SQL Guide

Master relational databases in Python step by step

Ankit Pandey

What You Will Build

The project that you will work on with the help of this book has the goal of supporting many of the aspects of the operation of a fictional vintage home computer store that I'm going to call RetroFun.

RetroFun offers an impressive collection of home computers from the 1980s and 1990s for sale. In this book you will learn how to build some database operations for this made-up company, including:

Maintaining a list of products categorized by several attributes such as year of release, manufacturer, country of origin or CPU.

Keeping track of customers and their orders.

Maintaining star ratings and reviews made by customers.

Tracking page views for articles published on the company's blog.

Generating a lot of reports, both simple and complex, always keeping an eye towards efficiency and performance.

Prerequisites

You should be aware that this isn't a book for the complete beginner. To make the most out of it you should have some previous experience writing Python, and ideally also some basic relational database knowledge. If you have learned to work with databases with my Flask Mega-Tutorial, or with any other introductory Python course, you should be at the right level.

I recommend that you don't just read this book, but also work on all the exercises along with me. For this you will need a recent Python interpreter installed on your computer, and a text editor or IDE in which you feel comfortable writing Python code. Basic knowledge of the terminal or command prompt in your operating system would also help.

How To Work With The Example Code

You are encouraged to write and try all the code examples as you read this book. The code has been thoroughly tested using SQLAlchemy 2.0 on the three major open-source databases:

- SQLite
- MySQL
- PostgreSQL

Other databases are likely to work as well, as long as SQLAlchemy has support for them.

The code from this book should work with SQLAlchemy 1.4 with relatively minor changes, but the effort to back port the book examples is left to the reader. Support for SQLAlchemy versions 1.3 and older was not taken into consideration.

I have released the complete source code for this book on a GitHub repository. In addition to source code, this repository contains data files that can be used to populate the database you will build. At the appropriate times you will be given instructions on how to use these data files.

Conventions Used In This Book

This book frequently includes commands that you need to type in a terminal session. For these commands, a $ will be shown as a command prompt. This is a standard prompt for many Linux shells, but may look unfamiliar to Microsoft Windows users. For example:

```
$ python hello.py
hello
```

In a lot of the terminal examples, you are going to be required to have an activated *virtual environment* (do not worry if you don't know or remember what this is, you will find out very soon!). For those examples, the prompt will appear as (venv) $:

```
(venv) $ pip install sqlalchemy
```

You will also need to interact with the Python REPL, or interactive prompt. Examples that show statements that need to be entered in a Python interpreter session will use a >>> prompt, as in the following example:

```
>>> print('hello!')
hello
```

In all cases, lines that are not prefixed with a >>> prompt are output printed by the command right above, and should not be typed.

Many of the statements that you will need to type in the REPL are database queries that are formed by a call to SQLAlchemy's select() function followed by an often long sequence of method calls. Here is an example of how these queries might look when typed in a single long line:

```
>>> q = select(select_expression).method1(expression1).method2(expression2)
```

These statements can be quite unreadable when shown as above, so the convention used in this book is to show them broken up into multiple lines as follows:

```
>>> q = (select(select_expression)
    .method1(expression1)
    .method2(expression2))
```

Note that to be able to break long lines on the periods as shown above, Python requires the entire expression to be enclosed in parentheses. You can collapse these multi-line statements into single-line when you type them in the REPL.

Acknowledgements

This book was inspired by many of the questions I have received over the years from readers of my Flask Mega-Tutorial, so I'm extremely thankful to them for engaging with me and sharing their problems and ideas. To them, I also owe the realization that Python database programming is an area that hasn't been well explored in technical literature or video content.

Writing a technical book is hard, especially when the book is structured as a tutorial with detailed steps that the reader is expected to follow. While I have put a lot of care and attention in creating this content so that readers can have a smooth experience as they move through the chapters, I relied on reviewers to alert me of mistakes and inconsistencies that I inadvertently introduced. I would like to recognize the work of Martin Bell, Rostislav Roznoshchik and my son Dylan Grinberg as technical reviewers.

Finally, I would like to thank Mike Bayer and Federico Caselli. Mike is the creator of SQLAlchemy and Alembic. He and Federico are the current maintainers, and both have been extremely helpful and patient with my questions. Their assistance gave me a greater understanding of the major changes that have been introduced in releases 1.4 and 2.0 of SQLAlchemy. Mike was also kind enough to review the draft of this book and has made some useful suggestions.

Database Setup

Welcome! This is the start of a journey which I hope will provide you with many new tricks to improve how you work with relational databases in your Python applications. Given that this is a hands-on book, this first chapter is dedicated to help you set up your system with a database, so that you can run all the examples and exercises.

Project Directory

Your first task is to create a project directory where you will store files associated with the project featured in this book.

Open a terminal or command prompt, find a suitable parent directory and create a directory there. Then change into that directory.

```
$ mkdir retrofun
$ cd retrofun
```

Note

"RetroFun" is the name of the fictional company for which the database project featured in this book is for.

Python and SQLAlchemy Installation

As is standard in Python projects, you should create a Python virtual environment where all the dependencies can be installed. The command to do this is:

```
$ python -m venv venv
```

The python -m venv portion is what tells Python to create the virtual environment, by running the venv module that is part of the Python standard library. The second and final venv included in the command is the

name I have chosen for the virtual environment. You are welcome to use a different name if you prefer.

After this command completes, your project will have a subdirectory named *venv*, containing a private copy of your Python interpreter.

Whenever you are ready to start working on this project, you have to tell your terminal session that you want to use the virtual environment. This action is called "activating" the virtual environment.

If you are using a UNIX based shell such as bash, regardless of operating system, the activation command is:

```
$ source venv/bin/activate
```

If you are using a Command Prompt on Microsoft Windows, the activation command is different:

```
$ venv\Scripts\activate
```

Finally, if you are using a PowerShell terminal the following is the activation command:

```
$ venv\Scripts\activate.ps1
```

Regardless of the activation command that you use, your shell prompt should change to indicate that the virtual environment has been activated. The prompt should look more or less as follows:

```
(venv) $ _
```

Note

Virtual environment activations are only active in the shell session in which they are issued. If you have multiple terminals open, the activation command must be given for each terminal session. Activations must be issued again after a computer reboot or restart.

You can now install SQLAlchemy in the virtual environment:

```
(venv) $ pip install sqlalchemy
```

Version 2.0 or newer of SQLAlchemy is required for the code featured in this book.

Database Choices

The code featured in this book is generic enough to be used with any relational database system supported by SQLAlchemy. The code examples have been tested against three popular open-source databases:

- SQLite
- MySQL
- PostgreSQL

If you are interested in a particular database system, and it is not in the list above, then you should ensure that SQLAlchemy supports it, either through a built-in or a third-party integration.

If, on the other side, you have no particular preference, then my recommendation is that you start with SQLite, which is by far the easiest to set up and manage. Since the code uses common features present in all the database systems, you can switch to a different database when and if needed.

Have you made a decision? Now it is time to create a brand-new database for use with this book. If you have your preferred set of tools to do this you are welcome to use them, but in case you need some guidance, the sections that follow offer step-by-step instructions for the three databases listed above.

SQLite Database Installation

SQLite is a C-language library that implements a small, fast and self-contained relational database engine. This database is particularly interesting because it does not require a separate server process to run.

The SQLite library is bundled with the Python interpreter, so support for this database is available to use in Python and SQLAlchemy without installing any additional software or perform any configuration.

If you would like to have a tool that you can use to inspect and manage SQLite databases outside of Python and SQLAlchemy, you can download the sqlite3 command-line shell for your operating system from the official SQLite download page.

MySQL Database Set Up

MySQL is an open-source relational database owned by Oracle Corporation. Unlike SQLite, this database includes server and client components, both of which need to be installed.

MySQL Server

If you have access to a running MySQL server, then you can just create a new database and database user and skip to the client section below. The installation instructions in this section demonstrate how to install a server along with the popular phpMyAdmin management application.

If you don't already have access to a MySQL installation, the easiest way to get one up and running is to use Docker. If you would like to follow the instructions below to install MySQL, first install Docker Desktop.

Copy the following definitions to a file named *docker-compose.yml* in your project directory:

```
version: '3'

services:
  db:
    image: mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: changethis!
    ports:
      - "3306:3306"
    volumes:
      - db-data:/var/lib/mysql
  admin:
    image: phpmyadmin
    restart: always
    environment:
```

```
    - PMA_ARBITRARY=1
  ports:
    - 8080:80
volumes:
 db-data:
```

This Docker Compose configuration file starts a service called db that runs a MySQL server connected to port 3306 of your computer, plus a second service called admin that runs phpMyAdmin on port 8080. The database storage is configured on a separate volume called db-data, to make it possible to upgrade the database container without losing data.

Note the MYSQL_ROOT_PASSWORD line, which has the value changethis!.

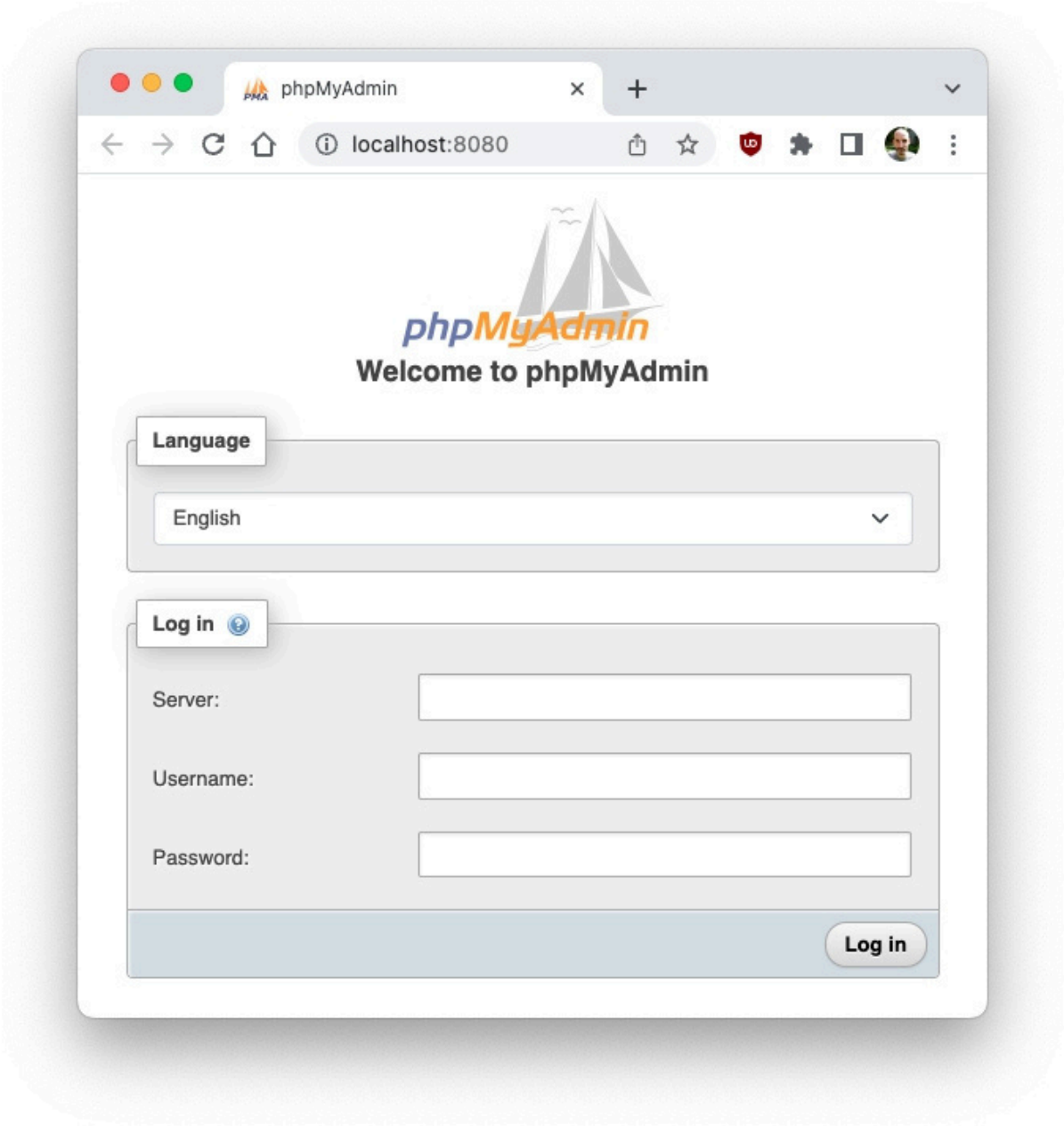
This line defines the administrator password for the MySQL server. Edit this line to set a secure password of your liking.

Once you have this file saved in your project directory, return to the terminal and run the following command to start your MySQL server:

```
$ docker-compose up -d
```

The first time you run this command it will take a while, as Docker has to download the MySQL and phpMyAdmin container images from the Docker Hub repository. Once the images are downloaded, it should take a few seconds for the containers to be launched, and at that point MySQL should be deployed on your computer and ready to be used.

You can open the phpMyAdmin database management tool by typing *http://localhost:8080* in the address bar of your web browser.



To log in, enter the following credentials:

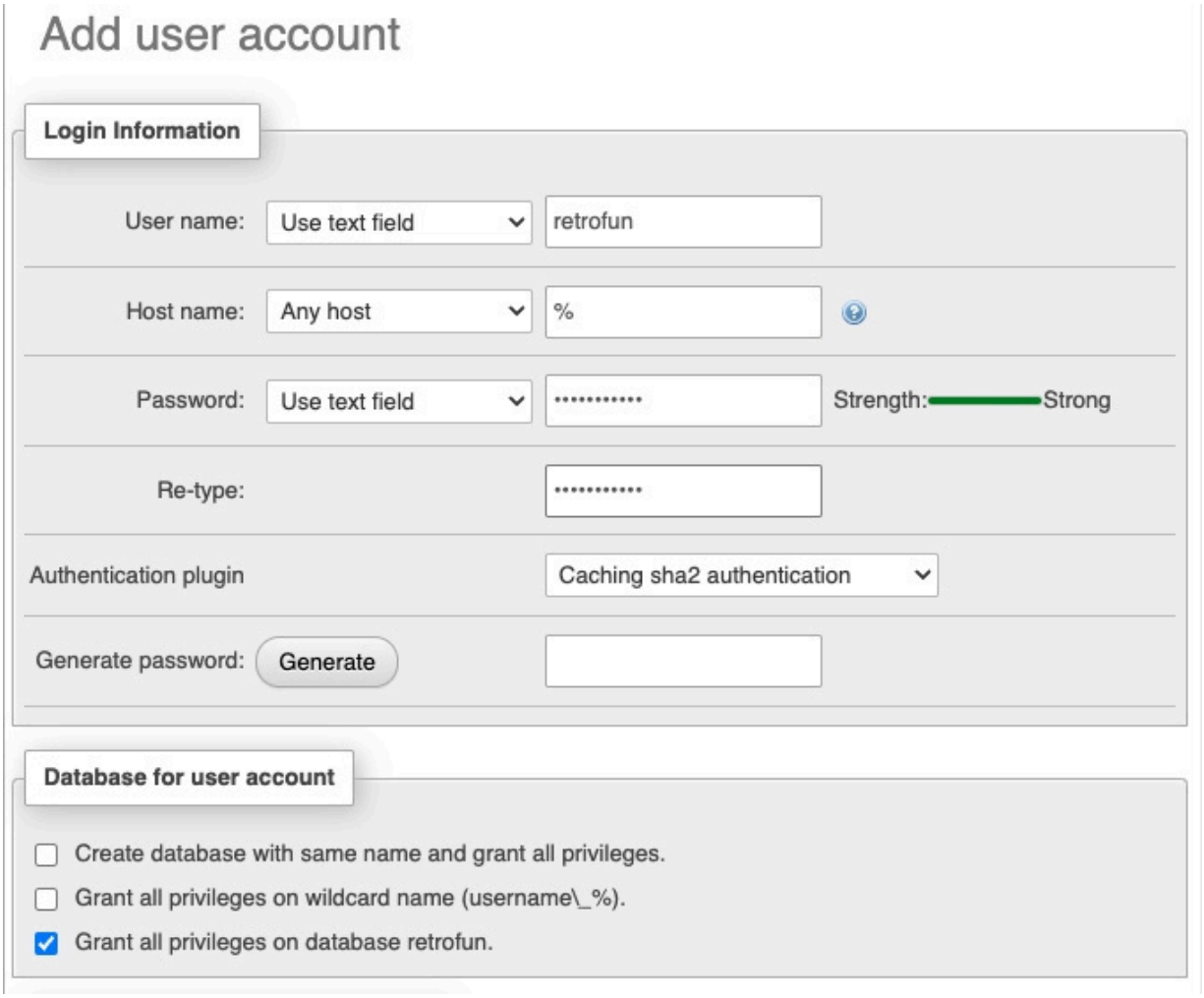
- Server: **db**
- Username: **root**
- Password: the root password that you entered in the *docker-compose.yml* file

Once you access the phpMyAdmin interface, click on the "Databases" tab. Near the top you should see the "Create Database" section.

