sqlalchemy import create_engine, text
import pandas as pd
```

1. The next cell establishes a connection to the database server. You will need to adjust this code based on your server's setup:

```
cnxn_string = (
    "postgresql+psycopg2://{username}:{pswd}@{host}:{port}/{database}"
)
engine = create_engine(
    cnxn_string.format(
        username="postgres",
        pswd="my_password",
        host="localhost",
        port=5432,
        database="sqlda"
    )
)
query = "SELECT * FROM customers"
customer_data = pd.read_sql_query(query, engine)
customer_data.head()
```

Here, the `customers` table's content is uploaded into the `customer_data` DataFrame.

1. pandas provides a rich set of tools that you can utilize for data analysis. For example, if you want to see the number of customers per state, you can first select the `state` and `customer_id` columns into a subset, group the rows by state, use the `count()` functions to count customers by state, and save the result into another DataFrame:

```
customer_per_state = \
    customer_data[['state', 'customer_id']] \
        .groupby('state').count()
customer_per_state.head(3)
```

1. The next cell sets up the `matplotlib` environment for drawing. `matplotlib` is a Python library that is widely used for data visualization, that is, drawing charts based on data. It comes together with pandas and should have already been installed in your environment. Running this command allows `matplotlib` to output the visualization directly to Jupyter Notebook:

```
%matplotlib inline
```

1. With the display setup, you will use the `matplotlib` package to draw a bar chart in the notebook based on the customer per state data that we obtained in *Step 5*:

```
import matplotlib.pyplot as plt
customer_per_state.plot(kind='bar')
plt.show()
```

The result in the notebook looks like this:

```
import matplotlib.pyplot as plt
customer_per_state.plot(kind='bar')
plt.show()
```

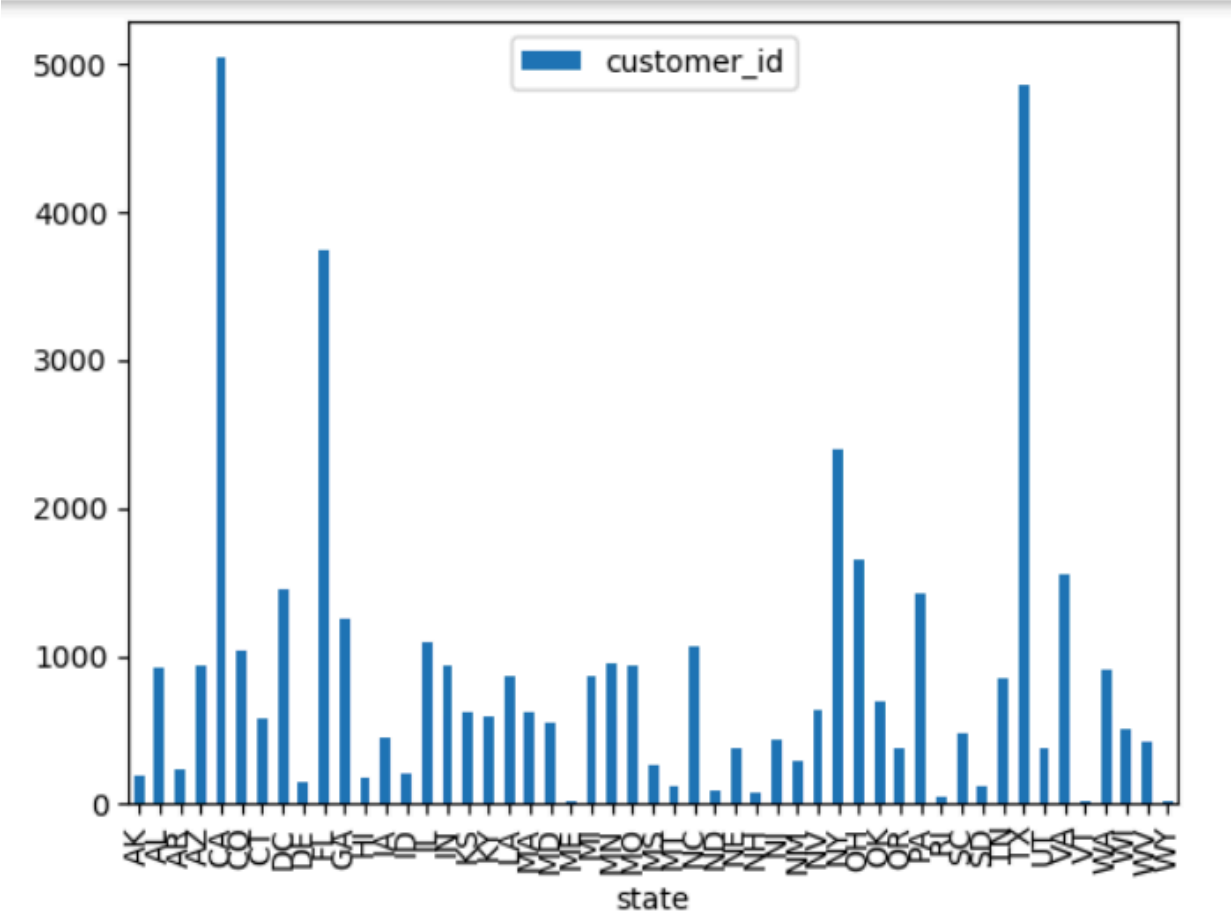

Figure 4.8: Visualization using Matplotlib

1. Once the desired result is received, save the data in a `customer_per_state` table by running the following code:

```
customer_per_state.to_sql(
    'customer_per_state',
    engine,
    if_exists='replace'
)
```

You should see the following output, which indicates the table creation is successful and 51 rows have been saved in the new table:

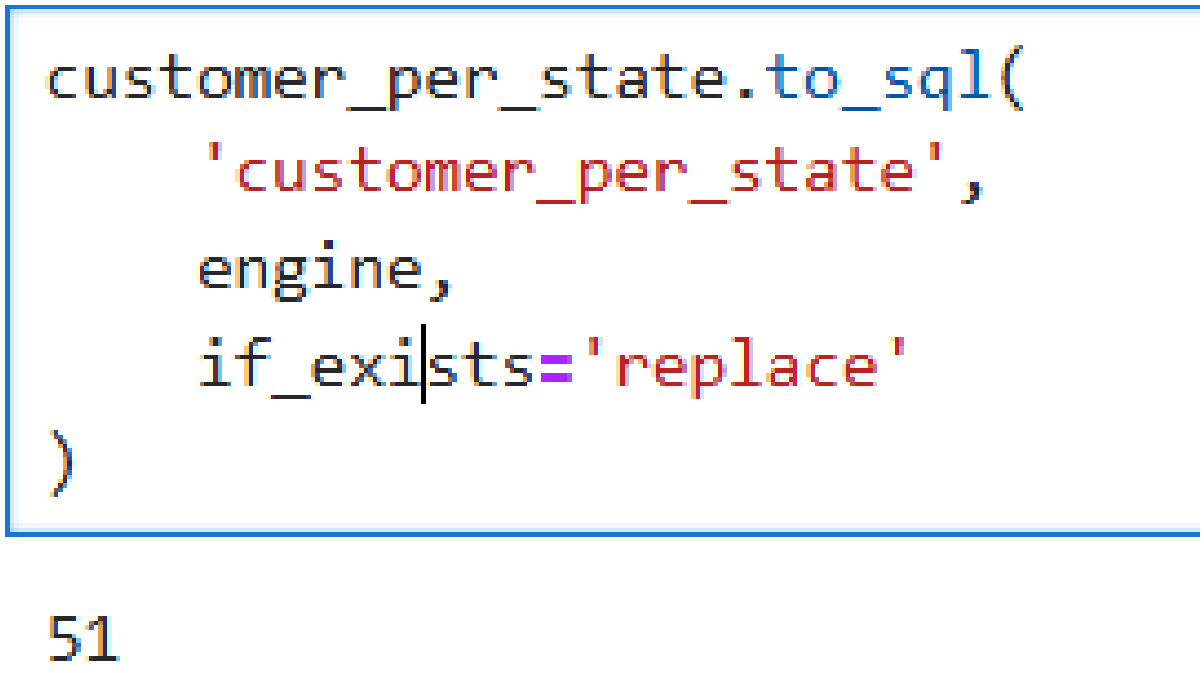

Figure 4.9: Saving a pandas DataFrame to a PostgreSQL table

1. Finally, you can log into `psql` and run a `SELECT` statement to confirm the existence of this new table, then run a `DROP` statement to remove it from the database:

```
SELECT * FROM customer_per_state LIMIT 5;
DROP TABLE customer_per_state;
```

There are certainly improvements that you can make to the visual effect of the preceding bar chart, but this exercise should have provided you with a clear workflow of how to use Python for relational database operations. You should be able to apply what you have learned in this chapter in the following activity to help your team perform product analysis.

Activity 4

Having mastered the data operation functionalities of Python, your team would like to explore the different powers that a pandas DataFrame can provide. You now need to fetch the products from the `products` table, and use a filter to get products that `production_start_date` falls in 2024. Once you get this data, save the result to a table called `products_2024`. Finally, as you complete all these, query the content of the new table to confirm the data has been properly loaded and drop the table to keep the database clean.

Summary

This chapter provided a comprehensive guide on using Python for data management and analytics with PostgreSQL, including setting up the environment, connecting to the database, and performing data operations and visualizations using libraries such as `psycopg2`, SQLAlchemy, and pandas. With these topics, you will be able to set up a Python environment, communicate with the relational database, and manipulate data within that database.From *Chapter 1*, *Introduction to Data Management Systems*, to this chapter, you have learned about the architecture and lifecycle of the **relational database management system** (**RDBMS**). You have also learned about the tools used to access and transfer data.Now that the system is operational and data can be properly saved in the databases, you will start to learn how to manipulate and utilize the data using SQL. Starting from the next chapter, you will use SQL statements for data retrieval and modification, and complex analytics.

5 Presenting Data with SELECT

Join our book community on Discord


https://packt.link/EarlyAccessCommunity

In previous chapters, you learned about the basic lifecycle of data operations. You are now able to create and drop tables and insert data into tables. You also learned how to import data into a relational database as well as export it. All these operations complete the starting and ending operations of the following CRUD lifecycle:

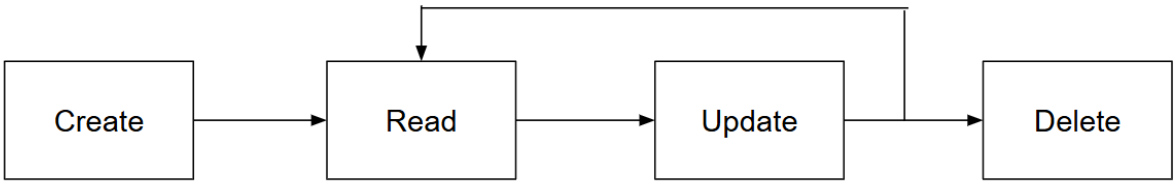

Figure 5.1: CRUD operations

The ultimate goal of data management is to utilize the data to generate analytical information. Reading the stored data is the fundamental step in this process. In this chapter, you will start to learn how to read data from existing tables. In the next chapter, you will learn further operations for data transformation and about the update stage of the lifecycle.The following topics are covered in this chapter:

- Using `SELECT` expressions:
  - Expression alias
  - The `LIMIT` clause
  - The `ORDER BY` clause
  - The `DISTINCT` and `DISTINCT ON` functions
- Filtering query results:
  - The `AND` / `OR` and `NOT` clauses
  - The `IN` / `NOT IN` clauses
  - The `IS NULL` / `IS NOT NULL` clauses

After learning about these topics, you will be able to extract data from a database via SQL, with conditions and selections applied.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in *Chapter 1*, *Introduction to Data Management Systems*.

Using SELECT expressions

So far, you have used the most basic `SELECT` query of this pattern:

```
SELECT * FROM <table_name>;
```

This query pulls all rows from a single table. For each row, all columns are displayed. This can be used to check what data exists in the table. However, selecting all columns blindly is usually not necessary and can cause significant performance issues. Many times, you can specify the list of columns by replacing the `*` sign with a list of column names, such as the following:

```
SELECT product_id, model, base_msrp FROM products;
```

This will return the `product_id`, `model`, and `base_msrp` information of all the products in this table. You can arrange the columns in any order, regardless of the order in the table definition. PostgreSQL will display the columns based on the order in the `SELECT` statement:

```
product_id |         model         | base_msrp
-----------+-----------------------+-----------
          1 | Lemon                 |    399.99
          2 | Lemon Limited Edition |    799.99
          3 | Lemon                 |    499.99
          5 | Blade                 |    699.99
          7 | Bat                   |    599.99
          8 | Bat Limited Edition   |    699.99
         12 | Lemon Zester          |    349.99
          4 | Model Chi             | 115000.00
          6 | Model Sigma           |  65500.00
          9 | Model Epsilon         |  35000.00
         10 | Model Gamma           |  85750.00
         11 | Model Chi             |  95000.00
```

It is important to understand that the `SELECT` query is applied to all rows in the table. It will be applied to each row, regardless of the execution of other rows. This explains a common confusion when running the following SQL:

```
SELECT 1 FROM products;
```

Many beginners think this statement will return a single `1`. In reality, the `SELECT 1` operation will be applied to all 12 rows in the `products` table. The result of this operation (which is `1`) will occur 12 times. As such, you will see 12 rows of `1` returned:

```
?column?
----------
        1
        1
        1
        1
        1
        1
        1
        1
        1
        1
        1
        1
(12 rows)
```

You can select not only the columns from tables but also the result of calculations of columns, either against a constant or between columns. This is called an **expression**. For example, the following expression applies a 90% discount to the `base_msrp` price of products (i.e., it is multiplied by 0.9):

```
SELECT model, base_msrp * 0.9 FROM products;
```

This results in the following:

```
         model         |  ?column?
-----------------------+------------
 Lemon                 |    359.991
 Lemon Limited Edition |    719.991
 Lemon                 |    449.991
 Blade                 |    629.991
 Bat                   |    539.991
 Bat Limited Edition   |    629.991
 Lemon Zester          |    314.991
 Model Chi             | 103500.000
 Model Sigma           |  58950.000
 Model Epsilon         |  31500.000
 Model Gamma           |  77175.000
 Model Chi             |  85500.000
(12 rows)
```

What you get is the original price multiplied by 0.9, which is applied to the price in each row.The preceding result shows the expression value, but the expression header is `?column?`. This is because you have not assigned a name to the expression. In the next subsection, you will learn how to specify a name for this expression.

Expression alias

In the `SELECT` statement, you can use the `AS` keyword to assign a meaningful name to an expression. For example, the following statement assigns the name `Discounted_Price` to the calculation result:

```
SELECT model, base_msrp * 0.9 AS Discounted_Price
FROM products;
```

This is the first few rows returned:

```
         model         | discounted_price
-----------------------+------------------
 Lemon                 |          359.991
 Lemon Limited Edition |          719.991
 Lemon                 |          449.991
 Blade                 |          629.991
```

An alias can be assigned to any expression. This includes the calculation of columns, a single column, and constant values. Also, the `AS` keyword can be omitted. Here is an example and the first few rows of its result:

```
SELECT
  Model Product_Name,
  0.9 Discount,
  base_msrp * 0.9 Discounted_Price
FROM products;
```

This is how it displays it:

```
     product_name      | discount | discounted_price
-----------------------+----------+------------------
 Lemon                 |      0.9 |          359.991
 Lemon Limited Edition |      0.9 |          719.991
 Lemon                 |      0.9 |          449.991
```

Now that you have learned how to customize the columns that you want to view, the next step is to customize the rows that you want to view. You will start by limiting the number of results first.

The LIMIT clause

Most tables in SQL databases tend to be quite large, and, therefore, returning every single row is unnecessary. Sometimes, you may want only the first few rows. For this scenario, the `LIMIT` keyword comes in handy. You have already tried the following query several times in previous chapters:

```
SELECT * FROM products LIMIT 5;
```

When you are not familiar with a table or query, it is a common concern that by running a `SELECT` statement, you will bring back too many rows, which can take up a lot of time and machine bandwidth. As a common precaution, you want to use the `LIMIT` keyword to only retrieve a small number of rows to the client tool when you run the query for the first time.However, it is worth noting that the `LIMIT` keyword will not reduce the query workload on the server. It simply reduces the number of rows transferred from the DBMS to the client tool. The DBMS will still run against the whole dataset. Hence, it will not help the server's performance optimization. Only the client tool benefits from the row reduction.The next common challenge is how the rows are organized. As stated before, relational operations do not enforce ordering. So, if you want to see the result sorted per your instruction, you will need to explicitly ask SQL to do so, which is the topic of the next subsection.

The ORDER BY clause

SQL queries will *not* order rows. They will display the data in the order that they have received (that is, in the order in which the database finds the data) if they are not given specific instructions to do otherwise. For many use cases, this is acceptable. However, you will often want to see rows in a specific order.For instance, say you want to see all the products listed by the date when they were first produced, from earliest to latest. The method for doing this in SQL would be using the `ORDER BY` clause, as follows:

```
SELECT model, production_start_date
FROM products
ORDER BY production_start_date
LIMIT 5;
```

As shown in the following output, the products are ordered by the `production_start_date` field:

```
         model         | production_start_date
-----------------------+-----------------------
 Lemon                 | 2015-10-28 00:00:00
 Lemon Limited Edition | 2016-08-30 00:00:00
 Lemon                 | 2018-12-27 00:00:00
 Blade                 | 2020-02-17 00:00:00
 Model Chi             | 2020-02-17 00:00:00
(5 rows)
```

The column used in `ORBER BY` must be from the underlying table, but does not need to be in the `SELECT` clause. If an order sequence is not explicitly mentioned, the rows will be returned in ascending order. Ascending order simply means the rows will be ordered from the smallest value to the highest value of the chosen column or columns. In the case of text, that means arranging in alphabetical order. You can make the ascending order explicit by using the `ASC` keyword. For example, the last query can be achieved with the following:

```
SELECT model FROM products
ORDER BY production_start_date ASC
LIMIT 5;
```

This SQL will return in the same order as the preceding SQL.If you want to extract data in descending order, you can use the `DESC` keyword. If you wanted to fetch product models manufactured from newest to oldest, you would write the following query:

```
SELECT model FROM products
ORDER BY production_start_date DESC
LIMIT 5;
```

The result will be sorted by descending order of `production_start_date` , latest first.Also, instead of writing the name of the column you want to order by, you can refer to the position number of that column in the query's

`SELECT` clause. For instance, say you wanted to return all the models in the `products` table ordered by product ID. You could write the following:

```
SELECT product_id, model FROM products
ORDER BY product_id;
```

However, because `product_id` is the first column in the `SELECT` statement, you could instead write the following:

```
SELECT product_id, model FROM products
ORDER BY 1;
```

This SQL will return the same result as previously.Finally, you can order by multiple columns by adding additional columns after `ORDER BY`, separated with commas, optionally with `ASC` or `DESC` for each. For instance, you want to order all the rows in the table first by the year of the model from newest to oldest, and then by the **manufacturer's suggested retail price** (**MSRP**) from least to greatest. You would then write the following query:

```
SELECT * FROM products
ORDER BY year DESC, base_msrp ASC;
```

You can also use the position instead of column names here:

```
SELECT * FROM products
ORDER BY 3 DESC, 5 ASC;
```

The result is the same.With `LIMIT` and `ORDER BY`, you are able to bring back data as is. There are times, however, that you need to manipulate the data for display. The first of these scenarios is to get the distinct values from tables, which will be discussed in the next subsection.

The DISTINCT and DISTINCT ON functions

Looking at the previous results, you can see that some product names are identical. When looking through a dataset, you may be interested in determining the unique values in a column or group of columns. This is the primary use case of the `DISTINCT` keyword.For example, there are two models in the `products` table that are called `Lemon`. If you wanted to know all the unique models in the `products` table, you could use the following query:

```
SELECT DISTINCT model FROM products;
```

This should give the following result:

```
          model
-----------------------
 Model Epsilon
 Model Chi
 Model Gamma
 Lemon
 Lemon Limited Edition
 Lemon Zester
 Bat Limited Edition
 Bat
 Model Sigma
 Blade
(10 rows)
```

Here, the model name `Lemon` only shows up once. You can also use it with multiple columns to get all the distinct column combinations present. For example, to find all distinct years and what product types were released for those model years, you can simply use the following:

```
SELECT DISTINCT year, product_type FROM products;
```

This should give the following output:

```
year | product_type
------+--------------
 2025 | automobile
 2020 | automobile
 2016 | scooter
 2025 | scooter
 2023 | automobile
 2019 | scooter
 2021 | automobile
 2020 | scooter
 2022 | scooter
 2017 | scooter
 2023 | scooter
(11 rows)
```

Another keyword related to `DISTINCT` is `DISTINCT ON`, which allows you to ensure that only one row is returned for each value of an expression. This value expression can be returned with the first value of other columns. The general syntax of a `DISTINCT ON` query is as follows:

```
SELECT DISTINCT ON (distinct_column)
  distinct_column,
  column_1,
  column_2, …
FROM table
ORDER BY order_column;
```

Here, `distinct_column` is the column(s) you want to be distinct in your query. Then, `column_1`, `column_2`, and so on are the columns you want in the query, and `order_column` allows you to determine the first row that will be returned for a `DISTINCT ON` query if multiple columns have the same value for `distinct_column`. The first column of `order_column` should be `distinct_column`. If an `ORDER BY` clause is not specified, the first row will be decided randomly.For example, you want to get a unique list of dealerships with their most senior salesperson. This query would look as follows:

```
SELECT DISTINCT ON (dealership_id)
  dealership_id, first_name, last_name
FROM salespeople
ORDER BY dealership_id , hire_date;
```

Here, the output rows are ordered by `dealership_id` and `hire_date`. For each distinct dealership, all the salespeople are sorted based on `hire_date`, and the first row is the one with the earliest `hire_date`, which means it belongs to the most senior salesperson for each dealership. The `DISTINCT ON` clause will select the first row, the most senior salesperson, to match each value in the `dealership_id` column. Thus, for each dealership, you got the first name and last name of the most senior salesperson. It should return this output (first few rows):

```
 dealership_id | first_name | last_name
---------------+------------+-----------
             1 | Dukie      | Oxteby
             2 | Adrianne   | Otham
             3 | Benn       | Jakobssen
             4 | Benji      | Balsillie
             5 | Hewie      | Hutcheon
```

This table now guarantees that every row has a distinct `dealership_id`. If there are multiple users within the same dealership, the salespeople who were hired first by the dealership will be pulled by the query.The discussion so far in this chapter has focused on getting data without filtering based on certain criteria. In the next section, you will learn how to filter out the rows that you don’t need.

Filtering query results

In most usage scenarios, you will only need to analyze a part of the data that meets a certain criterion, not the entire table. So, it makes sense to apply a filter condition to the `SELECT` statement and only fetch the data that meets your needs. The `WHERE` clause is a piece of conditional logic that limits the amount of data returned. You can use the `WHERE` clause to specify conditions based on which the `SELECT` statement will retrieve specific rows. In a

`SELECT` statement, you will usually find this clause placed after the `FROM` clause.The condition in the `WHERE` clause is generally a Boolean statement that can be either `True` or `False` for every row. For numeric, text, and datetime columns, these Boolean statements can use equals (`=`), greater than (`>`), greater than or equal to (`>=`), less than (`<`), or less than or equal to (`<=`) operators to compare the columns against a value. You can also use the "not equal" operator. There are two equivalent not-equal operators, `<>` and `!=`. You can use either based on your personal preferences.For example, say you want to see the model names of the products with the model year of 2017 from the sample dataset. You would write the following query:

```
SELECT model FROM products WHERE year=2017;
```

The output of this SQL is as follows:

```
           model
-----------------------
 Lemon Limited Edition
(1 row)
```

The `WHERE` clause can also be used to reduce the number of rows returned if you just need a part of the dataset. This is similar to the `LIMIT` clause. However, the `WHERE` clause applies Boolean conditions on the underlying dataset, while the `LIMIT` clause simply picks the first few rows returned. As such, the `WHERE` clause is applied in the database server and reduces the size processed by the database. Usually, this helps to improve the overall performance.You were able to filter out the products matching a certain criterion using the `WHERE` clause. If you want a list of products before 2017, you could simply modify the `WHERE` clause to say `year<2017`. If you want a list of products started any year other than 2017, you could simply modify the `WHERE` clause to say `year!=2017`. But what if you want to filter out rows using multiple criteria at once? Alternatively, you might also want to filter out rows that match two or more conditions. You can do this by adding the `AND` / `OR` clause in the queries, as explained in the next subsection.

The AND/OR and NOT clauses

To apply multiple conditions to the `SELECT` statement, you need to put multiple `WHERE` clauses together using the `AND` or `OR` clause. The `AND` clause helps us retrieve only the rows that match two or more conditions. The `OR` clause, on the other hand, retrieves rows that match any one (or many) of the conditions in a set of two or more conditions.For example, you want to return models that were not only built in 2017, but also have an MSRP of less than $1,000. You can write the following query:

```
SELECT model, year, base_msrp
FROM products
WHERE year=2017 AND base_msrp<=1000;
```

The result will look like the following:

```
           model         | year | base_msrp
-----------------------+------+-----------
 Lemon Limited Edition | 2017 |    799.99
(1 row)
```

Here, you can see that the year of the product is `2017` and `base_msrp` is lower than $1,000. This is exactly what you are looking for.You can also use the `OR` clause to find the rows that meet at least one condition out of a series of conditions. Suppose you want to return any models that were released in the year 2017 or have an MSRP of less than $1,000. You can write the following query:

```
SELECT model, year, base_msrp
FROM products
WHERE year=2017 OR base_msrp<=1000;
```

The result is as follows:

```
          model         | year | base_msrp
-----------------------+------+-----------
 Lemon                 | 2016 |    399.99
```

```
 Lemon Limited Edition | 2017 |    799.99
 Lemon                 | 2019 |    499.99
 Blade                 | 2020 |    699.99
 Bat                   | 2022 |    599.99
 Bat Limited Edition   | 2023 |    699.99
 Lemon Zester          | 2025 |    349.99
(7 rows)
```

You already know that there is only one product, `Lemon Limited Edition`, that started in 2017. The rest of the products in the example have an MSRP of less than $1,000. You are seeing the combined dataset of `year=2017` together with `base_msrp<=1000`. That is exactly what the `OR` operator does.When using more than one `AND` / `OR` condition, you may need to use parentheses to separate and position pieces of logic together. This will ensure that your query works as expected and is as readable as possible. For example, if you wanted to get all products with models between the years 2019 and 2021, as well as any products that are scooters, you could write the following:

```
SELECT *
FROM products
WHERE
year > 2019 AND year < 2021 OR product_type='scooter';
```

The result contains all the scooters as well as an automobile that has a year between 2019 and 2021.However, to clarify the `WHERE` clause, it would be preferable to write the following:

```
SELECT *
FROM products
WHERE (year>2019 AND year<2021)
OR product_type='scooter';
```

While the result is the same as the preceding, parenthesis makes this SQL easier to understand.PostgreSQL also provides a `NOT` clause to indicate rows that do not match the conditions following the `NOT` operator. For example, the following code will return rows that do not match the `OR` condition in parentheses:

```
SELECT model, year, base_msrp
FROM products
WHERE NOT (year=2017 OR base_msrp<=1000);
```

The result contains all models with a year other than 2017, as well as having an MSRP higher than 1,000:

```
     model     | year | base_msrp
---------------+------+-----------
 Model Chi     | 2020 | 115000.00
 Model Sigma   | 2021 |  65500.00
 Model Epsilon | 2023 |  35000.00
 Model Gamma   | 2023 |  85750.00
 Model Chi     | 2025 |  95000.00
(5 rows)
```

With the combination of comparison operators such as `>` and `<`, as well as logical operators such as `AND` and `OR`, you can perform most filtering operations for numeric and alphabetical values. However, there are special filter criteria, such as selecting from a list of items or `NULL` operations, which will be introduced in the next section.

The IN/NOT IN clause

Now that you can write queries that match multiple conditions, you also might want to refine your criteria by retrieving rows that contain (or do not contain) one or more specific values in one or more of their columns. This is where the `IN` and `NOT IN` clauses come in handy. For example, if you are interested in returning all models from the years 2017, 2019, or 2022, you can write a query such as this:

```
SELECT model, year
FROM products
WHERE year = 2017
OR year = 2019
OR year = 2022;
```

The result will look like the following, showing three models from these years:

```
           model         | year
-------------------------+------
 Lemon Limited Edition | 2017
 Lemon                 | 2018
 Bat                   | 2022
(3 rows)
```

However, it is tedious to write these similar `OR` clauses again and again, which occupies a lot of space. Using `IN`, you can instead write the following:

```
SELECT model, year
FROM products
WHERE year IN (2017, 2019, 2022);
```

This code is much cleaner and makes it easier to understand what is going on, and it will return the same result as previously.Conversely, you can also use the `NOT IN` clause to return all the values that are not in a list of values. For instance, if you wanted all the products that were not produced in the years 2017, 2019, and 2022, you could write the following:

```
SELECT model, year
FROM products
WHERE year NOT IN (2017, 2019, 2022);
```

Now, you see the products that are in the years other than the three mentioned in the SQL statement.So far, all the filter criteria are based on actual values, such as equals or greater than. However, what if you want to test the lack of values? Some entries in a column may be missing. This could be for a variety of reasons. Perhaps the data was not collected or was not available at the time the data was collected. Perhaps the absence of a value is representative of a certain state in a row and provides valuable information. Whatever the reason, you are often interested in finding rows where the data is not filled in for a certain column. In SQL, empty values are represented by the `NULL` value. For instance, in the `products` table, the `production_end_date` column having a `NULL` value indicates that the product is still being made. Testing the existence of `NULL`, thus, is very important in business operations. SQL provides a dedicated function to test the existence of `NULL` values, in other words, the non-existence of actual values.

The IS NULL/IS NOT NULL clauses

To list all products that are still being made, you must detect whether the `production_end_date` column contains `NULL`. You can use the following query:

```
SELECT model, production_end_date
FROM products
WHERE production_end_date IS NULL;
```

The following is the output of the query:

```
          model        | production_end_date
-----------------------+---------------------
 Bat                   |
 Bat Limited Edition   |
 Lemon Zester          |
 Model Epsilon         |
 Model Gamma           |
 Model Chi             |
(6 rows)
```

This lists all products with a `NULL` value for `production_end_date`.Conversely, if you are only interested in products that are not being produced anymore, you can use the `IS NOT NULL` clause, as shown in the following query:

```
SELECT *
FROM products
```

```
WHERE production_end_date IS NOT NULL;
```

Note that the `NULL` value is not `0` or empty space, and it can only be detected by using the `IS NULL` or `IS NOT NULL` operators. For example, the following `WHERE` clauses will not return the rows that contain `NULL` values:

- `base_msrp = 0`
- `name = ''`

Even a comparison with `NULL` will not bring you back the rows with `NULL` names:

```
name = NULL
```

None of these comparisons is capable of identifying the rows with `NULL` values. That's because any Boolean comparison involving a `NULL` value will result in `NULL`, not `True` or `False`. All the preceding comparisons will result in `NULL` when the column value is `NULL`.You have learned that when DBMS executes the `SELECT` statement, it runs the `SELECT` operation on each row. This includes verifying whether the row meets the `WHERE` condition. If the `WHERE` condition for this row is `False` or `NULL`, the row will not be selected into the final result dataset. That's why the preceding comparisons cannot properly detect `NULL` values. A `NULL` value will make these comparisons `NULL`, and the rows will not be selected. You have to use the `IS NULL` or `IS NOT NULL` operators for `NULL`-related operations.In the following exercise, you will use what you have learned in this chapter to read data from the database, with the format and condition that you applied.

Exercise 5.1: Reading data from the database

You have learned about the most important components of the `SELECT` clause so far. Now, you will use these new skills in this exercise.As a new data analyst at ZoomZoom, you just gained access to the production `sqlda` database. You have received the table list in *Chapter 1, Introduction to Data Management Systems*, and have gained a basic understanding of the purpose of these tables. Now, you will use your SQL skills to gain a better understanding of the data inside these tables. You would like to start by checking a small sample set of data to identify the basic characteristics of the dataset. Then, you will use the `WHERE` clause to filter out unnecessary data so that your research is more focused. You will apply a series of filter conditions to study the dataset from different angles. All these are covered in this exercise.Perform the following steps to complete the exercise:

1. Open `psql` and connect to the `sqlda` database.
2. You will start from the `customers` table. First, you will look at the top three rows to see what columns are in the table:

```
SELECT * FROM customers LIMIT 3;
```

The result looks like this:

```
customer_id | title | first_name | last_name | suffix |          email          | gender |   ip_a
------------+-------+------------+-----------+--------+-------------------------+--------+------
        716 |       | Jarred     | Bester    |        | jbesterjv@nih.gov       | M      | 216.5
        644 |       | Aeriell    | Zanicchi  |        | azanicchihv@a8.net      | F      | 154.2
        709 |       | Chane      | McGrory   |        | cmcgroryjo@gravatar.com | M      | 182.1
(3 rows)
```

1. The first thing that catches your eye is the `date_added` column. You would like to see the three most recent customers by running the following query:

```
SELECT first_name, last_name, date_added
FROM customers
ORDER BY date_added DESC
LIMIT 3;
```

Here, you order the result by `date_add`. You use the `DESC` order because you want to see the most recent (largest) days first. The result is the following:

```
first_name | last_name |     date_added
------------+-----------+---------------------
 Patrizius  | Feetham   | 2025-02-23 00:00:00
 John       | Manclark  | 2025-02-23 00:00:00
 Athena     | Rickerby  | 2025-02-23 00:00:00
(3 rows)
```

1. Any time you receive new datasets, the first question you would like to ask is about the quality of the data. Is this data trustworthy? For example, you can order the customer data by `date_added`, but is the `date_added` column clean? Does it contain a `NULL` value, which may indicate errors in the data entry process? To answer this question, you can run the following query:

```
SELECT first_name, last_name, date_added
FROM customers
WHERE date_added IS NULL
LIMIT 3;
```

The result indicates that there is no missing data in the `date_added` column:

```
first_name | last_name | date_added
------------+-----------+------------
(0 rows)
```

You should do quality checks on all the datasets that you receive, but the quality rules may be different. For example, there is a `NULL` value in the `product_end_date` column in the `products` table. But this `NULL` value indicates that the product has not been terminated and is still active. Thus, the `NULL` value in this column does not indicate any quality issues.

1. Another interesting aspect of customers is their geographic distribution. Here, you can start with the big picture by looking at the states that the customers are in. It is foreseeable that you may see many customers in the same state. So, you will use a `DISTINCT` keyword to remove the repeated state names:

```
SELECT DISTINCT state FROM customers;
```

The first few rows of the result are as follows:

```
state
-------
 KS
 CA
 NH
 OR
```

As discussed before, without the `ORDER BY` clause, SQL will not enforce sorting. So, the first few rows on your screen may be different from the rows showing here. But still, the result should show that there are 50 states and a `NULL` value in this table.

1. You are interested in discovering customer behaviors by geographical locations over time, so you would like to run queries with both geographical location restrictions and date range. This can be done by using the `AND` / `OR` clause. For example, the following query will return three customers living in California and Florida who enrolled after 01/01/2025:

```
SELECT first_name, last_name, state, date_added
FROM customers
WHERE state in ('CA', 'FL')
AND date_added >= '01/01/2025'
LIMIT 3;
```

This returns the following output:

```
first_name | last_name | state |     date_added
------------+-----------+-------+---------------------
 Ferrell    | Beese     | CA    | 2025-01-07 00:00:00
 Cecile     | Metzig    | FL    | 2025-01-18 00:00:00
```

```
 Randee     | West      | CA    | 2025-01-11 00:00:00
(3 rows)
```

In this exercise, you used the skills you learned in this chapter to perform a preliminary analysis of the `customers` table. In a real-world system, you will do similar things to each table and every column. Understanding the data is the first step in data analysis, and the `SELECT` statement can offer great help in this step. You will try some of these actions in the activity that comes next.

Activity 5

In this chapter, you have learned about different aspects of the `SELECT` statement. Now, put your knowledge into action and answer the following questions:

1. Write a query that pulls all salespeople who were hired in 2024 and 2025 but have not been terminated (i.e., the `hire_date` value is later than `2024-01-01`, and `termination_date` is `NULL`, ordered by `hire_date`, latest first).
2. Write a query that pulls all first names, last names, and emails for ZoomZoom customers in New York City in the state of New York. They should be ordered alphabetically, with the last name followed by the first name.
3. Write a query that returns all customers with a phone number, ordered by the date the customer was added to the database.

Summary

This chapter focused on presenting data using the `SELECT` statement. It began with an introduction to reading data from existing tables using the simplest form of the `SELECT` statement. It then explained how to specify a list of columns, provide an alias, limit the result size, sort results for displaying, and select distinct values. The `WHERE` clause was then introduced to filter query results based on specific conditions, including using Boolean operators (`=`, `>`, `>=`, `<`, `<=`, `<>`, and `!=`) and logical operators (`AND` / `OR` / `NOT`), `IN` / `NOT IN`, and `IS NULL` / `IS NOT NULL`. The query logic for handling `NULL` values was also explained. With these skills, you will be able to query any table with desired conditions and selections to gain insights into the data.Often, you will need to transform the data so that it is easier to understand. For example, you may want to get the current date, or to filter string fields based on the combination of words. This requires further data transformation work. Furthermore, you may need to update the existing data to a preferred value. Sometimes, you may even want to adjust the table structures. All these will be introduced in the next chapter.

6 Transforming and Updating Data

Join our book community on Discord


https://packt.link/EarlyAccessCommunity

In previous chapters, you learned how to populate data using different commands and statements. As data enters a table, it is common that you will need to change some of the values. First of all, the source data may change. A customer may move to a new address. An employee may get a promotion. The changes in the real world require changes in the tables. Existing data may also need corrections or updates, such as correcting a typo during data entry time. That's why update is such an important operation in the data CRUD (short for *create, read, update, delete*) lifecycle.The change may happen not only to table data but also to table structures. Tables are a reflection of entities and relationships in the real world. As entities and relationships evolve, new attributes come in, and old attributes get dropped. Twenty years ago, it was hard to imagine a customer profile without a home phone. These days, a lot of customers have cell phones, but not home phones. This is a great example of how data may evolve over time. In order to adjust to the definition changes of real-world entities and relationships, the table definitions also need to be adjusted. This is also a part of the update work in the CRUD process.You also learned how to select data from existing tables. You learned how to perform arithmetic calculations between columns and constants and apply an alias to the expression to draw meaningful information from raw data. However, in many cases, you will need to create more advanced transformations based on existing data, extracting useful information, and drawing helpful conclusions. This requires more data transformation functionality of the `SELECT` statement. Furthermore, as you perform data transformation, you may want to change the table definitions to permanently store the derived data.This chapter will introduce the `UPDATE` statement, which updates the data inside the table. It will also introduce different data transformation functionalities of the `SELECT` statement so that you can transform the data into desirable results. Finally, it will introduce the `ALTER` statement, which will make the adjustments needed to the table structure.The following topics are covered in this chapter:

- Updating table data
- Running data transformation functions
- Creating user-defined functions
- Changing the table definition

With these topics, you will be able to manipulate existing data and existing table structure to adapt to the changes in the real world. You will also be able to apply complex transformations, with either system functions or your own functions.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in *Chapter 1*, *Introduction to Data Management Systems*.

Updating table data

As discussed in *Chapter 2*, *Creating Tables with a Solid Structure*, table updating is one of the four operations in the CRUD data lifecycle. Updating can be done either in-place (that is, modifying the data in the original row) or outside (that is, deleting the old row and inserting the new row). To update the values of the data presented in a table in-place, you can use the `UPDATE` statement:

```
UPDATE {table_name} SET
  {column_1} = {column_value_1},
  {column_2} = {column_value_2},
  …
WHERE {conditional};
```

Here, `{table_name}` is the name of the table with data that will be changed, `{column_1}`, `{column_2}`, … is the list of columns whose values you want to change, `{column_value_1}, {column_value_2}, …` is the list of new values you want to update into those columns, and `WHERE` is a conditional statement like the one you would find in a `SELECT` query. For the rows that meet the condition of the `WHERE` clause in the `{table_name}` table, the `UPDATE` statement will change the values of these columns to the designated values. The new values can be constant or the calculated results of an expression.To illustrate the use of the `UPDATE` statement, imagine that, for the rest of the year, the company has decided to sell all scooter models before 2021 for $299.99. You could change the data in the `products` table using the following query:

```
UPDATE Products
SET base_msrp = 299.99
WHERE product_type = 'scooter'
AND year<2021;
```

The `base_msrp` column of all three records will be updated to $299.99.You can also update the `base_msrp` column to be only 90% of its original value:

```
UPDATE Products
SET base_msrp = base_msrp * 0.9
WHERE product_type = 'scooter'
AND year < 2021;
```

You have learned how to modify the data in place. The other way of updating data is to delete existing data first, then insert new values. You have learned how to insert new rows. Deleting a row from a table can be done using the `DELETE` statement, which is covered in the next section.

Cleaning data

To delete data, you use the `DELETE` statement, as follows:

```
DELETE FROM {table_name}
WHERE {condition};
```

For instance, you must delete the products whose `product_type` value is `scooter` from the `products` table. To perform that, you can use the following query:

```
DELETE FROM products
WHERE product_type = 'scooter';
```

After running the `DELETE` statement, there was no `scooter` product in this table anymore, as all records were deleted.If you intentionally want to delete all the data in the table without deleting the table, you can write the

following query, which is `DELETE` without any condition:

```
DELETE FROM products;
```

Alternatively, you can also use the `TRUNCATE` keyword for the complete removal of data:

```
TRUNCATE TABLE products;
```

`TRUNCATE` is much faster than `DELETE` because it does not perform certain table maintenance work like `DELETE` does. However, `TRUNCATE` will not accept any filter condition and will remove all rows. Either way, full table deletion is an action that may have a significant impact. You must be very careful in doing so.Comparing the `DELETE` statement to the `DROP` statement, the `DROP` statement will completely drop the table, both the data in the table and the table definition. Once dropped, you will not see the table anymore. The `DELETE` statement only removes data. Even if you run `DELETE` without a `WHERE` condition, which deletes all rows in a table, the table still exists in the database, just containing no data.You will practice both `UPDATE` and `DELETE` statements in the following exercise.

Exercise 6.1: Updating and deleting data

In this exercise, you will update the data in a table using the `UPDATE` statement. Due to the higher cost of rare metals needed to manufacture an electric vehicle, the 2025 Model Chi will need to undergo a price hike of 10%; the current price is $95,000.In a real-world scenario, you will update the `products` table to increase the price of this product. However, because you will use the same `sqlda` database throughout the book, it would be better to keep the values in the original tables unchanged so that your SQL results remain consistent. For this reason, you will create new tables for all the `INSERT`, `ALTER`, `UPDATE`, `DELETE`, and `DROP` statement examples.Perform the following steps to complete the exercise:

1. Open `psql` and connect to the `sqlda` database.
2. Run the following query to create a `product_2025` table from the `products` table:

```
CREATE TABLE products_2025 AS (
  SELECT * FROM products
  WHERE year=2025
);
```

1. Run the following query to update the price of `Model Chi` by 10% in the `products_2025` table:

```
UPDATE Products_2025
SET base_msrp = base_msrp * 1.10
WHERE model='Model Chi'
AND year=2025;
```

1. Run the `SELECT` query to check whether the price of `Model Chi` in 2025 has been updated:

```
SELECT model, year, base_msrp
FROM products_2025;
```

The following is the output of the preceding code:

```
    model     | year |  base_msrp
--------------+------+-------------
 Lemon Zester | 2025 |      349.99
 Model Chi    | 2025 | 104500.0000
```

As you see from the output, the price of `Model Chi` is now $104,500, which was previously $95,000.

1. You then found out that the `Lemon Zester` product is no longer needed. However, you still need the `Model Chi` product, and the table should be kept. So, you will run the following query to delete it from the table:

```
DELETE FROM products_2025
WHERE model = 'Lemon Zester';
```

1. Run the following `SELECT` query to confirm the deletion. Since the `DELETE` statement only removes the data and keeps the table, you can still query the same table:

```
SELECT model, year, base_msrp
FROM products_2025;
```

The following is the output of the preceding code:

```
   model   | year |  base_msrp
-----------+------+-------------
 Model Chi | 2025 | 104500.0000
```

1. Now that you are done with the research, you can drop the table to save some database space by running the following query:

```
DROP TABLE products_2025;
```

1. If you run the following `SELECT` query now, it will fail because the table has been dropped and no longer exists in the database:

```
SELECT * FROM products_2025;
```

The error message is as follows:

```
ERROR:  relation "products_2025" does not exist
LINE 1: SELECT * FROM products_2025;
```

In this exercise, you learned how to update a table using the `UPDATE` statement and delete data using the `DELETE` statement. Next, you will learn how to transform data using different PostgreSQL-defined transformation functions in the `SELECT` statement.

Running data transformation functions

Often, the raw data presented in a query output may not be in the desired form. You may want to substitute values or map values to other values. To accomplish these tasks, SQL provides a wide variety of operators and functions. Operators are special keywords or symbols used to perform operations on data values, such as `+`, `-`, `*`, and `/`. Functions are keywords that take in inputs (such as a column or a scalar value) and process those inputs into some sort of output. You will learn about some useful functions for data transformation and cleaning in the following sections.

The CASE WHEN function

`CASE WHEN` is a function that returns different values based on a condition, usually the value in table columns. The general format of a `CASE WHEN` statement is as follows:

```
CASE
  WHEN condition1 THEN value1
  WHEN condition2 THEN value2
  …
  ELSE else_value
END;
```

Here, `condition1`, `condition2`, and so on are Boolean conditions; `value1`, `value2`, and so on are values to map to the Boolean conditions; and `else_value` is the value that is mapped if none of the Boolean conditions are met. The `ELSE` clause is optional. For each row, the DBMS starts at the top of the `CASE WHEN` statement and runs through each Boolean condition from the first one. For the first condition from the start of the

statement that evaluates as `True`, the statement will return the value associated with that condition. If none of the clauses are evaluated as `True`, the evaluation result is `False`. In this case, the value associated with the `ELSE` clause will be returned. If there is no `ELSE` clause and none of the Boolean conditions are met, the `CASE WHEN` function returns `NULL`.For example, you would like to add a column that labels a user as an `Elite Customer` type if they live in postal code 33111, or as a `Premium Customer` type if they live in postal code 33124. Otherwise, it will mark the customer as a `Standard Customer` type. This column will be a text description called `customer_type`. You can create this table by using a `CASE WHEN` statement, as follows:

```
SELECT
  CASE
    WHEN postal_code='33111' THEN 'Elite Customer'
    WHEN postal_code='33124' THEN 'Premium Customer'
    ELSE 'Standard Customer'
  END AS customer_type
FROM customers;
```

The `CASE WHEN` statement effectively mapped a postal code to a string describing the customer type. Using a `CASE WHEN` statement, you can map values in any way you please. Note that although the `CASE WHEN` statement requires the calculation of Boolean expressions, its result value can be of different data types, such as numeric or string. However, there are also functions that are only used for specific data types or only return values of specific data types, which will be covered below.

Functions for different data types

You have learned that data types are assigned to columns to indicate what types of specific operations you can perform on these columns. For example, you can perform arithmetical calculations between numerical values, or concatenate strings, or get the first few characters of strings. But you can't concatenate two date type values. PostgreSQL provides a rich set of functions for each data type's specific operations. We will introduce the datetime, geographical, and complex data type-related ones later in the book. For now, here are some most popular functions.

The datetime function

Almost all entities and relationships are associated with date and time. As such, date/time-related functions are one of the most useful groups of functions in PostgreSQL. You will learn more about them in *Chapter 12*, *Advanced Data Types: Date, Text, and Geospatial*. But here are some of the simplest ones that you will use in the next few sections and exercises:

Function	**Description**
`current_datecurrent_timecurrent_timestamp`	Get the current date, current time, and current datetime
`date_add`	Add an interval to a timestamp to get the new date
`date_part/extract`	Get the subfield (year, month, day, etc.) of a datetime

Table 6.1: Frequently used datetime functions

The string function

PostgreSQL provides a large set of string processing functions. You will learn about these functions systematically in *Chapter 12*, *Advanced Data Types: Date, Geospatial, and Text*. For now, you only need to learn about the concatenate function, which will be used in future exercises.

Function	Description
`||CONCAT()`	Both functions concatenate two strings, such as `'Hello ' || 'world!'` and `CONCAT('Hello ', 'world!')`. Both yield one string, `Hello world!`.

Table 6.2: Frequently used text functions

The casting function

Another useful data transformation is to change the data type of a column within a query. This is usually done to use a function only available to one data type, such as text, while working with a column that is in a different data type, such as numeric. To change the data type of a column, you simply need to use the `column::datatype` format, where `column` is the column name and `datatype` is the data type you want to change the column to.For example, to change the year in the `products` table to a text column in a query so that you can add a `Year of` string prefix, use the following query:

```
SELECT 'Year of ' || year::TEXT FROM products;
```

The function will convert the `year` column to text. You can now apply text functions such as concatenation to this transformed column, such as adding the `Year of` prefix. Please note that not every data type can be cast to a specific data type. For instance, `datetime` cannot be cast to float types. Your SQL client will throw an error if you ever try an unexpected conversion.

The NULL handling functions

`NULL` is a very special value that can occur in many different data types. In the real world, data is rarely clean, and `NULL` value handling is an essential function to ensure data integrity. As such, there is a designated set of functions for `NULL` handling.

The COALESCE function

A common requirement is to replace the `NULL` values with the value from another column or a constant. This can be accomplished easily by means of the `COALESCE` function. `COALESCE` checks the given column or values from left to right and returns the first one that is not null. If all values are null, then it returns null.To illustrate a simple usage of the `COALESCE` function, study the `customers` table. Since some of the records do not have the value of the `phone` field populated, the marketing team would like to see the `NO PHONE` value for those without a phone number. You can accomplish this request with `COALESCE`:

```
SELECT
  first_name, last_name,
  COALESCE(phone, 'NO PHONE') as phone
FROM customers;
```

Here, if the `phone` column contains a phone number (thus is not `NULL`), the `COALESCE` function will return the phone number. But if the `phone` column is `NULL`, the `COALESCE` function will return a string explicitly saying `NO PHONE`, making the output more readable. When dealing with creating default values and avoiding `NULL`, `COALESCE` will always be helpful.

The NULLIF function

`NULLIF` is used as the opposite of `COALESCE`. While `COALESCE` is used to convert `NULL` into a standard value, `NULLIF` is a two-value function and will return `NULL` if the first value equals the second value.For example, the marketing department would like to find people who have titles (Mr, Dr, Mrs, and so on) longer than three letters. In the sample database, the only known title longer than three characters is `'Honorable'`. Therefore, they would like you to create a list that contains the title and names of customers, but convert all titles that are spelled as `Honorable` to `NULL`. This could be done with the following query:

```
SELECT customer_id,
  NULLIF(title, 'Honorable') as title,
```

```
  first_name, last_name
FROM customers c;
```

This will remove all mentions of `Honorable` from the `title` column.You will learn how to use these functions in the next exercise.

Exercise 6.2: Data manipulation using functions

As a part of the data exploration work, you would like to find patterns inside the data. For example, say you are the CFO of the ZoomZoom company, and you would like to identify sales patterns and motivate the team by awarding loyal employees. This involves extracting key information from the raw data, such as finding sales transactions occurring in the same year/month or looking for the longest-serving salesperson. All these can be achieved using SQL functions.In this exercise, you will manipulate the tables from the `sqlda` database to yield different derivations. PostgreSQL provides a large variety of functions, and many tasks can be achieved by using different functions. This exercise is just a demonstration of how functions can be used in the context of data analytics. It serves as a starting point for you to explore different groups of functions and their usage in your work.Perform the following steps to complete the exercise:

1. Open `psql`, connect to the `sqlda` database.
2. First, you will identify the longest-serving salespeople. In order to do this, you will extract the current timestamp from the salespeople's hire date and convert it to a number of days. You will order the salespeople by the number of serving days, with the longest ones first. You will also concatenate the first name and last name of the salespeople so that it looks more natural. All these can be achieved using the following query:

```
SELECT
  first_name || ' ' || last_name as Name,
  hire_date,
  EXTRACT(day FROM Current_Timestamp - hire_date)
    AS DaysServed
FROM salespeople
ORDER BY DaysServed DESC
LIMIT 3;
```

The result of the query is as follows:

```
      name       |      hire_date      | daysserved
-----------------+---------------------+------------
 Norah Lie       | 2020-02-04 00:00:00 |       1855
 Alaric Sterrick | 2020-02-11 00:00:00 |       1848
 Benn Jakobssen  | 2020-02-13 00:00:00 |       1846
```

Note that since the `DayServed` field is calculated based on the date you run this query, your result will be different.

1. Now, looking at the `products` table, you want to see whether a product is active or not by checking whether the `production_end_date` is `NULL`. If so, you want to see an `Active` flag; if not, you want to see an `Inactive` flag. These can be achieved using the following query:

```
SELECT
  model,
  CASE
    WHEN production_end_date IS NULL THEN 'Active'
    ELSE 'Inactive'
  END
FROM products;
```

Here, you use a `CASE` function to generate the `Active` flag by checking the `NULL` status of the `production_end_date` column. Two rows from the result look like the following:

```
         model         |   case
-----------------------+---------
 Lemon Limited Edition | Inactive
 Bat Limited Edition   | Active
```

1. Next, you will look at the `sales` table. By running a simple `SELECT` statement, you can see that some sales are done with `dealership_id` and some are not:

```
customer_id | product_id | sales_transaction_date | sales_amount | channel  | dealership_id
------------+------------+------------------------+--------------+----------+---------------
       46962 |          7 | 2024-06-06 01:26:02    |       599.99 | internet |
       16289 |          7 | 2023-06-07 23:56:52    |      539.991 | internet |
       40557 |          7 | 2024-02-08 00:16:38    |       599.99 | internet |
       43462 |          7 | 2023-05-03 15:52:26    |       599.99 | internet |
        6587 |          7 | 2024-06-20 07:36:53    |       599.99 | internet |
```

The first few records indicate that those without `dealership_id` may be because they were done through the internet. Further investigation shows that there are only two types of channels, `internet` and `dealership`. So you would like to see whether, for all those whose sales channel is `dealership`, `dealership_id` is not `NULL`. This can certainly be done using a `CASE WHEN` statement. But in this example, we will use a combination of `COALESCE` and `NULLIF`.

1. The following query will use a condition with the `COALESCE` function. Starting from the channel, if its sales channel is not `dealership`, the `COALESCE` function will return a non-`NULL` value. Otherwise, the `COALESCE` function will use `dealership_id` as its return value. If `dealership_id` is `NULL`, since it is the last condition, the `COALESCE` function will return `NULL`. You can then identify whether there is any row with the `dealership` channel, but `dealership_id` is `NULL`:

```
SELECT * FROM sales
WHERE COALESCE(NULLIF(channel, 'dealership'), dealership_id::TEXT) IS NULL;
```

The end result is empty, which means for all `dealership` channel records, there is a `dealership_id` value defined. The data integrity is ensured:

```
customer_id | product_id | sales_transaction_date | sales_amount | channel | dealership_id
------------+------------+------------------------+--------------+---------+---------------
(0 rows)
```

PostgreSQL provides a large variety of functions that you can use. However, there are always times when you need to define your own logic. You can create the logic using **user-defined functions** (**UDFs**), which we will learn about next.

Creating user-defined functions

As in almost all other programming or scripting languages, functions in SQL contain sections of code that provide a lot of benefits, such as efficient code reuse and simplified troubleshooting processes. You can use functions to repeat or modify statements or queries without re-entering the statement each time or searching for its use throughout longer code segments. Another powerful aspect of functions is that they allow you to break code into smaller, testable chunks, which makes the application easier to manage.So, how do you define functions in SQL? There is a relatively straightforward syntax, with the SQL syntax keywords:

```
CREATE FUNCTION function_name (function_arguments)
RETURNS return_type AS $return_name$
DECLARE return_name return_type;
BEGIN
  <function statements>;
  RETURN <return_value>;
END;
$return_name$
LANGUAGE PLPGSQL;
```

The following is a short explanation of the functions used in the preceding code:

- `function_name` is the name issued to the function and is used to call the function at later stages.
- `function_arguments` is an optional list of function arguments. This could be empty, without any arguments provided, if you do not need any additional information to be provided to the function. To provide additional information, you can use either a list of different data types as the arguments (such as integer and numeric data types) or a list of arguments with parameter names (such as the `min_val` integer and the `max_val` numeric data type).
- `return_type` is the data type being returned from the function.
- The `DECLARE return_name return_type` statement is only required if `return_name` is provided, and a variable is to be returned from the function. `return_name` is the name of the variable to be returned (optional). If `return_name` is not required, this line can be omitted from the function definition.
- `function_statements` are the SQL statements to be executed within the function.
- `return_value` is the data to be returned from the function.
- `PLPGSQL` specifies the language to be used in the function. PostgreSQL allows you to use other languages; however, their use in this context lies beyond the scope of this book.

We will learn how to create a single function that will allow us to calculate information from our tables in the next exercise. But before that, you will learn how to use `psql` commands to retrieve information about functions.

The \df and \sf commands

You can use the `\df` command in PostgreSQL's `psql` tool to get a list of the functions available in memory, including the variables and data types passed as arguments.The `\sf function_name` command in PostgreSQL can be used to review the function definition for already-defined functions. For example, say you have created a function called `max_sale` and then execute the following query:

```
\sf max_sale
```

The output will show the definition of that function.With this knowledge of functions, you will practice building a UDF in the following exercise.

Exercise 6.3: Creating functions with arguments

In this exercise, your goal is to create a function with arguments and compute the output. You will construct a function that fetches the `salesperson_id` value of the longest-serving salespeople within a specific dealership. The ID of the dealership is to be provided to the function as a parameter. These are the steps to follow:

1. Create the function definition for a function called `longest_serving_salespeople` that returns a numeric value for `salesperson_id` and takes an `INT` value to specify the `dealership_id` value from which we will identify the longest-serving salespeople.
2. Declare the return variable as a numeric data type and begin the function.
3. Select the top `salesperson_id` value based on the order of hire date. Since you are looking for the earliest hiring date, an ascending order is used here. The `WHERE` clause specifies `dealership_id`.
4. Return the function variable, end the function, and specify the `LANGUAGE` statement.
5. Use the function to determine the longest-serving salespeople for dealership 1.

Perform the following steps to complete the exercise:

1. Open `psql` and connect to the `sqlda` database.
2. Enter the following code to create the function:

```
CREATE FUNCTION longest_serving_salespeople(dealership INT)
RETURNS numeric AS $serving$
DECLARE salespersonid numeric;
BEGIN
  SELECT salesperson_id
  FROM salespeople
  INTO salespersonid
  WHERE dealership_id = dealership
  ORDER BY hire_date ASC
  LIMIT 1;
RETURN salespersonid;
END; $serving$
LANGUAGE PLPGSQL;
```

1. Run the following command to use the function to get the longest-serving salespeople of dealership 1:

```
SELECT longest_serving_salespeople(1);
```

The result is the following:

```
longest_serving_salespeople
----------------------------
                          65
```

1. Use the `\df` command to check information about this function:

```
\df longest_serving_salespeople
Schema |            Name             | Result data type | Argument data types | Type
--------+-----------------------------+------------------+---------------------+------
public | longest_serving_salespeople | numeric          | dealership integer  | func
```

1. Use the `\sf` command to see the code of this function:

```
\sf longest_serving_salespeople
```

This will result in showing the function code.

1. You can now drop the function to clean up the database:

```
DROP FUNCTION longest_serving_salespeople;
```

With this exercise, you now understand how to implement your own logic as a UDF. At one point, a large number of UDFs were created as triggers. You will learn what a trigger is in the next subsection.

Triggers

PostgreSQL and most RDBMSs offer a functionality called a trigger. Triggers, known as events or callbacks in other programming languages, are useful features that execute the execution of SQL statements or functions in response to a specific event. Triggers can be initiated when one of the following happens:

- A row is inserted into a table
- A field within a row is updated
- A row within a table is deleted
- A table is truncated (that is, all rows are quickly removed from a table)

The timing of the trigger can also be specified to occur at one of the following:

- Before an insert, update, delete, or truncate operation
- After an insert, update, delete, or truncate operation
- Instead of an insert, update, delete, or truncate operation

Triggers are useful in making sure certain actions happen in a chain, thus ensuring the overall integrity of operations. This is also important for data analytics, as analytics typically require a high degree of data integrity. But triggers are also very complex to design and can cause significant performance degradation. They were used widely at a time when database design was separated from the overall application architecture design, and database operations were done using procedural languages. At that time, data architects used triggers to ensure the integrity of data. These days, as database design becomes more and more integrated with the overall application design, triggers are used less and less often. As such, this book will not discuss triggers in detail.You have learned how to transform data and save the changes into tables using `UPDATE`. You have also learned to use out-of-the-box functions to implement complex logic and even build your own functions. Then, you also took a quick look at triggers, which are used less often than before, but are still a part of the common SQL code base. However, sometimes, there is more to change than the data. You may need to change the table structure by adding and removing columns or changing the properties and constraints of columns. This can be done using the `ALTER` statement, which will be introduced in the next section.

Changing the table definition

To change the table definition, you use the `ALTER TABLE` statement. For example, to add new columns to an existing table, you use the `ALTER TABLE … ADD COLUMN` statement, as shown in the following query:

```
ALTER TABLE {table_name}
ADD COLUMN {column_name} {data_type};
```

For example, you wanted to add a new column to the `products` table that you will use to store the products' weights in kilograms, called `weight`. You could do this by using the following query:

```
ALTER TABLE products ADD COLUMN weight INT;
```

This query will make a new column called `weight` in the `products` table and will give it the integer data type so that only integer numbers can be stored within it.You can also use the `RENAME` clause of the `ALTER` statement to rename a column, which is quite straightforward:

```
ALTER TABLE {table_name}
RENAME COLUMN {column_name} TO {new_column_name};
```

In fact, you can even rename a table using the `ALTER` statement:

```
ALTER TABLE {table_name} RENAME TO {new_table_name};
```

If you want to remove a column from a table, you can use the `ALTER TABLE … DROP COLUMN` statement:

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

Here, `{table_name}` is the name of the table you want to change, and `{column_name}` is the name of the column you want to drop. Imagine that you decide to delete the `weight` column you just created. You could get rid of it using the following query:

```
ALTER TABLE products DROP COLUMN weight;
```

There are many other options of the `ALTER` statement for a variety of tasks, such as changing column data type, changing column constraints, and changing table keys. You can refer to the PostgreSQL official manual for details. For example, the following statement changes the data type of the `year` column to `decimal`:

```
ALTER TABLE products
ALTER COLUMN year TYPE decimal(10, 0);
```

A typical activity in a data management system is data cleansing. As data comes in from sources, you need to inspect the data, remove unnecessary records, transform data values based on your analytical needs, and record

metadata of the cleansing task, such as date and time of the task run. With everything you learned in this chapter, you are capable of doing this task now. You will practice this in the following activity.

Activity 6

You will write a function to simulate the data cleansing process of the `products` table. As you receive this table, you will perform the following data manipulations to make the data more integrated and more consumable.

1. Typically, in a production system, for compliance and liability reasons, you would like to keep the data from the source unchanged and stored for a long period. You will instead make a copy of these tables and perform your data operations on the copies. So, as the first step, duplicate the `products` table into a copy called the `products_new` table.
2. Create a function called `update_products`, which performs the following tasks:
    - Add a column to the `products_new` table called `last_update_date`, and populate the column with the current date. This will be the record of when the product information was last updated in the system, which is important for data quality checking.
    - Remove products that have been discontinued (having a non-`NULL` `production_end_date` date) and have a `production_start_date` date earlier than 2020. This will remove obsolete data that is no longer relevant.
    - If the `production_end_date` column is `NULL`, replace it with the date `2999-01-01`. This is a very common technique in `NULL` handling. As `NULL` data requires different logic in filter processing, many applications will use a boundary value to replace `NULL` so that the logic is simpler. A date far enough in the future, such as `2999-01-01`, is widely used to fill in the expiration date/end date fields for active records.
    - Finally, return the time when all the processing is over. The purpose of returning this value is to provide the ending time to the overall data processing workflow for workload tracking and performance monitoring.
3. With the function completed, run the function and check the value in the `products_new` table.
4. Once you confirm that the data meets the preceding requirements, drop the function and table to maintain database cleanness.

Summary

The chapter provided a guide on updating data in databases with the `UPDATE` statement, transforming data using PostgreSQL functions, creating user-defined functions with the `CREATE FUNCTION` statement, and changing table definitions with the `ALTER` statement. With this knowledge, you will be able to manipulate existing data and existing table structure to adapt to the changes in the real world. You will also be able to apply complex transformations with either system functions or your own functions.At this point, you have learned how to work on each table within the `CRUD` lifecycle. You can create and drop the tables, insert or copy data, transform and update data, and alter the table structure, against a single table. Starting from the next chapter, you will look at the interactions between tables, as well as further aggregation from the raw tables.

7 Defining Datasets from Existing Datasets

Join our book community on Discord


https://packt.link/EarlyAccessCommunity

In previous chapters, you have learned how to write extraction queries against a single table. In real-world usage scenarios, the data you are interested in needs multiple layers of preprocessing before you can present it, or is spread across multiple tables. For example, you may only be interested in active customers who made purchases in the past three years, or you would like to know the addresses of both customers and dealerships. In the former case, a common request is to reuse the query that preprocesses the data, either within the same statement or across different statements. In the latter case, you need to bring related tables together.SQL provides multiple approaches to both requests. To reuse queries, you can use a subquery, a **common table expression** (**CTE**), or a view. To utilize data from multiple tables, you can either join these tables if you need to merge the columns of records from different tables, or union these tables if you simply want to merge the records into one larger dataset. All these will be introduced in this chapter.The following topics are covered in this chapter:

- Creating derived datasets
- Joining tables
- Running set operations

With these topics, you will be able to build advanced research by combining data from multiple tables out of complex relationships.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in *Chapter 1*, *Introduction to Data Management Systems*.

Creating derived datasets

So far, you have been using `SELECT` queries to pull data from tables. You may have observed that the results of all `SELECT` queries are two-dimensional relations that look like the tables in a relational database. Knowing this, you may wonder whether there is some way to use the relations produced by the `SELECT` queries as if they were tables in your database. The answer is “yes.” You can simply take a query, insert it between a pair of parentheses, and give it an alias. The result of this is a relation that can be used in any other queries. This will help you build an analysis on top of an existing analysis, thus reducing errors and improving efficiency.To illustrate, look at two tables in the `sqlda` database— `dealerships` and `salespeople`. Both the `dealerships` table and the `salespeople` table contain a column called `dealership_id`. This `dealership_id` column in the `salespeople` table is a direct reference to the `dealership_id` column in the `dealerships` table. As discussed before, the `dealership_id` column is the primary key in the `dealerships` table and a foreign key in the `salespeople` table.

As these two tables are related, you can perform some interesting analyses between them. For instance, you may be interested in determining which salespeople work at a dealership in California. One way of retrieving this information is to first query which dealerships are in California. You can do this using the following query:

```
SELECT dealership_id
FROM dealerships
WHERE state='CA';
```

This query should give you the following results:

```
dealership_id
---------------
             2
             5
(2 rows)
```

Now that you know that the only two dealerships in California have the IDs of `2` and `5`, respectively, you can then query the `salespeople` table, as follows:

```
SELECT * FROM salespeople
WHERE dealership_id in (2, 5);
```

The following are the first few rows of the output of the code:

```
salesperson_id | dealership_id | title | first_name |  last_name  | suffix |    username    | ger
---------------+---------------+-------+------------+-------------+--------+----------------+---
             23 |             2 |       | Beauregard | Peschke     |        | bpeschkem      | Ma
             51 |             5 |       | Lanette    | Gerriessen  |        | lgerriessen1e  | Fe
             57 |             5 |       | Spense     | Pithcock    |        | spithcock1k    | Ma
```

While this method gives you the results you want, it is extremely manual to do it this way, which is slow in performance and error-prone. What would make this process easier would be to feed the query result into the `dealerships` table as a filter to the query on the `salespeople` table. To achieve this, the query on the `dealerships` table will be embedded in the query on the `salespeople` table. Thus, it is called the subquery. The query on the `salespeople` table is the main query. You will write the query this way:

```
SELECT * FROM salespeople
WHERE dealership_id in (
  SELECT dealership_id
  FROM dealerships
  WHERE state='CA'
);
```

It will provide you with the same result as previously.Subqueries can be used not only as filters but also as source datasets in the `FROM` clause, as demonstrated here:

```
SELECT * FROM (
  SELECT * FROM salespeople
  WHERE termination_date IS NULL
) AS s
WHERE dealership_id IN (
  SELECT dealership_id FROM dealerships
  WHERE dealerships.state = 'CA'
);
```

Here, the subquery following the `FROM` clause returns all active salespeople (`termination_date IS NULL`). The subquery in the `WHERE` clause provides the dealership filter for California. Note that there is an `AS` alias following the first subquery. In order for the subquery to be properly recognized in the `FROM` clause, you must provide it with an alias using the `AS` clause. You can also use an alias for regular tables. In fact, you will use table aliases extensively when we put two tables/subqueries in the same `SELECT` query, which will be covered later in this chapter.A common scenario in SQL programming is that you may need to use the same subqueries (or subqueries with similar structures) multiple times in a query. To avoid writing down their definitions multiple times, improve readability, and make code simple, you can put subqueries into a structure called a CTE, which is covered in the next section.

Common table expressions

A CTE is an explicitly defined version of subqueries. To define a CTE, you put a subquery into the `WITH` clause with an alias, such as the following:

```
WITH s AS (
  SELECT * FROM salespeople
  WHERE termination_date IS NULL
)
```

You now have a CTE called `s`, which can be used in the main query as if it is an existing table. For example, the preceding query can be written as follows:

```
WITH s AS (
  SELECT * FROM salespeople
  WHERE termination_date IS NULL
)
SELECT * FROM s
WHERE dealership_id IN (
  SELECT dealership_id FROM dealerships
  WHERE dealerships.state = 'CA'
);
```

You can define multiple CTEs in the same `WITH` clause, separated by commas. The CTE defined later can utilize the CTE defined earlier:

```
WITH s AS (
  SELECT * FROM salespeople
  WHERE termination_date IS NULL
), s_CA AS (
  SELECT * FROM s
  WHERE dealership_id IN (
    SELECT dealership_id FROM dealerships
    WHERE dealerships.state = 'CA'
  )
)
```

Here, the `s_CA` CTE uses the `s` CTE, which was defined earlier. Both `s_CA` and `s` can be used in the main query following the `WITH` clause.The one advantage of CTEs is that they can be designed to be recursive. Recursive CTEs can reference themselves. Because of this feature, you can use them to solve problems that other queries cannot. However, recursive CTEs are beyond the scope of this book.Subqueries and CTEs are a convenient way of reusing query definitions in a statement. However, they only exist within the statement in which they are defined, and are not accessible outside of that statement. If you want to use the same query definition in different statements, you need to define it as a view, which will be discussed in the next subsection.

Views

A view is a virtual dataset based on the result of a SQL query. Instead of storing the data, it stores the query definition. When you use a view in a SQL statement, the DBMS will run the query of the view and use the latest data as if it were a real table. For example, you can reuse the query definition mentioned previously as a view:

```
CREATE VIEW active_salespeople AS (
  SELECT * FROM salespeople
  WHERE termination_date IS NULL
);
```

Then, you can use the view in a SQL statement. Note that there is no data storage for this `active_salespeople` view. When you run this statement, the DBMS dynamically runs the query in its definition to get its current data, and feeds this current data to the execution process of the following query:

```
SELECT * FROM active_salespeople
WHERE dealership_id IN (
  SELECT dealership_id FROM dealerships
```

```
  WHERE dealerships.state = 'CA'
);
```

You will practice utilizing these subquery structures for your advanced data analysis in the following exercise.

Exercise 7.1: Utilizing subqueries

Your marketing team is wondering how important it is to have human contact during sales. They would like to get information on customers living in states where there is a dealership. In this exercise, you will use three different approaches (subquery, CTE, and view) to create the same filter condition.To complete the task, execute the following:

1. Open `psql` and connect to the `sqlda` database.
2. Use the following statement to retrieve the required dataset:

```
SELECT first_name, last_name, email
FROM customers
WHERE state IN (
  SELECT state from dealerships
);
```

1. Now, you would like to focus on customers that salespeople can reach and talk to. You will add a CTE to generate a list of customers that have phone numbers:

```
WITH c AS (
  SELECT first_name, last_name, state, email
  FROM customers
  WHERE phone IS NOT NULL
)
SELECT first_name, last_name, email
FROM c
WHERE state IN (
  SELECT state from dealerships
);
```

1. You found this query to be really useful. So, you would like to save the query definition in the CTE so that you can use it in other statements. You will run the following statement to save it into a view for future usage:

```
CREATE VIEW reachable_customer AS (
  SELECT first_name, last_name, state, email
  FROM customers
  WHERE phone IS NOT NULL
);
```

1. To confirm the view is working, use it in the query that we have used before. You can compare the result of this query to the one in Step 3:

```
SELECT first_name, last_name, email
FROM reachable_customer
WHERE state IN (
  SELECT state from dealerships
);
```

1. Finally, you will drop the view to maintain the cleanness of the database:

```
DROP VIEW reachable_customer;
```

With the preceding section, you can utilize the result of queries against one table within queries against another table. But what if you want to utilize two tables together? This requires the utilization of either `JOIN` or `UNION`, which are covered in the next section.

Joining tables

You learned about relational data modeling practices and normalization in *Chapter 1, Introduction to Data Management Systems*. In relational databases, data is often normalized to be stored across multiple tables to simplify the data model and avoid redundancy. However, there are times when you want to combine data from multiple tables into one large dataset. Depending on the nature of the tables, you may want to combine the rows of them. This is called a union. Or you may want to combine the columns of multiple datasets based on a related column (or columns) between them. This is called a join.You will learn about the fundamentals of `JOIN` in this section and `UNION` in the next section. There are three major types of joins. You will start by learning about the inner join.

Inner joins

An inner join connects rows in different tables, based on a condition known as the join predicate. In many cases, the join predicate is a logical condition of column relationships between two tables. In a join, each row in the first table is compared against each row in the second table. For row combinations that meet the inner join predicate, that row is returned in the query. Otherwise, the row combination is discarded. For example, both the `dealerships` table and the `salespeople` table have the `dealership_id` column. When you combine the rows from the `dealerships` table with all rows from the `salespeople` table, you may get a combination like this:

The `dealerships` **table**	**The** `salespeople` **table**
`dealership_id` columns	`dealership_id` columns
1, …	1, …
1, …	2, …
1, …	3, …
…	
10, …	10, …

Table 7.1: Combining data from two tablesClearly, the combinations of unmatched `dealership_id`, such as combining dealership 1's salespeople with dealership 10, will not make any sense and should be discarded. Only those combinations with the same `dealership_id` value can meaningfully map salespeople to their dealerships. This, in effect, is a filter condition: `dealerships.dealership_id = salespeoples.dealership_id`. Thus, you can join the two tables using an equal-to condition in the join predicate, as follows:

```
SELECT *
FROM salespeople
INNER JOIN dealerships
  ON salespeople.dealership_id = dealerships.dealership_id;
```

The following shows the first few rows of the output:

```
 salesperson_id | dealership_id | title |  first_name  |  last_name   | suffix |    username
----------------+---------------+-------+--------------+--------------+--------+-----------------
              1 |            17 |       | Electra      | Elleyne      |        | eelleyne0
              2 |             6 |       | Montague     | Alcoran      |        | malcoran1
              3 |            17 |       | Ethyl        | Sloss        | IV     | esloss2
```

When you put this in a formal definition, you get the basic join statement:

```
SELECT {columns}
FROM {table1}
INNER JOIN {table2}
  ON {table1}.{common_key_1}={table2}.{common_key_2};
```

Here, `{columns}` is the columns you want to get from the joined table, `{table1}` is the first table, `{table2}` is the second table, `{common_key_1}` is the column in `{table1}` you want to join on, and `{common_key_2}` is the column in `{table2}` to join on. You can also have multiple join conditions based on multiple tables, and combine the conditions using a logical operator such as `AND`. By introducing the `ON` clause, you define the condition(s) on which these two tables should be combined. For example, in the previous SQL, `dealership_id` in the `salespeople` table matches `dealership_id` in the `dealerships` table. This shows how the join predicate is met.

By running this join query, you have effectively created a new super dataset consisting of the two tables merged where the two `dealership_id` columns are equal.Note that the first table listed in the query, `salespeople`, is on the left-hand side of the result, while the `dealerships` table is on the right-hand side. This left-right order will become very important in the next section, when you learn about outer joins between tables. For now, as you are using an inner join, the order of tables is not important for join predicates that use an equal operation.You can now run a `SELECT` query over this super dataset in the same way as one large table. For example, going back to the multi-query issue to determine which sales query works in California, you can now address it with one easy query:

```
SELECT *
FROM salespeople
INNER JOIN dealerships
  ON salespeople.dealership_id = dealerships.dealership_id
WHERE dealerships.state = 'CA';
```

You will observe that the preceding output has both the `dealerships` data and the `salespeople` data. If you want to retrieve only the `salespeople` table portion of this, you can select only the `salespeople` columns using the following star syntax:

```
SELECT salespeople.*
FROM salespeople
INNER JOIN dealerships
  ON dealerships.dealership_id = salespeople.dealership_id
WHERE dealerships.state = 'CA'
ORDER BY 1;
```

There is another shortcut that can help while writing statements with several `JOIN` clauses. You can use an alias for table names to avoid typing the entire name of the table every time. Simply write the name of the alias with the `AS` keyword after the first mention of the table after the `JOIN` clause, and you can save a decent amount of typing. For instance, for the preceding query, if you wanted to alias `salespeople` with `s` and `dealerships` with `d`, you could write the following statement:

```
SELECT s.*
FROM salespeople AS s
INNER JOIN dealerships AS d
  ON d.dealership_id = s.dealership_id
WHERE d.state = 'CA';
```

Even further, you can omit the `AS` keyword. The result will be the same.Now that you have covered the basics of inner joins, it is time for outer joins.

Outer joins

As discussed, inner joins will only return rows from the two tables when the join predicate is met for both tables, that is, when both tables have rows that can satisfy the join predicate. Otherwise, no rows from either table are returned. This is illustrated in the following figure:

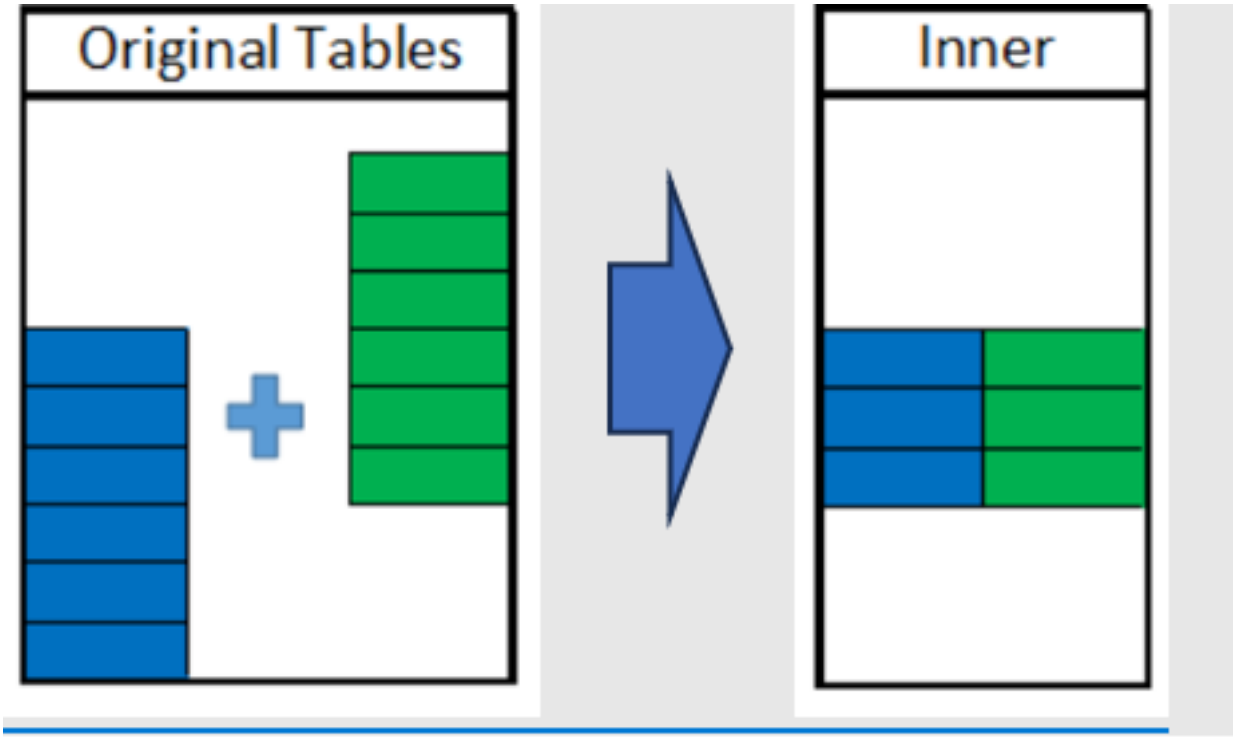

Figure 7.1: Inner join

In the preceding join, only matching rows are returned.It can happen that sometimes you want to return all rows from one of the tables, even if the other table does not have any row meeting the join predicate. In this case, since there is no row meeting the join predicate, the second table will return nothing but `NULL`. An outer join is a type of join that returns all rows from at least one table, along with matching rows from the other table, or even all rows of the other table, depending on the exact join type. Based on the condition of returning rows from the latter table, outer joins can be classified into three categories: left outer joins, right outer joins, and full outer joins. Let us understand each of these in the following sections.

Left outer joins

Left outer joins are where the left table (that is, the table mentioned first in a join) provides all the rows it has. If a matching row from the other table (the right table) is found, it behaves the same as an inner join, where these two rows from two tables are combined into one. If not, a list of `NULL` is used to fill the columns from the right table. Left outer joins are performed by using the `LEFT OUTER JOIN` keywords, followed by a join predicate. This can also be written in short as `LEFT JOIN`.To show how left outer joins work, examine two tables: the `customers` table and the `emails` table. For the time being, assume that not every customer has been sent an email, and you want to mail all customers who have not received an email. You can use a left outer join to make that happen since the left side of the join is the `customers` table. The following code snippet is utilized:

```
SELECT *
FROM customers c
LEFT OUTER JOIN emails e
  ON e.customer_id=c.customer_id
ORDER BY c.customer_id
LIMIT 1000;
```

The following are three rows in the middle of the output:

```
26 |       | Rhiamon      | Wynes          |        | rwynesp@hud.gov
27 |       | Anson        | Fellibrand     |        | afellibrandq@topsy.com
28 |       | Gradey       | Garrat         |        | ggarratr@bbb.org
```

To use the same illustration you saw in *Figure 7.1*, here is what a left outer join looks like:

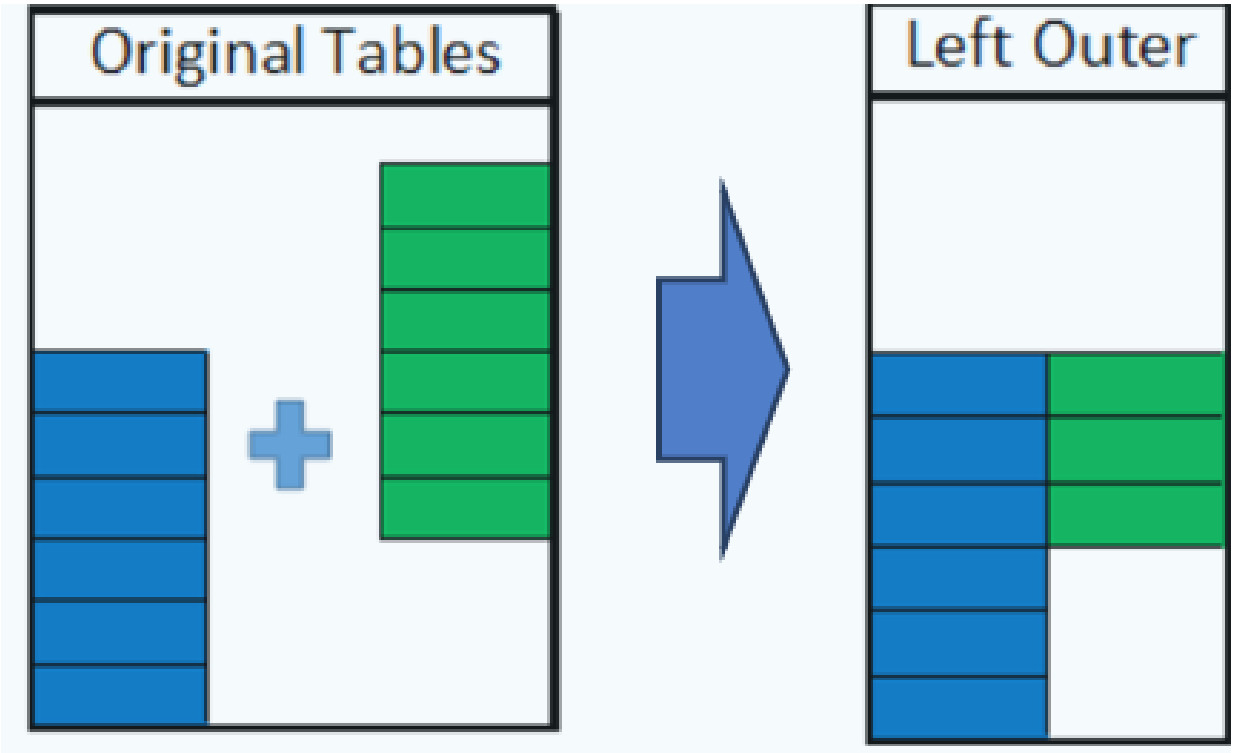

Figure 7.2: Left outer join

In this illustration, all rows in the left table stay in the result table, while only the matching rows in the right table exist in the result table. When you look at the output of the query, you should see that entries from the `customers` table are present. However, for some of the rows, such as for the `customer_id` value of `27`, which can be seen in the previous output, the columns belonging to the `emails` table are completely full of `NULL` values because this customer doesn't have a record in this table. This arrangement explains how the outer join is different from the inner join. Had the inner join been used, the row with the `customer_id` value of `27` would not have shown up because there is no matching record in the `emails` table.You can now use this query to find people who have never received an email. Because those customers who were never sent an email have a null `customer_id` column in the values returned from the `emails` table, you can find all these customers by checking the `customer_id` column in the `emails` table, as follows:

```
SELECT *
FROM customers c
LEFT OUTER JOIN emails e
  ON c.customer_id = e.customer_id
WHERE e.customer_id IS NULL
ORDER BY c.customer_id
LIMIT 1000;
```

In the result, all entries where the `email_id` column of the `emails` table is blank indicate that the customer of that row has not received any emails.

Right outer joins

A right outer join is very similar to a left join, except the table on the right (the second listed table) will now have every row show up, and the left table will have `NULL` values if the `ON` condition is not met. To illustrate, look at the following figure:

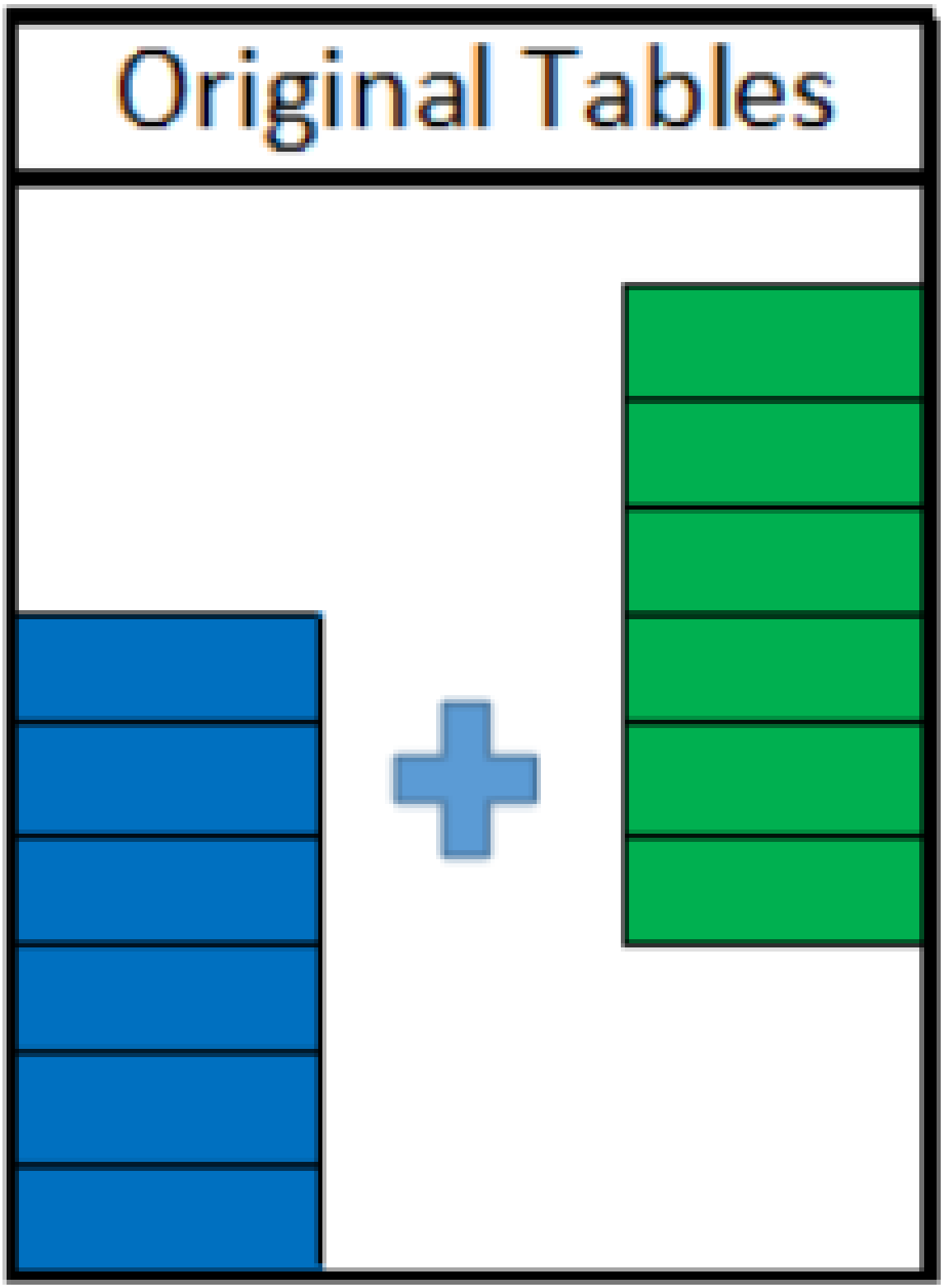
Original Tables

Figure 7.3: Right outer join

Here, all the rows in the right table remain in the result table, while only the matching rows from the left table stay. This is the mirroring of the left join concept. In fact, you can flip the left outer join query by right joining the `emails` table to the `customers` table with the following query:

```
SELECT *
FROM emails e
RIGHT OUTER JOIN customers c
  ON e.customer_id=c.customer_id
ORDER BY c.customer_id
LIMIT 1000;
```

When you run this query, you will notice that this output is similar to what was produced in the left outer join example, except that the data from the `emails` table is now on the left-hand side, and the data from the `customers` table is on the right-hand side. Once again, customer ID `27` has `NULL` for the email. This shows the symmetry between a right join and a left join.

Full outer joins

The full outer join will return all rows from the left and right tables, regardless of whether the join predicate is matched. For rows where the join predicate is met, the two rows are combined just like in an inner join. Rows in the left table that do not have a match in the right table will still be placed into the result table, with `NULL` filled in for the columns from the right table. Similarly, rows in the right table that do not have a match in the left table will still be placed into the result table, with `NULL` filled in for the columns from the left table. This can be explained using the following figure:

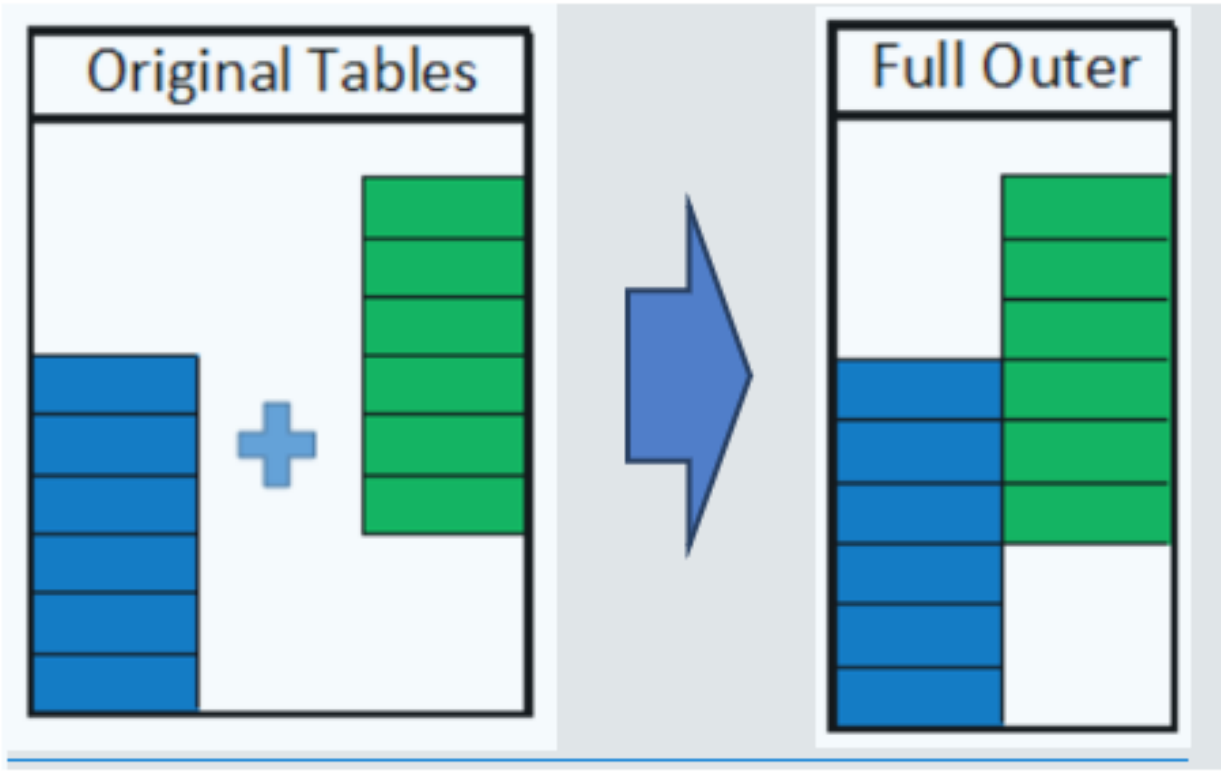

Figure 7.4: Full outer join

The full outer join is invoked by using the `FULL OUTER JOIN` clause, followed by a join predicate. Here is the syntax of this join:

```
SELECT *
FROM emails e
FULL OUTER JOIN customers c
  ON e.customer_id=c.customer_id;
```

A full outer join is very common when you have datasets from different sources that each cover a different channel of business activities. For example, in the United States healthcare system, medical examinations and dental visits are usually recorded in different systems. Patients may have both medical examinations and dental visits, or they may have only one of these. That means some patients only exist in one of the systems, while some exist in both. A full outer join of these two datasets can ensure you have the full list of patients.In this section, you learned how to implement three different types of outer joins. In the next section, you will learn about the cross join.

Cross joins

A cross join is a join type that has no join predicate. That means every row from the left table will be matched to all the rows in the right table, regardless of whether they are related or not. It is also referred to as the Cartesian product. It is named *Cartesian* after the French mathematician René Descartes, who raised the idea of this type of operation. It can be invoked using a `CROSS JOIN` clause, followed by the name of the other table. To better understand this, take the example of the `products` table.A common data mining analysis is called **market basket analysis**, which studies the selling patterns between multiple products. For example, diapers are usually sold together with baby wipes. So, if you are running a promotion for diapers and expect more customers to come to the diaper aisle or web page, you may want to place baby wipes there, too. To perform market basket analysis, you want to know every possible combination of two products that you could create from the `products` table. You can use a cross join to get the answer to the question using the following query:

```
SELECT
  p1.product_id, p1.model, P2.product_id, p2.model
FROM products p1
CROSS JOIN products p2;
```

In this case, you have joined every value in one table to every record of the same table. The result of the query has 144 rows, which is the equivalent of multiplying the 12 products by the same 12 products (12 * 12). You can also see that a cross join does not require a join predicate.In general, cross joins aren’t used much in practice as they can hamper the process if you are not careful. Cross-joining two large tables can lead to the origination of hundreds of millions of rows, which can stall and crash a database. So, if you decide to use a cross join, ensure you exercise utmost care.So far, you have covered the basics of using joins to fuse tables for a custom analysis of data. You will practice this in the following exercise.

Exercise 7.2: Using joins to analyze a sales dealership

In this exercise, you will use joins to bring related tables together. For instance, the head of sales at your company would like a list of all customers who bought a car. To do the task, you need to create a query that will return all customer IDs, first names, last names, and valid phone numbers of customers who purchased a car.To complete this exercise, perform the following steps:

1. Open `psql` and connect to the `sqlda` database.
2. Use an inner join to bring the `sales`, `customers`, and `products` tables together, which returns customer data for customer IDs, first names, last names, and the product model names that they purchased:

```
SELECT
  c.customer_id, c.first_name, c.last_name,
  p.model
FROM sales s
INNER JOIN customers c
  ON c.customer_id=s.customer_id
INNER JOIN products p
  ON p.product_id=s.product_id
WHERE p.product_type='automobile'
  AND c.phone IS NOT NULL;
```

Here are the first few rows of output:

```
customer_id |   first_name   |    last_name    |     model
------------+----------------+-----------------+---------------
       35946 | Merl           | Brumhead        | Model Sigma
       46003 | Anabelle       | Southby         | Model Chi
       19035 | Belia          | Turbat          | Model Sigma
       35824 | Wyatan         | Dickie          | Model Chi
```

You can see that running the query helped you join the data from the `sales`, `customers`, and `products` tables and obtain a list of customers who bought a car and have a phone number.Up till now, in this book, you have learned how to join data. You can use joins to add columns from other tables horizontally. However, you may be interested in putting multiple queries together vertically, that is, by keeping the same number of columns but adding multiple rows. This can be achieved by using the `UNION` statement, which will be covered in the next section.

Running set operations

The `UNION` statement combines rows from multiple tables with the same column structures. For example, suppose you wanted to visualize the addresses of dealerships and customers on the same map. To do this, you would need the addresses of both customers and dealerships. You could build a query with all customer addresses as follows:

```
SELECT street_address, city, state, postal_code
FROM customers
WHERE street_address IS NOT NULL;
```

You then also retrieve dealership addresses with the following query:

```
SELECT street_address, city, state, postal_code
FROM dealerships
WHERE street_address IS NOT NULL;
```

To reduce complexity, it would be nice if there were a way to assemble the two queries into one list with a single query. This is where the `UNION` keyword comes into play. You can use the two previous queries and create the following query:

```
  SELECT street_address, city, state, postal_code
  FROM customers
  WHERE street_address IS NOT NULL
UNION
  SELECT street_address, city, state, postal_code
  FROM dealerships
  WHERE street_address IS NOT NULL
;
```

This produces a combination of two address datasets.Note that there are certain conditions that need to be kept in mind when using `UNION`. Firstly, `UNION` requires the subqueries to have the same number of columns and the same data types for the columns. If they do not, the query will fail. Secondly, `UNION` only returns unique rows from its subqueries. `UNION`, by default, removes all duplicate rows in the output. If you want to retain the duplicate rows, it is preferable to use the `UNION ALL` keyword. For example, if both of the previous queries return a row with address values such as `123 Main St`, `Madison`, `WI`, and `53710`, the result of the `UNION` statement will only contain one record for this value set, but the result of the `UNION ALL` statement will include two records of the same value, one from each query.To demonstrate the usage of `UNION ALL`, first, run a simple query that combines the `products` table with all the rows:

```
SELECT * FROM products
UNION
SELECT * FROM products
ORDER BY 1;
```

You can see that the query returns 12 rows and there are no duplicate rows, just the same as the original `products` table. However, say you run the following query:

```
SELECT * FROM products
UNION ALL
SELECT * FROM products
ORDER BY 1;
```

You will see that the query returns 24 rows, in which each row is repeated twice. This is because the `UNION ALL` statement keeps the duplicated rows from both `products` tables. As such, `UNION ALL` usually has a between performance than `UNION` because it does not need to run through an additional step of removing redundancy. `UNION` and `UNION ALL` are not the only two set operations PostgreSQL provides. There are also two other operations, `INTERSECT` and `EXCEPT`. `INTERSECT` returns rows that are present in both result sets, while `EXCEPT` returns rows that are only in the first result set and not in the second one. Essentially, `INTERSECT` finds the common data between two queries, and `EXCEPT` finds the difference between them.In the next exercise, you will implement some set operations for your business users.

Exercise 7.3: Generating an elite customer party guest list using UNION

In this exercise, you will assemble two queries using `UNION`. To help build marketing awareness for the new Model Chi, the marketing team would like to throw a party for some of ZoomZoom's wealthiest customers in Los Angeles, CA. To help facilitate the party, they would like you to make a guest list with ZoomZoom customers who live in Los Angeles, CA, as well as salespeople who work at the ZoomZoom dealership in Los Angeles, CA. The guest list should include details such as the first and last names and whether the guest is a customer or an employee.To complete the task, execute the following:

1. Open `psql` and connect to the `sqlda` database.
2. Write a query that will make a list of ZoomZoom customers and company employees who live in Los Angeles, CA. The guest list should contain first and last names and whether the guest is a customer or an employee:

```
  SELECT first_name, last_name,
    'Customer' as guest_type
  FROM customers
  WHERE city='Los Angeles' AND state='CA'
UNION
  SELECT first_name, last_name,
    'Employee' as guest_type
  FROM salespeople s
  INNER JOIN dealerships d
    ON d.dealership_id=s.dealership_id
  WHERE d.city='Los Angeles' AND d.state='CA';
```

Here are the first few rows of the output. You can see the guest list of customers and employees from Los Angeles, CA, after running the `UNION` query:

```
first_name  |   last_name   | guest_type
------------+---------------+------------
 Abbott     | Poupard       | Customer
 Abe        | Auden         | Customer
 Abelard    | Clayborn      | Customer
 Ad         | Bazely        | Customer
```

You have practiced creating dynamic datasets using subqueries, CTEs, and views. You have also practiced combining datasets from multiple tables using `JOIN` and `UNION`. Now, it is time to put all these in a real-world usage scenario, which you will work on in this chapter’s activity in the next section.

Activity 7

The business users have been complaining that while they are able to see the sales data and see `customer_id` values in the `sales` table, they have to pull data from the `customers` table with an ID filter to see the customer name. Can you run a query to return all columns of the `customers` table and the `sales` table combined, based on the criteria that the records from the two tables belong to the same customer?Now that the users can see each customer's name with the sales figure, they want to see the columns from the `customers` table, `products` table,

and `sales` table using one query. You should return all columns of the three tables.It is known that internet sales do not have associated dealerships. To get a comprehensive view of sales, the business users want to see all columns from the `sales` table, together with all the columns from the `dealerships` table, if the sales belong to a certain dealership.

Summary

This chapter focused on defining datasets from existing datasets. It began by highlighting how `SELECT` queries produce two-dimensional relations and how these relations can be used as tables in a database by creating a subquery. To avoid duplicated coding and to simplify complex subqueries, you can build CTEs and views based on the subqueries.Then, it introduced the concept of joining tables to combine data from multiple tables into one dataset. The basics of joins were explained, including inner joins and outer joins. Finally, this chapter discussed set operations, including `UNION`, `UNION ALL`, `INTERSECT`, and `EXCEPT`. With these topics, you will be able to build advanced research by combining data from multiple tables out of complex relationships.In the next chapter, we will discuss how to aggregate data from raw tables into grouped and distilled datasets.c

8 Aggregating Data with GROUP BY

Join our book community on Discord

 https://packt.link/EarlyAccessCommunity

In addition to just seeing individual rows of data, it is also interesting to understand the statistical properties of an entire column or table. For example, say you just received the sample dataset of ZoomZoom, which specializes in car and electronic scooter retailing. You are wondering about the number of customers that this ZoomZoom database contains. You could select all the data from the table and then see how many rows were pulled back, but it would require pulling the entire dataset and processing it manually, which is incredibly painful. Luckily, there are functionalities provided by SQL that can be used to perform this type of calculation on large groups of rows. These functionalities are called **aggregations**.In this chapter, you will learn about the functions involved in aggregation operations and apply different levels of aggregation with the `GROUP BY` clause. You will learn how to enhance `GROUP BY` with filter conditions using the `HAVING` clause, and how all these features are integrated into the `SELECT` statement.The following topics are covered in this chapter:

- Aggregating data
- Aggregating with the `GROUP BY` clause
- Applying the `HAVING` clause

With these topics, you will be able to pull out useful statistics from raw data in your database and understand the big picture.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in *Chapter 1*, *Introduction to Data Management Systems*.

Aggregating data

Statistical calculation is done through aggregate functions. Aggregate functions take in one or more columns with multiple rows and return a number based on those columns. To use aggregate functions, you simply put them in the `SELECT` statement with the designated table and column on which you want to aggregate. For example, this SQL returns the latest hire date of all salespeople:

```
SELECT MAX(hire_date) FROM salespeople;
```

Here, the `max` function looked at all the values in the `hire_date` column and returned the `max` value (i.e, the latest date). The following shows how the output appears:

```
          max
---------------------
 2024-10-20 00:00:00
```

The following table provides a summary of the major aggregate functions that are used in SQL:

Function	**Explanation**
`COUNT(columnX)`	Counts the number of rows in `columnX` that have a non- `NULL` value.
`COUNT(*)`	Counts the number of rows in the output table, regardless of whether there is `NULL` or not.
`MIN(columnX)`	Returns the minimum value in `columnX` . For text columns, it returns the value that would appear first alphabetically. `NULL` values will be ignored.
`MAX(columnX)`	Returns the maximum value in `columnX` . For text columns, it returns the value that would appear last alphabetically. `NULL` values will be ignored.
`SUM(columnX)`	Returns the sum of all values in `columnX` . `NULL` values will be ignored.
`AVG(columnX)`	Returns the average of all values in `columnX` . `NULL` values will be ignored.
`STDDEV(columnX)`	Returns the sample standard deviation of all values in `columnX` . `NULL` values will be ignored.
`VAR(columnX)`	Returns the sample variance of all values in `columnX` . `NULL` values will be ignored.
`REGR_SLOPE(columnX, columnY)`	Returns the slope of linear regression for `columnX` as the response variable and `columnY` as the predictor variable.
`REGR_INTERCEPT( columnX, columnY)`	Returns the intercept of linear regression for `columnX` as the response variable and `columnY` as the predictor variable.

`CORR(columnX, columnY)`	Calculates the Pearson correlation between `columnX` and `columnY` in the data.

Table 8.1: Major aggregate functionsThe most frequently used aggregate functions include `SUM`, `AVG`, `MIN`, `MAX`, `COUNT`, and `STDDEV`. Aggregate functions can be used within the `SELECT` statement to calculate aggregate values. You can also apply the `WHERE` clause to limit the aggregation to specific subsets of data. For example, if you want to know how many customers ZoomZoom has in California, you could use the following query:

```
SELECT COUNT(*) FROM customers WHERE state='CA';
```

This results in the following output:

```
count
------
 5038
```

You can also perform arithmetic calculations with aggregate functions. In the following query, you can divide the count of rows in the `customers` table by two:

```
SELECT COUNT(*)/2 FROM customers;
```

This query will return `25000`.You can use aggregate functions with each other in mathematical ways. If you want to calculate the average value of a specific column, you can use the `AVG` function. For example, to calculate the average **manufacturer's suggested retail price** (**MSRP**) of products at ZoomZoom, you can use the `AVG(base_msrp)` function in a query. In addition, you can also build the `AVG` function using `SUM` and `COUNT`, as follows:

```
SELECT SUM(base_msrp)/COUNT(*) AS avg_base_msrp
FROM products;
```

The result will be similar to the result of this query:

```
SELECT AVG(base_msrp) AS avg_base_msrp
FROM products;
```

A frequently seen scenario is a calculation involving the `COUNT` function. First of all, you can count the number of non-`NULL` values in a column. For example, you can use the `COUNT` function to count the total number of non-`NULL` ZoomZoom customers by running the following query:

```
SELECT COUNT(customer_id) FROM customers;
```

The `COUNT` function will return the number of rows with a non-`NULL` value in the column. Since the `customer_id` column is a primary key of the `customers` table and will not be `NULL`, the count of non-`NULL` `customer_id` values is the same as the number of rows in the `customers` table. Thus, the query will return the following output:

```
 count
-------
```

```
 50000
```

As shown here, the `COUNT` function works with a single column and counts how many non-`NULL` values it has. However, if the column has at least one `NULL` value, the result will not be the same as the total number of rows. To get a count of the number of rows in that situation, you could use the `COUNT` function with an asterisk in brackets, `(*)`, to get the total count of rows:

```
SELECT COUNT(*) FROM customers;
```

This query will also return `50000`. Now, let's say you run the following query:

```
SELECT COUNT(state) FROM customers;
```

The query will return the following:

```
count
-------
 44533
```

The result is less than 50,000, because there are rows where the state column contains `NULL`, and these rows are not counted. The preceding result shows that there are 44,533 rows with a valid non-`NULL` value in the `state` column.One of the major themes you will find in data analytics is that analysis is fundamentally only useful when there is a strong variation in the data. A column where every value is exactly the same is not a particularly useful column. To identify this potential issue, it often makes sense to determine how many distinct values there are in a column. To measure the number of distinct values in a column, you can use the `COUNT DISTINCT` function. The structure of such a query would look as follows:

```
SELECT COUNT (DISTINCT {column1}) FROM {table1};
```

Here, `{column1}` is the column you want to count and `{table1}` is the table with the column.For example, say you want to verify that your customers are based in all 50 states of the US, possibly with the addition of Washington, D.C., which is technically a federal territory but is treated as a state in your system. For this, you need to know the number of unique states in the customer list. You can use `COUNT(DISTINCT expression)` to process the query:

```
SELECT COUNT(DISTINCT state) FROM customers;
```

This query returns the following output:

```
count
-------
    51
```

This result shows that you do have a national customer base in all 50 states and Washington, D.C.. You can also calculate the average number of customers per state as well as assign an alias to the resulting column using the following SQL:

```
SELECT
  COUNT(customer_id)::numeric / COUNT(DISTINCT state)
```

```
    AS Customer_per_state
FROM customers;
```

This query returns the following output:

```
  customer_per_state
----------------------
 980.3921568627450980
```

Note that in the preceding SQL, the count of customer ID is cast as numeric. The reason you must cast this as numeric is that the `COUNT` function always returns an integer. PostgreSQL treats integer division differently from float division; in float division, it will ignore the decimal part of the result. For example, dividing 7 by 2 as integers in PostgreSQL will give you `3` instead of `3.5`. In the preceding example, if you do not specify the casting, the SQL and its result will be as follows:

```
SELECT
  COUNT(customer_id) / COUNT(DISTINCT state)
    AS Customer_per_state
FROM customers;
```

You will get this output, which is missing the decimal portion:

```
customer_per_state
--------------------
                980
```

To get a more precise answer with a decimal part, you must cast one of the numbers as a float. There is also an easier way to convert an integer into a float, which is to multiply it by `1.0`. As `1.0` is a numeric value, its calculation with an integer value will result in a numeric value, too. For example, the following SQL will generate the same output as the SQL in the previous code block :

```
SELECT
  COUNT(customer_id) * 1.0 / COUNT(DISTINCT state)
FROM customers;
```

In the next section, you will work on an exercise to learn how to use aggregate functions as part of data analysis.

Exercise 8.1: Using aggregate functions to analyze data

In this exercise, you will analyze and calculate the price of a product using different aggregate functions. For instance, say you are curious about the data at your company and interested in understanding some of the basic statistics around ZoomZoom product prices (e.g., the lowest price, highest price, average price, and standard deviation of the price for all the products the company has ever sold). You will achieve this in the coming exercise.Perform the following steps to complete this exercise:

1. Open `psql` and connect to the `sqlda` database.

2. Calculate the lowest price, highest price, average price, and standard deviation of the price using the `MIN`, `MAX`, `AVG`, and `STDDEV` aggregate functions, respectively, from the `products` table:

```
SELECT
  MIN(base_msrp),
  MAX(base_msrp),
  AVG(base_msrp),
  STDDEV(base_msrp)
FROM products;
```

The preceding code will produce an output similar to this:

```
  min   |    max    |        avg         |     stddev
--------+-----------+--------------------+----------------
 349.99 | 115000.00 | 33358.327500000000 | 44484.40866379
```

From the preceding output, you can see the minimum, maximum, average, and standard deviation of the price. You can identify questions, concerns, and further research directions by analyzing these numbers. For example, the significant difference between the minimum and maximum prices suggests that they belong to two completely different product categories. This may indicate the need to divide the products into smaller groups, which will be covered in the next section.

Note

Your results may vary in comparison to the preceding output, probably because your PostgreSQL instance may be configured to show a different number of decimal points in the output. The other reason for the difference in outputs could be that the data contained in the database has been modified from what it was when the original database was created from the dump file. However, the key objective here is to demonstrate how you can use the aggregate functions to analyze data.

In this exercise, you used aggregate functions to learn about the basic statistics of prices. Next, you will use aggregate functions with the `GROUP BY` clause.

Aggregating with GROUP BY clause

So far, you have used aggregate functions to calculate statistics for an entire table. However, most times, you are interested in not only the aggregate values for a whole table but also the values for subgroups in the table. To illustrate this, refer back to the `customers` table. You know that the total number of customers is 50,000. However, you might want to know how many customers there are in each state. But how can you calculate this?You could determine how many states there are with the following query:

```
SELECT DISTINCT state FROM customers;
```

You will see 50 distinct states, Washington, D.C., and `NULL` returned as a result of the preceding query, totaling 52 rows. Once you have the list of states, you could then run the

following query for each state:

```
SELECT COUNT(*) FROM customers
WHERE state = '{state}';
```

Although you can do this, it is incredibly painful and can take a long time if there are many states. The `GROUP BY` clause provides a much more efficient solution.

The GROUP BY clause

`GROUP BY` is a clause that divides the rows of a dataset into multiple groups based on some sort of key that is specified in the clause. An aggregate function is then applied to all the rows within each group to produce a single number for that group. The `GROUP BY` key and the aggregate value for the group are then displayed in the SQL output. The following diagram illustrates this general process:

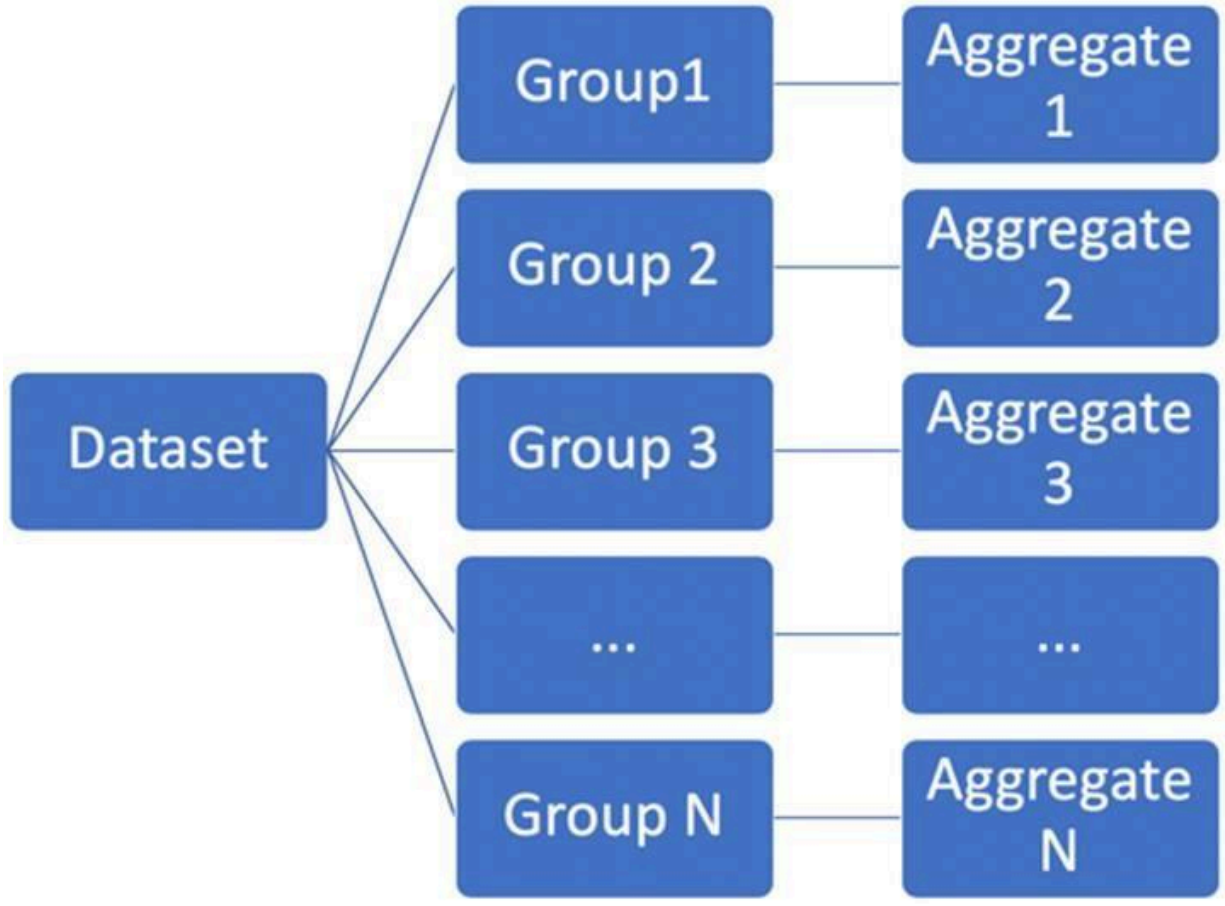

Figure 8.1: General GROUP BY computational model

In the preceding diagram, you can see that the dataset has multiple groups (*Group 1*, *Group 2*, …, *Group N*). Here, the aggregate function is applied to all the rows in *Group 1* and generates the result, *Aggregate 1*. Then, the aggregate function is applied to all the rows in *Group 2* and

generates the result, *Aggregate 2*, and so on.The `SELECT` statements with the `GROUP BY` clause have the following structure:

```
SELECT {KEY}, {AGGFUNC(column1)}
FROM {table}
GROUP BY {KEY}
```

Here, `{KEY}` is a column or columns or an expression that is used to create individual groups. For each value of `{KEY}`, a group is created. `{AGGFUNC(column1)}` is an aggregate function on a column that is calculated for all the rows within each group. `{table}` is the table or set of joined tables from which rows are separated into groups.To illustrate this point, you can count the number of customers in each US state using a `GROUP BY` query:

```
SELECT state, COUNT(*) FROM customers GROUP BY state;
```

The computational model looks like this:

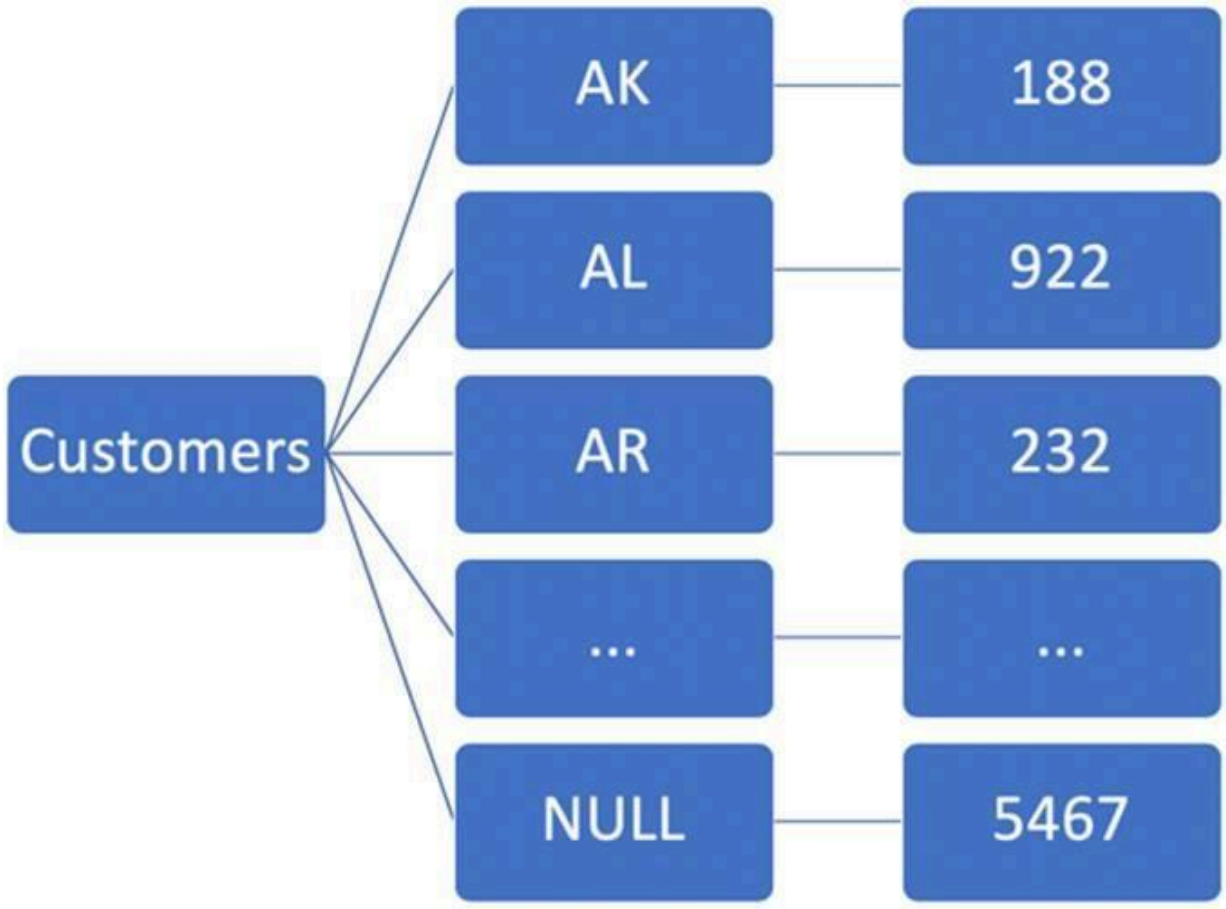

Figure 8.2: Customer count by the state computational model

Here, `AK`, `AL`, `AR`, and the other keys are abbreviations for US states. This grouping is a two-step process. In the first step, SQL will create groups based on the existing states, one group for each state, labeling the group with the state, like what is shown in the preceding figure.

SQL will then allocate customers into different groups based on their states as shown in the figure below.

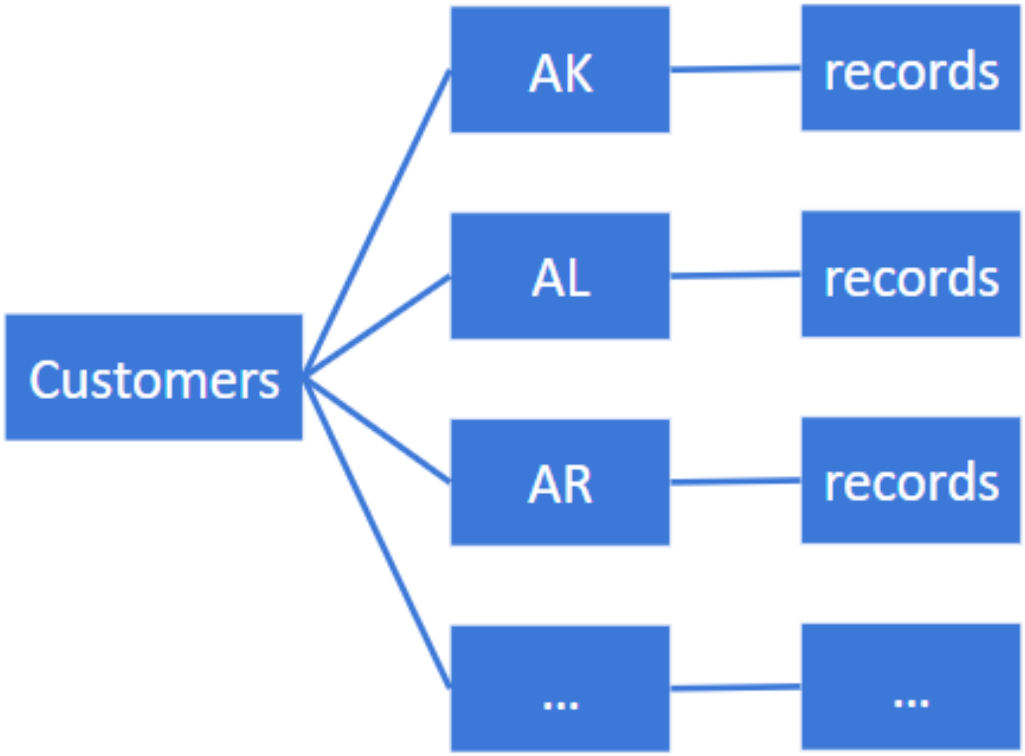

Figure 8.2: Customer grouping by states

Once all the customers are allocated to their respective state groups, the execution goes into the second step. In this step, SQL will apply the aggregate function to each group and associate the result with the group label, which is `state` in this case. The output of the SQL will be a set of aggregate function results with its `state` label. Have a look at the following figure:

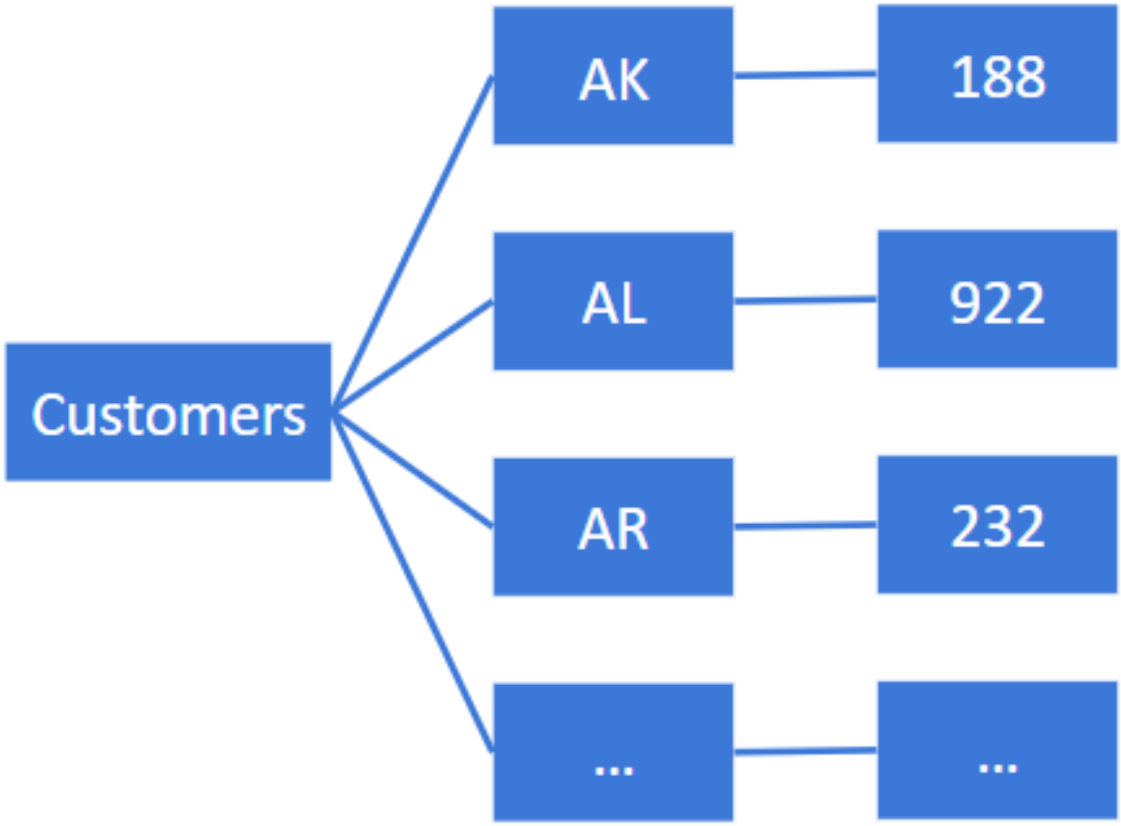

Figure 8.3: Customer grouping by with aggregation result

The first few records of the result are shown as follows:

```
 state | count
-------+-------
 KS    |   619
       |  5467
 CA    |  5038
 NH    |    77
 OR    |   386
```

The `{KEY}` value for the `GROUP BY` operation can also be an expression. The underlying example counts customers based on the year they were added to the database.In the following code, we can see that the year was the result of the `TO_CHAR` function on the `date_added` column:

```
SELECT TO_CHAR(date_added, 'YYYY'), COUNT(*)
FROM customers
GROUP BY TO_CHAR(date_added, 'YYYY');
```

The result of this SQL is as follows:

```
to_char | count
---------+-------
 2019    |  5348
 2020    |  5373
 2025    |   747
 2023    |  5349
 2017    |  5358
 2016    |  5292
 2024    |  5493
 2022    |  5375
 2015    |   743
 2018    |  5452
 2021    |  5470
```

You can also use the column number to indicate the `GROUP BY` column:

```
SELECT TO_CHAR(date_added, 'YYYY'), COUNT(*)
FROM customers
GROUP BY 1;
```

This SQL will return the same result as the previous one, which used the column name in the `GROUP BY` clause.If you want to return the output in alphabetical order, simply apply the `ORDER BY` clause using the following query:

```
SELECT state, COUNT(*)
FROM customers
GROUP BY state
ORDER BY state;
```

Alternatively, you can write the following with the column order number in `GROUP BY` and `ORDER BY` instead of column names:

```
SELECT state, COUNT(*)
FROM customers
GROUP BY 1
ORDER BY 1;
```

Often, though, you may be interested in ordering the aggregates themselves. You may want to know the number of customers in each state in increasing order so that you know which state has the least number of customers. You can then use this result to make a business decision, such as launching a new marketing campaign in the states where you don't have enough presence. This would require you to order the aggregates themselves. The aggregates can also be ordered using `ORDER BY`, as follows:

```
SELECT state, COUNT(*)
FROM customers
GROUP BY state
ORDER BY COUNT(*) ASC;
```

This query gives you the following output (first five rows):

```
state | count
-------+-------
 VT    |    16
```

```
 WY    |    23
 ME    |    25
 RI    |    47
 NH    |    77
```

You may also want to count only a subset of the data, such as the total number of male customers in a particular state, by applying the `WHERE` clause. To calculate the total number of male customers, you can use the following query:

```
SELECT state, COUNT(*)
FROM customers
WHERE gender='M'
GROUP BY state
ORDER BY State;
```

As shown here, grouping by one column can provide some great insight. You can get different aspects of the entire dataset, as well as any subset that you may think of. You can use these characteristics to construct a hypothesis and try to verify it. For example, you can identify the sales and the count of customers in each state, or better yet, the count of a specific subgroup of customers. From there, you can run your data analytics. If you can find a relationship between the sales amount and the particular group of customers, you may be able to figure out some way to reach out to more of these customers and thus increase the sales, or to figure out why other groups of customers are not as motivated.While `GROUP BY` with one column is helpful, you can go even further and use `GROUP BY` on multiple columns. For instance, say you wanted to get a count of the number of customers ZoomZoom has, not only in each state but also how many male and female customers it has in each state. You can find this using multiple `GROUP BY` columns, as follows:

```
SELECT state, gender, COUNT(*)
FROM customers
GROUP BY state, gender
ORDER BY state, gender;
```

This gives you the following result (first five rows):

```
state | gender | count
------+--------+-------
 AK    | F      |   101
 AK    | M      |    87
 AL    | F      |   433
 AL    | M      |   489
 AR    | F      |   112
```

Any number of columns can be used in a `GROUP BY` operation in the same way as illustrated in the preceding example. In this case, SQL will create one group for each unique combination of column values, such as one group for `state=AK` and `gender=F`, another for `state=AK` and `gender=M`, and so on, then apply the aggregate function to each group and label the result with a value from all the grouping columns.Now, test your understanding by implementing the `GROUP BY` clause in an exercise.

Exercise 8.2: Calculating the cost by product type using GROUP BY

In this exercise, you will analyze and calculate the cost of products using aggregate functions and the `GROUP BY` clause. The marketing manager wants to know the minimum, maximum, average, and standard deviation of the price for each product type that ZoomZoom sells for a marketing campaign. Perform the following steps to complete this exercise:

1. Open `psql` and connect to the `sqlda` database
2. Calculate the lowest price, highest price, average price, and standard deviation of the price using the `MIN`, `MAX`, `AVG`, and `STDDEV` aggregate functions from the `products` table and use `GROUP BY` to check the price of all the different product types:

```
SELECT product_type,
  MIN(base_msrp),
  MAX(base_msrp),
  AVG(base_msrp),
  STDDEV(base_msrp)
FROM products
GROUP BY 1
ORDER BY 1;
```

You should get the following result:

```
product_type |   min    |    max    |         avg          |      stddev
-------------+----------+-----------+----------------------+------------------
 automobile   | 35000.00 | 115000.00 |   79250.000000000000 |   30477.45068079
 scooter      |   349.99 |    799.99 | 578.5614285714285714 | 167.971085947212
```

From the preceding output, the marketing manager can check and compare the price of various products that ZoomZoom sells for the campaign.This exercise provides you with an example of how `GROUP BY` can be used in a query. Now that you understand the fundamentals of `GROUP BY`, you can dive deep into some of its advanced usage patterns.

Grouping sets

It is very common to check the statistical characteristics of a dataset from several different perspectives. For instance, say you wanted to count the total number of customers you have in each state, while simultaneously, you also wanted the total number of male and female customers you have in each state. One way you could accomplish this is by using the `UNION ALL` keyword, which was discussed in *Chapter 7*, *Defining Datasets from Existing Datasets*:

```
  SELECT state, NULL as gender, COUNT(*)
  FROM customers
  GROUP BY 1, 2
UNION ALL
  SELECT state, gender, COUNT(*)
  FROM customers
  GROUP BY 1, 2;
```

Fundamentally, what you do here is create multiple sets of aggregation, one grouped by state and another grouped by state and gender, and then union them together. Thus, this operation is called grouping sets, which means multiple sets are generated using different levels of

`GROUP BY`. However, using `UNION ALL` is tedious and can involve writing lengthy queries. An alternative way to do this is to use the `GROUPING SETS` clause of `GROUP BY`. This allows a user to create multiple sets of grouping in the same result set, similar to the `UNION ALL` statement. For example, using the `GROUPING SETS` keyword, you could rewrite the previous `UNION ALL` query, like so:

```
SELECT state, gender, COUNT(*)
FROM customers
GROUP BY GROUPING SETS (
  (state), (state, gender)
);
```

This query will generate the following result:

```
 state | gender | count
-------+--------+-------
 SD    | M      |    69
 GA    | M      |   647
 AK    | F      |   101
… (skipping some rows)
 NH    |        |    77
 OR    |        |   386
 ND    |        |    93
```

You can see that the first part of the result is the grouping set of `(state, gender)`, followed by the grouping set of `(state)`, which does not have any gender in the set. This is the same output as the previous `UNION ALL` query, which contains multiple sets of grouping.

Ordered set aggregates

Up until this point, none of the aggregates discussed depended on the order of the data. That is because none of the aggregate functions (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, and so on) you have encountered so far were ordinal. You can order the result using `ORDER BY`, but this is not required to complete the calculation, nor will the order impact the result. However, there is a subset of aggregate statistics that depend on the order of the column to calculate. For instance, the median of a column is something that requires the order of the data to be specified. To calculate these use cases, SQL offers a series of functions called ordered set aggregate functions. The following table lists the main ordered set aggregate functions:

Function	**Explanation**
`mode()`	Returns the value that appears most often. In the case of a tie, it returns the first value in order.
`Percentile_cont(fraction)`	Returns a value corresponding to the specified fraction in the ordering, interpolating between adjacent input terms if needed.
`Percentile_disc(fraction)`	Returns the first input value whose position in the ordering equals or exceeds the specified fraction.

Table 8.2: Major ordered set aggregate functionsThese functions are used in the following format:

```
SELECT {ordered_set_function}
WITHIN GROUP (ORDER BY {order_column})
FROM {table};
```

Here, `{ordered_set_function}` is the ordered set aggregate function, `{order_column}` is the column used to order the results returned by the function, and `{table}` is the table the column is in. For example, you can calculate the median price of the `products` table by using the following query:

```
SELECT
  PERCENTILE_CONT(0.5)
    WITHIN GROUP (ORDER BY base_msrp) AS median
FROM products;
```

The reason you use `0.5` is that the median is the 50th percentile, which is 0.5 as a fraction. This gives you the following result:

```
median
--------
 749.99
```

With ordered set aggregate functions, you now have the tools for calculating virtually any aggregate statistic of interest for a dataset.So far, the discussion on `GROUP BY` focused on getting the result set from raw data. In the next section, you will learn about the processing of results, notably, applying a filter on the results.

Applying the HAVING clause

You learned about the `WHERE` clause in previous chapters when you worked on `SELECT` statements, which select only certain rows meeting the condition from the original table for later queries. You also learned how to use aggregate functions with the `WHERE` clause in the previous section. Bear in mind that the `WHERE` clause will always be applied to the original dataset. This behavior is defined by the SQL `SELECT` statement syntax, regardless of whether there is a `GROUP BY` clause or not. As discussed earlier, `GROUP BY` is a two-step process. In the first step, SQL selects rows from the original table or table set to form aggregate groups. In the second step, SQL calculates the aggregate function results. When you apply a `WHERE` clause, its conditions are applied to the original table or table set, which means it will always be applied in the first step. Sometimes, you are also interested in filtering the aggregate function results. This can only happen after the aggregation has been completed and results yielded; thus, it is part of the second step of `GROUP BY` processing. For example, when doing the customer counts, perhaps you are only interested in states that have at least 1,000 customers. Your first instinct may be to write something such as this:

```
SELECT state, COUNT(*)
FROM customers
```

```
WHERE COUNT(*)>=1000
GROUP BY state;
```

However, you will find that the query does not work and gives you the following error:

```
ERROR:  aggregate functions are not allowed in WHERE
LINE 3: WHERE COUNT(*)>=1000
              ^
```

This is because `COUNT(*)` is calculated at the second step on the aggregated groups. Thus, this filter can only be applied to the aggregated groups, not the original dataset. So, using the `WHERE` clause on aggregate functions will produce an error. To use the filter on aggregate function results, you need to use a new clause: `HAVING`. The `HAVING` clause is similar to the `WHERE` clause, except it is specifically designed for `GROUP BY` queries. It applies to the filter condition on the aggregated groups instead of the original dataset at the second step of `GROUP BY` execution. The general structure of a `GROUP BY` operation with a `HAVING` statement is as follows:

```
SELECT {KEY}, {AGGFUNC(column1)}
FROM {table1}
WHERE {Original_dataset_Conditions}
GROUP BY {KEY}
HAVING {AGGFUNC(any_column)_Conditions}
```

Here, `{KEY}` is a column or an expression that is used to create individual groups, `{AGGFUNC(column1)}` is an aggregate function on a column that is calculated for all the rows within each group, `{table}` is the table or set of joined tables from which rows are separated into groups, and `{Original_dataset_Conditions}` is any filter condition that you will apply to the columns in the original table. `{AGGFUNC(any_column )_Conditions}` is a condition that you would put in a filter condition involving an aggregate function.Next, you will enhance your understanding by implementing an exercise while using the `HAVING` clause.

Exercise 8.3: Calculating and displaying data using the HAVING clause

In this exercise, you will calculate and display data using `GROUP BY` with the `HAVING` clause. The sales manager of ZoomZoom wants to know the customer count for the states that have at least 1,000 customers who have purchased any product from ZoomZoom. Perform the following steps to help the manager extract the data:

1. Open `psql` and connect to the `sqlda` database.
2. Calculate the customer count by states with at least 1,000 customers using the `HAVING` clause:

```
SELECT state, COUNT(*)
FROM customers
GROUP BY state
HAVING COUNT(*) >= 1000
ORDER BY state;
```

This query will give you the following output:

```
 state | count
-------+-------
 CA    |  5038
 CO    |  1042
 DC    |  1447
 FL    |  3748
 GA    |  1251
 IL    |  1094
 NC    |  1070
 NY    |  2395
 OH    |  1656
 PA    |  1419
 TX    |  4865
 VA    |  1555
       |  5467
```

Here, you can see the states that have more than 1,000 ZoomZoom customers, with `CA` having `5038`, the highest number of customers, and `CO` having `1042`, the lowest number of customers. You can also see that there are 5,467 rows where `state` is `NULL`. These are the rows that did not get counted when you ran the `COUNT(state)` statement earlier, resulting in 44,533 of `COUNT(state)` out of the 50,000 rows in the `customers` table.With all these discussions and exercises, you are now able to discover the statistical characteristics of datasets and their subsets. In the next activity, you will resolve some analytical questions by applying these skills.

Activity 8

Please help your business users with the following requests:

1. The business users would like to calculate the total number of unit sales the company has made.
2. They also want to calculate the total sales amount in dollars for the states of New Jersey, California, and Florida. The sales of each state should be reported separately.
3. Finally, they would like to filter out the state where the total sales are less than $10M ($10,000,000).

Summary

This chapter focused on aggregation operations, including the aggregation functions provided by SQL, the `GROUP BY` clause, the `HAVING` clause, and how these features integrate into the `SELECT` statement. With these topics, you will be able to distill statistics from raw data stored in your database and come up with high-level observations.In the next chapter, you will learn about another set of functions, the window functions, which evaluate each row’s position within the dataset.

9 Inter-Row Operation with Window Functions

Join our book community on Discord

https://packt.link/EarlyAccessCommunity

You learned about scalar functions such as `CASE WHEN`, `COALESCE`, and `NULLIF` in *Chapter 5, Presenting Data with SELECT*. These functions receive data from a single row and produce a result for this row. The result of these functions is only determined by the data value in the row and has nothing to do with the dataset it is in. You also learned aggregate functions such as `SUM`, `AVG`, and `COUNT` in *Chapter 8, Aggregating Data*, with `GROUP BY`. These functions receive data from a dataset with multiple rows and produce a single result for this dataset. Both types of functions are useful in different scenarios.You may also want to know the characteristics of a data point regarding its position in the dataset. A common request is to rank rows based on a certain column's value, such as exam grade or product price. Rank is determined by both the measurement itself and the dataset it is in. A bicycle's price rank within the bicycle group is obviously different than its rank within the entire product list. For the same reason, within the dataset, there might be subgroups, which are also called partitions, that the rank is based on. For example, you may want to check the student's rank both within this semester's class and among all students of the same course this year. For each partition,

the rows related to the calculation form a window. The rank is defined based on the value of the row, the window on which it is applied, and the dataset itself. The function used to perform this type of calculation is called a window function.The following topics are covered in this chapter:

- Defining window functions
- Using advanced window definitions

With this, you will be able to identify the position of a row within its dataset and its relationships with other rows within the same dataset.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in Chapter 1, *Introduction to Data Management Systems*.

Defining window functions

Continuing with the discussion on window functions, consider, for instance, that you want to find the first customers of ZoomZoom to offer customer loyalty promotions. To put it more technically, this means you want to rank every customer in order according to the date they became a customer, with the earliest customer being ranked 1, the second-earliest customer being ranked 2, and so on. You can get all the customers using the following query:

```
SELECT first_name, last_name, date_added
FROM customers
ORDER BY date_added;
```

You can order the customers from the earliest to the most recent, copy the output to an Excel spreadsheet, and assign a row number to each row so that you have the rank for each customer. However, this is not automated and is error-prone. Here, SQL provides several ways using which you can achieve this ranking. Later in this chapter, you will learn how to assign numbers to ordered records by using the `RANK` window function. But before that, you can always order the customers by date, count how many customers have already joined before a specific date, and derive the rank of the customer by the date of

joining. Thus, you can first use an aggregate function to get the dates and order them that way:

```
SELECT date_added, COUNT(*)
FROM customers
GROUP BY date_added
ORDER BY date_added;
```

The following is the first few rows of the preceding code:

```
     date_added      | count
---------------------+-------
 2015-11-09 00:00:00 |    11
 2015-11-10 00:00:00 |    13
 2015-11-11 00:00:00 |    12
 2015-11-12 00:00:00 |    19
 2015-11-13 00:00:00 |    23
```

This result gives the dates in a ranked order. You can calculate each customer's rank by adding up the number of customers who joined ZoomZoom before that customer's join date. However, this approach is still manual, requires extra calculation, and still does not directly provide rank information. This is where window functions come into play. Window functions will perform the same processing: order relevant rows of the current row and calculate the rank of the current row based on the number of relevant rows. For things such as ranks, this is exactly what you need.To better understand this, you will see what a window function query looks like in the next section.

The basics of window functions

The following is the basic syntax of a window function:

```
SELECT {columns},
  {window_func} OVER (
    PARTITION BY {partition_key}
    ORDER BY {order_key}
  )
FROM {table};
```

Here, `{columns}` refers to the columns to retrieve from tables for the query, `{window_func}` is the window function you want to use, `{table}` is the table or joined tables you want to pull data from, and the `OVER` keyword indicates where the window definition starts. The window definition in this basic syntax

includes two parts, `{partition_key}` and `{order_key}`. The former is the column or columns you want to partition on, and the latter is the column or columns you want to order by.You might be saying to yourself that you do not know any window functions, but in fact, all aggregate functions can be used as window functions. Now, use `COUNT(*)` in the following query:

```
SELECT first_name, last_name, gender,
  COUNT(*) OVER () as total_customers
FROM customers
```

The first few rows of results are shown here:

```
  first_name    |      last_name       | gender | total_customers
----------------+----------------------+--------+-----------------
 Aaren           | Deeman               | F      |           50000
 Aaren           | Lamlin               | F      |           50000
 Aaron           | Glozman              | M      |           50000
 Aaron           | MacAleese            | M      |           50000
 Ab              | Calloway             | M      |           50000
 Ab              | Ellinor              | M      |           50000
 Ab              | Ferraraccio          | M      |           50000
 Ab              | Giraldo              | M      |           50000
Ab              | Giraldo              | M      |           50000
```

As shown, the query returns the `first_name` and `last_name` values of customers, just like a typical `SELECT` query. However, there is now a new column called `total_customers`. This column contains the count of users that would be created by the following query:

```
SELECT COUNT(*) FROM customers;
```

The preceding query returns 50,000, that is, all the rows in the table, and `COUNT(*)` in the query returns the count as any normal aggregate function would.Now, regarding the other parameters of the query, what happens if you add an `OVER` clause to convert this `COUNT` into a window function, keeping the function as `COUNT` but defining the window using `PARTITION BY`, such as in the following query?

```
SELECT first_name, last_name, gender,
  COUNT(*) OVER (PARTITION BY gender)
FROM customers;
```

The following is the output of the preceding code:

```
   first_name    |      last_name       | gender | total_customers
-----------------+----------------------+--------+-----------------
 Maressa         | Klezmski             | F      | 25044
 Myrtia          | Sprague              | F      | 25044
 Deloria         | Steinson             | F      | 25044
 Jarred          | Bester               | M      | 24956
 Chane           | McGrory              | M      | 24956
```

Here, you can see that the count has now changed to one of two values, `24956` or `25044`. As you use the `PARTITION BY` clause over the `gender` column, SQL divides the dataset into two partitions based on the unique values of this column. Inside each partition, SQL calculates the total count. For example, there are 24,956 males, so the `COUNT` window function for the male partition returns `24956`. For females, the count matches the number of females, and for males, it matches the number of males. You can double-check this with the following query:

```
SELECT gender, COUNT(*) FROM customers GROUP BY 1;
```

Now you see how the partition is defined and used with the `PARTITION BY` clause. When you use this window function, if you did not specify `PARTITION BY`, it is assumed that there is only one partition, which contains the entire dataset. If `PARTITION BY` is specified, the dataset is divided into partitions, and the function is applied within each partition and the result is assigned to each row.Now, what happens if you use `ORDER BY` instead in the `OVER` clause, as follows?

```
SELECT first_name, last_name, gender,
  COUNT(*) OVER (ORDER BY customer_id)
FROM customers;
```

The following are the first few rows of the output of the preceding code:

```
   first_name    |      last_name       | gender | count
-----------------+----------------------+--------+-------
 Arlena          | Riveles              | F      |     1
 Ode             | Stovin               | M      |     2
 Braden          | Jordan               | M      |     3
 Jessika         | Nussen               | F      |     4
 Lonnie          | Rembaud              | F      |     5
```

You will notice something akin to a running count for the total customers. Here, because there is no `PARTITION BY` defined, the whole dataset is processed as a partition. As `ORDER BY` is specified, the rows in the partition are ordered first.

For each unique value in the order, SQL forms a value group, which contains all the rows containing this value. The query then creates a window for each value group and the value groups before it. This is where the *window* in *window function* comes from. The window will contain all the rows in this value group and all rows that are ordered before this value group. An example is shown here:

Figure 9.1: Windows for customers using COUNT(*) ordered by the customer_id window query

Here, the dataset is ordered using `customer_id`, which happens to be the primary key. As such, each row has a unique value and forms a value group. The first value group contains only the first row. The second value group contains the second row. Then, the third value group contains the third row, and so on and so forth.On top of these value groups, windows are created on a per-row basis. The window for row 1 contains value group 1, which contains itself. The window for row 2 contains value group 2, which contains row 2, as well as the value groups before it. That means the window for row 2 contains both row 1 and row 2. Similarly, the window for row 3 contains value group 3 and the value groups before it, which contains all 3 rows. Once the windows are established, the window function is calculated. In this example, this means `COUNT` is applied to every window. Thus, window 1 (the first row) gets `1` as the result since its window 1 contains one row, window 2 (the second row) gets `2` since its window 2 contains two rows, and so on and so forth. The results are applied to every row in this value group if the group contains multiple rows.What happens when you combine `PARTITION BY` and `ORDER BY`? Have a look at the following query:

```
SELECT first_name, last_name, gender,
  COUNT(*) OVER (
    PARTITION BY gender
    ORDER BY customer_id
  )
FROM customers;
```

When you run the preceding query, you get the following result:

```
   first_name    |      last_name        | gender | count
-----------------+-----------------------+--------+-------
 Arlena          | Riveles               | F      |     1
 Jessika         | Nussen                | F      |     2
 Lonnie          | Rembaud               | F      |     3
(skipping some rows...)
 Ode             | Stovin                | M      |     1
 Braden          | Jordan                | M      |     2
 Cortie          | Locksley              | M      |     3
(skipping the rest of rows...)
```

Like the previous query, it appears to be some sort of rank. However, it ranks with consideration to gender. In this particular SQL, the query divides the table into two subsets based on the `gender` column. That is because the `PARTITION BY` clause, like `GROUP BY`, will first divide the dataset into groups (which is called a partition here) based on the value in the `gender` column. Each partition is then used as a basis for the count, with each partition having its own set of windows. The rows are ordered inside the partitions. Since the order is by the unique `customer_id`, each row forms its own value group. Windows are then created based on the value groups and their orders. Each row's window contains itself and the rows before it, and the `COUNT` function is applied to the values. The results are finally assigned to every row in the value groups.This process is illustrated in *Figure 9.2*:

Figure 9.2: Windows for customers listed using COUNT(*) partitioned by gender and ordered by the customer_id window query

This process produces the count you can see. The three keywords, `OVER()`, `PARTITION BY`, and `ORDER BY`, are the foundation of the power of window functions.Now that you understand window functions, you will apply them in the next exercise.

Exercise 9.1: Analyzing Customer Data Fill Rates over Time

In this exercise, you will apply window functions to a dataset and analyze the data. For the last six months, ZoomZoom has been experimenting with various promotions to make their customers more engaged in the sales activity. One way to measure the level of engagement is to measure people's willingness to fill out all fields on the customer form, especially their address. To achieve this goal, the company would like a running total of how many users have filled in their street addresses over time. Write a query to produce these results:

1. Open `psql` and connect to the `sqlda` database.
2. Use window functions and write a query that will return customer information and how many people have filled out their street address. Also, order the list by date. The query will look as follows:

```
SELECT
  customer_id,
  date_added::DATE,
  COUNT(
    CASE
      WHEN street_address IS NOT NULL THEN customer_id
      ELSE NULL
    END
  ) OVER (ORDER BY date_added::DATE)
    as non_null_add,
  COUNT(*) OVER (ORDER BY date_added::DATE)
    as total_add
FROM customers
ORDER BY date_added;
```

You should get a result like this (the first few rows):

```
customer_id | date_added | non_null_add | total_add
------------+------------+--------------+-----------
       30046 | 2015-11-09 |           10 |        11
```

```
 2625 | 2015-11-09 |           10 |        11
13390 | 2015-11-09 |           10 |        11
17099 | 2015-11-09 |           10 |        11
48307 | 2015-11-09 |           10 |        11
 6173 | 2015-11-09 |           10 |        11
18685 | 2015-11-09 |           10 |        11
 7486 | 2015-11-09 |           10 |        11
30555 | 2015-11-09 |           10 |        11
12484 | 2015-11-09 |           10 |        11
35683 | 2015-11-09 |           10 |        11
 8571 | 2015-11-10 |           22 |        24
17832 | 2015-11-10 |           22 |        24
46277 | 2015-11-10 |           22 |        24
```

1. Write a query to see how the numbers of people filling out the street field change over time.

In the previous step, you have already got every customer address ordered by the signup date. The two columns following the signup date column are the number of non-`NULL` addresses and the number of all customer addresses for each rolling day, that is, a sum from the beginning of sales to the current day. By dividing the number of non-`NULL` addresses by the number of all customer addresses, you can get the percentage of customers with non-`NULL` street addresses and derive the percentage of customers with `NULL` street addresses. Tracking this number will provide insight into the way customers interact with your sales force over time. Also, because both numbers of addresses are calculated for each rolling day, the percentage is also for each rolling day. This is an example of different window functions sharing the same window in the same query. You can also rewrite the following query using a `WINDOW` clause to make the query simpler, which will be introduced in the next section:

```
WITH daily_rolling_count as (
  SELECT
    customer_id,
    date_added::DATE,
    COUNT(
      CASE
        WHEN street_address IS NOT NULL
          THEN customer_id
        ELSE NULL
      END
) OVER (ORDER BY date_added::DATE)
  AS non_null_add,
COUNT(*) OVER (ORDER BY date_added::DATE)
  AS total_add
```

```
  FROM customers
)
SELECT DISTINCT
  date_added,
  non_null_add,
  total_add,
  1 - 1.0 * non_null_add/total_add
    AS null_address_percentage
FROM daily_rolling_count
ORDER BY date_added DESC;
```

The result is the following:

```
date_added | non_null_add | total_add | null_address_percentage
------------+--------------+-----------+-------------------------
 2025-02-23 |        44533 |     50000 |  0.10934000000000000000
 2025-02-22 |        44516 |     49980 |  0.10932372949179671869
 2025-02-21 |        44502 |     49965 |  0.10933653557490243170
 2025-02-20 |        44483 |     49943 |  0.10932463007828924974
 2025-02-19 |        44470 |     49930 |  0.10935309433206489085
```

This result will give you the list of the rolling percentage of `NULL` street addresses on each day. You can then provide the full dataset to data analytics and visualization software such as Excel to study the general trend of the data, discover patterns of change, and make suggestions on how to increase the engagement of customers to the company management.With the fundamental concepts of window functions introduced, you will now dive deep into some advanced usage patterns in the next sections.

Using advanced window definitions

Now that you understand the basics of window functions, it is time to introduce a few advanced topics that will make it easier to use them. These include a list of common window functions, the `WINDOW` keyword, and the window frame. Then you will use them in an exercise.

Common window functions

The following table summarizes a variety of statistical functions that are useful. It is also important to emphasize again that all aggregate functions can also be used as window functions (`AVG` , `SUM` , `COUNT` , and so on):

Name	Description
ROW_NUMBER	Number of the current row within its partition, starting from 1. Usually, it is used with a sorted list. For example, the `ROW_NUMBER` values for the series `(10, 9, 9, 8, 8, 7)` are `(1, 2, 3, 4, 5, 6)` .
DENSE_RANK	Rank the current row within its partition without gaps. For example, the `DENSE_RANK` values for the series `(10, 9, 9, 8, 8, 7)` are `(1, 2, 2, 3, 3, 4)` .
RANK	Rank the current row within its partition with gaps. For example, the RANK values for the series `(10, 9, 9, 8, 8, 7)` are `(1, 2, 2, 4, 4, 6)` .
LAG	Return a value evaluated at the row that is at a specified physical offset row before the current row within the partition.
LEAD	Return a value evaluated at the row that is offset rows after the current row within the partition.
NTILE	Divide rows in a partition as equally as possible and assign each row an integer starting from 1 to the argument value.

Table 9.1: Statistical window functionsWith all these functions, you can perform a lot of analysis. However, each function requires its own window definition, which can make the query really long. To improve the efficiency of query writing, PostgreSQL provides the `WINDOW` keyword to help you define a window and use it in multiple functions.

The WINDOW keyword

In many scenarios, your analysis involves running multiple functions against the same window so that you can compare them side by side, and you are very likely running them within the same query. For example, when you are doing some gender-based analysis, you may be interested in calculating a running total number of customers as well as the running total number of customers with a title, using the same partition that is based on gender. You may end up with the following query:

```
SELECT first_name, last_name, gender,
  COUNT(*)
    OVER (
      PARTITION BY gender ORDER BY customer_id
```

```
    ) as total_customers,
  SUM(CASE WHEN title IS NOT NULL THEN 1 ELSE 0 END)
    OVER (
      PARTITION BY gender ORDER BY customer_id
    ) as total_customers_title
FROM customers;
```

Although the query gives you the result, it repeats the `OVER` clause, which is the same for the two functions. Fortunately, you can simplify this by using the `WINDOW` clause to define a generic window for multiple functions in the same query. The `WINDOW` clause facilitates the aliasing of a window.You can simplify the preceding query by writing it as follows:

```
SELECT first_name, last_name, gender,
  COUNT(*)
    OVER w as total_customers,
  SUM(CASE WHEN title IS NOT NULL THEN 1 ELSE 0 END)
    OVER w as total_customers_title
FROM customers
WINDOW w AS (
  PARTITION BY gender ORDER BY customer_id
);
```

This query should give you the same result as the preceding query. However, you do not have to write a long `PARTITION BY` and `ORDER BY` query for each window function. Instead, you simply use an alias with the defined `WINDOW w`. So far in the discussions, a window contains all the rows from the first to the current row in the partition. However, you can also adjust the beginning and end of the windows using the window frame clause. This will be covered in the next subsection.

Window frame

You can define a window frame to specify which rows you want to include in your windows. A `window` function query using the window frame clause would look as follows:

```
SELECT {columns},
  {window_func} OVER (
    PARTITION BY {partition_key}
    ORDER BY {order_key}
{rangeorrows}
  BETWEEN {frame_start} AND {frame_end}
```

```
  )
FROM {table};
```

Here, as you have already seen, `{columns}` refers to the columns to retrieve from tables for the query, `{window_func}` is the window function you want to use, `{partition_key}` is the column or columns you want to partition on, `{order_key}` is the column or columns you want to order by, and `{table}` is the table or joined tables you want to pull data from. What is new here is the window frame clause:

```
{rangeorrows} BETWEEN {frame_start} AND {frame_end}
```

In this clause, `{frame_start}` is a keyword indicating where to start the window frame (instead of the beginning of the partition), `{frame_end}` is a keyword indicating where to end the window frame (instead of the current value group), and `{rangeorrows}` is either the `RANGE` keyword or the `ROWS` keyword. To give further details, `{frame_start}` and `{frame_end}` can be one of the following values:

- `UNBOUNDED PRECEDING`: A keyword that, when used for `{frame_start}`, refers to the first record of the partition
- `{offset} PRECEDING`: A keyword referring to `{offset}` (an integer) rows or ranges before the current row
- `CURRENT ROW`: Refers to the current row
- `{offset} FOLLOWING`: A keyword referring to `{offset}` (an integer) rows or ranges after the current row
- `UNBOUNDED FOLLOWING`: A keyword that, when used for `{frame_end}`, refers to the last record of the partition

By adjusting the window, various useful statistics can be calculated. One such useful statistic is the rolling average. The rolling average is simply the average for statistics in a given time window. Say you want to calculate the seven-day rolling average of sales over time for ZoomZoom. You will need to get the daily sales first by running `SUM … GROUP BY sales_transaction_date`. This will provide you with a list of daily sales, each row being a day with sales. When you order this list of rows by date, the six preceding rows plus the current row will provide you with a window of seven rolling days. Calculating `AVG` over these seven rows will give you the seven-day rolling average of the given day.This calculation can be accomplished with the following query:

```
WITH
  daily_sales as (
    SELECT
      sales_transaction_date::DATE,
      SUM(sales_amount) as total_sales
    FROM sales
    GROUP BY 1
  ),
  moving_average_calculation_7 AS (
    SELECT
      sales_transaction_date,
      total_sales,
      AVG(total_sales) OVER (
        ORDER BY sales_transaction_date
        ROWS BETWEEN 6 PRECEDING and CURRENT ROW
      ) AS sales_moving_average_7,
      ROW_NUMBER() OVER (
        ORDER BY sales_transaction_date
      ) as row_number
    FROM
      daily_sales
    ORDER BY 1
  )
SELECT
  sales_transaction_date,
  CASE
    WHEN row_number>=7 THEN sales_moving_average_7
    ELSE NULL
  END AS sales_moving_average_7
FROM
  moving_average_calculation_7;
```

A natural question when considering the *N*-day moving window is how to handle the first *N*-1 days in the ordered column. In the previous query, the first six rows are defined as `NULL` using a `CASE` statement because in this scenario, the seven-day moving average is only defined if there is seven days' worth of information. Without the `CASE` statement, the window calculation will calculate values for the first seven days using the first few days. For these days, the seven-day moving average is the average of whatever days are in the window. For example, the seven-day moving average for the second day is the average of the first day and the second day, and the seven-day moving average for the sixth day is the average of the first six days. In fact, the `sales_moving_average_7` calculation in the CTE is such an average. It calculates the `AVG(total_sales)` value of the day and the six days prior. If there are fewer than seven days, the average is applied to whatever days it can find. Both of these approaches, of calculation and the `NULL` approach,

mentioned previously can make sense in their respective situations. It is up to the data analyst to determine which one makes the most sense for a particular scenario.Another point of consideration is the difference between using `RANGE` or `ROW` in a frame clause. In the previous example, you used `ROW` as the daily sales, which contained one row per day. `ROW` refers to actual rows and will take the rows before and after the current row to calculate values. `RANGE` should be applied to the values of `{frame_start}` and `{frame_end}` in the `{order key}` column. It differs from `ROW` in that it is based on the value of the order key instead of the number of rows. For example, if you define `{frame_start} as INTERVAL '3' DAY PRECEDING`, that means the starting date is three days before the current row's `sales_transaction_date`. The calculation will include all rows starting that day, instead of counting the number of rows. There are many different options for applying the `RANGE` clause. Please consult the official PostgreSQL documentation if you have any questions.In the following exercise, you will use a rolling window to calculate statistics with ordered data.

Exercise 9.2: Team Lunch Motivation

In this exercise, you will use a window frame to find some important information in your data. To help improve sales performance, the sales team has decided to buy lunch for all salespeople at the company every time they beat the figure for the best daily total earnings achieved over the last 30 days. Write a query that produces the total sales in dollars for a given day and the target that the salespeople must beat for that day, starting from January 1, 2022.Perform the following steps to complete this exercise:

1. Open `psql` and connect to the `sqlda` database.
2. Calculate the total sales for a given day and the target using the following query:

```
WITH
  daily_sales as (
    SELECT
      sales_transaction_date::DATE,
      SUM(sales_amount) as total_sales
    FROM sales
    GROUP BY 1
  ),
  sales_stats_30 AS (
    SELECT
```

```
      sales_transaction_date,
      total_sales,
      MAX(total_sales) OVER (
        ORDER BY sales_transaction_date
        ROWS BETWEEN 30 PRECEDING and 1 PRECEDING
      ) AS max_sales_30
    FROM daily_sales
    ORDER BY 1
  )
SELECT
  sales_transaction_date,
  total_sales,
  max_sales_30
FROM sales_stats_30
WHERE sales_transaction_date>='2021-12-31';
```

You should get the following results:

```
sales_transaction_date |    total_sales     |    max_sales_30
-----------------------+--------------------+--------------------
 2021-12-31            |         546899.892 |         536899.862
 2022-01-01            |  6249.874999999999 |         546899.892
 2022-01-02            | 119599.90800000002 |         546899.892
(skipping rows...)
 2022-01-30            | 357099.81599999993 |         546899.892
 2022-01-31            |  75049.80900000007 | 357099.81599999993
 2022-02-01            | 405149.90299999993 | 357099.81599999993
 2022-02-02            | 258149.75699999993 | 405149.90299999993
 2022-02-03            | 514299.76399999985 | 405149.90299999993
 2022-02-04            | 357549.74899999995 | 514299.76399999985
 2022-02-05            | 290049.84699999995 | 514299.76399999985
```

As you can see, there is a large number of sales on December 31, 2021. Because of this, for the next 30 days, the target is always the value on December 31, 2021. Only after this 30-day window passes does the target shift to the next highest value following 12/31/2021.. Notice the use of a window frame from `30 PRECEDING` to `1 PRECEDING`. By using `1 PRECEDING`, you are removing the current row from the calculation. The result is a 30-day rolling max in the 30 days before the current day.

1. Now you will calculate the total sales each day and compare it with that day's target, which is the 30-day moving average you just calculated in the previous step. The total sales on each day have already been calculated in the preceding SQL in the first common table expression and are later referenced in the main query. So, you can write the following SQL:

```
WITH
  daily_sales as (
    SELECT
      sales_transaction_date::DATE,
      SUM(sales_amount) as total_sales
    FROM sales
    GROUP BY 1
  ),
  sales_stats_30 AS (
    SELECT
      sales_transaction_date,
      total_sales,
      MAX(total_sales) OVER (
        ORDER BY sales_transaction_date
        ROWS BETWEEN 30 PRECEDING and 1 PRECEDING
      ) AS max_sales_30
    FROM
      daily_sales
    ORDER BY 1
  )
SELECT
  sales_transaction_date,
  total_sales,
  max_sales_30
FROM sales_stats_30
WHERE total_sales > max_sales_30
AND sales_transaction_date>='2021-12-31';
```

The result of this query is the maximum daily sales in the previous 30 days for each day, starting from January 1, 2022.With the topics covered in this chapter, you can utilize window functions to analyze each row’s position inside its dataset or partition. You can also define flexible windows for customized analysis. In the following activity, you will use these skills to help your business team with time-series analysis.

Activity 9

From the procurement department’s perspective, the most important measure is the rolling 30-day sales, which helps them determine the replenishing schedule. As such, they would like to calculate the rolling 30-day average for the daily total sales amount for the year 2024. They would also like to calculate which decile each date would be in compared to other days based on their daily 30-day rolling sales amount.Hint: In order to calculate the rolling 30-day average for the daily total sales amount for the year 2024, you will need to calculate the

total sales amount by day for all of the days in the year 2024, plus the 29 days in 2023 that precede 2024.

Summary

This chapter discussed the use of window functions in SQL, which allow for inter-row operations within datasets. Unlike simple functions that operate on a single row, window functions consider the position of data points within the dataset to produce results such as ranks and running totals. With these topics, you will be able to identify the position of a row within its dataset, and its relationships with other rows within the same dataset.So far, you have learned about the SQL statements for all four operations in the **CRUD** (**create, read, update, delete**) data life cycle. You have also learned how to move data in and out of databases and files, as well as accessing databases from Python. Furthermore, you have learned how to use different functions for data transforms or information extraction. At this point, you are already fully equipped to handle common database questions.Starting from the next chapter, you will dive into advanced topics. You will learn how to write performant SQL, how to handle specific data types, and how to apply concepts of statistics to your data analytics. The next chapter will focus on database performance improvement, which will provide you with a solid understanding of the SQL execution plan and database indexing.

10 Performant SQL

Join our book community on Discord


https://packt.link/EarlyAccessCommunity

So far, you have learned how to create SQL statements for different types of requests. You have also seen that you can achieve the same goal with different queries. In reality, not all SQL statements are equal. Some run faster, some run slower. These days, analyses or queries are increasingly becoming a part of a larger service or product. One such example is a travel advisory system that incorporates the current location with nearby businesses and/or travel sites. For such a system to be effective and provide up-to-date navigation information, the database must be analyzed at a rate that keeps up with the speed of the car and the progress of the journey. Any delays in the analysis would significantly impact the application's commercial viability.While it is certainly not the job of a data scientist or data analyst to ensure that the production process and the database are working at optimal efficiency, it is critical that the queries of the underlying analysis are as effective as possible. In this chapter, we will discuss the driving forces behind the SQL execution and how to improve performance.The following topics will be covered in this chapter:

- Scanning the database
- Scanning the index

With these topics, you will have a better understanding of the data retrieval mechanisms for PostgreSQL, which will help you write optimized queries that have better performance.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in *Chapter 1*, *Introduction to Data Management Systems*.

Scanning the database

You have learned that all SQL operations are carried out by **database management systems** (**DBMSs**) such as PostgreSQL. Typically, the DBMS will run these operations in a server's memory, which holds the data to be processed. The problem with this approach is that memory storage is not large enough for modern databases, which are frequently in the scale of gigabytes, if not terabytes. Data in most modern databases is saved on hard disks and uploaded into memory when it is used in a database operation. Yet again, a DBMS can only upload a small part of the database into memory. Whenever it figures that it needs a certain dataset, it must go to the hard disk to retrieve the unit of storage (which is called a **hard disk page** or **block**) that has the required data in it. The process that the PostgreSQL server uses to search through the hard disk to locate the page is known as scanning.SQL-compliant databases, such as PostgreSQL, provide several different methods for scanning, searching, and selecting data. The right scan method to use is dependent on the use case and the state of the

database at the time of scanning. How many records are in the database? Which fields are you interested in? How many records do you expect to be returned? How often do you need to execute the query? These are just some of the questions that you may want to ask when selecting the most appropriate scanning method. However, in order to find out the most appropriate method, you must first be able to identify what scanning method PostgreSQL uses for each query, which is covered in the next section.

Query planning

Before investigating the different methods of executing queries, it is useful to understand how the PostgreSQL server makes various decisions about the types of queries to be used. SQL-compliant databases possess a powerful tool known as a **query planner**, which implements a set of features within the server to analyze a request and decide how to execute the statement. The query planner optimizes different variables within the request with the aim of reducing the overall execution time. These variables are described in greater detail in the official PostgreSQL documentation and include parameters that correspond to the cost of sequential page fetches, CPU operations, and cache size.Interpreting the planner is critical if you want to achieve high performance from a database. Doing so allows you to modify the contents and structure of queries to optimize performance. Unfortunately, query planning can require some practice to be comfortable with. Even the PostgreSQL official documentation notes that plan-reading is an art that deserves significant attention.In this chapter, you will not see the details of how a query planner implements its analysis since there are great technical details involved. However, it is important to understand how to interpret the plan reported by the query planner. You will start with a simple plan and then work your way through more complicated queries and query plans. In the following exercise, you will learn about the `EXPLAIN` command, which displays the plan for a query before it is executed. `EXPLAIN` is a statement provided by PostgreSQL. When you use the `EXPLAIN` command in combination with a SQL statement, the SQL interpreter will not execute the statement but rather return the steps that are going to be executed (a query plan) by the interpreter to return the desired results. An example of using the `EXPLAIN` command is shown in the following exercise.

Exercise 10.1: Interpreting the query planner

In this exercise, you will interpret a query plan of the `emails` table of the `sqlda` database using the `EXPLAIN` command. Then, you will employ a more involved query, searching for dates between two specific values in the `clicked_date` field.Follow these steps to complete the exercise:

1. Open `psql` and connect to the `sqlda` database.
2. Enter the following command to get the query plan of the `emails` table:

```
EXPLAIN SELECT * FROM emails;
```

Information similar to the following will then be presented:

```
Seq Scan on emails  (cost=0.00..9605.58 rows=418158 width=79)
```

Note

For all exercises in this chapter, query analysis metrics will vary depending on system configuration. Thus, you may get outputs that vary from those presented in the exercises and activities. The key point is that the outputs provided in this chapter demonstrate the working of the principles.

The first aspect of the plan that is provided, `Seq Scan`, is the type of scan executed by the query. This `SELECT` command executes a sequential scan, where the database server reads each page from the hard disk, traverses through each record in the pages, compares each record to the criteria in the sequential scan if there is a `WHERE` clause, and returns those records that match the criteria.A sequential scan is the easiest to understand and is guaranteed to work in every scenario. Sequential scan is not the fastest or most efficient option; however, it will always produce the correct result. This is essentially a brute-force scan and, thus, can always be called upon to execute a search. In certain situations, a sequential scan is the most efficient method and will be automatically selected by the PostgreSQL server. This is particularly the case if any of the following is true:

- The table is quite small
- The field used in searching contains many duplicates
- The planner determines that the sequential scan would be equally or more efficient for the given criteria compared to any other scan

Following the `Seq Scan` keyword and the table of its target (`emails`) are a series of measurements: `cost=0.00..9605.58 rows=418158 width=79` . The first measurement reported by the planner, as shown here, is the startup cost. The startup cost is the time spent before the scan starts. This time may be required to first sort the data or complete other preprocessing applications. It is also important to note that the time measured is reported in cost units as opposed to seconds or milliseconds. Often, the cost units are an indication of the number of disk requests or page fetches made, rather than being a measure of absolute time. The reported cost is typically more useful as a means of comparing the performance of various queries, rather than as an absolute measure of time.The next number in the sequence indicates the total cost of executing the query if all available rows are retrieved: `cost=0.00..9605.58` . There are some circumstances in which not all the available rows may be retrieved, but you will learn about that in the *Scanning the index* section of this chapter.The next figure in the plan indicates the total number of rows that are available to be returned if the plan is completely executed. The final figure, as suggested by its name, indicates the width of each row in bytes: `rows=418158 width=79` . Note that the scan happens row by row, so the column(s) you select (including `*`) will not impact the execution plan. You can change the `*` symbol in the previous query to any column in the `emails` table, and the execution plan will not change. The width of the result, however, will change according to the columns you selected. Selecting specified columns instead of using `*` will thus reduce the width of the result, resulting in a small dataset and delivering better performance.

1. Run a query plan for selecting from the `emails` table and set the limit to `5` . This will give you an insight into how PostgreSQL adjusts its execution plan when the SQL changes. Enter the following statement in the PostgreSQL interpreter:

```
EXPLAIN SELECT * FROM emails LIMIT 5;
```

This repeats the previous statement, but the result is limited to the first five records. This query will produce the following output from the planner:

```
Limit  (cost=0.00..0.11 rows=5 width=79)
  ->  Seq Scan on emails  (cost=0.00..9605.58 rows=418158 width=79)
```

Referring to the preceding output, you can see that there are two individual rows in the plan. This indicates that the plan is composed of two separate steps, with the lower line of the plan being executed first. This lower line is a repeat of what is shown in *step 2*. The upper line of the plan is the component that limits the result to only five rows. The `LIMIT` process is an additional cost of the query; however, it is quite insignificant compared to the lower-level plan, which retrieves approximately 418,158 rows at a cost of 9,605.58 page requests. The `limit` stage only returns 5 rows at a cost of 0.11 page requests. Overall, you can see that the `LIMIT` clause does not reduce the execution time, as the full table scan still happened in the database server. So, you should not rely on the `LIMIT` clause to improve the query performance.The overall estimated cost of a request comprises the time taken to retrieve the information from the disk and the number of rows that need to be scanned. The `seq_page_cost` and `cpu_tuple_cost` internal parameters define the cost of the corresponding operations within the tablespace for the database. While not recommended at this stage, these two variables can be changed to modify the steps prepared by the planner.

1. Now, employ a more involved query searching for dates between two specific values in the `clicked_date` column. Enter the following statement into the PostgreSQL interpreter:

```
EXPLAIN
SELECT * FROM emails
WHERE clicked_date
  BETWEEN '2014-01-01' and '2014-02-01';
```

This will produce a query plan similar to this:

```
Gather  (cost=1000.00..9037.59 rows=1 width=79)
  Workers Planned: 2
```

```
->  Parallel Seq Scan on emails  (cost=0.00..8037.49 rows=1 width=79)
      Filter: ((clicked_date >= '2014-01-01 00:00:00'::timestamp without time zone) AND (clicke
```

The first aspect of this query plan to note is that it comprises a few different steps. The lower-level query is to be completed in parallel, as indicated by `Parallel Seq Scan`. PostgreSQL also indicates that it will use two workers to execute this filtered scan. Whether PostgreSQL will use a parallel scan or not depends on the setup of the server, as well as the power of the computer hardware. If the PostgreSQL server feels that a parallel scan is too complex for the hardware or server to handle, it may choose a regular sequential scan. In this example, PostgreSQL believes that parallel scan can provide better performance and decides to utilize two workers for it. Each individual sequence scan should return approximately 54 rows, taking a cost of 8,037.49 to complete. The upper level of the plan is a `Gather` state, which sets up the workers and is executed at the start of the query. You can see here for the first time that the upfront costs are non-zero (`1000`), indicating that the `Gather` step takes some time before the worker execution starts.In this exercise, you worked with the query planner and the output of the `EXPLAIN` command. These relatively simple queries highlighted several features of the SQL query planner as well as the detailed information that is provided by it. It will serve you well in your data science endeavors with a good understanding of the query planner and the rich information returned. Just remember that this understanding will come with time and practice.A sequential scan is the most fundamental approach to scanning. However, most queries will only use a small portion of the data stored in the database. For a large table, going through all available data to retrieve a small number of rows is a huge waste of resources. In the next section, you will learn how to use the index to skip some available pages and focus your scanning on the relevant pages.

Scanning the index

Index scans improve the performance of your database queries. Index scans differ from sequential scans in that index scans execute a preprocessing step before the search of database records can occur.The simplest way to think of an index scan is just like using the index of a text or reference book. When creating such a book, a publisher parses through the contents of the book and writes the page numbers corresponding to each alphabetically sorted topic. Just as the publisher makes the initial effort of creating an index for the reader's reference, you can create a similar index within the PostgreSQL database before you run the queries. This index within the database creates a prepared and organized set of references to the data under specified conditions. When a query is executed and an index is present that contains information relevant to the query, the planner may elect to use the data that was preprocessed and prearranged within the index. Without using an index, the database needs to repeatedly scan through all records, checking each record for information of interest. Even if all the desired information is at the start of the table, without indexing, the search will still scan through all the way to the end. Clearly, this would take a significantly longer time than necessary.There are several different indexing strategies that PostgreSQL can use to create more efficient searches, including **B-trees**, **hash indexes**, and others. Each of these different index types has its own strengths and weaknesses and is, therefore, used in different situations. One of the most frequently used indexes is the B-tree, which is the default indexing strategy used by PostgreSQL and is available in almost all DBMSs. You will learn about B-trees in the coming section.

The B-tree index

The **B-tree index** is a type of extended binary search tree and is characterized by the fact that it is a self-balancing structure, maintaining its own data structure for efficient searching. A generic B-tree structure can be found in *Figure 10.1*, in which you can see that each node in the tree has no more than two elements and each node has at most three children. The branches of the tree terminate at leaf nodes, which, by definition, have no children:

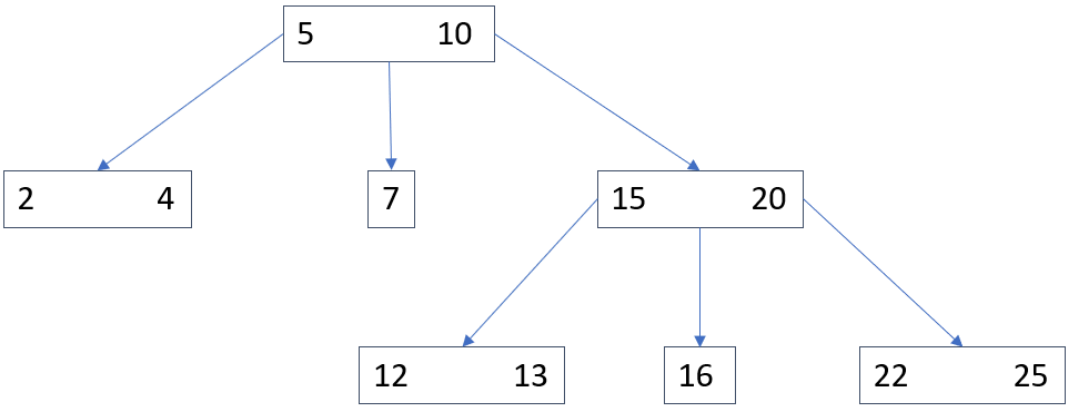

Figure 10.1: Generic B-tree

Using the preceding figure as an example, say you were looking for the page that is represented by number 13 in the B-tree index. You would start at the first node and select whether the number was less than 5 or greater than 10. This would lead you down the right-hand branch of the tree, where you would again choose between less than 15 and greater than 20. You would then select less than 15 and arrive at the location of `13` in the index.You can immediately see that this operation would be much faster than looking through all available values. You can also see that, for performance, the tree must be balanced to allow for an easy path for traversal. Additionally, there must be sufficient information to allow splitting because if you had a tree index with only a few possible values to split on, the traversal would take more time than scanning.Considering B-trees in the context of database searching, you would notice that you need a condition to divide the information with and need sufficient information for a meaningful split. You do not need to worry about the logic of following the tree, as that will be managed by the database itself and can vary depending on the conditions for searching. Even so, it is important for you to understand the strengths and weaknesses of the method to allow you to make appropriate choices when creating the index for optimal performance.To create an index for a set of data, you use the following syntax:

```
CREATE INDEX <index name>
ON <table name>(table column);
```

You can also add additional conditions and constraints to make the index more selective:

```
CREATE INDEX <index name>
ON <table name> (table column) WHERE [condition];
```

You can also specify the type of index:

```
CREATE INDEX <index name> ON <table name>
USING TYPE(table column)
```

For example, say you execute the following query to create a B-tree type index on the `customer_id` column:

```
CREATE INDEX ix_customers ON customers
USING BTREE(customer_id);
```

This outputs the following message:

```
CREATE INDEX
```

This indicates that the index was created successfully.If there is already an index with the same name existing in the database, you can use a `DROP INDEX <index_name>` command to drop and recreate it.In the next exercise, you will start with a simple plan and work your way through more complicated queries and query plans, using index scans.

Exercise 10.02: Creating an index scan

In this exercise, you want to investigate the distribution of customers from the customers table, but the size of this table makes your query very slow. As such, you will create a number of different index scans and investigate the performance characteristics of each of the scans:

1. Open `psql` and connect to the `sqlda` database.
2. Starting with the `customers` table, use the `EXPLAIN` command to determine the cost of the query and the number of rows returned in selecting all the entries with a `state` value of `FO`:

```
EXPLAIN SELECT * FROM customers WHERE state='FO';
```

The output of the preceding code will be similar to the following. Please note that the actual numbers may vary, but the structure will be similar:

```
Seq Scan on customers  (cost=0.00..1660.00 rows=1 width=140)
  Filter: (state = 'FO'::text)
```

Note that there is only 1 row returned, and that the setup cost is `0`, but the total query cost is `1660`.

1. Determine how many unique state values there are, using the `EXPLAIN` command:

```
EXPLAIN SELECT DISTINCT state FROM customers;
```

The output is similar to the following:

```
HashAggregate  (cost=1660.00..1660.51 rows=51 width=3)
  Group Key: state
  ->  Seq Scan on customers  (cost=0.00..1535.00 rows=50000 width=3)
```

Sequential scan is used in this query.

1. Next, create an index called `ix_state` using the `state` column of `customers`. The default index type is B-tree:

```
CREATE INDEX ix_state ON customers(state);
```

1. Rerun the `EXPLAIN` statement from *step 2*:

```
EXPLAIN SELECT * FROM customers WHERE state='FO';
```

The output of the preceding code is similar to this:

```
Index Scan using ix_state on customers (cost=0.29..8.31 rows=1 width=140)
  Index Cond: (state = 'FO'::text)
```

Notice that an index scan is being used with the index you created in *step 4*. You can also see that you have a non-zero setup cost (`0.29`), but the total cost is reduced from the previous `1660` to only `8.31`. This shows the power of the index scan.

1. Now, consider a slightly different example, looking at the time it takes to return a search on the `gender` column. Use the `EXPLAIN` command to return the query plan for a search for all records of males within the database:

```
EXPLAIN SELECT * FROM customers WHERE gender='M';
```

The output is as follows:

```
Seq Scan on customers  (cost=0.00..1660.00 rows=24957 width=140)
  Filter: (gender = 'M'::text)
```

As there is no index on the `gender` column, and the existing index on the `state` column is not relevant, PostgreSQL will still use a sequential scan for this statement.

1. Create an index called `ix_gender` using the `gender` column of `customers`:

```
CREATE INDEX ix_gender ON customers(gender);
```

1. Confirm the presence of the index using `\d`, which lists all the columns and indexes for the particular table:

```
\d customers;
```

Scrolling to the bottom, you can see the indexes as well as the column from the table used to create the index:

```
                          Table "public.customers"
    Column     |            Type           | Collation | Nullable | Default
---------------+---------------------------+-----------+----------+---------
customer_id    | bigint                    |           |          |
(Skip the rows in between ...)
date_added     | timestamp without time zone |           |          |
Indexes:
    "ix_customers_customer_id" btree (customer_id)
    "ix_gender" btree (gender)
    "ix_state" btree (state)
```

1. Rerun the `EXPLAIN` statement from *step 6*:

```
EXPLAIN SELECT * FROM customers WHERE gender='M';
```

The following is the output of the preceding code:

```
Bitmap Heap Scan on customers  (cost=285.71..1632.67 rows=24918 width=140)
  Recheck Cond: (gender = 'M'::text)
  ->  Bitmap Index Scan on ix_gender  (cost=0.00..279.47 rows=24918 width=0)
        Index Cond: (gender = 'M'::text)
```

Notice that the query cost has not changed much, despite the use of the bitmap index scan. This is because the selection based on the `gender` column is not selective enough. There is still a lot of page reading needed, so a bitmap index scan is needed. A bitmap index scan is an approach between a sequential scan and an index scan. It constructs a list of pages that may potentially store the queried rows, then the database will read through these pages to look for the rows, which is called a **bitmap heap scan**. As such, it is generally better than a sequential scan if there is a certain level of selectivity among pages to avoid a full scan, but not as good as an index scan, which can target pages more accurately. Both a bitmap index scan and a bitmap heap scan are created by PostgreSQL dynamically and are not controlled by users.The reason a bitmap scan is used here is that there are only two possible values, `M` and `F`. The gender index essentially marks the pages with two labels: one for containing males and the other for females. And many pages may contain both. The DBMS still needs to read at least half of the pages to get one gender. It still needs to construct a bitmap (a special type of list) of pages to perform selective fetching.

1. Use `EXPLAIN` to return the query plan, searching for latitudes less than 38 degrees and greater than 30 degrees:

```
EXPLAIN SELECT * FROM customers
WHERE (latitude < 38) AND (latitude > 30);
```

The following is the output of the preceding code:

```
Seq Scan on customers  (cost=0.00..1785.00 rows=17944 width=140)
  Filter: ((latitude < '38'::double precision) AND (latitude > '30'::double precision))
```

Notice that the query is using a sequential scan with a filter because there is no index set on the `Filter` condition, so PostgreSQL has to scan the entire table row by row. The initial sequential scan returns `17944` before the filter and costs `1785` with a `0` startup cost.

1. Now, create an index on the filtered column so that PostgreSQL has some prior knowledge of how data is stored based on latitude. Create an index called `ix_latitude` using the `latitude` column of `customers`:

```
CREATE INDEX ix_latitude ON customers(latitude);
```

1. Rerun the query of *step 10* and observe the output of the plan:

```
Bitmap Heap Scan on customers  (cost=384.22..1688.38 rows=17944 width=140)
  Recheck Cond: ((latitude < '38'::double precision) AND (latitude > '30'::double precision))
  ->  Bitmap Index Scan on ix_latitude  (cost=0.00..379.73 rows=17944 width=0)
        Index Cond: ((latitude < '38'::double precision) AND (latitude > '30'::double precision))
```

You can see that this plan is more involved than the previous plan, with a bitmap heap scan and a bitmap index scan being used.

1. The `EXPLAIN` command will not actually execute the query. It only offers an estimate. You can actually run the query and get its execution information by using the `EXPLAIN ANALYZE` command. You will do this in this step to query the content of the `customers` table with `latitude` values between 30 and 38, and get the execution steps and time spent:

```
EXPLAIN ANALYZE
SELECT * FROM customers
WHERE (latitude < 38) AND (latitude > 30);
```

The following output will be displayed:

```
Bitmap Heap Scan on customers  (cost=384.22..1688.38 rows=17944 width=140) (actual time=53.413..5
  Recheck Cond: ((latitude < '38'::double precision) AND (latitude > '30'::double precision))
  Heap Blocks: exact=1033
  ->  Bitmap Index Scan on ix_latitude  (cost=0.00..379.73 rows=17944 width=0) (actual time=53.19
        Index Cond: ((latitude < '38'::double precision) AND (latitude > '30'::double precision))
Planning Time: 0.169 ms
Execution Time: 57.981 ms
```

From the last two rows, you can see that there is `0.169` ms of planning time and `57.981` ms of execution time, with the index scan taking almost the same amount of time to execute as the bitmap heat scan takes to start.

1. Create another index for `latitude` between 30 and 38 on the `customers` table:

```
CREATE INDEX ix_latitude_less
ON customers(latitude)
WHERE (latitude < 38) and (latitude > 30);
```

1. Re-execute the query in *step 10* and compare the query plans:

```
Bitmap Heap Scan on customers  (cost=298.25..1602.41 rows=17944 width=140) (actual time=2.316..7
  Recheck Cond: ((latitude < '38'::double precision) AND (latitude > '30'::double precision))
  Heap Blocks: exact=1033
  ->  Bitmap Index Scan on ix_latitude_less (cost=0.00..293.77 rows=17944 width=0) (actual time=2
Planning Time: 0.293 ms
Execution Time: 7.905 ms
```

When you use a generic column index that includes all the elements in the column, the planning time is 0.169 ms, and the execution time is 57.981 ms. With a more targeted index that only includes a part of the values in the column, the numbers were `0.293 ms` and `7.905 ms`, respectively. Using this more targeted index, you were able to shave 50.076 ms off the execution time at the cost of an additional 0.124 ms of planning time.Thus far, you can improve the performance of your query as indexes have made the searching process more efficient. You may have had to pay an upfront cost to create the index, but once created, repeat queries can be executed more quickly.The B-tree index is the most widely used index in relational database management systems. However, it still requires searching through the tree hierarchy. A hash index will help you identify the location of specific values in one step, without browsing through the hierarchy. You will learn about the hash index in the next section.

The hash index

The other indexing type you will learn about is the hash index. It is relatively limited in the comparative statements it can run, with equality (=) being the only one available.So, given that the feature is somewhat limited in options for use, why would anyone use it? Hash indices can describe large datasets (in the order of tens of thousands of rows or more) using very little data and map values to locations in a much shorter time, allowing more of the data to be kept in memory and reducing search times for some queries. This is particularly important for databases that are at least several gigabytes in size.A hash index is an indexing method that utilizes a hash function to achieve its performance benefits. A hash function is a mathematical function that takes data or a series of data and returns a unique series of alphanumeric characters, depending on what information was provided and the unique hash code used.For instance, say you have a customer named Josephine Marquez. You could pass this information to a hash function, which could produce a hash result such as `01f38e`. Suppose you also had records for Josephine's husband, Julio; the corresponding hash for Julio could be `43eb38a`. A hash map uses a key-value pair relationship to find data. You will use the values of a hash function to provide the key, using the data storage location in the corresponding row of the database as the value. As long as the key is unique to the value, you can access the location you requested in one step. This method can also reduce the overall size of the index in memory if only the corresponding hashes are stored, thereby dramatically reducing the search time for a query.Similar to the syntax for creating a B-tree index, a hash index can be created using the following syntax:

```
CREATE INDEX <index name> ON <table name>
USING HASH(table column)
```

The following example shows how to create a hash index on the `gender` columns in the `customers` table:

```
CREATE INDEX ix_gender ON customers
USING HASH(gender);
```

In the previous section, it was mentioned that the query planner can ignore the indices created if it deems them to not be significantly faster or more appropriate for the existing query. As the hash scan is somewhat limited in use, it may not be uncommon for a different search to ignore the indices. Now, you will perform an exercise to implement the hash index. This will also show you the difference in performance between different index types.

Exercise 10.03: Generating hash indexes to investigate performance

In this exercise, you will generate several hash indexes and investigate the potential performance increases that can be gained from using them. You will start the exercise by rerunning some of the queries from previous exercises and comparing the execution times:

1. Open `psql` and connect to the `sqlda` database.
2. Drop all existing indexes using the `DROP INDEX` command for each of the indexes that you have created previously (`ix_gender` , `ix_state` , `ix_latitude` , and `ix_latitude_less`); otherwise, you will run into an issue in the following steps:

```
DROP INDEX <index name>;
```

1. Use `EXPLAIN ANALYZE` on the `customers` table where the gender is male, but without using a hash index:

```
EXPLAIN ANALYZE
SELECT * FROM customers
WHERE gender='M';
```

An output similar to the following will be displayed:

```
Seq Scan on customers  (cost=0.00..1660.00 rows=24957 width=140) (actual time=0.168..130.498 rows
  Filter: (gender = 'M'::text)
  Rows Removed by Filter: 25044
Planning Time: 0.217 ms
Execution Time: 12.833 ms
```

From the output, you can see that the estimated planning time is 0.217 ms and the execution time is 12.833 ms. Note that you may not have the same time with this query plan, and the plan may not always produce the same

values. The key here is to compare the values with the values when PostgreSQL uses an index for execution, not the absolute values.

1. Create a B-tree index on the `gender` column and repeat the query to determine the performance using the default index:

```
CREATE INDEX ix_gender ON customers
USING btree(gender);
```

The following is the output of the preceding code:

```
Bitmap Heap Scan on customers  (cost=285.71..1632.67 rows=24957 width=140) (actual time=1.002..7
  Recheck Cond: (gender = 'M'::text)
  Heap Blocks: exact=1035
    ->  Bitmap Index Scan on ix_gender (cost=0.00..279.47 rows=24957 width=0) (actual time=0.875
        Index Cond: (gender = 'M'::text)
Planning Time: 0.173 ms
Execution Time: 8.303 ms
```

From the output, you can decipher that the query planner has selected the B-tree index, but the costs of the scans do not differ much, although the planning and execution time estimates have been modified. This is because there are only two values in the column. Thus, the selectivity of this index is not high.

1. Repeat the query at least five times manually and observe the time estimates after each execution. The results of the five individual queries should be similar to the one shown previously, just that the planning and execution times differ for each separate execution of the query.
2. You created a B-tree index called `ix_gender` in *step 3*. Now, drop the index so that you can create a hash index on the `gender` column to compare the hash index with the B-tree index:

```
DROP INDEX ix_gender;
CREATE INDEX ix_gender ON customers
USING HASH(gender);
```

1. Repeat the query from *step 3* to see the execution time:

```
EXPLAIN ANALYZE SELECT * FROM customers
WHERE gender='M';
```

The following output will be displayed:

```
Seq Scan on customers  (cost=0.00..1660.00 rows=24957 width=140) (actual time=0.029..14.028 rows=
  Filter: (gender = 'M'::text)
  Rows Removed by Filter: 25044
Planning Time: 0.981 ms
Execution Time: 14.979 ms
```

PostgreSQL determined that there was no benefit to using the hash index on the `gender` column, so the index was not used by the planner. This is because the `gender` column could have only two possible values, and the selectivity is very low.

1. Use the `EXPLAIN ANALYZE` command to profile the performance of the query that selects all customers where the state is `FO`:

```
EXPLAIN ANALYZE SELECT * FROM customers
WHERE state='FO';
```

The following output will be displayed:

```
Seq Scan on customers  (cost=0.00..1660.00 rows=1 width=140) (actual time=13.321..13.321 rows=0 l
  Filter: (state = 'FO'::text)
  Rows Removed by Filter: 50000
Planning Time: 0.118 ms
Execution Time: 13.338 ms
```

1. Similar to what you just did to the index on the `gender` column, create a hash index for the `state` column and compare the performance. Then, use `EXPLAIN ANALYZE` to profile the performance of the hash scan:

```
CREATE INDEX ix_state ON customers
USING HASH(state);
EXPLAIN ANALYZE SELECT * FROM customers
WHERE state='FO';
```

The following is the output of the preceding code:

```
Index Scan using ix_state on customers (cost=0.00..8.02 rows=1 width=140) (actual time=0.032..0.0
  Index Cond: (state = 'FO'::text)
Planning Time: 0.359 ms
Execution Time: 0.054 ms
```

You can see that, for this specific query, a hash index is particularly effective, reducing both the planning/setup time and cost of the B-tree index, as well as reducing the execution time to less than 1 ms from 13.338 ms.For small tables, you may find out that using a hash index is no different than using a B-tree index. Remember, a hash index is a one-step mapping for row locations, while the same operation in a B-tree requires searching through multiple levels of the tree structure. So, when the table is larger and when the filter is more selective, the hash index will provide better performance.In this exercise, you used hash indexes to find the effectiveness of a particular query. You saw how the execution time goes down when using a hash index in a query. Now, you will learn about the general rules on how to effectively use the index.

Effective index use

So far in this chapter, you have looked at different scanning methods and the use of both B-trees and hash scans as a means of reducing query times. You have also seen different examples of where an index was created for a field or condition and was explicitly not selected by the query planner when executing the query, as it was deemed a more inefficient choice.This is actually a good time to talk about the appropriate use of indexes to reduce query times, since, while indexes generally are obvious choices for increasing query performance, choosing the columns and the right types can be a difficult task. Consider the following situations:

- **The field you have used for your index is frequently changing**: When you insert or delete rows in a table, PostgreSQL will automatically update the indexes. This is a convenient feature. However, updating indexes will consume resources and can degrade the performance of table updates. You need to consider the cost, means, and strategy of frequent re-indexing versus other performance considerations, such as the query benefits introduced by the index, the size of the database, or even whether changes to the database structure could avoid the problem altogether.
- **You are frequently looking for records containing the same search criteria within a specific field**: In *Exercise 10.02*, *Creating an index scan*, you considered an example similar to this when looking for customers within a database whose records contained `latitude` values of less than 38 and greater than 30, using `SELECT * FROM customers WHERE (latitude < 38) and (latitude > 30).`

In this example, it may be more efficient to create a partial index using a subset of data, like this:
`CREATE INDEX ix_latitude_less ON customers(latitude) WHERE (latitude < 38) and (latitude > 30).`
In this way, the index is only created using the data you are interested in, and is thereby smaller in size, quicker to scan, easier to maintain, and can also be used in more complex queries. This is widely used when your data is significantly skewed.

- **The database is not particularly large**: In such a situation, the overhead of creating and using the index may simply not be worth it. Sequential scans for small tables in RAM are quite fast, and if you create an index on a small dataset, there is no guarantee that the query planner will use it or get any significant benefit from using it.

For an overview of the scan types that you have seen so far, please refer to the following table:

Scan Type	Description	Performance	Best Use Cases
Sequential Scan	Reads all rows in the table, one by one	Slower on large tables without filtering; faster on small tables	- Small tables - Queries with no useful index - Returns a large portion of rows
Index Scan	Uses a B-tree or hash index to find rows matching the condition	Fast when few rows match and the index is selective	- Well-indexed columns - Queries returning a small number of rows
Bitmap Index Scan	Uses an index to build a bitmap of matching rows, then fetches data in batches	Efficient for moderately large result sets	- Multiple conditions on different columns - Queries returning many rows

Table 10.1: Table scan typesSo far, all the query plans in this chapter have only dealt with single-table queries. As you can imagine, when the query contains more tables, its query plan will become more complex. This is especially true when you try to join two or more tables because, at this point, you are not only picking data from the hard disk but also trying to match data (with a join key) in one table to the data in another. The interpretation and understanding of these plans are no doubt very important, but are beyond the scope of this book. If you are interested in learning more about this topic, you should get familiar with single-table query plans first, then seek further studies on the official PostgreSQL website.In the following activity, you will try to improve the query performance by creating indexes on your own.

Activity 10

The largest table in any e-commerce systen is usually the `sales` table, and it is almost always the most frequently queried table. As such, you would like to perform some research on the behaviors of the same query under different indexes.You would like to use two typical queries for research:

- Select all available records within the `sales` table that were filtered by `customer_id = 1`
- Select all available records within the `sales` table that were filtered by `customer_id < 100`

To check the query plan to see how these queries are executed without an index, with a B-tree index, and with a hash index, you will perform the following tasks:

1. Create the statements for the two queries mentioned previously.
2. Check the query plan of these two statements without an index.
3. Create a B-tree index on the `sales` table based on the `customer_id` column.
4. Check the query plan of these two statements with a B-tree index.
5. Drop the B-tree index. Create a hash index on the `sales` table based on the `customer_id` column.
6. Check the query plan of these two statements with a hash index.
7. Clean up the database by dropping the hash index.

Summary

This chapter covered the important topic of optimizing SQL queries for performance. It explained the various methods for scanning databases, discussed the PostgreSQL server query planner, and introduced the `EXPLAIN` command and the `EXPLAIN ANALYZE` command, which were used to display the plan for a query. With these topics, you should now have a better understanding of the data retrieval mechanisms for PostgreSQL, which will help you write optimized queries that have better performance.In the next two chapters, you will dive into the processing of specific data types. In *Chapter 11*, you will start with `JSON` and `Array`. Then, in *Chapter 12*, you will move on to `Date`, `Text`, and `Geographical` data.

11 Processing JSON and Arrays

Join our book community on Discord


https://packt.link/EarlyAccessCommunity

You have already seen two different ways of recording data in *Chapter 1*, using the **JavaScript Object Notation** (**JSON**) document model and using the relational model. You have also seen that JSON is a natural way of recording real-world data, but the relational model provides a much simpler approach for most use cases. That's why all the data processing you have learned about so far is SQL statements, which only handle relational data.It must be pointed out, however, that JSON is still a convenient way of describing real-world issues. As such, modern SQL does support complex data structures such as JSON and arrays. These data structures are a natural way of data description and show up constantly in technology applications. Being able to use them in a database makes it easier to perform many kinds of analysis work.The following topics are covered in this chapter:

- Understanding types of data
- Using JSON
- Using arrays

With these topics, you will be able to parse and utilize these two complex data types, which extends your ability to handle data that traditional relational models cannot handle.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in *Chapter 1*, *Introduction to Data Management Systems*.

Understanding types of data

Data can be broken down into three main categories: **structured**, **semi-structured**, and **unstructured**. You have already seen two different ways of recording data, JSON and relational. The former is called semi-structured data and the latter structured data.

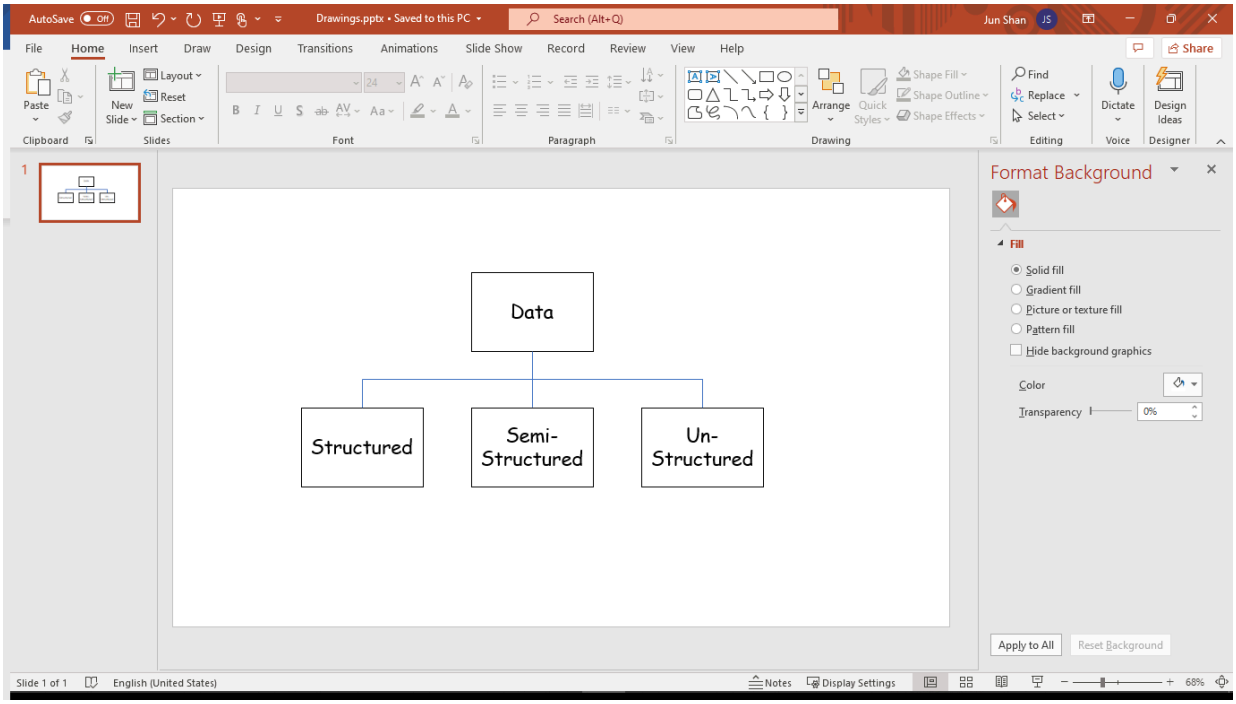

Figure 11.1: The classification of types of data

Structured data has an atomic definition for all the variables, such as data type, value range, and meaning for values. In many cases, even the order of variables is clearly defined and strictly enforced. For example, the record of a student in a school registration card contains an identification number, name, and date of birth, each with a clear meaning and stored in order. Given the wide adoption of relational databases, structured data these days is almost equivalent to relational data.Semi-structured data does not have a predefined format and order, but each of its measurement values is tagged with the definition of that measurement. JSON and array documents are typical examples of semi-structured data.The last type of data is unstructured data, which, on the other hand, does not have any definition as clear as structured data and thus is harder to extract and parse. It may be some binary blob that comes from electronic devices, such as video and audio files. It may also be a collection of natural tokens (words or emojis), such as social network posts and human speech.SQL, by definition, is a tool designed to process structured data. It does not have built-in functionalities to handle unstructured data. As for semi-structured data, there are two issues for SQL to fully handle. First of all, SQL cannot reference a data element if the schema is unknown. For example, for SQL to present a customer's name properly, it must know whether there is a `full_name` field, or two fields of `first_name` and `last_name`. It lacks the control structure of programming languages to detect and process fields dynamically. Second, SQL cannot loop through a data structure if it does not know the levels of embedding or the count of a list. These two issues greatly hinder SQL's ability to handle JSON and arrays. As such, the operations you can perform in SQL are limited to situations where you have a defined schema, that is, you already know what fields you are looking for and the depth and the width of the data. In this chapter, you will see that all the operations are done within this "defined schema" limit. If you need to perform more flexible operations, you will need to use certain types of NoSQL databases.

Using JSON

While the relational model is extremely convenient and powerful, sometimes your data structures can be complex. You might want to store multiple values of different types in a single field, and you might want data to be keyed with labels rather than stored sequentially. These are common issues with transaction-level data, as well as alternative data. For example, a health care patient database may contain a field called `prescription`, which contains all the prescriptions of a patient. Some patients may not have any prescriptions; thus, this field may be empty. Other patients may have multiple prescriptions, and each patient's prescription may be different from the others. One patient may have a hypertension drug of 10 mg per day. Another may have an insomnia medicine of

two pills per night. Yet another patient may have both. It is very hard to store these in a predefined format, but they can usually be stored as key-value pairs using the JSON format. JSON represents data as key-value pairs. Keys are strings, and values can be strings, numbers, Booleans, objects, or null. There is no requirement for the existence of certain keys, and no requirement for the order. As such, JSON is very flexible and can be used to store complex data.You have seen in *Chapter 1*, *Introduction to Data Management Systems*, that you can convert a JSON document into a row in a relational table. You can also convert a record from your table into JSON easily by defining column names as keys and row values as values. In fact, PostgreSQL provides you with a `row_to_json` function, which accepts a table name as the parameter, and transforms all columns in each row to a row containing a JSON document:

```
SELECT row_to_json(c) FROM customers c limit 1;
```

Here is the output of the preceding query, which converts the first row of the `customers` table into one JSON document:

```
 {"customer_id":716,"title":null,"first_name":"Jarred","last_name":"Bester","suffix":null,"email'
(1 row)
```

This is a little hard to read, but you can add the `pretty_bool` flag to generate a readable version. In the following query, the second parameter of the `row_to_ json` function is the `pretty_bool` flag and it is set to `TRUE`:

```
SELECT row_to_json(c, TRUE) FROM customers c limit 1;
```

Here is the output of the preceding query:

```
              row_to_json
-------------------------------------
 {"customer_id":716,                 +
  "title":null,                      +
  "first_name":"Jarred",             +
  "last_name":"Bester",              +
  "suffix":null,                     +
  "email":"jbesterjv@nih.gov",       +
  "gender":"M",                      +
  "ip_address":"216.51.110.28",      +
  "phone":null,                      +
  "street_address":null,             +
  "city":null,                       +
  "state":null,                      +
  "postal_code":null,                +
  "latitude":null,                   +
  "longitude":null,                  +
  "date_added":"2021-09-19T00:00:00"}
```

As you can see, once you reformat the JSON output from the query, `row_to_json` presents a simple, readable, text representation of your row. The JSON structure contains keys and values. In this example, the keys are simply the column names, and the values come from the row values. JSON values can be numeric values (integers or floats), Boolean values (`True` or `False`), text values (wrapped with double quotation marks), or simply `NULL` .While JSON is a universal format for storing data, it is inefficient because everything is stored as one long text string. To retrieve a value associated with a key, you would need to first parse the text, and this has a relatively high computational cost associated with it. If you just have a few JSON objects, this performance overhead might not be a big deal. However, it might become a burden if you are trying to perform a JSON operation on a large dataset, such as selecting the JSON object with `"customer_id": 7` from millions of other JSON objects in your database.In the next section, you will learn about JSONB, a binary JSON format that is optimized for PostgreSQL. This data type allows you to avoid a lot of the parsing overhead associated with a standard JSON text string. You will then proceed to learn how to use PostgreSQL operators and functions to extract and analyze JSON and JSONB values.

JSONB: Pre-parsed JSON

As you have seen previously, JSON is stored and transferred as a text string. For the computer to understand what key it contains and what value corresponds to each key, the computer must break up the string into key-value pairs.

This will increase the time and resources required to handle JSON data. PostgreSQL provides a data type called JSONB, which is a JSON but stored in a pre-parsed format. Upon receiving a JSON string for a JSONB column, PostgreSQL will decompose the string into binary format. This is advantageous as there is a significant performance improvement when querying the keys or values in a JSONB field. The keys and values do not need to be parsed. They have already been extracted and stored in an accessible binary format.

Accessing data from a JSON or JSONB field

You use JSON keys to access the associated value using the `->` operator. As discussed, you must already know the key names (schema) of the JSON document that you are analyzing. Here is an example:

```
SELECT
  '{
    "a": 1,    "b": 2,    "c": 3
  }'::JSON -> 'b' AS data;
```

In this example, you have a three-key JSON value, and you are trying to access the value for the `b` key. The output is a single number `2`. This is because the `-> 'b'` operation gets the value for the `b` key from the preceding `JSON {"a": 1, "b": 2, "c": 3}` string.PostgreSQL also allows more complex operations to access the nested JSON format by using the `#>` operator. Look at the following example:

```
SELECT
  '{
    "a": 1,
    "b": [{"d": 4}, {"d": 6}, {"d": 4}],
    "c": 3
  }'::JSON #> ARRAY['b', '1', 'd'] AS data;
```

On the right side of the `#>` operator, a text array defines the path to access the desired value. Its operation can be broken down into three steps:

1. Select the `b` value, which is a list of nested JSON arrays.
2. Select the element in the array denoted by `1`, which is a nested JSON object, `{"d": 6}`. Note that with the suffix `1`, the second element is returned because array indexes start at 0.
3. Select the value associated with the `d` key, and the output is `6`.

These functions work with JSON or JSONB fields (keep in mind that they will run much faster on JSONB fields). JSONB, however, also enables additional functionality. For example, you want to filter rows based on a key-value pair, such as filtering on the `customer_id` field inside the sales transaction record of the JSON format. You can use the `@>` operator, which checks whether the JSONB object on the left contains the key value on the right. Here is an example:

```
SELECT *
FROM customer_sales
WHERE customer_json @> '{"customer_id":20}'::JSONB;
```

The preceding query outputs the corresponding JSONB record:

```
{"email": "ihughillj@nationalgeographic.com", "phone": null, "sales": [], "last_name": "Hughill",
```

With JSONB, you can also make your output more readable using the `jsonb_pretty` function:

```
SELECT JSONB_PRETTY(customer_json)
FROM customer_sales
WHERE customer_json @> '{"customer_id":20}'::JSONB;
```

Here is the output of the preceding query:

```
                   jsonb_pretty
-------------------------------------------------
 {                                               +
     "email": "ihughillj@nationalgeographic.com",+
```

```
    "phone": null,                                +
    "sales": [                                    +
    ],                                            +
    "last_name": "Hughill",                       +
    "date_added": "2021-08-08T00:00:00",          +
    "first_name": "Itch",                         +
    "customer_id": 20                             +
}
```

One issue with the JSON format is that it is not accepted by all the data processing software on the market. You will need to break JSON into a relational dataset, which means the result must be a two-dimensional table with two columns. One column contains the key and the other contains the value. You can also select just the keys from the JSONB field, and un-nest them into multiple rows using the `jsonb_object_keys` function. Using this function, you can also extract the value associated with each key from the original JSONB field using the `->` operator. Here is an example:

```
SELECT
  JSONB_OBJECT_KEYS(customer_json) AS keys,
  customer_json -> JSONB_OBJECT_KEYS(customer_json)
FROM customer_sales
WHERE customer_json @> '{"customer_id":20}'::JSONB;
```

The following is the output of the preceding query:

```
    keys     |              ?column?
-------------+-----------------------------------
 email       | "ihughillj@nationalgeographic.com"
 phone       | null
 sales       | []
 last_name   | "Hughill"
 date_added  | "2018-08-08T00:00:00"
 first_name  | "Itch"
 customer_id | 20
```

Leveraging JSON path

In addition to the previous functions (such as `jsonb_object_keys`) and operators (such as `->`), PostgreSQL also offers a special JSON path language that can be leveraged to query data within a JSONB field. The first of these functions can check whether a path exists in your JSON object:

```
SELECT jsonb_path_exists(customer_json, '$.sales[0]')
FROM customer_sales
LIMIT 3;
```

The following is the output of the document:

```
jsnob_path_exists
-----------------
t
t
t
(3 rows)
```

The `jsonb_path_exists` function has two required parameters: the JSONB value and the JSON path. The JSON path expression uses the JSON path language. Within this JSON path language, `$` represents the root of the JSON value, and the `.key` notation is used to access the value for a given key. In this case, you can access the `sales` element directly under `root` using `$.sales`. The `[0]` value represents that you want the first value contained in the sales array. Alternatively, you could have specified `[*]` to represent all elements in the sales array. This query simply goes through the JSON value in each row, checks whether the JSON value contains a sales field under its root or not, and returns a Boolean value of `true` or `false` based on the result.In addition to checking whether a JSON path exists (with or without additional filter criteria), you can also query the result as follows:

```
SELECT
  jsonb_path_query(customer_json,
```

```
    '$.sales[0].sales_amount')
FROM customer_sales
LIMIT 3;
```

The following is the output of the document:

```
jsnob_path_query
-----------------
479.992
314.991
319.992
(3 rows)
```

In this case, the `jsonb_path_query` function grabs the first sale using the positional index, `[0]`, and grabs the value associated with the `sales_amount` key. The `jsonb_path_query` function will expand a result with more than one match to multiple rows:

```
SELECT
  jsonb_path_query('{"test":[1, 2, 3]}', '$.test[*]');
```

The following is the output of the code:

```
jsnob_path_query
-----------------
1
2
3
```

Note

If a path does not exist that meets the filter criteria (if any), `jsonb_path_query` will remove that entire row from the output. This is a bit counterintuitive because, normally, row filtering can only happen due to expressions evaluated in the `WHERE` clause, so this functionality can produce unexpected results.

But what if you want to grab the array of sales amounts in cases where there are multiple sales or no sales? In the following examples, you might want to instead use `jsonb_path_query_array`. In the following example, you return the entire array of sales amounts that are greater than $400:

```
SELECT
  jsonb_path_query_array(
    customer_json,
    '$.sales[*].sales_amount ? (@ > 400)'
  )
FROM customer_sales
LIMIT 3;
```

The following is the output of the code:

```
jsnob_path_query_array
-----------------
[479.992]
[]
[]
```

In this case, the first record contains the `$.sales[*].sales_amount` path and has one sale over the threshold, so the `jsonb_path_query_array` function returns the sales value array. The second and third rows have sales in the `$.sales[*].sales_amount` path but none of the values are over the threshold. So, the `jsonb_path_query_array` function returns the `NULL` array for both rows.With the preceding functions and operators, you can test the existence of JSON paths and query them. You can combine these using a `CASE` statement, in which you will test the existence of a specific path. If the path exists, you can query the value or perform some further operations, which will be discussed in the next subsection.

Creating and modifying data in a JSONB field

You can add and remove elements from JSONB. For example, to add a new key-value pair, `"c": 2`, you can do the following:

```
select
  jsonb_insert('{"a":1,"b":"foo"}', ARRAY['c'], '2');
```

Here is the output of the preceding query:

```
{"a": 1, "b": "foo", "c": 2}
```

If you wanted to insert values into a nested JSON object, you could do that too:

```
select
  jsonb_insert(
    '{"a":1,"b":"foo", "c":[1, 2, 3, 4]}',
    ARRAY['c', '1'], '10');
```

This would return the following output:

```
{"a": 1, "b": "foo", "c": [1, 10, 2, 3, 4]}
```

In this example, `ARRAY['c', '1']` represents the path where the new value should be inserted. In this case, it first grabs the `c` key and the corresponding array value, and then inserts the value `(10)` at position `1`.To remove a key, you can simply subtract the key that you want to remove. Here is an example:

```
SELECT '{"a": 1, "b": 2}'::JSONB - 'b';
```

In this case, you have a JSON object with two keys: `a` and `b`. When you subtract `b`, you are left with just the a key and its associated value:

```
{"a": 1}
```

So far in this section, you have learned the definition of JSON, how to use JSON data in PostgreSQL, the benefits of the JSONB data type, and how to explore and process JSONB data using specific functions. In addition to the methodologies described here, you might want to search through multiple layers of nested objects. As stated earlier, SQL can only handle known schemas. All the statements you have written so far are based on a schema or path of which you already know the keys and number of values. You will practice these skills in the following exercise.

Exercise 11.1: Searching through JSONB

In this exercise, you will identify the values using data stored as JSONB. Many source systems today will send the transaction information to downstream systems such as data analytics software in the format of a JSON string. You will need to properly identify values from JSON strings before many of the downstream systems can utilize the content. Suppose you want to identify all customers who purchased a Blade scooter; you can do this using data stored as JSONB. Complete the exercise by implementing the following steps:

1. Open `psql` and connect to the `sqlda` database.
2. In this step, you will explode each sale into its own row using the `JSONB_ARRAY_ELEMENTS` function:

```
CREATE TEMP TABLE customer_sales_single_sale_json
AS (
  SELECT
    customer_json,
    JSONB_ARRAY_ELEMENTS(
        customer_json -> 'sales') AS sale_json
  FROM customer_sales
  LIMIT 10
);
```

Note the `TEMP` keyword in this statement. You can create `TEMP` tables or views in your session. These tables/views will only exist for the duration of your session. Once you log out, they will be automatically dropped. These

tables/views are very useful for temporary storage of data.

1. Filter this output and grab the records where `product_name` is `Blade`:

```
SELECT DISTINCT customer_json
FROM   customer_sales_single_sale_json
WHERE  sale_json ->> 'product_name' = 'Blade';
```

The `->>` operator is similar to the `->` operator, except it returns text output rather than JSONB output. This outputs the following result:

```
{"email": "nespinaye@51.la", "phone": "818-658-6748", "sales": [{"product_id": 5, "product_name"
```

1. Use the `JSONB_PRETTY()` function to format the output and make the result easier to read:

```
SELECT DISTINCT JSONB_PRETTY(customer_json)
FROM customer_sales_single_sale_json
WHERE sale_json ->> 'product_name' = 'Blade';
```

Here is the output of the preceding query:

```
                    jsonb_pretty
-----------------------------------------------------
{                                                    +
   "email": "nespinaye@51.la",                       +
   "phone": "818-658-6748",                          +
   "sales": [                                        +
       {                                             +
           "product_id": 5,                          +
           "product_name": "Blade",                  +
           "sales_amount": 559.992,                  +
           "sales_transaction_date": "2020-07-19T06:33:44"+
       }                                             +
   ],                                                +
   "last_name": "Espinay",                           +
   "date_added": "2020-07-05T00:00:00",              +
   "first_name": "Nichols",                          +
   "customer_id": 15                                 +
}
```

You can now easily read the formatted result after using the `JSONB_PRETTY()` function.

1. Perform this same action with the JSON path expressions:

```
CREATE TEMP TABLE blade_customer_sales AS (
  SELECT
    jsonb_path_query(
      customer_json,
      '$ ? (@.sales[*].product_name == "Blade")'
    ) AS customer_json
  FROM customer_sales
);
SELECT JSONB_PRETTY(customer_json)
FROM blade_customer_sales;
```

1. Finally, count the number of customers who purchased a Blade scooter:

```
SELECT COUNT(1) FROM blade_customer_sales;
```

The following is the output of the code:

```
Count
------
986
(1 row)
```

In this exercise, you identified the values using data stored as JSONB. You used `JSONB_PRETTY()` to complete this exercise. At this point, you have learned many different aspects of JSON processing. In the next section, you will learn about a closely related topic: array processing.

Using arrays to process element collections

While the PostgreSQL data types that you have explored so far allow you to store many different types of data, occasionally you will want to store a series of values in a single cell. For example, you might want to store a list of the products that a customer has purchased, or the employee ID numbers associated with a specific dealership. For this scenario, PostgreSQL offers the `ARRAY` data type, which allows you to store a list of values.PostgreSQL arrays allow you to store multiple values of the same type in a field in a table. For example, consider the following first record in the `customers` table:

```
 customer_id | title | first_name | last_name | suffix |       email       | gender |  ip_address
-------------+-------+------------+-----------+--------+-------------------+--------+-------------
         716 |       | Jarred     | Bester    |        | jbesterjv@nih.gov | M      | 216.51.110
```

Each field contains exactly one value (the `NULL` value is still a value). However, there are some attributes that might contain multiple values with an unspecified length. For instance, say you wanted to have a `purchased_products` field. This could contain zero or more values within the field. Imagine the customer purchased the Lemon and Bat Limited Edition scooters. You can define an array in a variety of ways. One of the ways to get started is simply by creating an array using the following command:

```
SELECT
  ARRAY['Lemon', 'Bat Limited Edition']
    AS purchased_products;
```

The following is the output of the code:

```
purchased_products
------------------------------
 {Lemon,"Bat Limited Edition"}
```

PostgreSQL knows that the Lemon and Bat Limited Edition values are of the `TEXT` data type, so it creates a `TEXT` array to store these values.While you can create an array for any data type, each array is limited to values for one data type only. So, you could not have an integer value followed by a text value or vice versa (this will produce an error).You can also create arrays using the `ARRAY_AGG` aggregate function. This aggregate function will create an array of all the values in the group. This is useful when you want to have a consolidated list of sub-attributes for each value in a parent attribute. For example, the following query aggregates all the vehicles for each product type:

```
SELECT
  product_type,
  ARRAY_AGG(DISTINCT model) AS models
FROM products
GROUP BY 1;
```

The following is the output of the preceding query, in which all the models of automobile form an array that corresponds to the automobile product type, and all the models of scooter form an array that corresponds to the scooter product type:

```
product_type |                                      models
-------------+-------------------------------------------------------------------------------
 automobile   | {"Model Chi","Model Epsilon","Model Gamma","Model Sigma"}
 scooter      | {Bat,"Bat Limited Edition",Blade,Lemon,"Lemon Limited Edition","Lemon Zester"}
```

You can also specify how to order the elements by including an `ORDER BY` statement in the `ARRAY_AGG` function, as in the following:

```
SELECT
  product_type,
  ARRAY_AGG(model ORDER BY year) AS models
```

```
FROM products
GROUP BY 1;
```

This is the output:

```
 product_type |                                        models
--------------+-------------------------------------------------------------------------------
 automobile   | {"Model Chi","Model Sigma","Model Gamma","Model Epsilon","Model Chi"}
 scooter      | {Lemon,"Lemon Limited Edition",Lemon,Blade,Bat,"Bat Limited Edition","Lemon Zeste
(2 rows)
```

But there might be situations where you would want to reverse this operation. This can be done by using the `UNNEST` function, which creates one row for every value in the array:

```
SELECT UNNEST(ARRAY[123, 456, 789]) AS example_ids;
```

Here is the output of the preceding query:

```
example_ids
-------------
123
456
789
(3 rows)
```

You can also create an array by splitting a string value using the `STRING_TO_ARRAY` function. A common scenario is that when you use external transaction systems, many systems these days will generate text outputs containing all the information in one string. You will need to break the string into multiple parts and parse each part accordingly. Here is an example:

```
SELECT STRING_TO_ARRAY('hello world', ' ');
```

In this example, the sentence is split using the second string (`' '`), and you end up with the following result:

```
string_to_array
--------------------------
{hello,world}
(1 row)
```

Similarly, you can run the reverse operation and concatenate an array of strings into a single string:

```
SELECT
  ARRAY_TO_STRING(
    ARRAY['Lemon', 'Bat Limited Edition'], ', '
  ) AS example_purchased_products;
```

In this example, you can join the individual string with the second string using `', '`:

```
example_purchased_products
---------------------------
Lemon, Bat Limited Edition
```

There are other functions that allow you to interact with arrays. Here are a few examples of the additional array functionalities that PostgreSQL provides:

Desired Operation	**Postgres Function Example**	**Output**
Concatenate two arrays	`ARRAY_CAT(ARRAY[1, 2],ARRAY[3.,414])` or `ARRAY[1, 2] | | ARRAY[3, 4]`	`{1, 2, 3, 4}`
Append a value to an array	`ARRAY_APPEND(ARRAY[1, 2], 3)` or `ARRAY[1, 2] | | 3`	`{1, 2, 3}`
Check whether a value is contained in an array	`3 = ANY(ARRAY[1, 21])`	`f`

Check whether two arrays overlap	`ARRAY[1, 2, 3] && ARRAY[3, 4]`	`t`
Check whether an array contains another array	`ARRAY[1, 2, 3] @> ARRAY[2, 1]`	`t`

Figure 11.2: Examples of additional array functionalityIn *Exercise 11.2*, *Analyzing sequences using arrays*, you will apply these operators and array functionality to capture sequences of marketing touchpoints.

Exercise 11.2: Analyzing sequences using arrays

In this exercise, you will use arrays to analyze sequences. ZoomZoom sends emails to customers in series. For example, before the December holiday season, they will send out an email providing a product catalog of all the things they sell. During the season, they will send out updates on what product is selling well and what discounts are provided. After the season, they will send out thank-you emails and offer further products and discounts. The marketing team wants you to identify the three most common email sequences. You will help them to better understand how different these sequences are, by looking at whether these sequences are supersets of one another:

1. Open `psql` and connect to the `sqlda` database.
2. First, create a table that represents the email sequence for every customer:

```
CREATE TEMP TABLE customer_email_sequences AS (
  SELECT
    customer_id,
    ARRAY_AGG(
      email_subject ORDER BY sent_date
    ) AS email_sequence
  FROM emails
  GROUP BY 1);
```

1. Next, identify the three most common email sequences. Given that you have already gotten the email sequences, you can do this by using `ORDER BY` with `LIMIT 3`. As the `ORDER BY` clause is based on the occurrence of email sequences, the `SELECT` statement will yield the sequences with the most frequent ones first. Then the `LIMIT 3` clause will make the `SELECT` statement yield the top three sequences:

```
CREATE TEMP TABLE top_email_sequences AS (
  SELECT
    email_sequence,
    COUNT(1) AS occurrences
  FROM customer_email_sequences
  GROUP BY 1
  ORDER BY 2 DESC
  LIMIT 3);
SELECT email_sequence
FROM top_email_sequences;
```

The code will generate three rows. They are too long to display entirely, so only the first one is shown here:

```
{"The 2013 Lemon Scooter is Here","Shocking Holiday Savings On Electric Scooters","A Brand New Sc
```

1. Lastly, you would want to check which of these arrays is a superset of the other arrays. It is possible that some customers joined later than others, so they only received part of the email sequence. You need to identify these sub-sequences as a part of the complete email sequence. The only issue is that the email sequence fields are very long and are not intuitive to read through with human eyes. To help with this, it is helpful to give your rows a numeric ID for identification:

```
ALTER TABLE top_email_sequences
ADD COLUMN id SERIAL PRIMARY KEY;
```

1. Next, you can cross-join the table to itself, and use the `@>` operator to check whether an array containing an email sequence contains another email sequence:

```
SELECT
  super_email_seq.id AS superset_id,
  sub_email_seq.id AS subset_id
FROM top_email_sequences AS super_email_seq
CROSS JOIN top_email_sequences AS sub_email_seq
WHERE
  super_email_seq.email_sequence @>
  sub_email_seq.email_sequence
AND super_email_seq.id != sub_email_seq.id;
```

The following is the output of the code:

```
superset_id | subset_id
------------+-----------
           1 |         2
           1 |         3
           3 |         2
```

From this, you can gather that the first email sequence contains the second and third most common email sequences, while the third most common email sequence is a superset of the second most common sequence. This type of analysis is generally helpful when looking at what customer touchpoints might lead someone to make a purchase or not. For example, some customers joining late may not have received the first email, the holiday season product catalog. But if a similar percentage of these customers and the customers who have received the first email made a purchase after receiving the holiday season discount email, you may reasonably suspect that the holiday season discount email is the main reason for purchase, not the product catalog.With this discussion on JSON and arrays, you are now equipped to help your team with complex data type analysis. You will do this in the following activity.

Activity 11

A salesperson asks you whether you can use your new search skill to find a customer by the name of Arlena who has made some purchases before.Use the `customer_sales` table and create a searchable view with one record per customer. This view should use the `customer_id` column as the primary key and be searchable on everything related to that customer: name, email address, phone number, and purchased products. It is acceptable to include other fields as well. For your convenience, you can also include the original JSON column as is.Query your new searchable view using the `Arlena` keyword filter. Note that the results of some JSON functions are still `JSON`, not `TEXT`. You may need to perform type conversion based on the function you choose. Also, there must be a filter condition that this customer made some purchases before (the sales array contains at least one record, which is record 0).

Summary

This chapter provided a comprehensive guide on processing JSON and arrays in SQL, covering the types of data, using JSON and JSONB, manipulating arrays, and practical exercises to apply these concepts in real-world scenarios. With these topics, you will be able to parse and utilize these two complex data types, which extends your ability to handle data that traditional relational models cannot handle.In the next chapter, you will learn how to handle data types with special calculation rules, such as date/time, geographics, and text.

12 Advanced Data Types: Date, Text, and Geospatial

Join our book community on Discord


https://packt.link/EarlyAccessCommunity

So far, every operation you have done has been based on normal numeric or text calculations. However, certain data types follow their own computing rules. For example, adding 1 to 31 gives you 32, but adding 1 day to January 31st will yield February 1st, not January 32nd. The computing rules of the date/time are different from the rules of normal numbers, so are its storage and display.Equally important is the text processing power of SQL. Human activities generate textual data every moment. As such, businesses must be able to cleanse and analyze textual data for insights, such as extracting key information, performing transformations, or summarizing large bodies of text. All these require powerful text processing capabilities.The last important data type is geospatial data. In the past two decades, there has been a significant increase in the usage of geospatial data. With the wide usage of mobile devices, geospatial data has become a critical part of any consumer-facing application. The geospatial-related calculation, such as distance, is now a must-have functionality for any data

platform.To help you master the processing of these data types, the following topics are covered in this chapter:

- Using date and time in data analytics
- Understanding text processing
- Applying geospatial data

With these topics, you will be able to process data types with their own processing rules, thus extending relational databases' ability to handle complex real-world questions.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in *Chapter 1, Introduction to Data Management Systems*.

Using date and time in data analytics

You are familiar with dates and times, but do you know how these quantitative measures are represented? They are represented using numbers, but not with a single number. Instead, they are measured with a set of numbers, with one number each for the year, month, day, hour, minute, second, and millisecond.This is a complex representation, comprising several different components. For example, knowing the current minute without knowing the current hour does not serve any purpose. Additionally, there are complex ways of interacting with dates and times. For example, the current time can be represented differently depending on where you are in the world.As a result of these intricacies, you need to take special care when working with this type of data. In fact, PostgreSQL, like most databases, offers special data types that can represent these types of values. You will start by examining the `DATE` type.

The DATE type

Dates can be easily represented using strings or numbers (for example, both January 1, 2025 and 01/01/2025 clearly represent a specific date), but dates have different properties that you might want to use in your analysis, for instance, the year or the day of the week that the date represents. Working with dates is also necessary for time series analysis, which is one of the most common types of analysis. The SQL standard includes the `DATE` data type, and PostgreSQL offers great functionality for interacting with this data type.The most common concern about the `DATE` data type is the display format. Different regions use different formats to represent the same date. For example, the same date, January 14th, 2025, is written as 01/14/2025 in some countries but 14/01/2025 in others. You can set your database to display dates in the format that you are most familiar with. PostgreSQL uses the `DateStyle` parameter to configure these settings. To view your current settings, you can use the following command:

```
SHOW DateStyle;
```

The following is the output of the preceding query in a system where both the PostgreSQL server and the `psql` client are installed in the same Windows server whose system locale is set to the United States:

```
DateStyle
-----------
ISO, MDY
```

The first output specifies the default output format to be **International Organization for Standardization** (**ISO**) format, which displays the date as *Year-Month-Day*, and the second parameter specifies the ordering for input or output. In addition, since both the server and client are using a United States locale, *Month/Day/Year* can also be used as the input format. Start by testing out the local input format:

```
SELECT '1/14/2025'::DATE;
```

The following is the output of the query:

```
date
------------
 2025-01-14
```

As you can see, when you input a string, `1/14/2025`, using the *Month/Day/Year* format, PostgreSQL understands that this is January 14th, 2025. It displays the date using the ISO format specified previously, in the form of *YYYY-MM-DD*. PostgreSQL configuration also accepts the ISO input because this is what is specified by the `DateStyle` parameter:

```
SELECT '2025-01-14'::DATE;
```

This will give you the same result as the previous query. However, let's say you use the following code:

```
SELECT '14/01/2025'::DATE;
```

You will get an error message telling you the month cannot be `14`:

```
ERROR:  date/time field value out of range: "14/01/2025"
LINE 1: SELECT '14/01/2025'::DATE;
               ^
HINT:  Perhaps you need a different "datestyle" setting.
```

Note that *14/01/2025* is indeed a valid date format in some countries. If your PostgreSQL server or client is installed in an operating system with a different system locale than the one mentioned previously, the result of the previous command may be different. For example, if you wanted to set it to the European format of *Day, Month, Year*, you would set `DateStyle` to `GERMAN, DMY`. You can configure the output for your database using the following command:

```
SET DateStyle='GERMAN, DMY';
```

Then, running the previous query will bring you the following success:

```
    date
------------
 14.01.2025
```

You can then change back to ISO by running the following query:

```
SET DateStyle=ISO, MDY';
```

For this chapter, you will use the ISO display format (*Year-Month-Day*) and the *Month/Day/Year* input format. You can configure this format by using the preceding command.The natural extension of the `DATE` data type is the data type representing the time, such as 10 a.m. or 2 p.m. in a day. The interesting fact is that when people talk about time, they usually refer to the combination of a day and a time. Simply referring to a time is not enough to determine the exact moment that something happens. For example, “the class starts at 6 p.m.” very likely implies that the class starts at 6 p.m. on Mondays for this semester. To avoid any confusion, the SQL standard offers the `TIMESTAMP` data type, which represents a date and a time, down to a microsecond, for example, `2025-06-05 13:47:44.472096`.You can see the current timestamp using the `NOW()` function or the `CURRENT_TIMESTAMP` function, and you can specify your time zone using `AT TIME ZONE '<time zone>'`. Here is an example of these two functions with the Eastern and Pacific Standard time zones specified:

```
SELECT
  NOW() AT TIME ZONE 'EST' as Eastern,
  CURRENT_TIMESTAMP AT TIME ZONE 'PST' as Pacific;
```

The following is the output of the query:

```
          eastern           |          pacific
----------------------------+----------------------------
 2025-05-06 21:25:32.177817 | 2025-05-06 18:25:32.177817
```

You can also use the `TIMESTAMP` data type without the time zone specified. You can get the current time zone with the `NOW()` function:

```
SELECT NOW();
```

The following is the output of the query. The `-04` part at the end of the string indicates the output time zone:

```
now
-------------------------------
 2025-05-06 13:47:44.472096-04
```

In addition, you can display the current date using the `current_date` function.

Note

In general, it is recommended that you use the timestamp with the time zone specified. If you do not specify the time zone, the value of the timestamp could be questionable (for example, the time could be represented in the time zone where the company is located, in **Coordinated Universal Time** (**UTC**) time, or the customer's time zone).

The `DATE` and `TIMESTAMP` data types are helpful not only because they display dates in a readable format, but also because they store these values using fewer bytes than the equivalent string representation (a `DATE` type value requires only 4 bytes, while the equivalent text representation might be 8 bytes for an 8-character representation such as `20250101`). Additionally, PostgreSQL provides special functionalities to manipulate and transform dates. This is particularly useful for data analytics and will be introduced in the next subsection.

Transforming DATE data types

Often, you will want to decompose your dates into their component parts. For example, while daily sales are important, you may also be interested in the summary for each year and month, so that you can review the monthly trend of your sales. You may see from this trend which month is your best-selling one, so that you can prepare your inventory in advance. To do this, you can use `EXTRACT(component FROM date)`. Here is an example:

```
SELECT current_date,
  EXTRACT(year FROM current_date) AS year,
  EXTRACT(month FROM current_date) AS month,
  EXTRACT(day FROM current_date) AS day;
```

The following is the output of the code:

```
current_date | year | month | day
-------------+------+-------+-----
2025-05-07   | 2025 |     5 |   7
```

Similarly, you can abbreviate these components as `y`, `mon`, and `d`, and PostgreSQL will understand what you want:

```
SELECT current_date,
  EXTRACT(y FROM current_date) AS year,
  EXTRACT(mon FROM current_date) AS month,
  EXTRACT(d FROM current_date) AS day;
```

The output of the code is the same as the previous query.In addition to `year`, `month`, and `day`, you will sometimes want additional components, such as day of the week, week of the year, or quarter. These are available by using specific words, such as `dow`, `week`, `quarter`, and so on. You can refer to PostgreSQL's documentation for details.

Note

`EXTRACT` always outputs a number. So `dow` (day of week) starts at `0` (Sunday) and goes up to `6` (Saturday). Instead of `dow`, you can use `isodow`, which starts at `1` (Monday) and goes up to `7` (Sunday).

In addition to extracting date parts from a date, you may want to simply truncate your date or timestamp. For example, say you want to view the year and month summary for sales, but you only have the date on the sales table. To aggregate the sales for year/month, you need to remove the day and timestamp from the date and get the `year+month` output. This can be done using many functions, such as `DATE_TRUNC()`, `DATE_PART()`, or `EXTRACT()`, each with a slightly different syntax and purpose. In the following example, you will use the `TO_CHAR()` function, which extracts the designated parts of a date and organizes them into one string:

```
SELECT NOW(), TO_CHAR(NOW(), 'yyyymm') AS yearmonth;
```

The following is the output of the code:

```
              now              |      yearmonth
-------------------------------+-----------------------
2025-05-06 19:40:08.691618+00 | 202505
```

The date part extraction functions, such as `TO_CHAR()` and `EXTRACT()`, are particularly useful for `GROUP BY` statements. For example, you can use it to group sales by year and month and get the total year's monthly sales:

```
SELECT
  TO_CHAR(sales_transaction_date, 'yyyymm') yearmonth,
  SUM(sales_amount) AS total_quarterly_sales
FROM sales
GROUP BY 1
ORDER BY 1 DESC;
```

The first three rows of the result are as follows:

```
 yearmonth | total_quarterly_sales
-----------+-----------------------
 202501    |     5706307.722000084
 202412    |     5455222.157000108
 202411    |     4280693.541000041
```

Note that the `TO_CHAR()` function generates a string result, so the `yearmonth` column is now a string, not a date column.In addition to representing dates, you can also represent time intervals between different dates and times. This data type is useful if you want to analyze how long something takes. You will learn about it in the next subsection.

Intervals

A common use case for date/time data is to calculate the duration between two data points. For example, when customers receive a promotional email, they may not open it immediately. The interval between the date the email is received and the date the email is opened can indicate how attractive the email is to the customers. If you want to know how long it takes a customer to open an email after receiving it, you need to calculate the interval between these two dates.You can start with a simpler example, such as measuring the length of February. You can calculate the interval between the first day of February and the first day of March:

```
SELECT
  TIMESTAMP '2025-03-01 00:00:00' –
    TIMESTAMP '2025-02-01 00:00:00' AS days_in_feb;
```

The following is the output of the code:

```
days_in_feb
-------------
 28 days
```

Alternatively, intervals can be used to add the number of days to a timestamp to get a new timestamp, such as what the date is seven days from now:

```
SELECT
  TIMESTAMP '2025-03-06 00:00:00' + INTERVAL '7 days'
    AS new_date;
```

The following is the output of the code:

```
      new_date
---------------------
 2025-03-13 00:00:00
```

While intervals offer a precise method for doing timestamp arithmetic, the `DATE` format can be used with integers to accomplish a similar result. In the following example, you simply add `7` (an integer) to the date to calculate the new date:

```
SELECT DATE '2025-06-05' + 7 AS new_date;
```

The following is the output of the code:

```
new_date
------------
2025-06-12
```

Similarly, you can subtract two dates and get an integer result:

```
SELECT
  '2025-03-01'::DATE - '2025-02-01'::DATE
    AS days_in_feb;
```

The following is the output of the code:

```
days_in_feb
-------------
```

While the `DATE` data type offers ease of use, the timestamp with the `TIME ZONE` data type offers precision. If you need your date/time field to be precisely the same as the time at which the action occurred, you should use the timestamp with the time zone. If not, you can use the `DATE` field.In the following exercise, you will practice your newly learned skills of date/time processing.

Exercise 12.1: Analytics with time-series data

ZoomZoom ramped up its efforts to recruit more customers during the year 2024, hoping that it could sell more vehicles as the number of new customers grows. In this exercise, you will perform a basic analysis using time series data to gain insight into whether the sales were affected by the number of new customers. While it makes sense to have a day-by-day comparison, daily sales/recruitments can fluctuate significantly. It is generally recommended to start from a larger time span, such as monthly sales/recruitment, and break down the numbers once you find any patterns.Perform the following steps to complete the exercise:

1. Open `psql` and connect to the `sqlda` database.
2. First, look at the number of monthly sales. You can use the following aggregate query with the `TO_CHAR` method:

```
SELECT
  TO_CHAR(sales_transaction_date, 'yyyymm')
    AS month_date,
  COUNT(1) AS number_of_sales
FROM sales
WHERE
  EXTRACT(year FROM sales_transaction_date)=2024
GROUP BY 1
ORDER BY 1;
```

Here, the `COUNT(1)` expression adds one for each row, which is similar to the `COUNT(*)` function.After running this SQL, you will get the following result (first three rows):

```
month_date | number_of_sales
------------+-----------------
 202401     |              989
 202402     |              929
 202403     |             1111
```

1. Run another query to get the number of new customers joining each month:

```
SELECT
  TO_CHAR(date_added, 'yyyymm') AS month_date,
  COUNT(1) AS number_of_new_customers
FROM customers
WHERE EXTRACT(year FROM date_added) = 2024
GROUP BY 1
ORDER BY 1;
```

The following are the first three rows of the preceding query's output:

```
  month_date | number_of_new_customers
------------+-------------------------
 202401     |                     466
 202402     |                     439
 202403     |                     455
```

You can probably see that the flow of new potential customers is fairly steady and hovers around 400–500 new customer sign-ups every month, while the number of sales (as queried in step 2) varies considerably. So, it looks like the effort to sign up new customers may not be directly related to the sales amount.In this exercise, you used a PostgreSQL function to extract the `year` and `month` parts from a date and used the extracted information to aggregate the sales data and customer recruitment data to form a time series for comparison. In the next section, you will learn about another data type that has its own domain of operations, the text data.

Understanding text processing

Text data, or string, contains valuable insights. For instance, you can imagine a salesperson keeping notes on prospective clients: *Very promising interaction, the customer is looking to make a purchase tomorrow* is valuable data, as is *The customer is uninterested. They no longer have a*

need for the product. While this text can be valuable for someone to manually read, it can also be valuable in the analysis. Keywords in these statements, such as *promising*, *purchase*, *tomorrow*, *uninterested*, and *no*, can be extracted using the right techniques to try to identify top prospects in an automated fashion.In this section, you will look at how you can use some PostgreSQL functionality to identify patterns that will help you identify trends. You will also leverage text-related functions in PostgreSQL for better searching.

String characteristics and manipulation

PostgreSQL provides the following functions for understanding and transforming the string. You have learned about and used the `||` operator for string concatenation. Here is a more complete list of commonly used functions:

Function name	**Function description**
`length(string)`	Number of characters in the string.
`lower/upper`	Convert the string into all lower/upper case.
`substring`	Extract the substring of a string starting at a certain position.
`substr`	Extract the substring (same as `substring`).
`left/right`	Return the first/last `n` characters in the string. When `n` is negative, return all but the last/first `|n|` characters.
`position`	Return the position of a substring within the string.
`strpos`	Location of specified substring (same as `position`).
`split_part`	Split the string on the delimiter and return the given field (counting from one).
`trim`	Remove the blank characters (space by default) from the start/end/both ends of the string.
`ltrim/rtrim`	Remove the blank characters (space by default) from the start/end of the string.
`reverse(str)`	Return the reversed string.
`replace`	Replace all occurrences in string of `substring from` with `substring to` .

`repeat`	Repeat the string the specified number of times.

Table 12.1: String manipulation functionsYou will use some of these text functions for analysis in the coming exercise. But before that, you should have a fundamental understanding of string pattern matching, which is discussed in the next section.

Identifying string patterns

A very common usage pattern of strings is to identify whether a string contains a substring (or more complex match patterns). For example, does the product name contain the word `scooter` in it? When translated to a pattern-matching algorithm, what you are looking for is a string pattern that starts with zero or more arbitrary characters, followed by the substring `scooter`, then followed by another set of zero or more arbitrary characters. The `LIKE` operator can help you with this pattern matching.The `LIKE` operator evaluates whether a string contains a special pattern or not. It allows you to filter data based on partial matches using wildcard characters such as `%` (matches any number of characters) and `_` (matches a single character) instead of exact matches. For example, `LIKE '%scooter%'` will look for exactly what was described previously, of a string pattern that starts with zero or more arbitrary characters, followed by the substring `scooter`, then followed by another set of zero or more arbitrary characters. Similarly, `LIKE 'Scooter%'` will look for any string that starts with the substring `Scooter`, then followed by another set of zero or more arbitrary characters. Note that the `%` wildcard matches any number of characters, including `0`. So, the `LIKE '%scooter%'` pattern will also match any string that starts or ends with the substring `scooter`, even the word `scooter` itself.On the other hand, the `_` wildcard matches exactly a single character. So the `LIKE 'Scooter_'` pattern will only match `Scooter1`, `ScooterX`, or `Scooter` (with a space at the end), but will not match `Scooter` or `Scooter12`. Also, all these matches are case-sensitive. So, the `LIKE '%scooter%'` pattern will not match `ScooterX` because the letter `S` has a different case between the pattern and the string. PostgreSQL does provide a case-insensitive operator, `ILIKE`, which acts the same as `LIKE`, except that it is case-insensitive. So, the `ILIKE %scooter%` pattern will

match `ScooterX`.You will try some text processing analysis in the following exercise.

Exercise 12.2: Text processing

The marketing team of Zoomzoom is looking for ways to brand their products better. They plan to promote the sales by utilizing names that can attract consumers' attention, such as LIMITED or SUPER. In this exercise, you will help them by providing different ways to adjust the names of products:

1. Open `psql` and connect to the `sqlda` database.
2. First, looking at the `products` table, you noticed some products have the branding `Limited Edition` in their names. You would like to identify all products with `Limited Edition` in the model name. You also want to see the release year, but concatenate the phrase `Release Year:` in front. Since the `Year` column is a number, you will need to convert it to text first before it can be concatenated. These can be achieved using the following query:

```
SELECT
  model,
  'Release Year: ' || year::char(4) ReleaseYear
FROM products
WHERE model LIKE '%Limited Edition%';
```

Here, you apply a `LIKE` function in the `WHERE` clause. The `%` wildcard means it can be replaced by any text. Adding `%` before and after a `Limited Edition` text constant means whenever the constant shows up in a string, the filter condition is met. Similarly, if you want to identify strings starting with `Lemon`, you can use `Lemon%`. Any string starting with `Lemon`, followed by any text, will meet the filter condition:

```
          model          |    releaseyear
-------------------------+--------------------
 Lemon Limited Edition | Release Year: 2017
 Bat Limited Edition   | Release Year: 2023
```

1. Now, you want to launch a campaign on scooter products. You would like to change the word `Model` to `Super Model` and get the length of

the new model name so that it won't exceed the marketing poster's space limit of 20 characters. You will do this by running the following query:

```
SELECT
  model,
  REPLACE(model, 'Model', 'Super Model'),
  LENGTH(
    REPLACE(model, 'Model', 'Super Model'))
FROM products
WHERE model LIKE '%Model%';
```

The result looks like this:

```
     model     |       replace        | length
---------------+----------------------+--------
 Model Chi     | Super Model Chi      |     15
 Model Sigma   | Super Model Sigma    |     17
 Model Epsilon | Super Model Epsilon  |     19
 Model Gamma   | Super Model Gamma    |     17
 Model Chi     | Super Model Chi      |     15
```

1. Finally, you will learn that there are many different ways of doing the same thing. PostgreSQL provides you with a lot of tools, and you can choose any of them based on your personal preference. As an example, say you want to find the models whose names start with `Model`. You can use any of the following queries:

```
SELECT model FROM products
  WHERE model LIKE 'Model%';
SELECT model FROM products
  WHERE LEFT(model, 5) = 'Model';
SELECT model FROM products
  WHERE SUBSTRING(model, 1, 5) = 'Model';
SELECT model FROM products
  WHERE SUBSTR(model, 1, 5) = 'Model';
SELECT model FROM products
  WHERE POSITION('Model' IN model) = 1;
SELECT model FROM products
  WHERE STRPOS(model, 'Model') = 1;
```

The first query uses the `LIKE` operator to match the pattern. The following three queries use different functions, `LEFT`, `SUBSTRING`, or `SUBSTR`, to get the first five characters of the column and compare them with the `Model`

pattern. The last two use the substring position functions to find the exact match. All these queries return the same result. This should help you understand the flexibility provided by PostgreSQL.In this section, you learned about the functions and operators to process text data. In the next section, you will work on another data type, geospatial.

Applying geospatial data

Consumer behavior generally will vary among different geographical areas. People living in large cities certainly behave differently from people living in the countryside. So, in addition to looking at time-series data and text patterns to better understand trends, Postgres also allows you to use geospatial information (such as city, country, or latitude and longitude) to better understand your customers. For example, governments use geospatial analysis to better understand regional economic differences, while a ride-sharing platform might use geospatial data to find the closest driver for a customer.You can represent a geospatial location using latitude and longitude coordinates, and this will be the fundamental building block for you to begin geospatial analysis.

Latitude and longitude

Locations are often thought about in terms of the address— the city, state, country, or postal code that they are assigned to. For example, you can look at the sales transaction in the ZoomZoom sales table by city and come up with meaningful results about which cities are performing well. But often, you also need to understand geospatial relationships numerically to understand the distances between two points or relationships that vary based on where you are on a map. After all, if you live on the border between two cities, it is unlikely that your spending behavior will suddenly change if you walk into a store in the other city.Latitude and longitude allow you to look at the location in a continuous context. This allows you to analyze the numeric relationships between the location and other factors (for example, sales). Latitude and longitude also enable you to look at the distances between two locations. Latitude tells you how far north or south a point is. A point at +90° latitude is at the North Pole, while a point at 0°

latitude is at the equator, and a point at -90° is at the South Pole. On a map, lines of constant latitude run east and west. Longitude tells you how far east or west a point is. On a map, lines of constant longitude run north and south. Greenwich, England, is the point of 0° longitude. Points can be defined using longitude as west (-) or east (+) of this point, and values range from -180° west to +180° east. These values are equivalent because they both point to the vertical line that runs through the Pacific Ocean, which is halfway around the world from Greenwich, England.In PostgreSQL, you can represent latitude and longitude using two floating-point numbers. In fact, this is how latitude and longitude are represented in the ZoomZoom `customers` table:

```
SELECT latitude, longitude
FROM customers
WHERE latitude IS NOT NULL
LIMIT 3;
```

Here is the output of the preceding query:

```
latitude | longitude
----------+-----------
  31.6948 |    -106.3
  30.6589 |   -88.178
  40.0086 |  -76.5972
```

Here, you can see that all the latitudes are positive because the United States is north of the equator. All the longitudes are negative because the United States is west of Greenwich, England. You can also notice that some customers do not have `latitude` and `longitude` values filled in, because their location is unknown.While these values can give you the exact location of a customer, you cannot do much with that information, because distance calculations require trigonometry and make simplifying assumptions that the Earth is perfectly round. Thankfully, PostgreSQL has the tools to solve this problem. You can calculate distances in PostgreSQL using two packages, `earthdistance` and `cube` . You can install these two packages by running the following two commands in `pgAdmin` :

```
CREATE EXTENSION cube;
CREATE EXTENSION earthdistance;
```

These two extensions only need to be installed once on each server by running the two preceding commands. The `earthdistance` module depends on the `cube` module, so you must install the `cube` module first. Once you install the `earthdistance` module, you can define a `POINT` data type:

```
SELECT POINT(longitude, latitude)
FROM customers
WHERE longitude IS NOT NULL
LIMIT 3;
```

Here is the output of the preceding query:

```
       point
--------------------
 (-106.3,31.6948)
 (-88.178,30.6589)
 (-76.5972,40.0086)
```

Note

A `POINT` data type is defined as a combination of two numbers enclosed in parentheses, with the first number being the longitude and the second being the latitude, such as `(-90, 38)`. This is contrary to the convention of latitude followed by longitude. The rationale behind this is that longitude more closely represents points along an *x* axis, latitude more closely represents points on the *y* axis, and in mathematics, graphical points are usually noted by their *x* coordinate followed by their *y* coordinate. It is easier to follow this *(x, y)* pattern when drawing geographical points on a map, which is being used more and more frequently in recent years.

The `earthdistance` module also allows you to calculate the distance between points in miles:

```
SELECT point(-90, 38) <@> point(-91, 37) AS dist;
```

Here is the output of the preceding query:

```
 dist
-------------------
 88.1949338379752
```

In this example, you defined two points, (38° N, 90° W) and (37° N, 91° W), and were able to calculate the distance between these points using the `<@>` operator. This operator calculates the distance in miles (in this case, these two points are 88.2 miles apart).In the next exercise, you will see how you can use these distance calculations in a practical business context.

Exercise 12.3: Geospatial analysis

In this exercise, you will identify the closest dealership for each customer. ZoomZoom marketers are trying to increase customer engagement by helping customers find their nearest dealership. The product team is also interested in knowing what the typical distance is between each customer and their closest dealership.Follow these steps to implement the exercise:

1. Open `psql` and connect to the `sqlda` database.
2. Create a table with the longitude and latitude points for every customer:

```
CREATE TEMP TABLE customer_points AS (
  SELECT
    customer_id,
    point(longitude, latitude) AS lng_lat_point
  FROM customers
  WHERE longitude IS NOT NULL
  AND latitude IS NOT NULL
);
```

1. Create a similar table for every dealership:

```
CREATE TEMP TABLE dealership_points AS (
  SELECT
    dealership_id,
    point(longitude, latitude) AS lng_lat_point
  FROM dealerships
);
```

1. Cross-join these tables to calculate the distance from each customer to each dealership (in miles):

```
CREATE TEMP TABLE customer_dealership_distance
AS (
  SELECT
    customer_id,
    dealership_id,
    c.lng_lat_point <@> d.lng_lat_point AS dist
  FROM customer_points c
  CROSS JOIN dealership_points d
);
```

1. Finally, for each customer ID, you select the dealership with the shortest distance. So far, you have got the location for customers and dealerships and obtained distances from each customer to each dealership. The next task is to find the customer-dealership combination that has the shortest distance for the customer using a `DISTINCT ON` clause. The `DISTINCT ON` clause guarantees only the first record for each unique value of the column in parentheses. In this case, you will get one record for every `customer_id` value, and because this is sorted by distance to dealerships, you will get the first dealership that has the shortest distance:

```
CREATE TEMP TABLE closest_dealerships AS (
  SELECT DISTINCT ON (customer_id)
    customer_id,
    dealership_id,
    dist
  FROM customer_dealership_distance
  ORDER BY customer_id, dist
);
```

1. Now that you have the data to fulfill the marketing team's request, you can calculate the typical distance from each customer to their closest dealership. You have learned that there are two common ways to represent the typical value of a dataset: mean and median. You will get both using the following query:

```
SELECT
  AVG(dist) AS avg_dist,
  PERCENTILE_DISC(0.5)
```

```
    WITHIN GROUP (ORDER BY dist) AS median_dist
FROM closest_dealerships;
```

The `PERCENTILE_DISC()` function picks the discrete value of certain percentiles from the dataset. So, `PERCENTILE_DISC(0.5)` will bring you the median, which is the 50% percentile value of the dataset. Here is the output of the preceding query:

```
      avg_dist      |    median_dist
--------------------+-------------------
 146.77826608034164 | 91.23958293233487
```

The result is that the average distance is about 147 miles, but the median distance is about 91 miles.With all the topics introduced in this chapter, you now have the ability to handle special data types in PostgreSQL. Next, you will apply these skills to the following activity.

Activity 12

Your business analysts are interested in discovering more patterns from the customers' purchasing history. They have some hypotheses that they would like you to provide supporting data for. Please use the skills you learned in this chapter to help them out:

- They are wondering whether there is any difference in purchasing (average daily sales) between weekdays and weekends. You will need to get the average daily sales for each weekday and weekend day and compare them side by side.
- They are wondering whether there is any difference in the purchase of limited-edition products between different weekdays and weekends.

Summary

This chapter covered advanced data types in SQL, focusing on date, text, and geospatial data types. It introduced the importance of these data types and their unique computational rules, storage, and display requirements. With these topics, you will be able to process data types with their own processing rules, thus extending relational databases' ability to handle

complex real-world questions.At this stage, you have completed the learning of SQL features. In the final two chapters of this book, you will utilize your knowledge of SQL to perform analytical tasks. Stay tuned for *Chapter 13, Inferential Statistics*.

13 Inferential Statistics using SQL

Join our book community on Discord


https://packt.link/EarlyAccessCommunity

You have learned how to perform complex data operations such as filtering, sorting, and aggregating data using window functions from previous chapters. In this chapter, we will introduce several key concepts in inferential statistics that can help you to make an educated guess about a big group based on limited samples.The following topics will be covered:

- Moving from Analytics to Statistics
- Estimating: point estimate and confidence interval
- Testing hypotheses
- Analyzing correlation and performing regression

The goal is to get you familiar with these ideas so to make it easier in practice to uncover hidden patterns, draw conclusions, and make informed decisions to enhance efficiency and improve effectiveness by using these tools.

Moving from analytics to statistics

Moving from analytics to statistics is like going from reading a story to understanding how it was written. Analytics helps us see what's happening in the data—like spotting trends or patterns in sales, website visits, or customer behavior. It's great for getting quick answers. But statistics takes things a step further. With statistics, we learn how to test ideas, make predictions, and understand how confident we can be in our results. It's not just about looking at numbers—it's about making smarter decisions based on them.Let us understand them in depth in the following sections.

Understanding Fundamental Concepts: population vs samples, parameters vs statistics

In this section, we will explore the foundational concepts of statistical inference. Here we study a subset of individuals (the sample) to draw conclusions about the larger group (the population). Parameters describe characteristics of the population, while statistics are the calculated values from the sample used to estimate those parameters. Grasping these distinctions is key to designing sound studies, interpreting results accurately, applying data-driven insights effectively, and making data-informed decisions efficiently.Let's begin by distinguishing two foundational concepts in statistics: populations and samples. From there, we'll explore the difference between parameters and statistics, followed by an overview of common probability distributions and the two core statistical measures—central tendency and variability. These elements not only form the backbone of statistical inference but also play a crucial role in exploratory data analysis, guiding how we interpret and summarize data.

Population vs Samples

Let us understand what these terms mean:

- **Population**: A population, as the name implies, is the entire group that you are interested in studying, i.e., it includes all members of a defined group. For example, if you want to study the average weight of adults in a country, the population will consist of all adults in that country. Populations can be large and often impractical to study in their entirety.
- **Sample**: A sample is a subset of the population selected for analysis. Samples are used because it is usually not feasible to collect data from the entire population. By studying a sample, you can make inferences about the population. For instance, you might measure the weights of 1000 adults from the country as a "proxy" for the average weight of all adults. The United States government conducts a decennial census, since the government cannot do that for everyone, it will sample a fraction of population to infer about the population.

Population vs sample may also depend on the context. Let us say we are interested in automobile product types sold in the sales table. Since the table contains all sale records, we have the entire population. However, if we are only querying, say, the first 100 rows, or the sales table does not have all the sales records then we are looking at sample.Under certain circumstances, the underlining data is very large, and we are constrained by computing resources. What we could do in these circumstances is to extract some records randomly to get a glimpse of how the data looks like. The following SQL will randomly sample 100 records:

```
SELECT * FROM SALES ORDER BY RANDOM() LIMIT 100;
```

Parameters vs statistics

Related to population vs sample, you probably hear quite often the buzz words of parameters vs. statistics. Let us now understand the distinction between these two concepts:

- **Parameters**: Parameters are numerical values that describe an entire population (e.g., population mean). They are fixed values, but in practice, they are often unknown because it is challenging to collect data from the entire population. Examples of parameters include the population mean (μ) and population standard deviation (σ) for a normal distribution.

The following table summarizes some of the key probability distributions used in statistical inference. It highlights their central parameters, characteristics, and typical applications.

Distribution	Key Parameters	Characteristics	Applications
Normal	Mean (μ), Standard Deviation (σ)	Continuous, symmetric, bell-shaped curve; defined by μ and σ	Confidence intervals, hypothesis tests, error modeling
t-Distribution	Degrees of Freedom (ν)	Continuous, symmetric with heavier tails than the normal; becomes similar to the normal as ν increases	Inference on means when the population variance is unknown; small sample tests
Chi-Square	Degrees of Freedom (k)	Continuous, skewed right, defined for non-negative values; distribution of a sum of squared standard normals	Variance testing, goodness-of-fit tests, independence tests in contingency tables
F-Distribution	Two degrees of freedom (d_1, d_2)	Continuous, skewed right; the ratio of two scaled chi-square variates	ANOVA, regression model comparisons, tests for equality of variances
Binomial	Number of Trials (n), Probability of Success (p)	Discrete, symmetric or skewed based on p; models the number of successes in n trials	Modeling binary outcomes (e.g., success/failure), tests of proportions
Poisson	Rate (λ)	Discrete, skewed right; mean equals the variance, modeling counts of events over time or space	Modeling count data (e.g., arrival counts, event frequencies)

Table 13.1: Key Distributions and Their Analytical RolesEach of these distributions provides the mathematical foundation for calculating probabilities, critical values, and p-values. Different probability distributions serve distinct roles in statistical testing based on their characteristics and parameters. For example, the **normal distribution**, defined by mean and standard deviation, is widely used for modeling errors and conducting hypothesis tests and confidence intervals due to its symmetric bell shape. The **t-distribution**, with heavier tails and defined by degrees of freedom, is more suitable for small sample inference when population variance is unknown. The **chi-square distribution**, skewed right and based on squared standard normals, supports variance testing and categorical data analysis like goodness-of-fit and independence tests. The **F-distribution**, a ratio of chi-square variables, is often seen application in comparing variances and evaluating models in ANOVA and regression.For discrete outcomes, the **binomial distribution** models binary events across trials, aiding in proportion tests, while the **Poisson distribution**, defined by a rate parameter, is suited for modeling event counts over time or space. Each distribution aligns with specific data types and testing needs, ensuring accurate statistical inference.

- **Statistics**: Statistics are numerical values that describe the main features of a dataset. They serve as the “best guess” for the corresponding parameters of the population. Two main categories of such statistics are **measures of central tendency** and **measures of variability**. Let us look at some of the most commonly used statistical properties in these two measures.

Measures of Central Tendency

Measures of central tendency describe the center or typical value of a dataset. They provide a single value that represents the “middle” of the data distribution. The most common measures of central tendency are mean, median, and mode. Let us understand each of them here:

- **Mean**(inter-changeably sample average): This is used to estimate the actual average of the population, and the bigger and more representative the sample is, the closer the sample mean gets to the true mean. The mean is calculated by summing all the values in a dataset and dividing by the number of values:
-
- x: the observed values of a sample item
- is the sum of all values in the sample.
- n is the total number of observations in the sample

The following SQL will calculate the average number of customers across the cities.

```
SELECT ROUND(CAST(AVG(number_of_customers) as numeric),0) AS avg_num_cust FROM top_cities_data;
```

This is the output we get:

```
avg_num_cust
          534
```

- **Median**: The median is the 50^{th} percentile value of a dataset when the values are arranged in ascending or descending order. It is less affected by outliers and skewed data compared to the mean.

Example: Calculating the median number of customers across all cities

```
SELECT PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY number_of_customers) AS median FROM top_cities
```

This is the output we get:

```
median
    422
```

- **Mode**: The mode is the value that appears most frequently in a dataset. A dataset may have one mode, more than one mode, or no mode at all.

Example: Finding the Mode of number of customers in the `top_cities_data` table:

```
SELECT COUNT(number_of_customers) AS frequency
FROM top_cities_data
ORDER BY frequency DESC
LIMIT 1;
```

The following is the output of the query:

```
frequency
        40
```

Measures of Variability

Measures of variability describe the spread or dispersion of a dataset. They provide insights into how much the values in a dataset vary from the central tendency. The most common measures of variability are range, variance, and standard deviation.

- **Range**: The range is the difference between the maximum and minimum values in a dataset. It provides a simple measure of the spread of the data.

Example: Finding the range of number of customers from the same top_cities_data table:

```
SELECT MAX(number_of_customers)-MIN(number_of_customers) AS Range FROM top_cities_data;
```

```
range
  1079
```

- **Variance**: The variance measures the average squared deviation of each value from the mean. It provides a measure of how much the values in a dataset vary from the mean.
- **Standard Deviation**: The standard deviation is the square root of the variance. It tells you how much individual values “deviate” from the average. Taking the square root brings the units back to the original scale of the data, making it intuitively easier to interpret. A higher standard deviation means greater variability in the dataset.

Example: Calculating the Variance and standard deviation of number of customers in the `top_cities_data` table. You can verify that standard deviation is the square root of variance!

```
SELECT VAR_SAMP(number_of_customers) AS sample_variance,
STDDEV_SAMP(number_of_customers) AS sample_standard_deviation
FROM top_cities_data;
```

Here is the output that we see:

```
sample_variance   | sample_standard_deviation
------------------+--------------------------
 66129.887179487179 |          257.157319902598
```

You may simply use `VARIANCE` and `STDDEV` to calculate variance and standard deviation for a sample.Central tendency e.g., mean, alone doesn’t show variability: we know where the center of the data is, but we have no idea how spread out the values are. On the other hand, standard deviation depends on the mean – We need the mean first as the reference because standard deviation calculates how far each data point is from that average. Calculating both measures of statistics gives us a fuller picture of your data. We should always look at these statistical measures collectively when doing data analysis.Let us do another exercise, we want to find out average price and the associated standard deviation for automobile product from the Zoomzoom table. we may do the following:

```
SELECT COUNT(*) as cnt,
ROUND(CAST(AVG(sales_amount) AS numeric),0) AS avg_sales,
ROUND(CAST(STDDEV(sales_amount) AS numeric),0) AS stddev_sales,
PRODUCT_ID
FROM SALES
WHERE product_id IN (4,6,9,10,11)
GROUP BY product_id
ORDER BY product_id;
```

This is the result we get:

```
cnt  | avg_sales | stddev_sales | product_id
------+-----------+--------------+------------
 1924 |    109872 |         8224 |          4
 2104 |     62655 |         4644 |          6
 1590 |     33375 |         2513 |          9
 1082 |     81534 |         6251 |         10
  154 |     90682 |         6791 |         11
```

We find out `product_id` of 4 has the highest average sale amount, but the sample standard deviation is also the largest.Now that you have learned populations vs samples, and the difference between parameters and statistics. Let us explore another important concept in inferential statistics: point estimate and confidence intervals.

Estimating: point estimate and confidence interval

While a point estimate provides a single value, it does not convey the uncertainty associated with the estimate. A confidence interval addresses this by providing a range of values within which the true population parameter is likely to lie, along with a specified level of confidence (the probability that the value of a parameter falls within a specified range of values, e.g., 95%).A confidence interval consists of two parts:

- Point Estimate: The central value of the interval.
- Margin of Error: The range around the point estimate that accounts for sampling variability.

Where:

- represents sample mean
- Z = Z-score depends on confidence level, e.g., 1.96 for 95%. You can find Z-score for a given confidence level in Z-table.
- s = standard deviation
- n = Sample size

We may construct confidence intervals for the average sale amount for the automobile product type as shown below for a 95% confidence level:

```
SELECT COUNT(*) as cnt,
ROUND(CAST(AVG(sales_amount) AS numeric),0) AS avg_sales,
ROUND(CAST(STDDEV(sales_amount) AS numeric),0) AS stddev_sales,
ROUND(CAST(AVG(sales_amount)-1.96* STDDEV(sales_amount)/SQRT(COUNT(*)) AS numeric),0) AS lower_CI
ROUND(CAST(AVG(sales_amount)+1.96* STDDEV(sales_amount)/SQRT(COUNT(*)) AS numeric),0) AS upper_CI
PRODUCT_ID
FROM SALES
WHERE product_id IN (4,6,9,10,11)
GROUP BY product_id
ORDER BY product_id;
```

This is the result we get:

```
cnt  | avg_sales | stddev_sales | lower_ci | upper_ci | product_id
------+-----------+--------------+----------+----------+------------
 1924 |    109872 |         8224 |   109504 |   110239 |          4
 2104 |     62655 |         4644 |    62456 |    62853 |          6
 1590 |     33375 |         2513 |    33252 |    33499 |          9
 1082 |     81534 |         6251 |    81161 |    81906 |         10
  154 |     90682 |         6791 |    89609 |    91754 |         11
```

The interpretation is we are 95% confident that the true mean sale value falls between 109504 and 110239 for `product_id = 4`. Estimation is a crucial aspect of inferential statistics, allowing us to make informed guesses about population parameters based on sample data. Point estimates provide a single best guess, while confidence intervals offer a range of values that account for sampling variability. By using SQL to calculate sample statistics and applying statistical formulas, we can construct meaningful estimates that guide decision-making and provide insights into the population..….. We have learned how to use confidence intervals to estimate a range where a

population value is likely to fall, giving us a sense of certainty around our sample results. But what if we want to test a specific claim—like whether a new method really improves performance? That's where hypothesis testing comes in. Instead of estimating, we ask a direct question and use data to decide whether the evidence supports or contradicts our assumption.

Testing hypotheses

Have you ever wondered how researchers or scientists prove their ideas? How do businesses decide if a new product performs better than the old one? Or how does a pharmaceutical company determine if a new medicine really works?These types of questions rely on hypothesis testing. At its core, hypothesis testing is like a trial for an idea—instead of assuming something is true, we test the evidence and see if the data supports it.Hypothesis testing is a fundamental pillar of statistical inference, providing a structured framework for making decisions about populations based on sample data. It allows us to formally evaluate competing claims or hypotheses about the characteristics of a population. By following a systematic series of steps, we can determine whether there is enough statistical evidence to reject a null hypothesis in favor of an alternative hypothesis.Imagine we own a bagel shop and believe that changing the menu increases sales. Instead of just assuming it works, we gather data on sales before and after the change. Hypothesis testing helps us determine whether the sales increase is real or just happened by chance.The process involves formulating hypotheses, selecting an appropriate statistical test based on the nature of the data, calculating a test statistic, and ultimately making a decision based on a predetermined significance level. The summary below will guide us through this crucial process and help you choose the right statistical tool for your specific research question.Here are the hypothesis testing steps:

1. **State a Hypothesis**: In hypothesis testing, the hypothesis that is being tested is known as the alternative hypothesis (*H_a* or *H_1*). Often, it is expressed as a statistical relationship between variables. The null hypothesis (*H_0*), on the other hand, states that there is no statistical relationship between the variables being tested, typically, the exact opposite of the alternative hypothesis.

In the above bagel shop example, the null hypothesis may be there is no significant difference between before and after the menu change. The alternative hypothesis may be stated as sales increase. Once we have the hypothesis, we next need to set the Significance Level (alpha) which is the probability of rejecting the null hypothesis when it is true. Common values are 0.05 (5%) or 0.01 (1%). When we have a lot of data, we may choose the smaller significance level of 1%.

1. **Collect Data and calculate test statistics**: Once we have the hypothesis, we will need of course collect the data and measure relevant information (e.g., sales before and after the strategy). Using the sample data collected, we compute the value of the chosen test statistic.

When it's time to test the hypothesis, it's important to leverage the correct testing method. Factors to consider when choosing the right test:

- **Objective of the analysis**: what is the business goal from the analysis: compare means, proportions or associations?
- **Type of data**: is the data numerical (continuous or discrete) or categorical?
- **Sample size**: large or small?
- **Population variance**: population variance known or unknown?
- **Number of groups**: the number of groups being compared?
- **Dependency**: Whether the populations are independent or dependent
- **Distribution of data**: does your data follow a normal distribution or do you have prior knowledge of proper distribution for the data

The two most common hypothesis testing methods are one-sided and two-sided tests, or one-tailed and two-tailed tests, respectively. Typically, we would want to leverage a one-sided test when we have a strong conviction about the direction of change we expect to see due to the hypothesis test. We would leverage a two-sided test when we are less confident in the direction of change. The following table summarizes typical tests:

Test Name	Type	Purpose	When to Use

Z-test (One-Sample, Two-Sample for Proportions)	Parametric	Compares means or proportions	Used when population variance is known or sample size is large
t-test (One-Sample, Independent Samples, Paired Samples)	Parametric	Compares means of two groups	One-sample: Against a known value; Independent samples: Two separate groups; Paired samples: Same group at different times
ANOVA (One-Way, Repeated Measures)	Parametric	Compares means of three or more groups	One-way: Independent groups; Repeated measures: Dependent groups
Mann-Whitney U Test (Wilcoxon Rank-Sum)	Non-parametric	Compares two independent groups	Alternative to t-test when normality assumption is violated
Wilcoxon Signed-Rank Test	Non-parametric	Compares paired samples	Alternative to paired t-test when normality assumption is violated
Kruskal-Wallis Test	Non-parametric	Compares means of three or more groups	Alternative to one-way ANOVA when normality/equal variance assumptions are violated
Friedman Test	Non-parametric	Compares repeated measures	Alternative to repeated measures ANOVA when normality/sphericity assumptions are violated
Chi-Square Goodness-of-Fit Test	Categorical variable	Compares observed vs. expected frequencies	Tests if observed frequencies differ significantly from expected
Chi-Square Test of Independence	Categorical variable	Examines association between two categorical variables	Determines if variables are statistically dependent
Binomial Test	Categorical variable	Tests probability of success in two-category trials	Used for independent Bernoulli trials

Table 13.2: Hypothesis Testing Methods OverviewForm the table, we understand the following:

- **Determine the p-value**: P-value describes the probability that, assuming the null hypothesis is correct, the probability one may still observe results that are at least as extreme as the results of your hypothesis test. The smaller the p-value, the more likely the alternative hypothesis is correct, and the greater the significance of the results.
- **Compare and Decide***:* The final step is to state the conclusion in the context of the original research question. If the p-value is less than or equal to the significance level (alpha), reject the null hypothesis. Avoid saying we "prove" the alternative hypothesis; instead, say there is "sufficient evidence" (so accept) or "insufficient evidence" to support it (so reject).

Once we've tested a hypothesis and drawn conclusions about a single variable or comparison, we might start wondering about relationships between variables. For example, does studying more hours correspond to better test scores? That’s where correlation comes in—it helps us measure quantitatively and understand how two variables move together, and whether that connection is strong, weak, or just a coincidence.

Analyzing correlation and performing regression

Correlation is a statistical measure that quantifies the strength and direction of a linear relationship between two quantitative variables. To understand correlation, it's helpful to consider variance and covariance. Variance measures the spread or dispersion of a single variable around its mean. Covariance, on the other hand, extends this idea to two variables. It measures the direction of the linear relationship between them, indicating whether they tend to vary together (positive covariance) or in opposite directions (negative covariance). However, the magnitude of covariance is not standardized and depends on the units of the variables, making it difficult to directly interpret the strength of the relationship. This is where correlation becomes useful. The Pearson correlation coefficient (r) can be seen as a standardized version of the covariance. It's calculated by dividing the covariance of the two variables from a sample of (n) paired data points (,) by the product of their standard deviations (which are the square roots of their respective variances):where:

- and are the individual data points for the two variables.
- is the sample mean of the (x) values.
- is the sample mean of the (y) values.

Correlation coefficient lies within a range between -1 and +1, making it unitless and directly interpretable in terms of both the direction and the strength of the linear relationship. The interpretation of (r) is as follows:

- **(0<r<=1):** Indicates a positive linear relationship. As one variable increases, the other variable increases proportionally. The data points would scatter around a straight line with a positive slope (exactly on a straight line when r equals 1). The more r is closer to 1, the stronger the positive linear relationship. .
- **(r = 0):** Indicates no linear relationship between the two variables. Changes in one variable are not associated with any consistent linear change in the other. When we plot the two variables on a x-y coordinates, they would appear randomly scattered in the coordinates likely without particular pattern of either upward or downward “trend”. However, this does not rule out the possibility of a non-linear relationship.
- **(-1<=r<0):** Suggests a negative linear relationship. As one variable increases, the other variable decreases proportionally. The data points would lie exactly on a straight line with a negative slope when r equals -1.

Now consider an example where we want to examine the relationship between public transportation percentage and population in the `PUBLIC_TRANSPORTATION_BY_ZIP` table. The correlation coefficient can be conveniently calculated using the `CORR` function:

```
SELECT ROUND(CAST(CORR(public_transportation_pct,public_transportation_population) as numeric),4)
FROM PUBLIC_TRANSPORTATION_BY_ZIP;
```

This is how the output looks:

```
correlation_coefficient
-------------------------
                  0.7478
```

There is a positive relationship (r≈0.75) between population and public transportation percentage, meaning the larger the population and the higher public transportation percentage will be, which makes intuitive sense.Correlation is a valuable tool in inferential statistics for understanding relationships between variables. By calculating the correlation coefficient using SQL, we can quantify the strength and direction of these relationships, providing insights that inform predictions and decision-making. Whether we analyze employee data, customer behavior, or financial trends, correlation helps uncover meaningful patterns that drive informed inferences.It's important to note that correlation only describes linear relationships; two variables can have a strong non-linear association even if their linear correlation is close to zero. Furthermore, correlation does not imply causation; a strong correlation between two variables doesn't necessarily mean that one causes the other. There might be a confounding variable influencing both, or the relationship could be coincidental. So caution should always be exercised when interpreting correlationsWhile correlation measures the strength and direction of a relationship between two variables, regression goes a step further by estimating this relationship which can be used to make predictions. It can be used to model the relationship between one dependent variable and one or multiple

independent variables. The former is called simple linear regression and the latter multiple linear regression. The model takes the form:

- is called response variable, target variable, or dependent variable
- : is called regressors, explanatory variables, predictor variables, or independent variables.
- is number of independent variables
- : commonly known as regression coefficients, effects, or simply factors. where is the intercept, all other represent the individual effect of respectively (1 to *p*). The interpretation is the expected change in the response variable when increases by one unit with other predictor variables held constant.
- : is formally known as error terms. This variable captures all other "unknown" factors which influence the dependent variable other than the predictors.

Estimating the regression coefficients for linear model can be done by minimizing the error term . A common strategy is to do least square estimation by minimizing the sum of squared errors (SSE) which results in (Interested readers may refer to statistics text books on the detailed derivation):

Interpreting Regression Results

Key components to look at when interpreting regression results include:

- **Coefficients**: These indicate the strength and direction of the relationship between the independent and dependent variables.
- **R-squared**: This statistic measures the proportion of the variance in the dependent variable that is predictable from the independent variables. A higher R-squared value indicates a better fit of the model. The tighter the relationship, the larger the portion of the variability explained. If independent variables and the dependent variable are "perfectly correlated", then 100% of the variance of the dependent variable can be explained by the relationship.
- **P-values**: These help determine the statistical significance of the coefficients. A low p-value (typically < 0.05) suggests that the corresponding independent variable has a significant effect on the dependent variable. You will find more detailed discussion on P-value later

Understanding a simple linear regression example

We learned there is a strong correlation between population and public transportation percentage from the earlier exercise. Now we want to build a simple regression model using population (x) to predict percentage of public transportation (y). We can achieve this by using the internal function **REG_INTERCEPT** and **REGR_SLOPE**:

```
select REGR_SLOPE(public_transportation_pct,public_transportation_population) as beta_1,
REGR_INTERCEPT(public_transportation_pct,public_transportation_population) as beta_0
FROM PUBLIC_TRANSPORTATION_BY_ZIP;
```

Beta_0 (intercept) represents the predicted public transportation percentage when population is zero. For a given population of 500, the model will predict roughly 4. The complete regression model in SQL can be written as:

```
WITH regmodel AS (
SELECT REGR_SLOPE(public_transportation_pct,public_transportation_population) as beta_1,
REGR_INTERCEPT(public_transportation_pct,public_transportation_population) as beta_0,
regr_r2(public_transportation_pct
FROM PUBLIC_TRANSPORTATION_BY_ZIP
)
SELECT 500 as population,
regmodel.beta_1 * 500 + regmodel.beta_0 AS prediction
from regmodel;
```

This is the output:

```
        beta_1        |       beta_0
----------------------+--------------------
 0.004262755440353541 | 1.9806864083561269
```

To understand how well our regression model predicts the outcome, we can look at the R^2 which reflects the proportion of variance in the dependent variable, explained by the independent variables:

```
WITH regmodel AS (
SELECT REGR_SLOPE(public_transportation_pct,public_transportation_population) as beta_1,
REGR_INTERCEPT(public_transportation_pct,public_transportation_population) as beta_0,
regr_r2(public_transportation_pct, public_transportation_population) AS r_squared
FROM PUBLIC_TRANSPORTATION_BY_ZIP
)
SELECT 500 as population,
regmodel.beta_1 * 500 + regmodel.beta_0 AS prediction,
regmodel.r_squared as r_squared
from regmodel;
```

This is the output we get:

```
 population |    prediction     |     r_squared
------------+-------------------+-------------------
        500 | 4.112064128532897 | 0.559235535090672
```

We found R^2 is about 0.56, not bad since this is an extremely simple one-variable model.There are a few assumptions behind linear regression:

- **Linearity**: The relationship between the independent variable(s) and the dependent variable should be approximately a straight line.
- **Independence of Errors**: Each data point should be independent of the others. The error (or residual) of one observation should not be related to the error of another.
- **Homoscedasticity (Equal Variance)**: The spread of the errors should be roughly the same across all levels of the independent variable(s). In other words, the “mistakes” your model makes should be similar regardless of the value predicted. You can plot residual scatter plot to verify that.
- **Normality of Errors**: The residuals (the differences between the observed and predicted values) are assumed to be normally distributed. This helps in making reliable statistical inferences about the relationship.

These assumptions help ensure that the model's estimates are reliable and that the statistical tests (like the t-test for the slope) are valid. Deviations from these conditions may lead to biased estimates or misinterpretation of the model’s significance. Like correlation, regression analysis does not infer causal relationship between two (or more) variables.If our model includes higher order polynomial terms or interactions, it is advisable to standardize the variables to reduces collinearity across related terms and thus improves stability of coefficient estimates and their interpretation.When we develop a model, we often choose to create several candidate models to compare. In this case , adjusted R-square is more appropriate as it adjusts R^2 for the number of predictors in the model, specifically penalizing for adding unnecessary complexity (Interested readers can read more details, e.g., in www.investopedia.com by searching R^2 and adjusted R^2)When there are many independent variables, not all of them will be associated with the dependent variable. We can use either forward or backward regression to add or drop variables that have no predictive power (independent variables not contributing to explaining variance of the dependent variable). For forward regression, we start with one independent variable, and then add another one, and so on. In backward regression, we first build the “saturated” model by including all independent variables of interest; and then drop independent variable that is not “contributing” to the model one at a time until no more variables shall be dropped.When deciding which variable(s) to add or drop, we may check the associated p-value. In linear regression, the p-value tests the null hypothesis that there is no relationship between a predictor variable and the response variable. A low p-value (typically less than 0.05) means we can reject the null hypothesis and conclude that there is a significant relationship between the independent variable and the response. If p-value is greater than 0.05, we cannot reject the null hypothesis of no relationship between the variables of interest, the independent variable can be dropped.It is not straightforward to calculate p-value in PostgreSQL. One workaround is to create a custom function (using a procedural language such as Python or R) to compute the cumulative distribution function (CDF) for the Student’s t-distribution. Once we have that function, we can calculate the standard error, t-statistic, and then the two-tailed p-value. Or we may directly, e.g., use Python for these tasks by loading the SQL table into the computing environment of python that was described in chapter 4.A good modeler will always do extensive exploratory data analysis (EDA) before starting a modeling journey. E.g., simple scatter plots of x against y will tell you whether linear model is even a good choice. Or whether data transformation may

be needed. Histogram plots will tell you whether there are any outliers which may adversely the fit of the regression. Popular programming language such as python and R come with a variety of libraries or modules to choose for EDA.

Summary

Inferential statistics are essential for making predictions and drawing conclusions about a population based on sample data. By using techniques such as hypothesis testing, confidence intervals, and regression analysis, we can go beyond merely describing the data so tomake informed decisions. SQL plays a crucial role in preparing and manipulating data for these statistical analyses, making it an invaluable tool for data analysts and researchers. Whether we are testing hypotheses, calculating confidence intervals, or performing regression analysis, SQL provides the foundation for robust inferential statistics.

14 A Case Study for Analytics Using SQL

Join our book community on Discord


https://packt.link/EarlyAccessCommunity

So far in this book, you have learned about the various features and functionalities offered by SQL. Now, if you take a step back and think about the fundamental question of “why” (i.e., why do you need these features and functionalities at all), you will see that it all stems from the need for data analysis.Data originates in individual activities. Each activity creates its own data point and context. When the data of all the activities in a certain domain is collected and grouped together, statistical patterns emerge. As discussed from the very first chapter of this book, ancient Egyptian pharaohs collected census information to determine the size of the army they could recruit, and modern companies dive into sales data to find the most attractive products for promotion. All these involve the same process of gathering activity level data (the number of people in each household, or the quantity and amount of each sales transaction in a certain period), organizing the data in a way that is easily retrievable, and delivering the data for statistical and predictive analysis.To master this process of data analytics, data gathering, organization, and delivery, in this chapter, you will go through the fundamental concepts and architecture of data analytics systems. Then you will work through multiple exercises and activities of a case study of data analytics, using SQL as the tool. The following topics are covered in this chapter:

- Understanding the data analytics system
- Applying data analysis using SQL

With these topics, you will combine your SQL knowledge with real-world scenarios and build an introductory understanding of the data management workflow in a modern data analytics system.

Technical requirements

In order to complete the exercises in this chapter, you should have set up the PostgreSQL server on your machine and imported the `sqlda` database, as instructed in *Chapter 1*, *Introduction to Data Management Systems*.

Understanding the data analytics system

Data originates from transactional systems such as e-commerce systems or social networking sites. These systems process live data actively and thus are called **Online Transaction Processing** (**OLTP**) systems. They typically store their data in operational databases, aka transactional databases, that are built for fast transaction processing and integrity. These databases are optimized for fast data storage and retrieval, typically focusing on shallow datasets (datasets without complex relationships), most-recent events, and deduplicated data. When the transactions complete, the data will be moved out of the operational databases to make room for more recent data.As important as they are, transactional databases are not designed for analytical purposes. They are designed to handle datasets for transactions, usually with limited size, duration, and context. Analytics-oriented systems

usually answer general business questions, which involve longer durations, touch on multiple aspects of context, and require complex relationships between datasets. Some examples of questions include the following:

- Which dealership sold the most/least electric scooters last month?
- What is the average selling speed of electric scooters in each dealership?
- What is the relationship between the gas price and electric scooter sales over time?
- Based on the answers to the previous questions, how should you arrange the shipment of electric scooters to each dealership, if you foresee there is an increase in gas prices in certain areas?

These questions require analysis over a large number of business activities. For example, there are 5 records of scooters sold in a dealership during the last quarter when/where the gas price was low, and 12 records of scooters sold in another dealership last year when/where the gas price was high. These records contain the base data and context (e.g., gas price, location). They contain tightly related datasets and span a longer time. To utilize this data, the analytical process must often invoke data from multiple activities, such as sales data, gas price data, geographical location data, and so on, and select a specific set of data based on context, which requires filtering on multiple tables and complex joins. Furthermore, it often involves data transformation (last month, last year, same month last year) and derived calculation (percentage, growth, profit/loss), which is time-consuming. In order to handle the need for analytics, you must store measurable events such as the number of scooters sold, as well as describing context such as time, location, product details, and gas price brackets. This structure enables users to slice and dice data efficiently (e.g., comparing sales across regions or during high gas price periods). The scope and complexity of these requirements are way beyond the capacity provided by transactional databases. To satisfy these requirements, people widely use dimensional data models, which are better suited to analytical purposes.You will learn about the concept of dimensional data modeling in the coming section.

Understanding dimensional models

The **dimensional model** is a structured approach to organizing data based on business activities. It is very similar to the relational model, where the model's foundation is still two-dimensional tables that are accessible using SQL. What makes it unique is that the dimensional data model organizes its data around two fundamental components: **facts** and **dimensions**, instead of entities and relationships.Facts in a dimensional model represent quantifiable values, such as sales revenue, units sold, or website traffic, that are subject to statistics and aggregation. They exist in the context of dimensions. Dimensions complement facts by providing further descriptive information that adds valuable information to the quantitative data, such as date/time, geographical location, product categories, or customer demographics. Dimensions typically consist of hierarchies of different levels of descriptors, such as date/time hierarchies (day - week - month - quarter - year), geographical regions (city - state - region - country - continent), product categories (product SKU - product - product category), or customer segments (gender, age group, cultural group).Facts are stored together with their related dimensions in fact tables. Fact tables serve as the centerpiece of the dimensional model, containing quantitative data or measures associated with specific events or transactions. Each row in a fact table represents a distinct event/transaction, such as one sales record, while each computable column represents a measure or metric, such as sales revenue, quantity sold, or profit margin. Each row also includes references to different dimensions, providing context for measures and enabling analysis across multiple dimensions.Dimension tables serve as repositories of descriptive attributes that add context to the quantitative measures stored in fact tables. They are the basis for filtering and aggregating data in analytical queries, enabling users to explore insights from various perspectives. They provide answers to fundamental questions such as who, what, when, where, and how, thereby enriching the understanding of the events captured in the fact tables. Each row in a dimension table represents a unique entity or category, while each column represents an attribute or characteristic of that context. For example, a customer dimension table may include attributes such as customer ID, name, address, and contact information. Each row in this table represents a unique customer. Similarly, a product dimension table may contain attributes such as product ID, name, category, and price. Each row in this table represents a specific product. Dimension tables are typically smaller in size compared to fact tables, as they store descriptive information regarding a finite number of contexts rather than historical records of events for quantitative measures.From the perspective of data analytics, facts are usually positioned at the center of analysis, defined by the dimensions surrounding them. Thus, when it comes to data model diagrams, it is natural to see that fact tables are placed at the center and dimension tables are arranged around them, in a manner that resembles a star. This configuration, known as a **star schema**, simplifies data retrieval and enhances query performance by minimizing the number of joins required to access relevant data. The following figure

demonstrates such a star schema, in which the facts (**Sales Amount**, **Sales Quantity**, and **Cost**) are stored in the fact table (**Sales**) at the center, and related dimensional tables (**Salespeople**, **Customer**, **Product**, etc.) are placed around it.

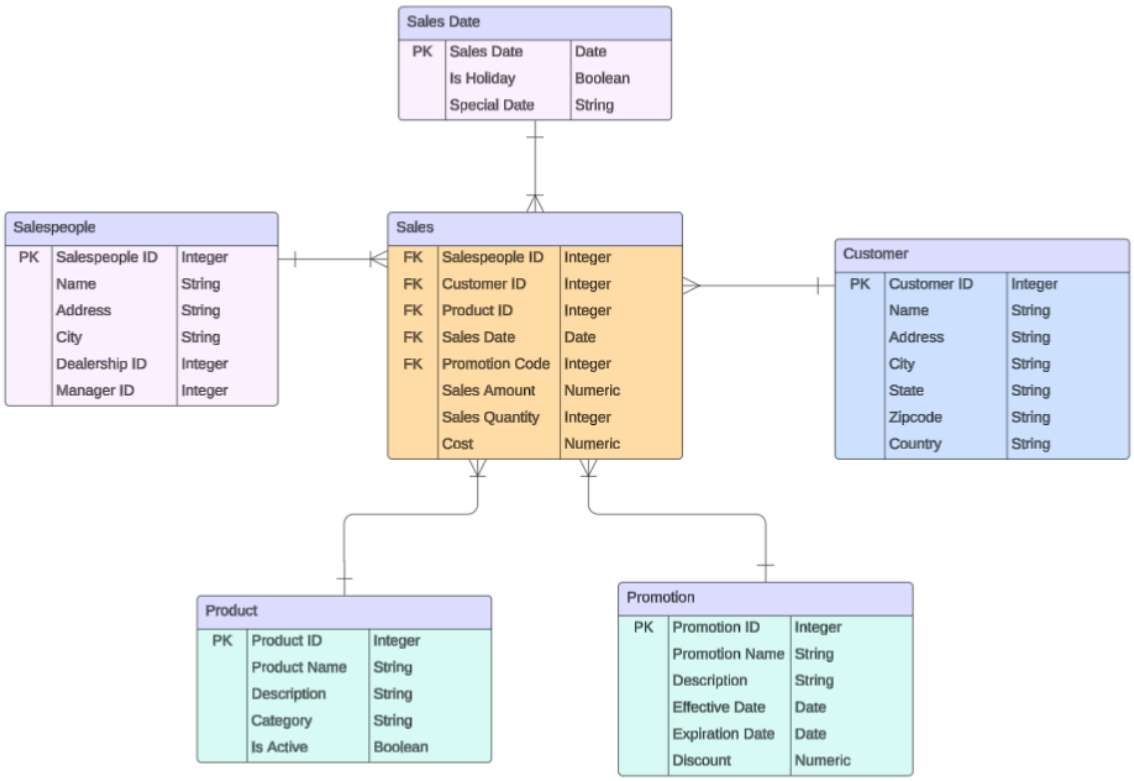

Figure 14.1: Star schema

By structuring data in a star schema, organizations can streamline analytical processes and accelerate insights generation.The dimensional model offers many advantages for analytics. As such, it is widely used in modern enterprise data management systems as the fundamental data storage model. A modern data management system usually stores the dimensional model in a data warehouse. Data warehouses, in essence, are repositories for data that's been transformed and loaded from various source systems. It must be noted, however, that data inside the data warehouse does not come directly in a dimensional format. The data comes from source transactional systems and must be formatted and transformed to fit into the data warehouse star schema. Therefore, there is a full ecosystem around the data warehouse for it to function properly. You will learn about these components in the data warehouse ecosystem in the next section.

Understanding data warehouse architecture

As discussed earlier, data warehouses do not exist alone. The following figure demonstrates this workflow and some related managed services.

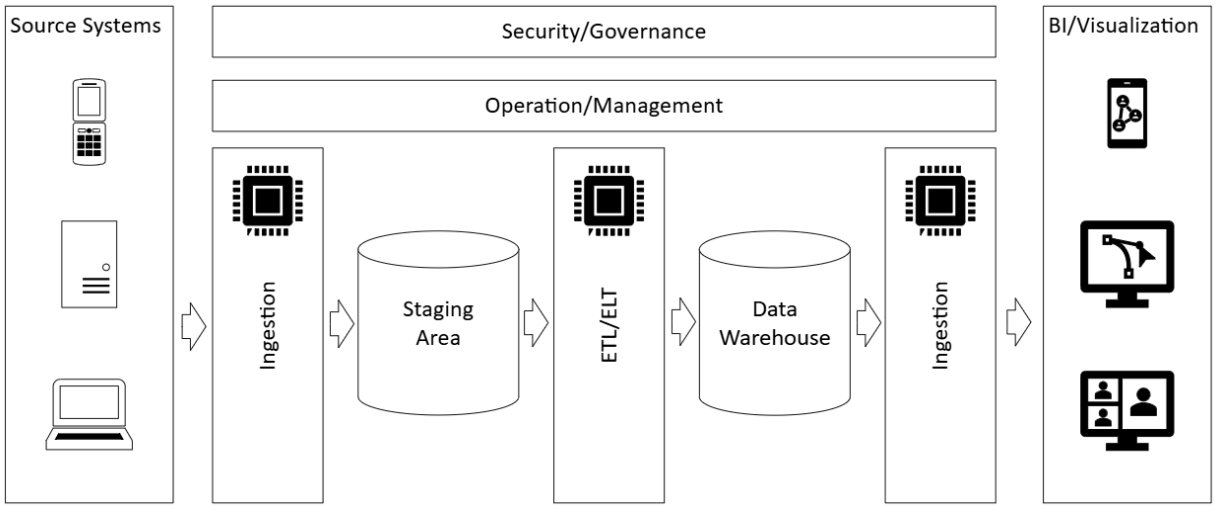

Figure 14.2: Data warehouse ecosystem

Data comes in from sources, usually transactional systems, in all sorts of formats. It must be extracted from the source systems, cleansed, conformed, joined, and aggregated before it can be loaded into the data warehouse. This is done using ingestion tools, commonly called **Extract, Transform, Load (ETL)** or **Extract, Load, Transform (ELT)** tools. These are two common data integration approaches used to move and prepare data from various sources into a centralized repository. In ETL, data is first extracted from source systems, transformed into the desired format or structure, and then loaded into the destination system. In contrast, ELT involves extracting and loading raw data directly into the target system, where transformation is performed using the destination's processing power. ETL is typically used when data transformations are complex and need to be done before loading, often in traditional on-premises systems. ELT can efficiently handle large-scale transformations post-load, especially when the source data is relatively structured.ETL/ELT tools typically utilize a temporary storage area to operate. This temporary storage is commonly called the **staging area**. After the ETL/ELT tools successfully load the data into the data warehouse, where structured and historical data is stored, the tables in the staging area are no longer needed and are usually dropped. Finally, you need **Business Intelligence (BI)** tools, which are frontend applications for data visualization, reporting, and dashboard creation. In recent years, data scientists also use programming tools such as Python to retrieve data from data warehouses for machine learning and artificial intelligence purposes. The services mentioned form the backbone of data processing systems.You can also see from the preceding figure that outside of all these data pipelines, there are also supporting tools for operational tasks such as user and security control, metadata management, data governance, workflow orchestration, and monitoring/logging. However, the design and implementation of these tools are beyond the scope of this book. What is relevant is the workflow of data processing and storage that you saw in the previous paragraph.In the next section, you will go through a series of exercises to get familiar with a case study that brings you into such a data processing workflow.

Applying data analysis using SQL

Given all the preceding discussions, you are now familiar with the common data processing flow. This includes ingesting source files into staging areas, cleansing and transforming raw data, loading transformed data into the data warehouse, and retrieving results using complex queries for downstream analysis. All these can be achieved using SQL. In the following exercises, you will practice your knowledge by applying SQL to each step. The end goal of the exercises is to move the data from a source system file to a data warehouse that is usable for data analysis tools.

Exercise 14.1: Copying from a file into the staging table

The first step of data processing is to ingest the data from the source into the staging area. Source data usually comes in from transactional systems as text files. So, you will start by copying a file into a staging table in this exercise.ZoomZoom has a smaller transactional system that runs in parallel with the main processing system. The company has decided to merge the data from this smaller system into the current `sqlda` database. The engineers of the smaller system have dumped their data into one JSON file. In this exercise, you will load the content of this data file into a staging table in the `sqlda` database.Perform the following steps to complete the exercise:

1. Download the JSON file from this path of the https://github.com/PacktPublishing/SQL-for-Data-Analytics-Fourth-Edition/blob/main/Datasets/customer_sales.txt repository.
2. Read through the file content: Before loading the data, you **must** have a thorough understanding of its structure and values. By looking at the first few rows of this file, it looks like this is a JSONL file. Traditionally, the most common text file format is CSV. In recent years, due to the inherent flexibility and readability, JSON has gradually become the mainstream format of data input. **JSON Lines (JSONL)** is a file format where each line contains a single, valid JSON object, delimited by newline characters. One such row looks like this:

```
{"email": "ariveles0@stumbleupon.com", "phone": null, "sales": [{"product_id": 7, "product_name"
```

Once you identify the format of the file, the next thing is to identify the content. Here are a few things to note by looking at the first few rows:

- Each row contains `customer_id`, `first_name`, `last_name`, `email`, and optionally, `phone`. Based on the convention you have seen so far, it is likely that `customer_id` is the primary key of the dataset.
- Each row contains a `sales` array, but for some rows, this array is empty, yet for other rows, it contains multiple JSON documents. The documents in the `sales` array contain product information, sales transaction date, and sales amount. The sales amount is the only value that is computable in this file and is likely the fact that you should build your star schema upon.
- It must be pointed out that all these analyses are based on the first few rows and, therefore, are exploratory. In the real world, the process of understanding data is an evolving process. You will explore the data, make assumptions, explore the data further, confirm some assumptions and/or reject others, then explore further, before arriving at a conclusion.
- Now that you have a good understanding of the data structure, you will create a staging table to receive the content. You can determine the most appropriate structure of the staging table by looking at the source file's structures and values. Normally, all columns in the staging table should be defined as TEXT fields, one for each attribute in the file. This provides you with maximum flexibility in case the values change format over time. Since this file is a JSONL file (i.e., only one JSON document in each row), you can define the table as containing only one JSONB column:

```
CREATE TABLE customer_sales_json (
  customer_json JSONB
);
```

Note

The data in this file was actually generated from the `customer_sales` table. Once the data is loaded, this `customer_sales_json` table will contain the same data as the `customer_sales` table.

- With the source file residing on your computer and the receiving table created, now you can use the `COPY` command to move data into the staging table:

```
\COPY customer_sales_json
FROM 'c:\Users\Public\customer_sales.txt'
```

As discussed in *Chapter 3*, *Exchanging Data Using COPY*, you need to adjust the path to where you saved the JSON file, with the proper format. Also, as the `\COPY` command is a `psql` command, you must have the whole command in one line.You can now check the populated `customer_sales_json` table by running a `SELECT` statement. Each row should be a valid JSON document.Loading data into the database is the first step in data processing. For your analysis to generate accurate results, your data must be accurate. As such, you must perform a

quality check on all data coming from outside. In the next exercise, *Exercise 14.2: Quality check for raw data*, you will apply some checks to the data in this staging table, before loading it into the star schema in *Exercise 14.3: Loading data into the star schema*.

Exercise 14.2: Quality check for raw data

There is a well-known phrase: garbage in, garbage out. If you feed inaccurate data to your analytical system, your analysis results will be inaccurate. In other words, your analysis is only as good as your data. As such, data quality control must always be an integrated part of the data loading process. In practice, you should do it as early as possible, preferably right after the data is loaded, to avoid wasting your resources on processing bad data. In this exercise, you will use SQL to perform some quick quality checks on the staging table.Perform the following steps to complete the exercise:

1. Open `psql` and connect to the `sqlda` database.
2. Data validity issues caused by errors in manual data input is one of the most common data quality issues. Spelling issues in customer names, typos in addresses, and invalid dates are some examples of them. To discover these kinds of issues, you should convert each text or JSON field into an appropriate data type and apply check constraints. The following SQL shows the discovery of an invalid date of 2018-02-29, which was intentionally placed in the source data file for demonstration purposes:

```
SELECT
  TO_TIMESTAMP(
    (csj.customer_json->>'date_added'),
    'YYYY-MM-DDXHH24:MI:SS'
  ) date_added
FROM customer_sales_json csj;
```

As you copy the file into the staging table using JSON format, the error is carried into the staging table as a string value in JSON documents. Thus, this error is not discovered until you explicitly convert it into a `TIMESTAMP`:

```
ERROR:  date/time field value out of range: "2018-02-29T00:00:00"
```

Now that you have discovered a data issue, how to fix the issue depends on the business requirements. In most cases, the input issue is a source issue. Data engineers will run a SQL query to identify the record(s) where the `date_added` field is `2018-02-29` and communicate with the source team to correct the values in the source system. The source system should then send out a corrected file. Then you should drop the existing staging table and load the new file. There are other ways to fix data issues, such as filling the field with a default value. The exact approach you choose depends on the business requirements.

1. Data validation can also be used to verify or confirm certain data characteristics. For example, in the previous exercise, you made the assumption that `customer_id` is the primary key of this dataset. But this assumption was only based on the observation of the first few rows. To confirm its validity, you can confirm that the number of unique `customer_id` values is the same as the number of total rows. First, you run the following SQL to get the total number of rows:

```
SELECT COUNT(*) FROM customer_sales;
```

This is the output you receive:

```
 count
-------
 50000
```

Then you run the following SQL to get the number of unique `customer_id` values:

```
SELECT COUNT(DISTINCT customer_json->'customer_id')
FROM customer_sales_json;
```

This is the output you receive:

```
 count
-------
 50000
```

The result shows that all the `customer_id` values are unique and can be used as the primary key.

1. The other common data validation is to verify whether the dimension data already exists in the data warehouse, and if so, whether there are any changes to the value. The `customer_id`, `first_name`, `last_name`, and `email` fields exist both in the `customers` table and in this staging table. You would like to first identify whether there are new customers in this dataset or not by checking if there is a customer ID in the new dataset that does not exist in the existing `customers` table:

```
SELECT *
FROM customer_sales_json
WHERE (customer_json->'customer_id')::integer NOT IN (
  SELECT customer_id FROM customers c
);
```

This is the result:

```
(0 rows)
```

This shows that all customers in the new dataset exist in the data warehouse's `customers` table. Now the question is, do these customers have the same name/email as in the existing table? Using first name and last name as examples, you will check for the same customer ID and whether the names are different by running the following SQL:

```
SELECT
  c.first_name, c.last_name,
  replace(
    (csj.customer_json->'first_name')::TEXT, '"', ''
  ) AS new_first_name,
  replace(
    (csj.customer_json->'last_name')::TEXT, '"', ''
  ) AS new_last_name
FROM customer_sales_json csj
JOIN customers c
  ON c.customer_id =
    (csj.customer_json->'customer_id')::integer
WHERE
  c.first_name != (csj.customer_json->>'first_name')
  OR c.last_name != (csj.customer_json->>'last_name');
```

This is what we see:

```
first_name | last_name | new_first_name | new_last_name
-----------+-----------+----------------+---------------
(0 rows)
```

The result is 0 rows again, indicating that all the first names and last names in the new dataset match the existing names in the **customers** table. In real-world scenarios, most dimensional data does change over time. For example, customers move to new addresses, and employees are promoted to new titles. This type of dimension is called a **Slowly Changing Dimension** (**SCD**). SCD processing is a common task in data loading, but it requires complex logic. As such, it is beyond the scope of this book.In the previous exercises, you loaded the data into the staging table and confirmed that the data is of good quality. Now you are ready to load the data from the staging table into the star schema, which you will do in the next exercise.

Exercise 14.3: Loading data into the star schema

As discussed earlier in this chapter, data warehouses store their data in star schemas. But the source data comes in as a flat text file, either CSV or JSON. As such, you must extract the contents and convert them into the star schema format. This involves extracting computable fields and their contexts, i.e., facts and dimensions, into their respective tables.In this exercise, you will use SQL queries to load data from the staging table into a star schema.

In real-world scenarios, you will load the data into existing fact and dimensional tables using `INSERT` statements with `SELECT` clauses. Once you are done with loading, you will drop the staging table to save some space. To simplify the process, in this exercise, you will create new fact and dimension tables using `CREATE` statements with `SELECT` clauses, and you will convert the staging table directly into a fact table.Perform the following steps to complete the exercise:

1. Open `psql` and connect to the `sqlda` database.
2. The first step in data extraction is to identify the fact. You have already identified that the `sales` field in the JSON file is the only computable field. However, there is no way to determine how many sales records each customer has. Some customers do not have any sales records. Others may have several. For example, there are two sales records under `customer_id 32` but none for `customer_id 2`. It is difficult to determine the exact number of records that you need to process.

The `JSONB_ARRAY_ELEMENTS()` function breaks up a JSONB record, with one element as a row. You can use it to break up the **sales** field as follows:

```
SELECT
  (csj.customer_json->'customer_id')::integer
    AS customer_id,
  jsonb_array_elements(csj.customer_json->'sales')
    AS sales_record
FROM customer_sales_json csj;
```

This is the result:

```
customer_id |                                                                        sales_record                                                                        
------------+------------------------------------------------------------------------------------------------------------------------------------------------------
           1 | {"product_id": 7, "product_name": "Bat", "sales_amount": 479.992, "sales_transacti
           4 | {"product_id": 12, "product_name": "Lemon Zester", "sales_amount": 314.991, "sales
```

You can further break down each sales record and put the result together with the customer information in the same row. The result is a flattened table with facts and dimensions that are readily accessible using SQL:

```
CREATE TABLE customer_sales_flattened AS
SELECT
  (csj.customer_json->'customer_id')::INTEGER AS customer_id,
  (csj.customer_json->>'first_name') AS first_name,
  (csj.customer_json->>'last_name') AS last_name,
  (csj.customer_json->>'email') AS email,
  (csj.customer_json->>'phone') AS phone,
  (jsonb_array_elements(
    csj.customer_json->'sales')->'product_id'
  )::INTEGER AS product_id,
  (jsonb_array_elements(
    csj.customer_json->'sales')->>'product_name'
  ) AS product_name,
  (jsonb_array_elements(
    csj.customer_json->'sales')->'sales_amount'
  )::NUMERIC(10,2) AS sales_amount,
  TO_TIMESTAMP(
    (
      jsonb_array_elements(
        csj.customer_json->'sales'
      )->>'sales_transaction_date'
    ), 'YYYY-MM-DDXHH24:MI:SS'
  ) AS sales_date
FROM customer_sales_json csj;
```

This is the result:

```
SELECT 37711
```

Here, you use the `->` operator to get JSON objects from JSONB documents, and use `->>` to get the text value from JSONB documents.

1. The JSON file has 50,000 rows, but the sales fact table only contains `37711` rows. This is because many customers do not have associated sales records. If you only use the fact table, you will miss many customer records, which can be useful for marketing purposes. To keep the customer information, you should extract customer information into a dimensional table:

```
CREATE TABLE customer_sales_dimension AS
SELECT
  (csj.customer_json->'customer_id')::INTEGER AS customer_id,
  (csj.customer_json->>'first_name') AS first_name,
  (csj.customer_json->>'last_name') AS last_name,
  (csj.customer_json->>'email') AS email,
  (csj.customer_json->>'phone') AS phone
FROM customer_sales_json csj;
```

Here is the result:

```
SELECT 50000
```

It is worth noting that, as examined in the previous exercise, the customer information is already stored in the `customers` table. This `customer_sales_dimension` table is created just for demonstration purposes. Also, it is the only dimension table you create in this exercise. In real-world processing, you can foresee that there will be many different dimension tables created from source files, thus forming a star schema.

Note

You can also use a special type of PostgreSQL object called Materialized Views for dimensional tables. Materialized views precompute and store the results of complex queries, making them ideal for generating dimension tables in a star schema.

1. Now that you have both the fact table and the customer dimension table, you can see that the customer's name, email, and phone information are stored in both the fact table and the dimension table. Fact tables are usually very large – much larger than dimension tables. So, the common practice is to remove the descriptive information (usually the fields with long strings) from the fact table and save these fields into the dimension tables.

```
\d customer_sales_flattened
```

When you check the structure of the fact table, you can see multiple such text fields:

```
 customer_id  | integer
 first_name   | text
 last_name    | text
 email        | text
 phone        | text
 product_id   | integer
 product_name | text
 sales_amount | numeric(10,2)
 sales_date   | timestamp with time zone
```

Since you have already moved these fields into dimensional tables in previous steps, you can use `ALTER` statements now to remove the duplicated customer-related columns from the fact table. Similarly, you can remove the `product_name` column too, once you confirm it is also a duplicate from the `product_name` column in the `products` table:

```
ALTER TABLE customer_sales_flattened DROP COLUMN first_name;
ALTER TABLE customer_sales_flattened DROP COLUMN last_name;
ALTER TABLE customer_sales_flattened DROP COLUMN email;
ALTER TABLE customer_sales_flattened DROP COLUMN phone;
ALTER TABLE customer_sales_flattened DROP COLUMN product_name;
```

Now, if you check the table structure again, you will see that the fact table contains only the dimension IDs (`customer_id`, `product_id`, and `sales_date`) and the fact. This greatly reduces the size of the fact table and makes querying this table much more efficient.In relational data modeling, this reduction of redundant columns, as

well as the flattening of embedded fields, is called normalization. Normalization was briefly mentioned in *Chapter 1*. It is a very common task performed in data processing workflows. However, it is usually determined by the data architects who design the entire database system and is beyond the scope of this book. Still, as a data scientist or data engineer, being aware of this technology, as covered in the preceding exercise, will provide you with a better understanding of your job duties for data processing.In this exercise, you have created a fact table and a dimension table, that is, you have effectively built a mini star schema. In real-world scenarios, you will build many more dimension tables and join data from different sources to form fact tables with more facts. But the fundamental workflow is the same.Now that the data is stored in the star schema, which was invented for analytical purposes, the next step of data utilization is to provide data to the users for analytical purposes. This is widely done using SQL queries. You will see some such examples in the next exercise.

Exercise 14.4: Delivery data for analysis

Now that the data is stored in the star schema of the data warehouse, it is ready to be used by data analysts and data scientists. Data is generally retrieved using SQL. However, users usually don't directly use data values. Instead, they will run the SQL from business intelligence software such as Microsoft Power BI and Tableau for data analysis and visualization, Python notebooks for data science research, and machine learning jobs for model training and inferencing. These tasks typically involve reading data from multiple tables and require large-scale filtering and aggregation. You will use `SELECT` statements with `WHERE`, `JOIN`, and `GROUP BY` clauses to extract datasets for analytical purposes. Furthermore, you may need to save the aggregated data for further analysis. For example, if your analytical queries usually run on the day level, you may want to create a new table that sums up the sales for each day. This aggregated table is smaller in size, and running SQL against it can yield better performance.In this exercise, you will go through some queries to get an understanding of these use cases.Perform the following steps to complete the exercise:

1. Open `psql` and connect to the `sqlda` database.
2. The first step in analytics is to join the fact table to the dimension tables and aggregate the data to a level that is appropriate for analysis. For example, the following SQL aggregates the sales amount to the customer city/state, product type, and sales date level. Since customer city/state and product type fields are not in the fact table, such aggregation requires the query to join dimension tables with the information (context) in the fact table.

```
SELECT
  c.city, c.state,
  p.product_type,
  csj.sales_date,
  SUM(csj.sales_amount) total_sales
FROM customer_sales_flattened csj
JOIN customers c
  ON csj.customer_id = c.customer_id
JOIN products p
  ON csj.product_id = p.product_id
GROUP BY
  c.city, c.state,
  p.product_type,
  csj.sales_date;
```

1. If your analysis frequently happens on or above the levels of the previous SQL, such as daily sales reports in the top customer city/state, as well as daily sales reports by product type, you may want to save this query for easy reuse. You can save the query using a view:

```
CREATE VIEW customer_sales_agg_sales AS
SELECT
  c.city, c.state,
  p.product_type,
  csj.sales_date,
  SUM(csj.sales_amount) total_sales
FROM customer_sales_flattened csj
JOIN customers c
  ON csj.customer_id = c.customer_id
JOIN products p
  ON csj.product_id = p.product_id
```

```
GROUP BY
  c.city, c.state,
  p.product_type,
  csj.sales_date;
```

Note that creating a view just saves the query definition. You still run the query every time you use the view and it will not save you time querying the underlying fact table. You can also create an aggregate table using the same query so that the aggregated data is physically saved in the database. This way, when you query against this aggregate table, you are running against a much smaller table and can enjoy better performance. However, to ensure this aggregated table contains the same data as the fact table, you must reload the aggregated table every time your fact table is reloaded, usually as an additional data loading step, immediately following the fact table loading.

1. After you create a view or a table, the aggregated data is defined and easier to access than the original raw data. You can now use the following SQL to retrieve the top sales city/state in the entire sales history, and you may also add a `WHERE` clause to get the top city/state for a given date.

```
SELECT city, state, SUM(total_sales) total_sales
FROM customer_sales_agg_sales
GROUP BY 1, 2
ORDER BY total_sales DESC
LIMIT 10;
```

This is the output:

```
     city      | state | total_sales
---------------+-------+-------------
               |       | 27817511.20
 Washington    | DC    |  7211614.68
 New York City | NY    |  4844308.83
 Houston       | TX    |  4012993.90
 El Paso       | TX    |  3915465.57
 Sacramento    | CA    |  3421386.61
 San Antonio   | TX    |  3256747.19
 Atlanta       | GA    |  3075850.95
 Dallas        | TX    |  3016611.01
 Los Angeles   | CA    |  2840496.61
```

This query, together with the previous queries creating the underlying view/table, is just a simple example of how you can tailor the star schema that you built to better fit into your analytical needs. You have gained an in-depth understanding of SQL through the previous chapters. Now you can use this understanding for all your data analytical needs.With all four exercises, you have gone through the process of loading data from source systems into the data warehouse and delivering result data for analysis. It is now time to review what you have learned in this chapter as well as in the whole book, to put all these pieces together.

Summary

You have just completed this real-world case study using SQL. In this chapter, you utilized the SQL skills you have gained throughout this book, from simple `SELECT` statements to aggregating complex data types. You started this case study with an understanding of the storage characteristics required by data analysis and the data warehouse ecosystem that is built to provide these characteristics. Then you walked through the data loading process step by step, from raw data copying all the way to the final queries for analysis. After completing this chapter, you will be able to create similar tasks in your own data projects.You have reached the end of this book. Throughout these chapters, you have learned about data and how you can make sense of it with one of the most important computer technologies, SQL. You learned how SQL's powerful functionality can be used to organize data, process it, and identify interesting patterns. Additionally, you saw how SQL can be connected to other systems and optimized to analyze at scale. This all culminated in using SQL on a case study to help facilitate the business analytical process.However, these skills are only the beginning of your work. Relational databases are constantly evolving, and new functionalities are being developed all the time. There are also many advanced techniques that this book did not cover. So, while this book is proud to serve you as an introductory guide to data

analytics, it is only the first step in what will definitely be a rewarding journey. We are sure you will enjoy the ride, and we wish you good luck!