Enter a name for the new database, such as **retrofun** and click the "Create" button.

A good practice when creating a new database is to also define a user specifically assigned to it. Using the root database user for day-to-day operations is too risky, because this account is too powerful and should only be used for important administration tasks.

Click on the "Privileges" tab for the new database. Near the bottom of the page there is a section titled "New" with an "Add user account" link. Click it to create a new user.



For the username you can choose any name that you like, but a naming convention that I find useful is to use the same name for the database and the user, so in this case it would be **retrofun**. Leave the host set to "%", then enter a password for the new user.

Confirm that the "Grant all privileges on database retrofun" option is enabled, and then scroll all the way to the bottom of the page and click the "Go" button to create the user. This user will have full access to the database, but it will not be able to access or create other databases, which is a good security principle to follow.

From now on, you can log in to phpMyAdmin using the user you just created, and your view of the database server will be constrained to only what's relevant to manage this particular database.

If you'd like to stop the MySQL server, you can do so with this command, issued from the directory in which you have your *docker-compose.yml* file:

```
$ docker-compose down
```

To start the server again, repeat the "up" command as before:

```
$ docker-compose up -d
```

Stopping and restarting the server as shown above does not cause any data loss.

MySQL Client

To access your MySQL database you have to install a Python client, sometimes also called driver. There are several MySQL drivers for Python that can be used here, so as before, you should use your favorite if you have one.

If you need a recommendation, my driver of choice is pymysql, which you can install into your Python virtual environment as follows:

```
(venv) $ pip install pymysql cryptography
```

The cryptography package installed above is an optional dependency of pymysql that is needed to perform authentication against the MySQL database.

Congratulations! You now have a complete set up, including a blank MySQL database that is ready to be used. If you followed the installation procedure described above, the connection settings for your database are:

- Hostname: localhost (but use db as hostname to connect from the phpMyAdmin container)
- Port: 3306
- Database: retrofun
- Username: retrofun

- Password: the password that you selected for the user
- Python driver: pymysql

PostgreSQL Database Set Up

PostgreSQL (often shortened to Postgres) is yet another major open-source relational database system, similar to MySQL in the sense that it also requires separate server and client.

PostgreSQL Server

If you have access to a PostgreSQL server then you can create a database and user to use with this book and skip to the next section to set up your client.

This section provides instructions to install PostgreSQL in your computer, along with the pgAdmin administration application. These instructions are based on Docker containers, and are compatible with the three major operating systems. The easiest way to install Docker and Docker Compose on your computer is through Docker Desktop.

Copy the following Docker Compose configuration file to a file named *docker-compose.yml* in your project directory:

```
version: '3'

services:
  db:
    image: postgres
    restart: always
    environment:
      POSTGRES_PASSWORD: changethis!
    ports:
      - "5432:5432"
    volumes:
      - db-data:/var/lib/postgresql/data
  admin:
    image: dpage/pgadmin4
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: changethis!
    ports:
      - 8080:80
    volumes:
      - admin-data:/var/lib/pgadmin
volumes:
```

```
db-data:
admin-data:
```

This configuration file defines a service called db with the PostgreSQL server running on port 5432 of your computer, and a second service called admin that runs pgAdmin on port 8080. Both services require storage, so two volumes are also created for them. Using separate volumes for storage allows the data to persist if the containers are stopped and restarted.

There are three lines in the above configuration that need to be reviewed and edited:

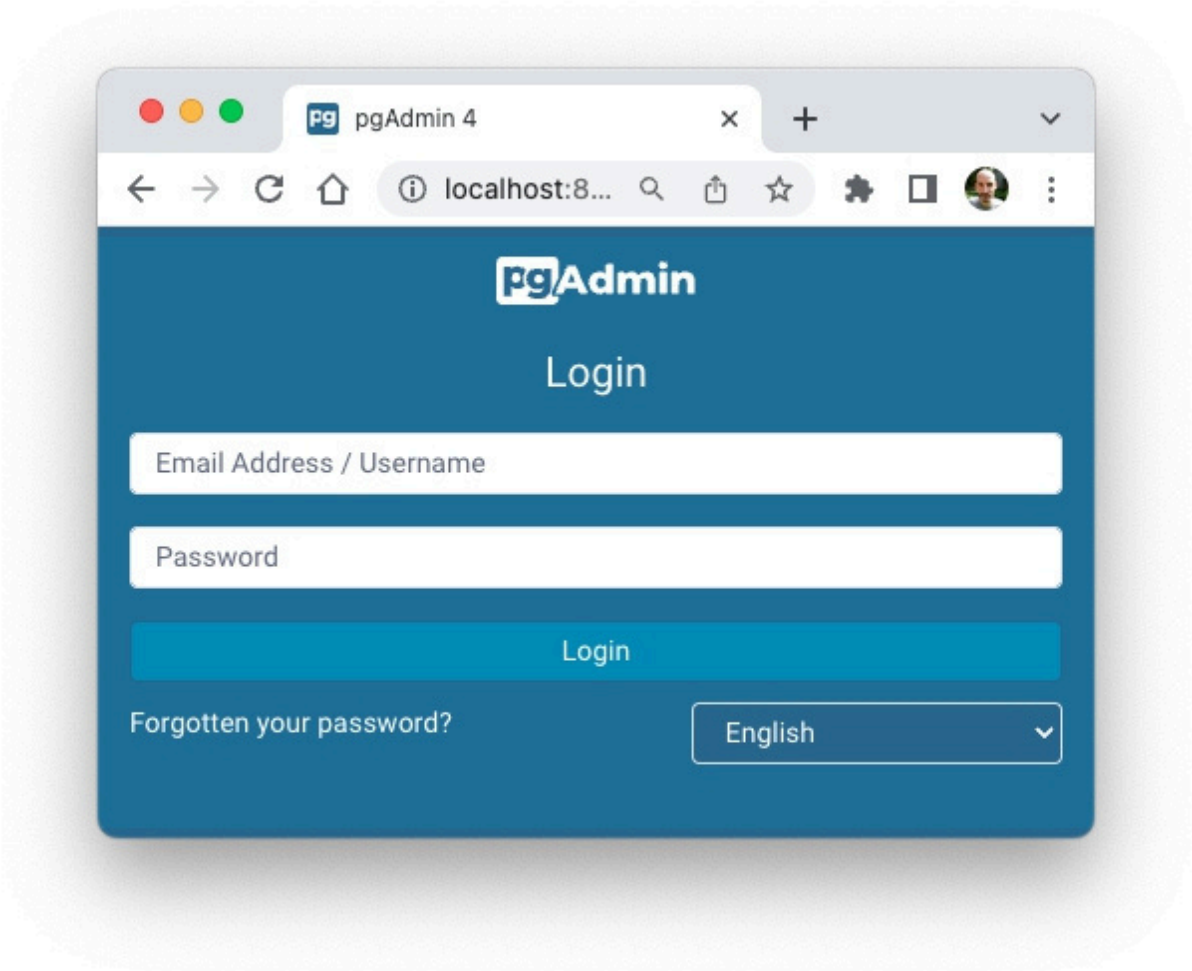
- Change POSTGRES_PASSWORD to the PostgreSQL administrator password of your liking.
- Change PGADMIN_DEFAULT_EMAIL to your own email address (used only to log in).
- Change PGADMIN_DEFAULT_PASSWORD to the pgAdmin administrator password of your liking.

Once you have the *docker-compose.yml* file ready, you can start the services with the following command:

```
$ docker-compose up -d
```

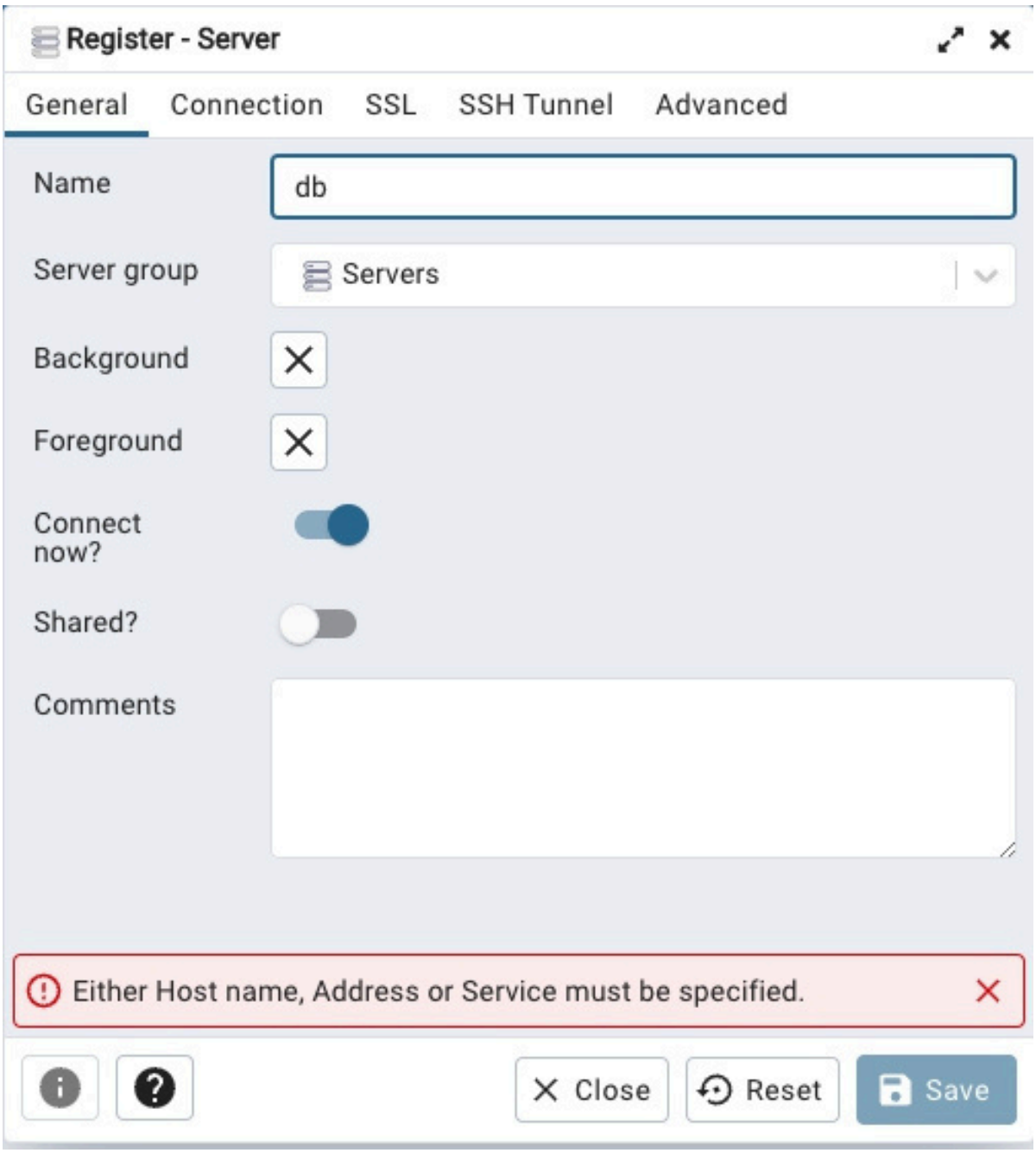
The first time you run this command Docker will have to download the Docker images for PostgreSQL and pgAdmin, so that may take a while. Once these images are downloaded, starting the services should take just a few seconds.

After the above command completes, give your computer a minute or two to get everything started and then connect to pgAdmin by typing *http://localhost:8080* on the address bar of your web browser.

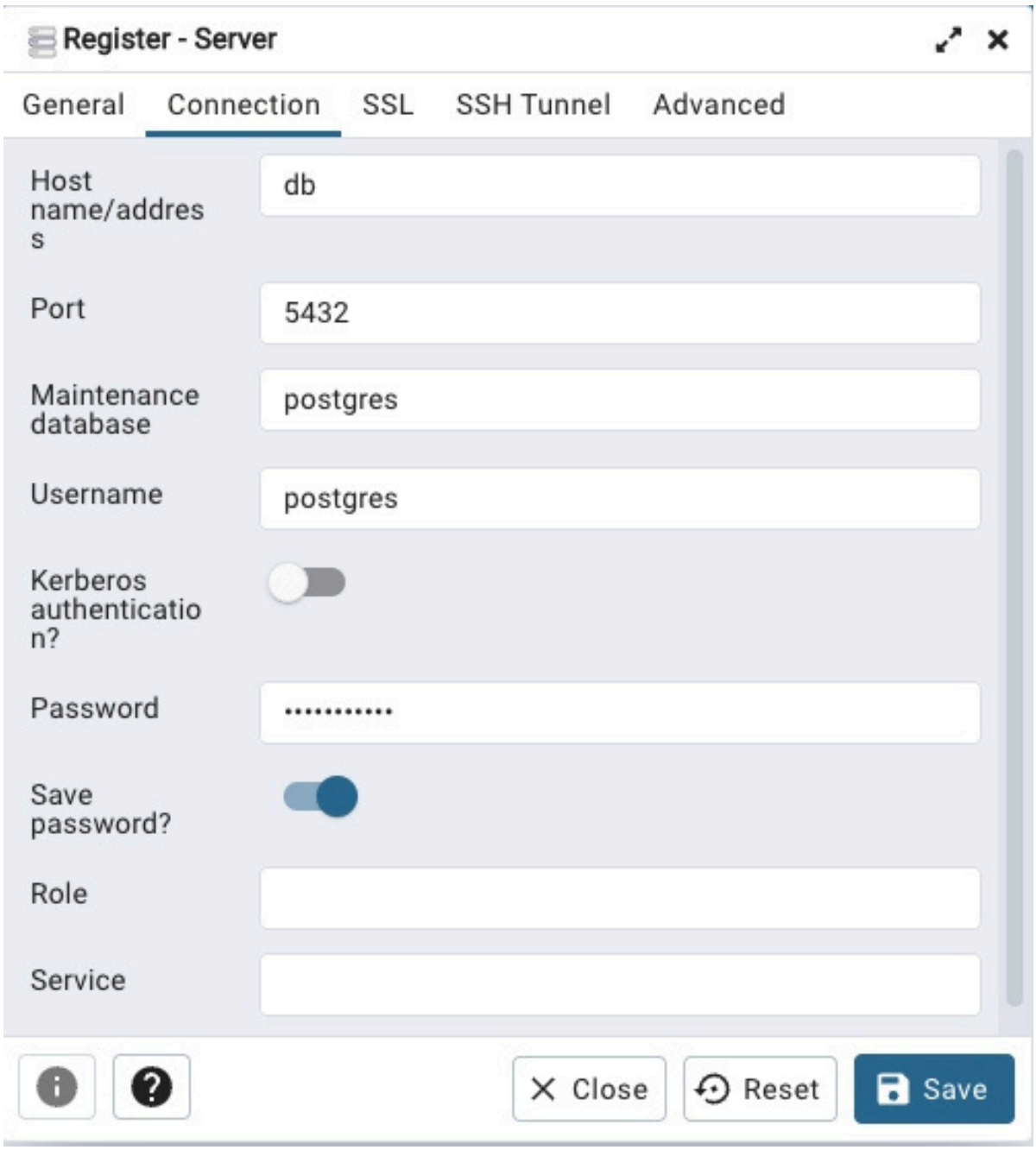


You can log in to the administration interface with the email and password that you selected for the PGADMIN_DEFAULT_EMAIL and PGADMIN_DEFAULT_PASSWORD settings in your *docker-compose.yml* configuration file.

The first task is to tell pgAdmin about the PostgreSQL server. Click the "Add New Server" icon to do this. In the "General" tab, enter a name for the server such as **db** in the "Name" field.



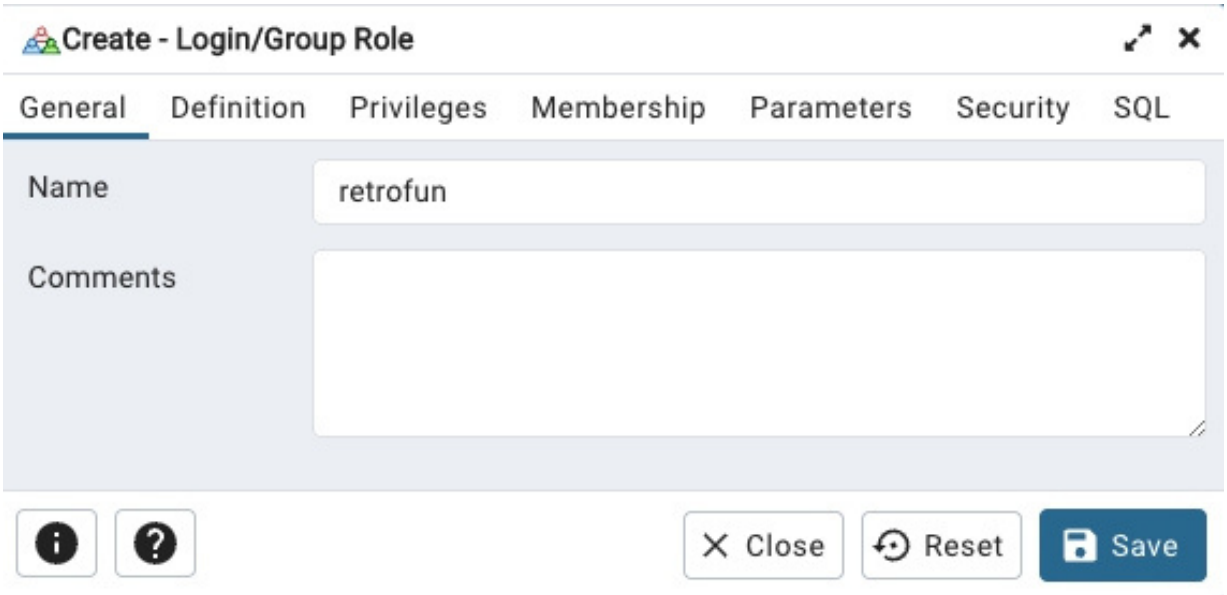
Then in the "Connection" tab, set "Host name" to **db**, which is the name of the PostgreSQL service as defined in the Docker Compose configuration. Leave the "Port" and "Maintenance Database" settings with their default values. Change "Username" to **postgres**, and write the password that you entered for the POSTGRES_PASSWORD setting in the "Password" field. You can enable the "Save password?" option if you don't want to have to re-enter the password in the future.



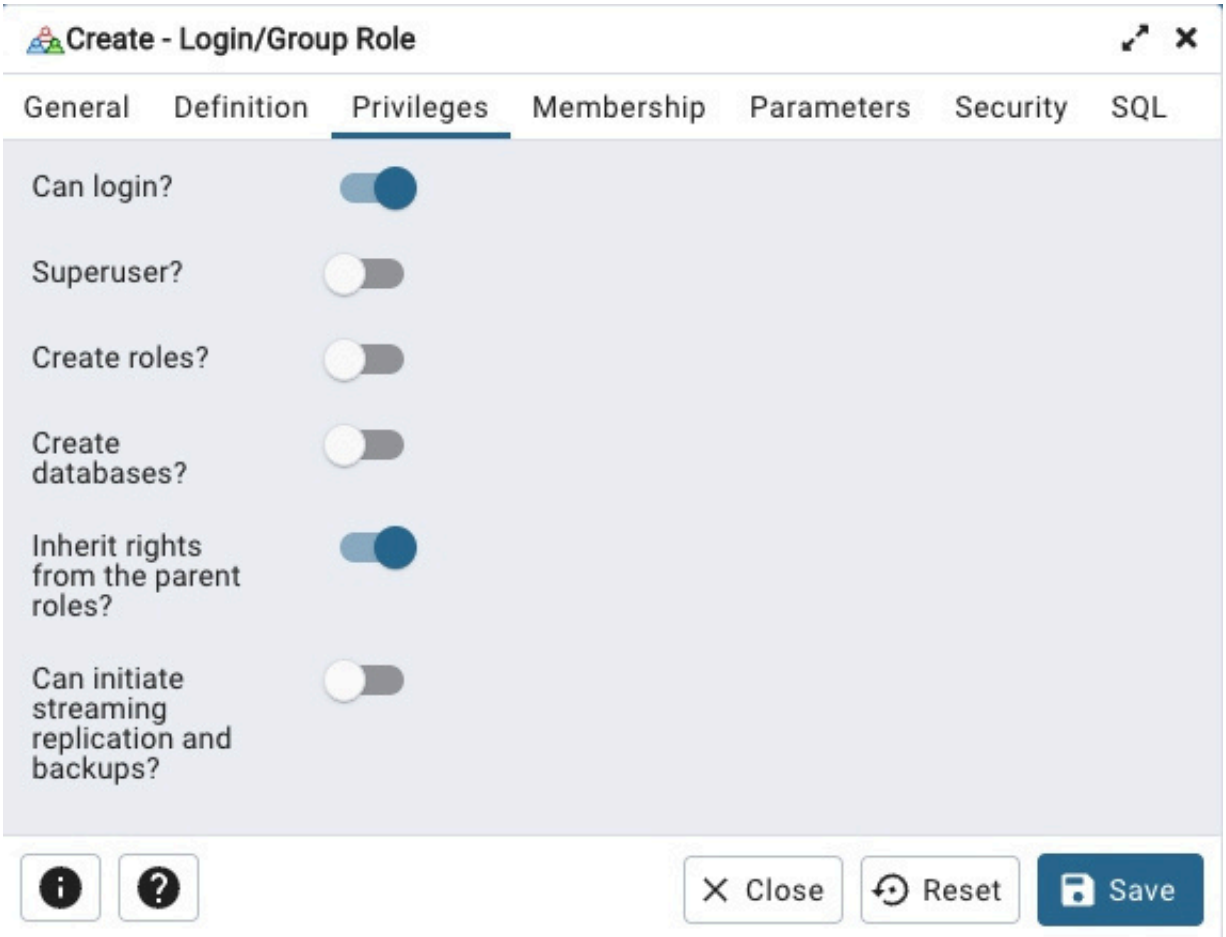
After you click the "Save" button, pgAdmin will add the server to the left sidebar, and will start showing you live statistics about its operation.

The next step is to create a brand-new database that you can use to run the examples in this book. As with MySQL, it is a good practice to create a dedicated user for each database. To create the user, right-click on the db name in the sidebar and select "Create", and then "Login/Group Role...".

In the "General" tab, enter a name for the new user, such as **retrofun**.



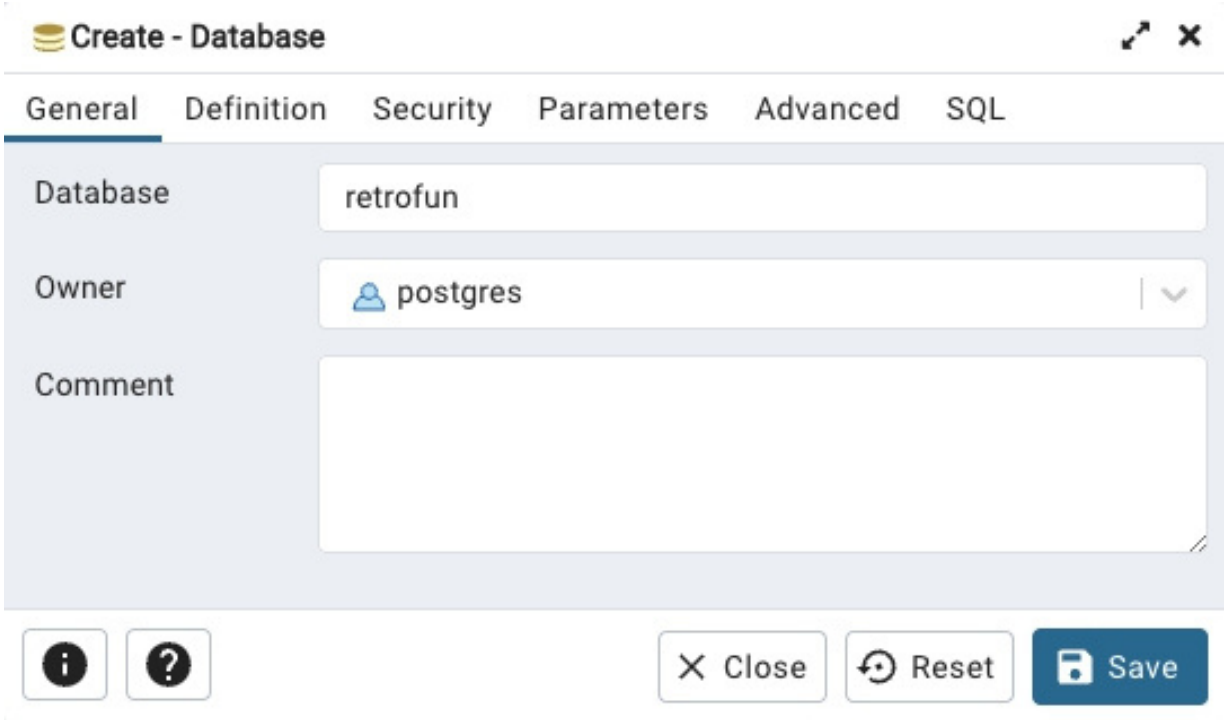
Switch to the "Definition" tab, and enter a password for the user. Then switch to the "Privileges" tab.



This user should have the "Can login?" and "Inherit rights from the parent roles?" options enabled. To increase security it is best to have all other privileges disabled, as they are not needed.

Click the "Save" button to add the user.

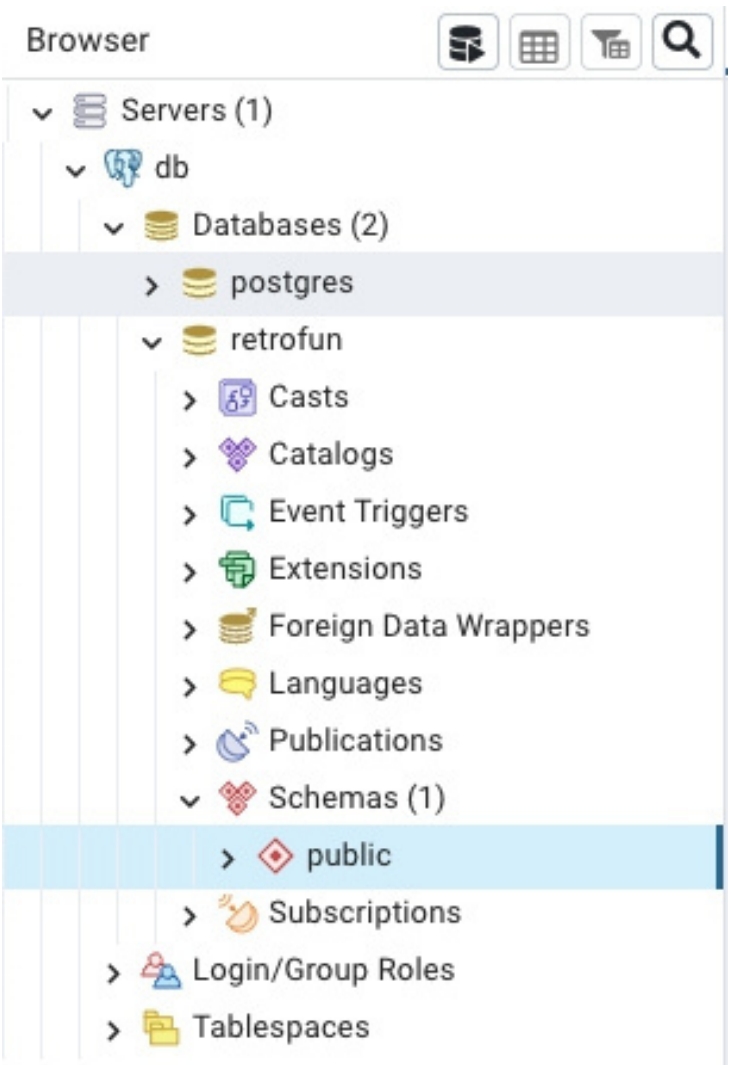
Then right-click on the database on the left once again, then select "Create", and then pick "Database...".



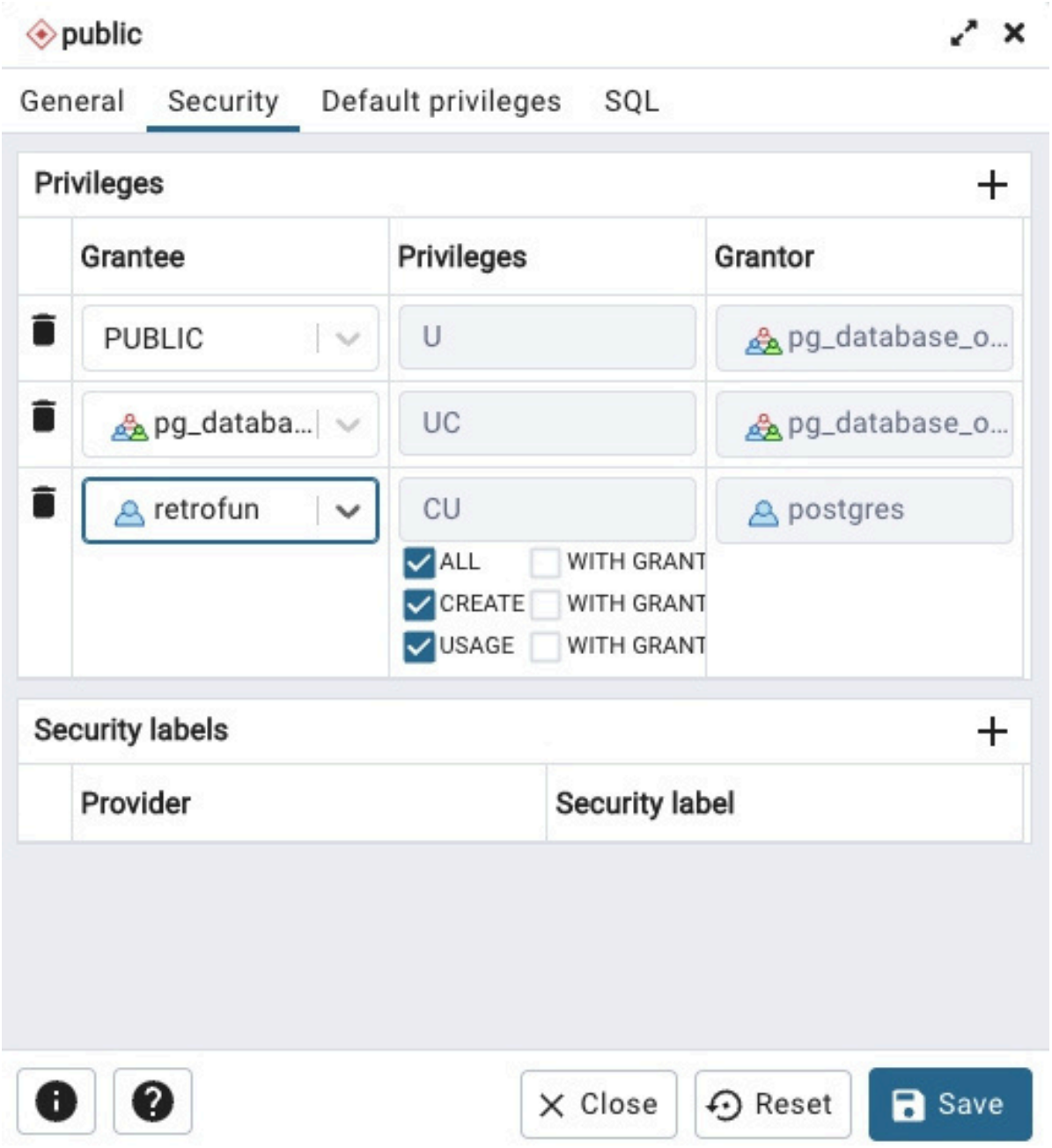
Give the new database a name, such as also **retrofun**. Naming the user and the database the same is a naming convention that I find convenient, since each user will be dedicated to only one database. The owner of the database should be the postgres user, which is the administrator.

Click "Save" to create your new database.

The next step is to configure the privileges of the retrofun user so that it can have full access to the new database. In the left sidebar, expand the tree view starting from the db server and continuing on to "Databases", the retrofun database, "Schemas", and finally "public".



Right-click on the `public` schema and select "Properties...". Then select the "Security" tab.



Click the "+" in the "Privileges" table to add a new entry. Under "Grantee", select the retrofun user. In the "Privileges" column check the "ALL" option to give the user full access to the schema.

Click "Save" to store the new privileges.

To stop the PostgreSQL server you can issue the following command from the directory in which you have your *docker-compose.yml* file:

```
$ docker-compose down
```

To start the server again, repeat the "up" command as before:

```
$ docker-compose up -d
```

Thanks to the data being stored in standalone volumes, you can freely stop and restart the server without losing any data.

PostgreSQL Client

The final step is to install a PostgreSQL driver for Python. SQLAlchemy supports a few PostgreSQL drivers, and you can choose any of them if you have a preference.

A driver that is extremely popular and has proven to be very stable is psycopg2, which you can install with this command:

```
(venv) $ pip install psycopg2-binary
```

To connect to your database from Python you will later need to know the connection details. If you followed the instructions above, then these are:

- Hostname: localhost (but use db as hostname to connect from the pgAdmin container)
- Port: 5432
- Database: retrofun
- Username: retrofun
- Password: the password that you selected for the user
- Python driver: psycopg2

Database Connection URLs

When using SQLAlchemy, a database to connect to is represented by a URL that has the following structure:

```
{dialect}{+driver}://{username}:{password}@{hostname}:{port}/{database}
```

URLs for MySQL and PostgreSQL are built using `mysql` or `postgresql` as dialect respectively, plus the connection details for your database.

The following examples assume that the user password is `my-password`:

```
# MySQL with pymysql
url = 'mysql+pymysql://retrofun:my-password@localhost:3306/retrofun'

# PostgreSQL with psycopg2
url = 'postgresql+psycopg2://retrofun:my-password@localhost:5432/retrofun'
```

Database URLs for SQLite are a bit different, because this is an in-process database without the concept of users or servers. For this database, the dialect name is sqlite and the driver can be omitted. The username, password, hostname and port are also omitted, since they do not have any meaning for this database. Finally, instead of a database name, a path to the database file is given.

The following examples show some possible database URLs for a SQLite database named retrofun.sqlite:

```
# database file in the current directory
url = 'sqlite:///retrofun.sqlite'

# database file in /home/miguel/retrofun directory
url = 'sqlite:////home/miguel/retrofun/retrofun.sqlite'

# database file in C:\users\miguel\retrofun directory (Microsoft Windows)
url = 'sqlite:///c:\\users\\miguel\\retrofun\\retrofun.sqlite'
```

If you look at these URLs carefully, you may think that they have too many forward slashes right after the sqlite: prefix, but these are all correct.

The first example uses a relative location (the current directory) for the database file. In this URL, the first two forward slashes are part of the sqlite:// URL prefix, and the third slash is the one that comes after the username, password, hostname and port, only in this case these four are empty so only the slash separator needs to be included.

In the second example there are four forward slashes after the dialect and driver. The first three slashes have the same purpose as in the first example. The fourth slash is the start of an absolute path for the SQLite database file, which in this example is */home/miguel/retrofun/retrofun.sqlite*.

The third and final example shows how an absolute path can be given when using the Microsoft Windows operating system. Here what follows the three forward slashes is an absolute path that starts with a disk drive and uses backslashes as path component separators. Python strings need the backslash character to be escaped by entering a second backslash.

The SQLite database provides one additional option: an in-memory database. This is useful for temporary databases, such as those used in unit tests. With an in-memory database, all the data is kept in the memory of the process, without persistence. This option is not going to be used in this book, but it is useful to be aware of it.

```
url = 'sqlite://'
```

In all the examples in this book, the database URL will be configured externally, in an environment variable named DATABASE_URL. To avoid having to set this variable in every shell session, create a file named *.env* (a dot followed by *env*, often called a "dotenv" file), open it in your text editor, and write the database URL that you would like to use in it as follows:

```
DATABASE_URL=sqlite:///retrofun.sqlite
```

The above example configures a SQLite database named *retrofun.sqlite* in the current directory.

The python-dotenv package allows an application to read variables from a *.env* file. Install it with pip:

```
(venv) $ pip install python-dotenv
```

Below you can see an example of how to read the DATABASE_URL variable from a Python program. Copy this code to a file named *db.py* in the project directory to try it out on your computer.

Listing 1 *db.py*: Display the database URL

```
import os
from dotenv import load_dotenv

load_dotenv()


print('Database URL:', os.environ['DATABASE_URL'])
```

Run this example to ensure that you have configured your database URL correctly:

```
(venv) $ python db.py
Database URL: sqlite:///retrofun.sqlite
```

Database Tables

This chapter provides an overview of the most basic usage of the SQLAlchemy library to create, update and query database tables.

SQLAlchemy Core and SQLAlchemy ORM

The SQLAlchemy library is divided into two modules called Core and ORM (short for Object-Relational Mapping).

The Core module contains the database integration logic for all the supported database dialects, a collection of classes to describe database tables, and a fairly sophisticated system for generating SQL statements using Python language constructs.

The ORM module introduces a layer of abstraction between the Python application and the database that allows many database operations to be automatically derived from actions performed on Python objects.

An application can choose to use SQLAlchemy Core exclusively, or it can combine elements from Core and ORM. In this book you will learn to use a combined approach.

The Database Engine

SQLAlchemy uses "engine" objects to manage connections to a database, both for Core and ORM applications. The create_engine() function constructs an engine given a database URL. Below you can see an updated version of the *db.py* file you created in the previous chapter, showing how to create an engine object from a DATABASE_URL environment variable imported from the *.env* file:

Listing 2 *db.py*: Create a SQLAlchemy engine object

```
import os
from dotenv import load_dotenv
from sqlalchemy import create_engine

load_dotenv()

engine = create_engine(os.environ['DATABASE_URL'])
```

The create_engine() function can be passed additional keyword arguments to configure the engine. Interesting options include:

- echo=True, to have SQLAlchemy log every SQL statement issued to the database. This is a very useful option when debugging.
- pool_size=<N>, to specify a custom size for the connection pool SQLAlchemy maintains (the default is a pool size of up to 5 simultaneous connections).
- max_overflow=<N>, the maximum number of connections above the pool size that can be created during usage spikes (the default is 10).
- future=True, to tell SQLAlchemy 1.4 to use the newer 2.0 based APIs.

Consult the documentation for create_engine() to see the complete list of options that are available.

Models

When using the ORM module, database tables are defined in the application as Python classes. The application must create a parent class for all these classes, where settings that are common to all the tables can be configured. This parent class, which SQLAlchemy calls the *declarative base* class, is often named Model, or in some cases Base. The collection of subclasses of the Model class represent the structure or schema of the database, and are generally referred to as the "models" of the application.

The Model class must inherit from SQLAlchemy's DeclarativeBase class.

Here is an updated version of *db.py* that defines Model as an empty class, without any custom settings:

Listing 3 *db.py*: Create a declarative base class

```
import os
from dotenv import load_dotenv
from sqlalchemy import create_engine
from sqlalchemy.orm import DeclarativeBase


class Model(DeclarativeBase):
    pass


load_dotenv()

engine = create_engine(os.environ['DATABASE_URL'])
```

To help keep things nicely organized, the models for the application you are going to build with the help of this book are going to be stored in their own file, which will be called *models.py* file. The next code example shows a first implementation of a model for a products database table:

Listing 4 *models.py*: Product model class

```
from sqlalchemy import String
from sqlalchemy.orm import Mapped, mapped_column
from db import Model


class Product(Model):
    __tablename__ = 'products'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(64))
    manufacturer: Mapped[str] = mapped_column(String(64))
    year: Mapped[int]
    country: Mapped[str] = mapped_column(String(32))
    cpu: Mapped[str] = mapped_column(String(32))

    def __repr__(self):
        return f'Product({self.id}, "{self.name}")'
```

If you have used older versions of SQLAlchemy, you may find the above model definition to be significantly different. In version 2.0, SQLAlchemy introduced an integration with Python type hints, and that is the reason why the currently recommended syntax for column definition has changed from older releases.

As indicated above, all application model classes must inherit from the Model declarative base class, which is imported from *db.py*.

Model subclasses are configured using class attributes. The __tablename__ attribute defines the name of the database table the class represents. A very common naming convention for database tables is to use the plural form of

the entity in lowercase, so in this case the table is given the products name. This contrasts with the convention used for the model class names, which prefers the singular form in camel case.

The remaining attributes defined in the class represent the columns of the table. The Mapped[t] type declaration is used to define each column, with t being the Python type assigned to the column, such as int, str, or datetime. For simple columns such as year above, this is all that is necessary. If the column needs to be given additional options, it is assigned to a mapped_column() constructor that provides those options.

In the Product model defined above, an option is used to identify the id column as a primary key, which means that the values in this column must uniquely identify each item stored in the table. Without any additional configuration, SQLAlchemy configures integer primary key columns with auto-incrementing numbers starting from 1. You will later learn other ways to define primary keys.

The remaining columns describe the attributes that products have. For columns that are of type str, a maximum length is added with a supplementary String() option. Not all databases require a length to be given for string columns, but it is best to always include a length just in case.

The __repr__() method included in this class is a special method that tells Python how an object of this class should be printed. Adding this method is optional, but it is useful as an aid when debugging or when trying things out in a Python shell, which is something you will often while working with this book.

To create an instance of a model class, a standard class constructor is used, passing the values for the model's attributes as keyword arguments. For example:

```
c64 = Product(name='Commodore 64', manufacturer='Commodore')
```

The above example initializes a new Product instance. SQLAlchemy provides a default constructor for all model classes that accepts value

assignments for its columns. In this particular example, all the attributes of c64 except name and manufacturer will be set to None, because they were not assigned a value when the object was created. Even though this object is a model instance, at this point it is just a plain Python object that is not stored or associated with any database.

Note

The concept of model classes is available only for applications that use the ORM module. When using Core, instances of the Table class are used to represent database tables.

Database Metadata

SQLAlchemy maintains the definitions of all the tables that make up a database in an object of class MetaData. For convenience, it initializes the declarative base class with a metadata attribute that has a default MetaData object. For the Model class, the metadata instance is available as Model.metadata. When a model class such as Product is defined, SQLAlchemy creates a corresponding table definition in this attribute.

The default MetaData configuration has one important limitation that is bound to cause problems when projects reach certain size or level of complexity. This is related to the naming_convention option, which tells SQLAlchemy how to name indexes and constraints it creates on a database. You will learn what these are later in this chapter, but for now, just consider that in the same way as tables, indexes and constraints need to have a name.

The default naming convention used in the MetaData object provides a naming rule for indexes, but not for constraints, so SQLAlchemy initializes all constraints without an explicit name, which results in them having arbitrary names chosen by the database. This is a problem if at some point a constraint needs to be modified or deleted, since SQLAlchemy wouldn't immediately know how to address the constraint by its name. To avoid this potential complication down the road, the Model declarative base can be initialized with a more complete set of naming conventions, as shown below:

Listing 5 *db.py*: Configure naming conventions for indexes and constraints

```
import os
from dotenv import load_dotenv
from sqlalchemy import create_engine, MetaData
from sqlalchemy.orm import DeclarativeBase



class Model(DeclarativeBase):
    metadata = MetaData(naming_convention={
        "ix": "ix_%(column_0_label)s",
        "uq": "uq_%(table_name)s_%(column_0_name)s",
        "ck": "ck_%(table_name)s_%(constraint_name)s",
        "fk": "fk_%(table_name)s_%(column_0_name)s_%(referred_table_name)s",
        "pk": "pk_%(table_name)s",
    })


load_dotenv()

engine = create_engine(os.environ['DATABASE_URL'])
```

The MetaData object has a create_all() method that has significant importance, as it creates all the database tables associated with the defined models. Here is an example usage:

```
Model.metadata.create_all(engine)
```

Here the create_all() method will issue SQL statements to the database represented by engine to create the database tables referenced by all the models. Following the code examples from previous sections, this call would create a products table, which is defined by the Product model.

An important limitation of create_all() is that it only creates tables that don't already exist in the database, which means that when a model class is changed, this method cannot be used to transfer the change to the corresponding database table.

A workaround that can be used to modify an existing table is to remove the old and outdated version of the table from the database before calling create_all() again. As a convenience, the MetaData object also has a drop_all() method, which removes all the tables from the database. The following example refreshes all the tables to their latest definitions:

```
Model.metadata.drop_all(engine)
Model.metadata.create_all(engine)
```

Unfortunately updating a database in this way is only practical for small tests or while prototyping, because drop_all() not only deletes the tables

but also all the data stored in them. You will later learn how to use Alembic to manage updates to the database in a much more effective way through *migration scripts*.

Note

When using Core, the database metadata object must be manually created by the application.

Sessions

Another important entity in ORM-based applications is the *session*. A session object maintains the list of new, read, modified and deleted model instances.

Changes that accumulate in a session are passed on to the database in the context of a database transaction when the session is *flushed*, which is an operation that in most cases is automatically issued by SQLAlchemy when it is needed. A flush operation writes the changes to the database, but keeps the database transaction open.

When the session is *committed*, the corresponding database transaction is committed as well, causing all the changes to be permanently written to the database.

Database transactions are one of the most important benefits of relational databases, designed to maintain the integrity of the data. The changes that are committed as part of a transaction are written as an atomic operation, so errors or unexpected interruptions will never result in partial or incomplete data being written. If an error or failure occurs while a session is active, a *rollback* operation on the session will roll the transaction back, and all the changes made up until that point in that session will be undone.

The following example shows how the c64 object created in the previous section can be added to a database session and committed:

```
from sqlalchemy.orm import Session

with Session(engine) as session:
```

```
                        try:
    session.add(c64)
    session.commit()
except:
    session.rollback()
    raise    print(c64)
```

The best way to manage a database session is to create it as a context manager. This ensures that the session is properly closed and disposed of at the end.

The session is initialized with the engine object. As with the engine, the future=True option can be used with SQLAlchemy 1.4 to configure the session to use 2.0 style APIs.

Session objects are designed to accumulate changes until they are either committed or rolled back. The add() method is used to insert a new object into the session. The try/except block ensures that the session is always committed or rolled back. If an error occurs while the session is being used or committed, the except section does the roll back, guaranteeing that all the partial changes that could not be committed are discarded.

As mentioned earlier, SQLAlchemy configures integer primary key columns to be auto-incrementing by default. When the session is flushed, which usually happens during a commit() call, the database will assign the next available number to the id attribute of the new item, or 1 if this is the first entry added. Any other attributes in the model object that were not set will be recorded with a NULL value in the database.

The print statement after the try/except block is designed to show the newly assigned id value, as implemented in the __repr__() method of the `Product class`.

SQLAlchemy provides more concise ways to work with sessions. A normal application will have lots of places in which sessions need to be created, and it may be inconvenient to have to pass the engine and other options in every one of them. The sessionmaker factory function provides a way to create a customized Session class that has all the options incorporated:

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(engine)
```

```
with Session() as session:
    # ...
```

Having to wrap all the database logic in a try/except block can also become very tedious. In the next example, an inner context manager started with the begin() method replaces the exception handling:

```
with Session() as session:
    with session.begin():
        session.add(c64)
    print(c64)
```

The context manager created by session.begin() implements the try/except logic internally. It automatically commits the session at the end, and if there are any raised exceptions, it rolls the session back as before. In all cases the session is properly closed by the outer context manager.

Given that the application will need easy access to session objects, it makes sense to also define the base Session class in *db.py*. Here is the complete version of this file that you will use through most of this book:

Listing 6 *db.py*: Create a session class

```
import os
from dotenv import load_dotenv
from sqlalchemy import create_engine, MetaData
from sqlalchemy.orm import DeclarativeBase, sessionmaker


class Model(DeclarativeBase):
    metadata = MetaData(naming_convention={
        "ix": "ix_%(column_0_label)s",
        "uq": "uq_%(table_name)s_%(column_0_name)s",
        "ck": "ck_%(table_name)s_%(constraint_name)s",
        "fk": "fk_%(table_name)s_%(column_0_name)s_%(referred_table_name)s",
        "pk": "pk_%(table_name)s",
    })


load_dotenv()

engine = create_engine(os.environ['DATABASE_URL'])
Session = sessionmaker(engine)
```

Note

Session objects are available only for applications that use the ORM module. When using Core, database transactions have to be manually managed by issuing appropriate SQL statements through an engine connection.

A First SQLALchemy Application

The previous sections in this chapter provide an overview of the most important components of a SQLAlchemy ORM application, which are:

- the engine
- the models
- the database metadata
- the session

Now it is time to see how these components are used in a complete application.

As mentioned in the Preface, all the examples in this book are designed around the needs of a made-up company called RetroFun that sells vintage home computers. The first complete script combines many of the snippets presented earlier into an application that imports the catalog of products offered by RetroFun from a CSV file and populates a products database table with them.

Copy the code below into a file named *import_products.py*, in the same directory as *db.py* and *models.py*.

Listing 7 *import_products.py*: Import products from CSV file

```
import csv
from db import Model, Session, engine
from models import Product


def main():
    Model.metadata.drop_all(engine)  # warning: this deletes all data!
    Model.metadata.create_all(engine)

    with Session() as session:
        with session.begin():
            with open('products.csv') as f:
                reader = csv.DictReader(f)
                for row in reader:
                    row['year'] = int(row['year'])
                    product = Product(**row)
                    session.add(product)
```

```
if __name__ == '__main__':
    main()
```

The application imports the Model and Session classes and the engine instance from *db.py*. It also imports the Product model class from *models.py*.

The main() function is where all the database operations are issued. First the drop_all() and create_all() methods of the metadata object are invoked. These ensure that the products table is recreated from scratch to match the definitions of the Product model.

Next, a database session is started using the double context manager method, so that all the changes made in the session are automatically committed atomically at the end.

What's left is the importing logic, which starts with a third context manager dedicated to opening the CSV file that contains the data to import. Using a context manager when opening a file is very convenient, as this ensures that the file is automatically closed at the end.

Python includes a csv module in its standard library. The DictReader class from this module is used to read rows from the CSV file one at a time in a for-loop. Each row is returned as a dictionary, where the keys are the column names, which are given in the first line of the CSV file. The values for all columns are returned as strings. The CSV column names were carefully chosen to match the names of the attributes of the Product model, but the year field has to be manually converted to an integer to match the Product model definition.

A Product model instance is created directly by passing the contents of each row dictionary as keyword arguments. Each of these product model instances is then added to the database session.

When the for-loop that iterates over the rows of the CSV file exits, the session.begin() context manager will flush and commit the session, and the outer context manager will then close the session. The flush operation will write all the products imported from the CSV file to a database transaction, and the commit operation will then make these changes

permanent. If an error occurs during this process, the session will be rolled back and nothing will be written to the database.

Are you ready to try this application? Make sure you have *import_products.py*, *db.py* and *models.py* in your project directory. You should also have a *.env* file in this directory with a DATABASE_URL variable configured with an active database according to the instructions in Chapter 1.

You will also need to have a copy of a file called *products.csv* with all the product data in the project directory. This file can be downloaded from the book's ~~GitHub repository~~:

Make sure your Python virtual environment is activated, and then run the script as follows:

```
(venv) $ python import_products.py
```

There shouldn't be any output, but when the script ends you should have a populated products table in your database. If you have a database administration tool to inspect your database, feel free to review the new table with it.

Queries

With a populated table in your database, this is the perfect time to begin to learn how to issue some queries. If you have used SQLAlchemy in the past, be aware that starting with version 1.4, SQLAlchemy introduced significant changes in how ORM queries are constructed. The legacy query implementation you may be familiar with is still available, but in this book only the new query style is used. The SQLAlchemy documentation refers to the new query style as the "2.0 query style", but this style of queries can also be used in the 1.4 releases when the engine and session objects are created with the future=True option.

The easiest way to work with your new database is to open a Python shell, from where you can issue queries interactively. You can import the engine object and the model and session classes as follows:

```
>>> from db import Session
>>> from models import Product
```

You can then create a session:

```
>>> session = Session()
```

This is a different way of creating a session that does not use a context manager. The context manager approach to sessions is very convenient in an application, but it gets in the way when working interactively in the Python prompt, so a direct creation is better in this context.

Query Definition

Relational databases use the SELECT keyword to implement queries. SQLAlchemy provides a select() function with similar functionality. The simplest query is the one that returns all the elements in a table. Here is how to define a query that retrieves all the products stored in the database:

```
>>> from sqlalchemy import select
>>> q = select(Product)
```

The select() function takes as arguments the items that need to be retrieved. When a model class is given as an argument, SQLAlchemy ORM retrieves all the attributes of the model and transparently returns Python objects.

Sometimes it is useful to see what is the SQL code that will be sent to the database when this query is executed. The SQL code associated with a SQLAlchemy query object can be seen when the query is printed:

```
>>> print(q)
SELECT products.id, products.name, products.manufacturer, products.year,
products.country, products.cpu
FROM products
```

Here you can see how the `Product` class passed in the select() function was transformed into a `SELECT` statement that retrieves all the attributes of the table.

Query Execution

After a query object is created, it has to be given to the session, which will send it to the database driver to execute through a connection maintained by the engine. The most generic way to do this is to use the execute() method of the session:

```
>>> r = session.execute(q)
>>> list(r)
[(Product(1, "Acorn Atom"),), (Product(2, "BBC Micro"),), ..., (Product(149, "GEM 1000"),)]
```

The execute() method returns a results object. This is an iterable object that retrieves the query results. In the above exercise the iterable was converted to a list to force all the results to be retrieved and displayed. Your terminal will show a long list of products.

The all() method on the results object returned by execute() achieves the same effect as the conversion done by list():

```
>>> session.execute(q).all()
[(Product(1, "Acorn Atom"),), (Product(2, "BBC Micro"),), ..., (Product(149, "GEM 1000"),)]
```

In addition to all(), the results object has other methods that retrieve the first result of a query, which is a very common need:

- first() returns the first result row, or None if there are no results. If there are any more rows in the result set, they are discarded.
- one() returns the first and only result. If there are zero or more than one result rows, an exception is raised.
- one_or_none() returns the first and only result, or None if there are no results. If there are two or more result rows, an exception is raised.

Having the results as an iterable is very efficient, as SQLAlchemy only retrieves the rows as they are needed, which means that for a query that returns a very large number of results you don't have to retrieve all the results into a list, and instead can process results as they are loaded. In an application, you would normally iterate over the results in a for-loop:

```
>>> r = session.execute(q)
>>> for row in r:
        print(row)
(Product(1, "Acorn Atom"),)
(Product(2, "BBC Micro"),)
```

```
 ...
(Product(149, "GEM 1000"),)
```

Note the structure of each result. Here is the first one, isolated from the rest:

```
(Product(1, "Acorn Atom"),)
```

This isn't a simple Product instance. SQLAlchemy returns each result as a tuple, because queries can sometimes return multiple values per row (you will see examples of this shortly). Since SQLAlchemy does not know how many results per row are expected, it always returns each row as a tuple. This query returns a single value per row, so each tuple has a single element in it.

If you know that you will be receiving a single value per row, then you can use the scalars() convenience method to execute the query:

```
>>> session.scalars(q).all()
[Product(1, "Acorn Atom"), Product(2, "BBC Micro"), ..., Product(149, "GEM 1000")]
```

With scalars(), SQLAlchemy returns a different "results" object that only iterates over the first value of each result row. If the query returned multiple values per row, then the additional values in each row are discarded. The all(), first(), one() and one_or_none() methods are also available on the results object returned by scalars().

Also as a convenience, there are some additional query execution methods that combine scalars() with first(), one() and one_or_none(). Given a query q, these are the additional methods:

- scalar(q) is the same as scalars(q).first() and returns the first value of the first result row, or None if the query has no results.
- scalar_one(q) is the same as scalars(q).one() and returns the first value of the only result row, or raises an exception if there are zero or more than one result.
- scalar_one_or_none(q) is the same as scalars(q).one_or_
  and returns the first value of the only result row or None if there are no results. It raises an exception if there are two or more results.

The next example shows a compact way to get the first value of the first result row:

```
>>> r = session.scalar(q)
>>> r
Product(1, "Acorn Atom")
```

Filters

A query that only includes a select() statement returns all available items, which is sometimes useful, but not very often. There are many situations in which an application may want to retrieve just a subset of all the items, possibly the items that fulfill some criteria.

The application can retrieve all the results as shown above and then discard the ones that aren't of interest, but this can be very inefficient, especially for very large tables. Databases are designed to perform filtering and return only the desired results in ways that are much more efficient than what the application can do on its own.

With SQLAlchemy, a filter can be added to a query object with the where() clause. The following example shows how to retrieve only products made by Commodore. Feel free to try this query out in your Python session.

```
>>> q = select(Product).where(Product.manufacturer == 'Commodore')
>>> session.scalars(q).all()
[Product(39, "PET"), Product(40, "VIC-20"), ..., Product(48, "Amiga")]
```

SQLAlchemy implements a highly sophisticated solution for defining filters that combines the class attributes of model classes with standard Python operators such as ==. The example that follows uses the >= operator in a query that retrieves all products made after 1990:

```
>>> q = select(Product).where(Product.year >= 1990)
```

Try executing this query with session.scalars() to see which products are returned.

The where() statement can be given multiple times to specify multiple conditions. The next example retrieves products made by Commodore only in 1980:

```
>>> q = (select(Product)
        .where(Product.manufacturer == 'Commodore')
        .where(Product.year == 1980))
```

Two or more filters can also be given as multiple arguments in a single where() for the same result:

```
>>> q = select(Product).where(Product.manufacturer == 'Commodore', Product.year == 1980)
```

Combining multiple filters as shown above effectively applies the AND logical operator to them. Sometimes a query may need to combine filters with the OR operator, which SQLAlchemy offers with the or_() function. The next example returns products built before 1970 or after 1990:

```
>>> from sqlalchemy import or_
>>> q = select(Product).where(or_(Product.year < 1970, Product.year > 1990))
```

Even though it is implied when passing multiple arguments to the `where()` clause, the AND logical operator can also be explicitly given using the and_() function. The NOT unary operator is also available as the not_() function.

Another very useful filter is the LIKE operator, which can be used to implement a simple search function. The following example retrieves all products that have the word Sinclair in their name:

```
>>> q = select(Product).where(Product.name.like('%Sinclair%'))
```

The like() method available on column attributes of models accepts a search pattern string and returns all results that match this pattern. The pattern defines the text to search for, expanded with a % character as a wildcard that matches zero, one or more characters, and a _ character to match just one character. Here are a few other example patterns for the like() filter:

- Sinclair% (items that start with Sinclair)
- %Sinclair (items that end with Sinclair)
- % Sinclair (items that end with a space, followed by Sinclair)

- R__% (items that start with the letter R followed by at least two more characters)
- _ (items that are one character long)

The like() function is case-sensitive. For case-insensitive searches, you can use the ilike() function instead.

A range of items can be requested with a where() clause defining two conditions for the lower and upper bounds respectively, but it is more clear to use the between() method exposed by column attributes of model classes. The example below returns products that were made in the 1970s:

```
>>> q = select(Product).where(Product.year.between(1970, 1979))
```

Have you tried looking at the SQL code generated by some of these queries? Here is how the last one looks:

```
>>> print(q)
SELECT products.id, products.name, products.manufacturer, products.year,
products.country, products.cpu
FROM products
WHERE products.year BETWEEN :year_1 AND :year_2
```

Here you can see that the literal values that are defined in query filters are not inserted in the rendered SQL. Instead, they are replaced with placeholder arguments such as the :year_1 and :year_2 above. This is a well established security practice that prevents SQL injection attacks, and SQLAlchemy implements it automatically.

For debugging purposes you may want to see the SQL query with the actual literals. While this can be insecure and should not be used to generate SQL statements intended to be executed, the following example shows how to tell SQLAlchemy to render the query along with all the literal parameters:

```
>>> print(q.compile(compile_kwargs={'literal_binds': True}))
SELECT products.id, products.name, products.manufacturer, products.year,
products.country, products.cpu
FROM products
WHERE products.year BETWEEN 1970 AND 1979
```

Order of Results

The queries above return the requested data in the order chosen by the database server, but relational databases are able to sort the results very efficiently to provide them in the order that the application finds most convenient. The order_by() method can be added to a query to specify the desired order.

The next example retrieves products alphabetically ordered by their names:

```
>>> q = select(Product).order_by(Product.name)
>>> session.scalars(q).all()
[Product(10, "464 Plus"), Product(11, "6128 Plus"), ..., Product(127, "ZX Spectrum")]
```

It is also possible to sort in reverse order by calling the desc() method on the column attribute given in the order_by() clause. The example below returns the products according to their year of manufacture, with the newest products first:

```
>>> q = select(Product).order_by(Product.year.desc())
>>> session.scalars(q).all()
[Product(6, "A7000"), Product(33, "Falcon"), ..., Product(74, "Honeywell 316")]
```

There are many situations in which a single ordering criteria is insufficient. For example, in the last example, all the computers that were built in the same year are returned in an arbitrary order. The order_by() method accepts multiple arguments, each adding a new level of sorting. The previous example can be improved with a secondary sorting by product name:

```
>>> q = select(Product).order_by(Product.year.desc(), Product.name.asc())
```

Note the asc() method, which is used to specify ascending order for the product name. Ascending order is the default, so there is no need to include this method, but sometimes it may make the query more clear when the order is explicitly stated.

Access to Individual Columns

In all the example queries so far, the requested data was entire rows out of the products table, which the SQLAlchemy ORM maps to instances of the Product model class. While querying ORM entities in this way is very

common, the select() function is very flexible and can work with more granular data as well.

For example, an application may only need to retrieve an individual column. In the next query, only the names of the products are retrieved:

```
>>> q = select(Product.name) >>>
session.scalars(q).all() ['Acorn Atom', 'BBC
Micro', ..., 'GEM 1000']
```

As discussed earlier, the select() function is not limited to retrieving a single value per result row, and it can actually request several in the same query. The following query obtains the name and manufacturer of each product:

```
>>> q = select(Product.name, Product.manufacturer)
>>> session.execute(q).all()
[('Acorn Atom', 'Acorn Computers Ltd'), ('BBC Micro', 'Acorn Computers Ltd'), ...]
```

Notice how this example has to be executed with `session.execute()` so that the pair of values in each result row are returned as a tuple.

Aggregation Functions

The select() function can also work with SQL functions that are evaluated on the retrieved data on the fly to transform it. A very useful function is count(), which replaces all the result rows with a count of how many there are. The next example finds out how many products are stored in the database:

```
>>> from sqlalchemy import func
>>> q = select(func.count(Product.id))
>>> r = session.scalar(q)
>>> r
149
```

The count() function used above reduces the list of results to a single value, and for that reason the scalar() method is used to retrieve it. In this example, using Product.id as argument to count results is arbitrary, any column attribute of the Product class can be given, and the result would be the same, because the data itself does not matter. There is an alternative form of the above query that does not require picking a random column to get a count of results:

```
>>> q = select(func.count()).select_from(Product)
>>> r = session.scalar(q)
>>> r
149
```

In this second form, the count() function is given no arguments to indicate that a count of results is desired, without specifying what data to count. When using this format, the select_from() method has to be added to configure the table to use in the query, because SQLAlchemy cannot automatically determine it from the arguments given to the select() function.

Another pair of useful SQL functions are min() and max(). The example that follows returns the first and last years in which products in the database were manufactured:

```
>>> q = select(func.min(Product.year), func.max(Product.year))
>>> r = session.execute(q)
>>> r.first()
(1969, 1995)
```

This query has to use execute() because it retrieves two values per row. The min() and max() functions reduce the list of results to a single row, so there is no point in using all() to retrieve the results as in previous examples. When it is known in advance that there is going to be one result row, the first() or one() methods are more convenient, with the latter raising an exception for queries that return anything other than a single result row.

Result Grouping

The database in its current form only has products as a first-class entity, but sometimes the application may be interested in a retrieving related data attributes such as the manufacturer. Here is an attempt to obtain a list of computer manufacturers from this table:

```
>>> q = select(Product.manufacturer).order_by(Product.manufacturer)
>>> session.scalars(q).all()
['Acorn Computers Ltd', 'Acorn Computers Ltd', ..., 'West Computer AS']
```

But of course, this has a problem. Even though the query retrieves only manufacturers, the queried table has products in it, so each result row corresponds to a product, and manufacturers that have more than one

product in the database appear multiple times. Acorn Computers Ltd, the first manufacturer when sorting alphabetically, appears as the first six results because it has six different computer models.

Whenever a database query can return duplicated results, the distinct() clause added to it tells the database to combine identical results:

```
>>> q = select(Product.manufacturer).order_by(Product.manufacturer).distinct()
>>> session.scalars(q).all()
['Acorn Computers Ltd', 'AGAT', ..., 'West Computer AS']
```

You may be tempted to combine the distinct() clause with the count() aggregation function to find out how many manufacturers exist in the database. Unfortunately, the distinct() clause does not work when using the count() aggregation function because the database evaluates the function before distinct(). When you need to count unique results, there is a distinct() method that can be called on the item being counted, inside the count() function:

```
>>> q = select(func.count(Product.manufacturer.distinct()))
>>> session.scalar(q)
76
```

Merging results with distinct() is limited because two rows of results have to be identical for the database to collapse them into one. For example, this solution would not be able to generate a list of manufacturers along with the first and last years each was active, because adding Product.year as a second value would change which results distinct() merges together.

The group_by() clause offers a result grouping solution that has more flexibility. The query above that returns the list of manufacturers can also be created with group_by() as follows:

```
>>> q = (select(Product.manufacturer)
        .group_by(Product.manufacturer)
        .order_by(Product.manufacturer))
```

The results are the same, but when using group_by(), additional columns can be added to the query, as long as they are aggregated into a single value for each group using a function. The next example obtains the list of manufacturers along with their first and last years of operation, and how many models they produced:

```
>>> q = (select(           Product.manufacturer,           func.min(Product.year),
        func.max(Product.year),
        func.count()
      )
     .group_by(Product.manufacturer)        .order_by(Product.manufacturer))
>>> session.execute(q).all()
[('Acorn Computers Ltd', 1980, 1995, 6), ..., ('West Computer AS', 1984, 1984, 1)]
```

When grouping in this way, the database uses functions such as min(), max() and count() to reduce the different values in the groups of entries that are being merged. A query that uses group_by() can have result values that are either explicitly referenced in the group_by() call, or that are aggregated with a function. Having any other result values would produce an error because it would not be possible to include multiple rows of values in a grouped result row.

Previously you've seen that the where() method can be used to filter the set of results returned by a query. The conditions given in where() are evaluated before results are grouped, so this clause cannot be used to filter grouped results. Similar to where(), the having() clause is used to filter the grouped and aggregated results. Below is a query that gets a list of manufacturers that have five or more models, along with the actual number:

```
>>> q = (select(
        Product.manufacturer,
        func.count()
      )
     .group_by(Product.manufacturer)
     .having(func.count() >= 5)
     .order_by(Product.manufacturer))
>>> session.execute(q).all()
[('Acorn Computers Ltd', 6), ('Amstrad', 7), ..., ('Timex Sinclair', 6)]
```

You may notice that in this example, the count() function appears twice. First it is used in the select() part so that it is included in the results, and then a second time in the having() method, so that entries can be filtered according to this value.

To ensure that the count of products per manufacturer is written only once, the label() method can be used to associate a label to the calculation, and then the label can be used in the two locations it is needed:

```
>>> num_products = func.count().label(None)
>>> q = (select(
        Product.manufacturer,
        num_products
```

```
    )
    .group_by(Product.manufacturer)
    .having(num_products >= 5)
    .order_by(Product.manufacturer))
```

The argument to the label() method is a name for the label, which is generated automatically by SQLAlchemy when given as None, ensuring that a unique name is picked. Letting SQLAlchemy pick the name is okay because what matters is that the label instance is assigned to the num_products variable, but in any case, if you prefer to also provide a name for the label, that is also allowed:

```
>>> num_products = func.count().label('num_products')
```

Pagination

For queries that return a large list of results, a common practice is to limit the number of results returned to some maximum number. The limit() method added to a query sets a maximum number of results. In the next query, up to three products are returned in alphabetical order:

```
>>> q = select(Product).order_by(Product.name).limit(3)
>>> session.scalars(q).all()
[Product(10, "464 Plus"), Product(11, "6128 Plus"), Product(6, "A7000")]
```

Putting a limit to the number of results is a good practice that prevents queries from retrieving too many results and becoming too large to handle. A common pattern when potentially long queries are issued is to provide *pagination* options, so that the results can be retrieved in chunks. In interactive or web applications users are often given the option to move forward and backwards on the results in increments of a given page size.

Implementing pagination of query results involves setting a page size with the limit() method, and indicating at which position to start retrieving results.

The simplest approach to select a start position is to add the offset() method to the query. This method sets a start index for the results to retrieve. The example query above could be generalized by adding offset(0) to it. A query to retrieve the second page of three results would be done as follows:

```
>>> q = select(Product).order_by(Product.name).limit(3).offset(3)
>>> session.scalars(q).all()
[Product(131, "Aamber Pegasus"), Product(84, "ABC 80"), Product(5, "Acorn Archimedes")]
```

Using offset() for pagination is often considered problematic, first because in most databases it does not have an efficient implementation, but more importantly because it may provide confusing results for datasets that change often. Continuing with the above examples, if a new product called "AAA" is added to the products table after a user viewed the second page of results, when the user requests the third page all the positions would have shifted one place down, and the "Acorn Archimedes" would appear again as the first item when offset(6) is requested. Similarly, if a product is deleted, all positions after the deleted item would shift one place up, and a product might be skipped when moving to the next page of results.

A more robust way of specifying where to start returning results is to use the last returned item as a reference. To request the second page of results using this method the query would be:

```
>>> q = select(Product).order_by(Product.name).where(Product.name > 'A7000').limit(3)
```

This query returns the same results as above, but has the advantage that items that are inserted or deleted will not cause any results to be repeated or omitted. The disadvantage is that when navigating the list of results backwards things get slightly more complicated. After seeing the second page of results, a query to go back to the first page would look like this:

```
>>> q = (select(Product)
         .order_by(Product.name.desc())
         .where(Product.name < 'Aamber Pegasus')
         .limit(3))
>>> session.scalars(q).all()
[Product(6, "A7000"), Product(11, "6128 Plus"), Product(10, "464 Plus")]
```

But this does not look like the first page anymore. The order_by() clause had to be reversed with desc() so that the query can reference the three items that appear right before the "Aamber Pegasus" product that starts the second page, and this causes the results to be in reverse order. The application will need to reverse these results before they are presented to the user.

Which of the two pagination solutions to use is a decision that needs to be made for each particular case. The offset() method is simpler to implement and allows a user to randomly request any page of results, since the offset can be calculated with just a multiplication. If the dataset rarely changes, this may be the best option.

The alternative solution using a where() clause is very robust as it will never duplicate or skip any items as the data changes, but that comes at a cost of a more complex implementation. Also, the where() solution does not allow random jumps, the user can only move forward or backwards one page at a time.

Obtain an Element by its Primary Key

A particularly useful query is one that retrieves the element in a database table that matches a given primary key value. This query can return one result when the item in question is found, or no results at all when the given key value does not exist. The query will never return more than one result, because primary keys are unique by definition. Using what you learned above, you can already build a query for this purpose. Below is an example query that retrieves the product that has its id set to 23:

```
>>> q = select(Product).where(Product.id == 23)
>>> session.scalar(q)
Product(23, "CT-80")
```

This is such a common query that SQLAlchemy implements a shortcut for it. The next example shows how to issue the same query as above using the get() method of the session object:

```
>>> session.get(Product, 23)
Product(23, "CT-80")
```

As with the longer form above, if the given primary key value does not exist in the database table, the return value is None.

Indexes

You may be wondering how can the database search information, and do so efficiently. Databases implement a variety of algorithms to navigate the data, and when given a specific query to execute they determine which of those algorithms are applicable and the most efficient to use.

There is one search algorithm that is always available: the *table scan*. A table scan operation consists in evaluating the query filters on all the rows of a table sequentially as the entries are read. This does not seem very efficient, does it?

Table scans are a last resort, an operation that the database will only use when no other option is available, or also when the data to search is small enough that there is no advantage in using a more sophisticated searching algorithm.

Your job as a database designer is to study the queries that the application makes and ensure that the data is properly *indexed* to support more advanced searching options when solving those queries.

When a column is marked as indexed, the database will maintain binary tree structures for the data in that column that allow for very efficient searching and sorting. Looking through the example queries shown in this chapter, searching is done on the following columns of the products table:

- `id`
- `name`
- `manufacturer`
- `year`

The `id` column is the table's primary key, which the database automatically indexes, so searches on this column are already optimized. The name, manufacturer and year columns, however, are used in where(), group_by() and order_by() clauses and are currently not indexed, which means that the table will have to be scanned when these columns are in a query.

What about the remaining two columns, country and cpu? These columns are not referenced in any of the example queries, so based on these queries there is no benefit in indexing them, and in fact, there are costs both in performance and disk space associated with maintaining indexes, so for these two columns it is best to not index them. This decision will need to be re-evaluated if other queries involving these columns are implemented later.

When using SQLAlchemy, a column can be marked as indexed with a index=True option added to it in the model definition. Here is the Product model class expanded with indexes:

Listing 8 *models.py*: indexes added to model

```
class Product(Model):
    __tablename__ = 'products'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(64), index=True)
    manufacturer: Mapped[str] = mapped_column(String(64), index=True)
    year: Mapped[int] = mapped_column(index=True)
    country: Mapped[str] = mapped_column(String(32))
    cpu: Mapped[str] = mapped_column(String(32))


    def __repr__(self):
        return f'Product({self.id}, "{self.name}")'
```

Constraints

Another good database design practice is to assign appropriate *constraints* to columns. The Product model already has a constraint called PRIMARY KEY, which is enabled on the id column with the primary_key=True option. This names the id column as the primary key of the products table.

Besides PRIMARY KEY, two other commonly used constraints are UNIQUE and NOT NULL.

A column that has a UNIQUE constraint does not allow duplicated values. In the Product model, this would be a good choice for the name column, to ensure that there are no two products with the same name. To add this constraint to a column, the unique=True option is used with SQLAlchemy.

The NOT NULL constraint prevents a column from ever having an empty or undefined value. This constraint can also be thought of in reverse, by saying

that columns that do not have the NOT NULL constraint are considered optional. Columns defined with the Mapped[t] typing syntax get the NOT NULL constraint by default, and to create a column that is allowed to have NULL values, the type hint should be changed to Mapped[Optional[t]]. The country and cpu can be considered optional at this point, but as always, this may need to change as the role of these columns is better defined.

Note

When working with the legacy column definition syntax in SQLAlchemy 1.x versions that is based on the Column() constructor, the nullable=True option is the default and is used to denote an optional column, while nullable=False must be added for columns for which a value is required.

Below you can see an updated Product model class, with constraints added.

Listing 9 *models.py*: constraints added to model

```
# ...
from typing import Optional

class Product(Model):
    __tablename__ = 'products'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(64), index=True, unique=True)
    manufacturer: Mapped[str] = mapped_column(String(64), index=True)
    year: Mapped[int] = mapped_column(index=True)
    country: Mapped[Optional[str]] = mapped_column(String(32))
    cpu: Mapped[Optional[str]] = mapped_column(String(32))

    def __repr__(self):
        return f'Product({self.id}, "{self.name}")'
```

Deletions

You've seen above that new objects are added to the database using the add() method of the session, and that this schedules the new object to be saved to the database in the next commit operation. The session also has a delete() method.

If you have left your Python prompt, start a new one and initialize it as before:

```
>>> from db import Session
>>> from models import Product
```

```
>>> session = Session()
```

Now try deleting a product:

```
>>> p = session.get(Product, 23)
>>> session.delete(p)
>>> session.commit()
```

Once a session having deleted objects is committed, these objects are effectively removed and cannot be retrieved anymore.

```
>>> p = session.get(Product, 23)
>>> print(p)
None
```

Exercises

Many of the chapters in this book include a list of exercises that can help you solidify the knowledge you acquired in the chapter. Solutions to all exercises are provided at the end of the book.

Before you attempt to solve these exercises, make sure that you have all the products imported. If you are in doubt that you have a complete database you can run the importer script once again:

```
(venv) $ python import_products.py
```

Start a Python session and write queries that return the following information:

1. The first three products in alphabetical order built in the year 1983.
2. Products that use the "Z80" CPU or any of its clones. Assume that all products based on this CPU have the word "Z80" in the cpu column.
3. Products that use either the "Z80" or the "6502" CPUs, or any of its clones, built before 1990, sorted alphabetically by name.
4. The manufacturers that built products in the 1980s.
5. Manufacturers whose names start with the letter "T", sorted alphabetically.

6. The first and last years in which products have been built in Croatia, along with the number of products built.

7. The number of products that were built each year. The results should start from the year with the most products, to the year with the least. Years in which no products were built do not need to be included.

8. The number of manufacturers in the United States (note that the country field for these products is set to USA)

One-To-Many Relationships

In the previous chapter you learned how to execute a variety of queries on the products table. Interestingly, some of those queries were designed to obtain product manufacturers and not products, and this required duplicates to be removed by grouping the results.

In many situations, grouping is a great solution, especially when aggregation functions are used to calculate information about each group that is not directly stored in the database. There are cases, however, in which the duplication that occurs when storing secondary data in the same table as the primary entity is problematic, as the table may grow much larger than it needs to be, and spelling differences in the duplicated data could produce incorrect grouped results.

In this chapter you are going to learn how to eliminate duplication by creating multiple related tables using one of the fundamental patterns in relational database design: the *one-to-many* relationship.

When to Use a Relationship

You may be wondering what makes the manufacturer column of the Product model a good candidate to move to its own separate model class and database table. There isn't a definitive answer on this, just that some entities are intuitively first-class and there is no doubt that they should be independent, while others cannot be identified so easily, and it may require one or more iterations on the database design before the benefit of creating a separate table becomes clear.

As an example, if the RetroFun database had to be extended to store orders from customers, nobody would disagree that these should be stored in their own table. But each order will need customer information. Should customers be in their own table or stored as additional columns with the order? For a business in which it is highly unlikely for customers to make

repeat orders you may consider the simpler approach of storing customer details directly with the orders, but for most businesses keeping customer information separately is likely more convenient. Database design decisions aren't absolute and should be made with the needs of the application in mind.

As a rule of thumb, when you find that information is being duplicated, you should consider the benefits of removing this duplication through the addition of a new table and a relationship. In the case of the manufacturer column, a separate table means that each company name will exist only once, so there will never be a risk of a spelling mistake affecting how products are grouped. Having the manufacturers in their own table would also make it possible to rename a company name or add other details such as the company address and phone number, and all these changes will have to be made only once and will be immediately accessible to all related products.

What about the other attributes of the Product model?

The country column is definitely a candidate for a relationship, as with this column once again there is duplication of country names. If grouping by country is desired, then countries would have to be entered with the exact same spelling in every occurrence to be grouped correctly. For example, "United States" and "united states" would be grouped separately because of the case difference.

The year column, on the other side, is just numbers, so initially it would not appear as if removing duplication can make a significant improvement. And looking at the contents of the cpu field in many of the products it is clear that the field is written as descriptive text and not as a reference to an entity, so it is also not a good candidate to relocate to a separate table, at least if kept in the current format.

In this chapter, you are going to learn how to move the manufacturers into a separate table, and how to build a relationship between products and manufacturers to preserve the information of which company built which product. Applying a similar change to the countries presents additional

challenges that will be discussed in the next chapter, so for now only the manufacturer will be addressed.

One-To-Many Relationship Implementation

Relational databases create links between entities stored in different tables through *relationships*. There are two main types of relationships:

- One-to-many
- Many-to-many

Relational database literature will also mention two additional relationship types: the many-to-one and the one-to-one, which are special forms of the one-to-many type.

A one-to-many relationship describes a situation in which two tables A and B are related in such a way that an entry in table A can be linked to any number of entries in table B, but each entry in table B is linked to at most one entry from A. In this scenario table A is the "one" and table B is the "many" of the relationship.

This pattern fits the relationship between computer manufacturers and their computer products. A manufacturer can produce many computer models, and each of these computer models was built by only one manufacturer. So the manufacturer is the "one" and the products are the "many".

The first step when defining a relationship is to create database tables (or models, when using SQLAlchemy ORM) for the two entities involved. The database in its current state has a products table, and now it needs a manufacturers table. Add a Manufacturer model at the bottom of *models.py* to represent manufacturers:

Listing 10 *models.py*: Manufacturers model

```
class Manufacturer(Model):
    __tablename__ = 'manufacturers'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(64), index=True, unique=True)
```

```
    def __repr__(self):
        return f'Manufacturer({self.id}, "{self.name}")'
```

With this, the manufacturer column from the Product model becomes name in the new Manufacturer model class. The class has its own id primary key, and a __repr__() implementation to have instances of this class print nicely when debugging. The name column has an additional unique=True option that adds a UNIQUE constraint to the column, because in this table each manufacturer will appear only once.

The manufacturer column of Product has to be removed, but what can it be replaced with? To establish the relationship, a manufacturer_id column is defined in its place:

Listing 11 *models.py*: Manufacturer foreign key in Product model

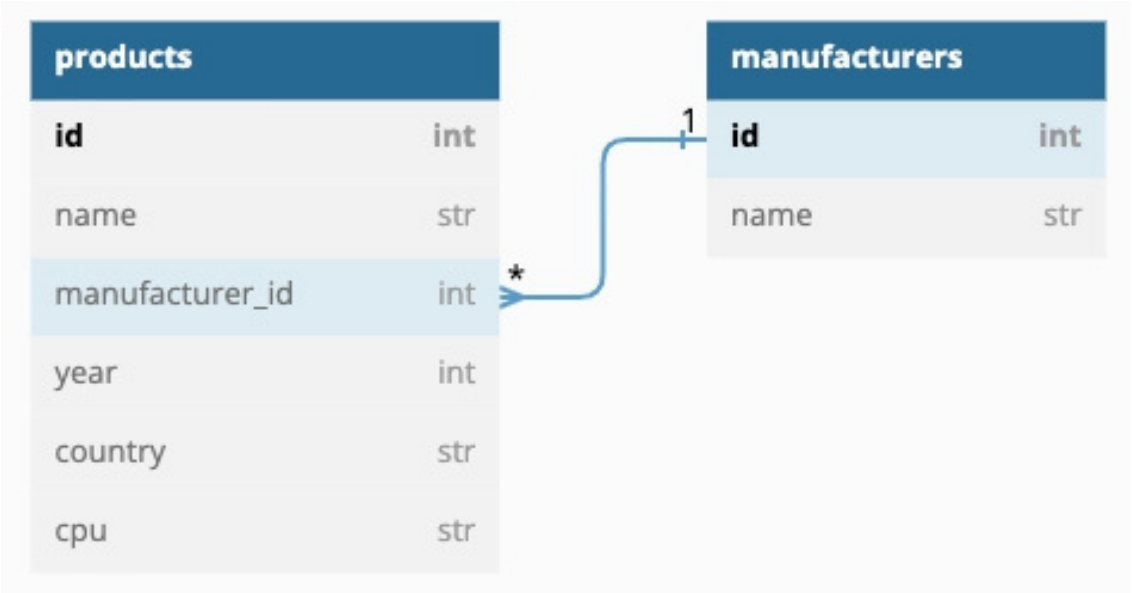
```
from sqlalchemy import ForeignKey

class Product(Model):
    __tablename__ = 'products'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(64), index=True, unique=True)
    manufacturer_id: Mapped[int] = mapped_column(
        ForeignKey('manufacturers.id'), index=True)
    year: Mapped[int] = mapped_column(index=True)
    country: Mapped[Optional[str]] = mapped_column(String(32))
    cpu: Mapped[Optional[str]] = mapped_column(String(32))


    def __repr__(self):
        return f'Product({self.id}, "{self.name}")'
```

The new manufacturer_id column is an integer, matching the type of the primary key of the new Manufacturer model. Columns that reference primary keys of another table are called *foreign keys*, and are given a foreign key constraint. A good naming convention to follow is to name foreign key columns with the name of the referenced entity followed by the _id suffix.

The ForeignKey class applies the foreign key constraint to the column. The argument passed to the class indicates what is the primary key target, which can be given as a column object (such as Manufacturer.id) or as a string with the format '<tablename>.<column>'. Using the string format makes it possible to use forward references to entities that are defined later in the file.

The new manufacturer_id column has an index, to help the database navigate this column efficiently. Some databases automatically index foreign key columns, but others do not, so it is best to be explicit and always add an index. It also has the NOT NULL constraint, indirectly due to the column's typing hint not including Optional. Stating that the foreign key cannot be null ensures that products without a link to a manufacturer are not allowed.

Below you can see a diagram of the database structure with this relationship.



It is useful to remember that one-to-many relationships always follow this style, where the "many" side (products) includes a foreign key column that references the "one" side (manufacturers).

SQLAlchemy Relationships

You now know how one-to-many relationships are implemented in a relational database. But you may be thinking that you will now have to tediously navigate using these primary and foreign keys, and that this can be error-prone, maybe in ways that are more dangerous than having data duplication.

Consider, for example, what an application would need to do to obtain the manufacturer of a given product. When manufacturers were in the `products` table, you could obtain the manufacturer very easily:

```
>>> p = session.get(Product, 127)
>>> p
Product(127, "ZX Spectrum")
>>> p.manufacturer
'Sinclair Research'
```

Having the data broken down into separate tables creates a complication, because now a product can only provide access to its manufacturer_id attribute, which is a number. Then this number has to be used to load the manufacturer from the new manufacturers table.

Fortunately the ORM module of SQLAlchemy provides high-level support for relationships, making most of the work of navigating foreign keys invisible. To gain access to these features, the two model classes involved in a relationship need relationship attributes that represent this relationship. Below you can see how these objects are defined for Product and Manufacturer. Note that these relationship objects are added to each model, without changing any of its existing attributes.

Listing 12 *models.py*: Define relationship objects for one-to-many relationship

```
from sqlalchemy.orm import relationship

class Product(Model):
    # ...
    manufacturer: Mapped['Manufacturer'] = relationship(
        back_populates='products')
    # ...

class Manufacturer(Model):
    # ...
    products: Mapped[list['Product']] = relationship(
        back_populates='manufacturer')
    # ...
```

The Product class now has a manufacturer attribute that represents the relationship as seen from the "many" side. This attribute is not a column that is physically stored in the database; it is a high-level replacement of manufacturer_id that transparently loads the related model object, as you will see soon.

The change in the Manufacturer class is also interesting. This class has a new products attribute, representing the same relationship but as seen from the "one" side. From this side, a manufacturer can have many related products, so this attribute is a list that is automatically populated with the corresponding product instances.

The typing hints given to the relationship attributes are based on the same Mapped[x] type used for columns, but for these the x is the model class for the other side of the relationship, either on its own for the "one" side or as a list for the "many" side. As in many other cases before, the class names can be given directly, or as a string. Using a string is often necessary to prevent errors when needing to use a forward reference. For consistency, you may opt to always use strings.

The two relationship() definitions have back_populates arguments, each set to the name of the relationship attribute on the other side. This is so that SQLAlchemy understands that these two attributes represent the two sides of the same relationship.

If these relationships seem confusing do not worry, you will find them much easier to understand soon, once you start using them.

A Revised Importer Application

The changes made to the models mean that the product importer script from the previous chapter has to be updated so that it now imports manufacturers to their own table.

Below is the updated importer script, which is still stored in *import_products.py*. The same CSV file is used.

Listing 13 *import_products.py*: Import products and manufacturers from CSV file

```
import csv
from db import Model, Session, engine
from models import Product, Manufacturer


def main():
    Model.metadata.drop_all(engine)  # warning: this deletes all data!
```

```
    Model.metadata.create_all(engine)

    with Session() as session:
        with session.begin():
            with open('products.csv') as f:
                reader = csv.DictReader(f)
                all_manufacturers = {}

                for row in reader:
                    row['year'] = int(row['year'])

                    manufacturer = row.pop('manufacturer')
                    p = Product(**row)

                    if manufacturer not in all_manufacturers:
                        m = Manufacturer(name=manufacturer)
                        session.add(m)
                        all_manufacturers[manufacturer] = m
                    all_manufacturers[manufacturer].products.append(p)


if __name__ == '__main__':
    main()
```

In this version an all_manufacturers dictionary is initialized empty above the CSV import for-loop. For each row of the CSV file that is processed, the manufacturer field is extracted from the row dictionary before the items in this dictionary are used to initialize the Product instance. This is necessary because manufacturer isn't a column in the Product model anymore.

The manufacturer name is checked for existence in the all_manufacturers dictionary. When not found, a new Manufacturer object is created and initialized with that name. The new manufacturer object is added to the SQLAlchemy session so that it is later saved, and it is also added to the dictionary, so that repeat appearances of this manufacturer all use the same instance.

In the final line of the loop, the new product is appended to the products relationship of the manufacturer, which works similarly to a list. This products relationship object, which represents the "many" side, has append() and remove() methods, allowing applications to add or remove objects from the relationship using the familiar list syntax. SQLAlchemy automatically translates these operations to the corresponding foreign key changes.

In case you are interested in the details, the append() call on the products relationship attribute achieves two things: first, it links the manufacturer to the product through the manufacturer_id foreign key, which will be automatically set when the session is committed; and second, it indirectly

includes the new product in the database session, because it is referenced by the manufacturer instance which has been explicitly added before. An explicit session.add(p) for the product would not cause any harm, but it isn't necessary. This automatic addition of a child to the session when the parent is already in it is called a *cascade*. You will learn about this and a few other types of cascades more in detail later.

When the session block ends, all the manufacturers and products that are in the session are saved to the database in a single atomic operation.

Ready to try this new importer? Run the script as follows:

```
(venv) $ python import_products.py
```

The drop_all() call at the start of the main() function will destroy the earlier version of the products table, and then create_all() will create the new products and manufacturers tables according to the new models.

Note

Wiping the entire database each time a change is made is somewhat extreme. In the real world a database cannot afford the disruption of having to be recreated entirely from scratch every time a structural change is made.

In a later chapter you will learn how to implement *database migrations* to manage changes in a more reasonable way.

Feel free to inspect the new database using your database administration tool if you like. Note how the manufacturer_id column in the products table has numbers, and how these numbers can be used to locate the manufacturer in the manufacturers table.

If you inspect your tables, you will notice that the manufacturer and products relationship objects that were defined in the model classes do not exist in the database. These are virtual attributes that are managed by SQLAlchemy and that only exist in the Python objects, so they do not have a representation in the database.

One-To-Many Relationship Queries

You are probably anxious to see how the SQLAlchemy relationship objects are used in queries. Start a Python shell, import the session class and the models, and create a session:

```
>>> from sqlalchemy import select
>>> from db import Session
>>> from models import Product, Manufacturer
>>> session = Session()
```

Load the "ZX Spectrum" product:

```
>>> p = session.scalar(select(Product).where(Product.name == 'ZX Spectrum'))
>>> p
Product(127, "ZX Spectrum")
```

Who is the manufacturer of this product? In the single table version, p.manufacturer would have returned the name as a string. Now there is a relationship object instead of the string column, and this new attribute transparently returns the model instance that represents the linked entity.

```
>>> p.manufacturer
Manufacturer(63, "Sinclair Research")
```

As you recall, the actual string name of this company is stored in the `name` attribute of the model. This name is also easily accessible:

```
>>> p.manufacturer.name
'Sinclair Research'
```

Did this manufacturer make other computer models? This can be checked by looking at the one-to-many relationship from the manufacturer's side, which returns a list:

```
>>> p.manufacturer.products
[Product(125, "ZX80"), Product(126, "ZX81"), Product(127, "ZX Spectrum"),
Product(128, "Sinclair QL")]
```

It is also possible to start navigating the database from a manufacturer. Let's find Texas Instruments by name and then look at the list of products it made:

```
>>> m = session.scalar(
    select(Manufacturer)
      .where(Manufacturer.name == 'Texas Instruments'))
```

```
>>> m
Manufacturer(66, "Texas Instruments")
>>> m.products
[Product(132, "TI-99/4"), Product(133, "TI-99/4A")]
```

One of the queries shown in the previous chapter returned product names and manufacturer names side-by-side. This was easy to do when both entities were defined in the same table, but doing this now requires combining information from two tables, an operation that relational databases call a *join*. Here is the query that does this:

```
>>> q = select(Product.name, Manufacturer.name).join(Product.manufacturer)
>>> session.execute(q).all()
[('Acorn Atom', 'Acorn Computers Ltd'), ..., ('GEM 1000', 'GEM')]
```

In this query, the select() statement names two attributes from different tables. Any time multiple tables are involved in a query, SQLAlchemy needs to know how to join the tables, and that is why the join() clause was added. When using the ORM module, the argument to join() can be one of the two relationship attributes, and SQLAlchemy figures everything out from it. Since the two relationship objects are linked through the back_populates options, in general it does not matter which of the two is given in the join() clause. In the example query above, passing Product.manufacturer to join() means that Product will be on the left side of the join, and Manufacturer will be on the right. If instead Manufacturer.products is passed, then the sides will be reversed, but the results will be the same. Later you will learn about cases in which it does matter what entity is on the left and the right sides of a join.

If you are familiar with how joins are constructed in SQL, you may want to print the query to understand how this is translated to the statement sent to the database:

```
>>> print(q)
SELECT products.name, manufacturers.name AS name_1
FROM products JOIN manufacturers ON manufacturers.id = products.manufacturer_id
```

Here you can appreciate how SQLAlchemy ORM defines the join condition on its own thanks to the knowledge it has of the relationship.

Another interesting query from last chapter returned the manufacturers in alphabetical order along with the count of products each made. This also requires a join now that the data is split across two tables:

```
>>> from sqlalchemy import func
>>> q = (select(
        Manufacturer,
        func.count(Product.id)
    )
    .join(Manufacturer.products)
    .group_by(Manufacturer)
    .order_by(Manufacturer.name))
>>> session.execute(q).all()
[(Manufacturer(1, "Acorn Computers Ltd"), 6), (Manufacturer(24, "AGAT"), 1), ...,
(Manufacturer(75, "West Computer AS"), 1)]
```

This query isn't the first to have two values per result row, but it is the first in which one of the results is a model and the other isn't. Note the Manufacturer model given in the select() statement, and again in the group_by(). When group_by() receives a model class as an argument instead of a single attribute, the grouping is done by all the attributes of the model combined. If you are interested in how this grouping is translated to SQL, you can look at the SQL code for this query:

```
>>> print(q)
SELECT manufacturers.id, manufacturers.name, count(*) AS count_1
FROM manufacturers JOIN products ON manufacturers.id = products.manufacturer_id
GROUP BY manufacturers.id, manufacturers.name ORDER BY manufacturers.name
```

I hope you can appreciate how SQLAlchemy greatly simplifies the creation of these queries. Consider that as columns are added or removed from the manufacturers table this query will automatically adjust and still be able to group the model as a whole, without any changes needed.

Lazy vs. Eager Relationships

Have you thought about what happens when you access one of these seemingly magical relationship attributes defined in model classes?

A good way to spy on the database activity is to enable the echo option in the SQLAlchemy engine object. Open *db.py* in your code editor and add `echo=True` to the `create_engine()` call:

```
engine = create_engine(os.environ['DATABASE_URL'], echo=True)
```

Save the change and then open a new Python session. Import all the necessary components again and get the "Texas Instruments" manufacturer as before:

```
>>> from db import Session >>> from models import Product,
Manufacturer >>> from sqlalchemy import select
>>> session = Session()
>>> m = session.scalar(
      select(Manufacturer)
        .where(Manufacturer.name == 'Texas Instruments'))
```

Right after you execute the query with scalar(), you will see some activity logged to your terminal. You can see an example of what you might see below, but keep in mind that the output can vary depending on the database that you use.

```
2023-01-03 18:44:32,185 INFO sqlalchemy.engine.Engine select pg_catalog.version()
2023-01-03 18:44:32,185 INFO sqlalchemy.engine.Engine [raw sql] {} 2023-01-03
18:44:32,188 INFO sqlalchemy.engine.Engine select current_schema()
2023-01-03 18:44:32,188 INFO sqlalchemy.engine.Engine [raw sql] {}
2023-01-03 18:44:32,190 INFO sqlalchemy.engine.Engine show
standard_conforming_strings
2023-01-03 18:44:32,190 INFO sqlalchemy.engine.Engine [raw sql] {}
2023-01-03 18:44:32,192 INFO sqlalchemy.engine.Engine BEGIN (implicit) 2023-01-03
18:44:32,197 INFO sqlalchemy.engine.Engine SELECT manufacturers.id,
manufacturers.name
FROM manufacturers
WHERE manufacturers.name = %(name_1)s
2023-01-03 18:44:32,197 INFO sqlalchemy.engine.Engine [generated in 0.00019s]
{'name_1': 'Texas Instruments'}
```

All the log lines up to and including the BEGIN statement are part of the session initialization. The SELECT statement is the actual execution of the scalar() call, which is followed by a summary line that shows the placeholder values that were used in the query.

This all looks great. Now when you try to access m or any of its direct attributes such as m.name there is no additional database activity logged, because all the attributes were loaded from the query and are cached in the database session. But see what happens when you try to access the relationship m.products:

```
>>> m
Manufacturer(66, "Texas Instruments")
>>> m.name
'Texas Instruments'
>>> m.products
```

At this point you are going to see additional database statements appear in the log. SQLAlchemy is running a database query on its own, so that it can provide the list of products that are related to the manufacturer.

```
2023-01-03 18:46:53,940 INFO sqlalchemy.engine.Engine SELECT products.id AS products_id,
products.name AS products_name, products.manufacturer_id AS products_manufacturer_id,
products.year AS products_year, products.country AS products_country,
products.cpu AS products_cpu
```

```
FROM products
WHERE %(param_1)s = products.manufacturer_id
2023-01-03 18:46:53,940 INFO sqlalchemy.engine.Engine [generated in 0.00019s] {'param_1': 66}
```

If you attempt to access the same relationship a second time, the response will be immediate, as the results are now cached within the context of the database session.

A similar behavior can be observed when navigating the relationship from the other side. Try loading the "ZX Spectrum" product once again:

```
>>> p = session.get(Product, 127)
```

The output should include a database query similar to this:

```
2023-01-03 18:50:42,294 INFO sqlalchemy.engine.Engine SELECT products.id AS products_id,
products.name AS products_name, products.manufacturer_id AS products_manufacturer_id,
products.year AS products_year, products.country AS products_country,
products.cpu AS products_cpu
FROM products
WHERE products.id = %(pk_1)s
2023-01-03 18:50:42,294 INFO sqlalchemy.engine.Engine [generated in 0.00022s] {'pk_1': 127}
```

You can now access all the attributes of the Product model, but the manufacturer relationship triggers more database activity:

```
>>> p.manufacturer
```

Before the result is printed to the console, another query executes implicitly:

```
2023-01-03 18:51:30,151 INFO sqlalchemy.engine.Engine SELECT manufacturers.id AS
manufacturers_id,
manufacturers.name AS manufacturers_name
FROM manufacturers
WHERE manufacturers.id = %(pk_1)s
2023-01-03 18:51:30,151 INFO sqlalchemy.engine.Engine [generated in 0.00020s] {'pk_1': 63}
```

As you see from these examples, SQLAlchemy ORM remembers the session that is in use and sends its own queries to it the first time access to a relationship attribute is made.

When SQLAlchemy works in this way, it is said to use "lazy" loading of relationships. Thanks to this, the application can forget that these attributes are relationships and use them as if they were regular attributes of the model.

While loading relationships lazily sounds great in principle, it has a downside. The fact that database queries are issued implicitly can make the application lose track of how many queries it is sending to the database, and possibly become inefficient if too many implicit queries bog down the database server.

An example of how lazy loading can cause trouble might be useful.

Remember the query that returns each product along with its manufacturer? Here it is one more time:

```
>>> q = select(Product.name, Manufacturer.name).join(Product.manufacturer)
```

This query is very efficient, since a single database operation returns all the product and manufacturer pairs. A developer that is not aware of the effects of lazy loading might decide to use a different approach to retrieve this same data, taking advantage of the manufacturer relationship attribute in a for-loop:

```
>>> q = select(Product)
>>> for p in session.scalars(q):
        print(p.name, p.manufacturer.name)
```

At first sight, this looks like a simple and safe way to get the same list of pairs, right? Can you guess how many database queries it takes to produce the list in this way?

Run the above for-loop in the Python session that has the echo option enabled, and you will see lots of database queries scroll by. The exact number of queries this loop requires is one for the initial query stored in the q variable, plus one additional lazy loading query per manufacturer.

There are 76 manufacturers, so the total count is 77 queries, to get the same information that was obtained above with just one!

Relationship Loaders

The good news is that SQLAlchemy offers some options to configure these relationships and make them more useful and efficient based on how they will be used.

SQLAlchemy uses a relationship *loader* to bring one or more related objects into the session. The default loader, which you have seen in action above, is called the select loader.

Another available loader is called joined. This loader reads the related objects from the database at the same time the parent is retrieved, by extending the main query with a join clause.

The select loader is a "lazy" loader, because the database query for the related objects is delayed until the relationship attribute is accessed for the first time. The joined loader is an "eager" loader, because the relationship data is requested at the same time the parent object is, no matter if the application wants it or not.

Using the joined loader, the for-loop example above would not issue any additional queries beyond the initial one. To enable this loader, the options() method can be added to the query as follows:

```
>>> from sqlalchemy.orm import joinedload
>>> q = select(Product).options(joinedload(Product.manufacturer))
```

This is telling SQLAlchemy to override the default lazy loading and bring the manufacturer relationship into the session using the joined loader. Feel free to try the for-loop above with this as the initial query to see the difference.

Instead of choosing the loader explicitly in each query, it is also possible to change the default loader that is used by the relationship. This is done by passing the desired loader in the lazy argument. Here is how the manufacturer relationship could be made to use the joined loader by default:

```
class Product(Model):
    # ...
    manufacturer: Mapped['Manufacturer'] = relationship(
        lazy='joined', back_populates='products')
    # ...
```

You may be wondering why relationships use select as the default loader instead of joined, which is, at least in some cases, more efficient. The fact is that it is really difficult to know which loader is best, as this largely

depends on the use the application gives to each relationship. The joined loader is useful when you know for sure that you'll need to access the related objects, but it may not be the best choice when these objects may or may not be needed, as many objects would be loaded unnecessarily into the session. It is also unlikely to perform well for complex queries or relationships with many items, because the cost of adding a join in those cases can be significant.

To make choosing the best loader even harder, the select and joined loaders are not the only available options to choose from. Below is the complete list of loaders that can be used:

- select: issues a select() statement lazily, when the relationship attribute is accessed for the first time. This is the default behavior. As a query option, this loader can be enabled with the lazyload() function.
- immediate: loads the related entities at the same time the parent is loaded with a separate select() statement. The only difference between select and immediate is that the latter issues all the relationship queries up front instead of on demand. As an option, this loader is enabled with the immediateload() function.
- joined: loads the related entities at the same time the parent is loaded by extending the parent's query with a join to the related table. Use the joinedload() function to enable it explicitly as an option in a query.
- subquery: loads the related entities immediately after the parent, in a single additional query that joins a copy of the original query (transformed into a subquery) with the related table. The option version of this loader is the subqueryload() function.
- selectin: loads the related entities immediately after the parent, in a single additional query that specifies the list of primary keys to load with the IN operator. As an option, this loader is enabled with the `selectinload()` option.
- write_only: disables the loading of the relationship. This loader is only available in SQLAlchemy 2.0 and later and can only be used in

the relationship, so it does not have an option version. This is an important loader that you will learn more about later in this book.

- noload: a somewhat similar option to write_only, but less flexible. Recommended in place of write_only when using SQLAlchemy 1.4. The option version of this loader is enabled with the noload() function.
- raise and raise_on_sql: two slightly different modes in which an exception is raised if a relationship needs to be loaded implicitly. These modes can be used to detect application code that triggers implicit I/O operations when these are not desired. The raiseload() function is the query option version.
- dynamic: a legacy loader that is incompatible with the new SQLAlchemy query APIs, so it should not be used except with legacy code.

It is expected for many of these options to seem obscure or to not make sense at all at this time. Over time, and as you start working with some of these, their purpose will become more clear. A good way to reduce the complexity of having so many options is to group these loaders according to when they load the relationship. Study the following table:

When	Loaders
Lazy load	select (default), dynamic (legacy)
Eager load	joined, selectin, subquery, immediate
Explicit load	write_only (SQLAlchemy 2.0 and up), noload, raise, raise_on_sql

Using this table, the list of choices is reduced to three functional groups, with the loaders within each group only having implementation differences, but operating similarly.

What are the best default loaders for the two relationships in *models.py*? With such a small application it is hard to make a decision. A good reason to change the default lazy loading mechanism is to improve performance when the database server is hit by many small relationship queries. At this

early time in the life of this project this isn't a concern, so a sensible decision is to keep using the select lazy loader for now and postpone any potential changes until later, when these relationships are used more.

Deletion of Related Objects with a Cascade

You've briefly seen how to delete entities from the database in the previous chapter. In general, deleting from the "many" side of a one-to-many relationship works in the same way and does not present a problem. To try this, grab a product and its manufacturer:

```
>>> p = session.get(Product, 24)
>>> p
Product(24, "Atari 400")
>>> m = p.manufacturer
>>> m
Manufacturer(8, "Atari, Inc.")
```

You can now delete the product:

```
>>> session.delete(p)
>>> session.commit()
```

And this works well. Deleting the manufacturer, however, is more difficult. Try to do it to see what happens:

```
>>> session.delete(m)
>>> session.commit()
[ traceback omitted ]
sqlalchemy.exc.IntegrityError: (sqlite3.IntegrityError) NOT NULL constraint failed: products.manufacturer_id
[SQL: UPDATE products SET manufacturer_id=? WHERE products.id = ?]
[parameters: ((None, 25), (None, 26), (None, 27), (None, 28), (None, 29), (None, 30))]
(Background on this error at: https://sqlalche.me/e/14/gkpj)
```

If the manufacturer only had the one product deleted above, then the deletion would have worked. But this manufacturer has a few more products, all of which still exist and have their manufacturer_id foreign key pointing back at this entry. If the manufacturer is removed then the foreign keys on these products would become invalid. SQLAlchemy recognizes this and attempts to set the foreign keys that are invalidated to NULL before deleting the manufacturer. However, the manufacturer_id column is not defined as optional, so the attempt to set it to NULL fails and produces the above error message.

Note

When an operation such as a commit fails, the session transitions into an error state and cannot be used anymore until it is rolled back. This does not cause any major problems when using a context manager because sessions are always rolled back on errors and closed on exit, but when managing the session life cycle manually in the Python prompt it is necessary to do the roll back explicitly. Here is how this is done:

```
>>> session.rollback()
```

Automatic operations such as the attempt to clear foreign keys of an item that is about to be deleted are called *cascades*, and are not limited to deletions. There are a few other situations in which SQLAlchemy also applies a change to child objects in a relationship as a result of an action performed on the parent object. Recall how in the product importer script, after a Product instance was created, it was not explicitly added to the session, because its parent (the Manufactuer instance) was already added and the parent's inclusion in the session "cascades" into its children.

The desired cascade behaviors are defined in each relationship object, and in many cases the default is the best configuration. To change cascade options, a string with comma-separated options is given in the cascade argument to the relationship() call.

Note

Unfortunately cascades are an area of SQLAlchemy that is particularly complex so only the two most common cascade configurations will be described here. If you are interested in the details of each individual cascade option, consult the Cascades section of the SQLAlchemy documentation.

The two most used cascade configurations are the following:

- 'save-update, merge': a conservative cascading behavior that is the default, recommended for most relationships. Since this is the default, there is no need to set these options explicitly. With this configuration, child objects are automatically included in the session if the parent has

been added to it. The behavior that sets foreign keys that are invalidated by deletions to NULL is actually defined by the absence of the delete option in this list.

- 'all, delete-orphan': an alternative, more aggressive cascading configuration that makes most operations done on the parent apply to its children, and in particular will delete children along with their parent. The all option causes some confusion because in spite of its name it covers all the cascades except delete-orphan, which causes children to also be deleted when they are removed from their relationship and become orphaned, even if their parent remains in the database.

The alternative cascade settings are better suited to the relationship between products and manufacturers. With this change, when a manufacturer is deleted, any products associated with it would be deleted as well. Here is how to reconfigure the relationship to implement this behavior:

Listing 14 *models.py*: Change cascade options for products to manufacturer relationship

```
class Manufacturer(Model):
    # ...
    products: Mapped[list['Product']] = relationship(
        cascade='all, delete-orphan', back_populates='manufacturer')
    # ...
```

Save this change, and then start a new Python session to try to delete the manufacturer again (if you don't want to continue seeing SQL logs in your sessions, feel free to remove the echo=True option in *db.py*).

```
>>> from db import Session
>>> from models import Product, Manufacturer
>>> session = Session()
>>> m = session.get(Manufacturer, 8)
>>> m
Manufacturer(8, "Atari, Inc.")
>>> session.delete(m)
>>> session.commit()
```

If you query the list of products again, you will find that all the products that were attached to the deleted Atari, Inc. company have now been deleted along with it.

Detaching Related Objects

Sometimes it is necessary to delete the relationship between two objects, without deleting the objects themselves. This can be thought of a "detach" operation that breaks the link between two objects.

For a one-to-many relationship there are two ways to detach two related objects, depending on the side from which this is done. When doing it from the "one" side, the relationship object presents all the related objects in the "many" side in a format similar to a list. In this case, the remove() method on the relationship object can be used to remove an element, following familiar list semantics.

The next example gets a product and its manufacturer, then unlinks them:

```
>>> p = session.get(Product, 1)
>>> p
Product(1, "Acorn Atom")
>>> m = p.manufacturer
>>> m
Manufacturer(1, "Acorn Computers Ltd")
>>> m.products.remove(p)
>>> session.commit()
```

After the session is committed, this product is not linked anymore to the manufacturer. But there is an unintended consequence. If you now try to get the product again, it isn't there anymore:

```
>>> p = session.get(Product, 1)
>>> print(p)
None
```

Recall that the cascade options used by the products relationship object were changed to ensure that products are deleted along with their manufacturer. The delete-orphan cascade option that was included in this relationship covers the case of a product becoming an orphan, and it states that the orphaned object should be deleted too.

What would happen if delete-orphan wasn't used? Then SQLAlchemy would set the manufacturer_id foreign key in the product to None to break the link to the parent, but this column cannot accept null values, so the result will be that the commit operation would fail, and the relationship link would not be removed.

For a one-to-many relationship in which it is acceptable to have objects from the "many" side in an orphaned state the foreign key column must be configured as nullable by adding the Optional type hint, as this will prevent the error.

Detaching a one-to-many relationship also works from the "many" side:

```
>>> p = session.get(Product, 2)
>>> p
Product(2, "BBC Micro")
>>> p.manufacturer = None
>>> session.commit()
```

When starting from the product side, the relationship is broken by setting the parent object to None. For this particular relationship, however, orphaned products are not allowed because the Optional type hint hasn't been used in the relationship object or the foreign key column, so this operation generates another error from SQLAlchemy.

Exercises

Now is your chance to practice some one-to-many relationship queries. Before starting, run the *import_products.py* script to restore any products or manufacturers you may have deleted. Then open a Python session and write queries that return:

1. The list of products made by IBM and Texas Instruments.
2. Manufacturers that operate in Brazil.
3. Products that have a manufacturer that has the word "Research" in their name.
4. Manufacturers that made products based on the Z80 CPU or any of its clones.
5. Manufacturers that made products that are not based on the 6502 CPU or any of its clones.

6. Manufacturers and the year they went to market with their first product, sorted by the year.

7. Manufacturers that have 3 to 5 products in the catalog.

8. Manufacturers that operated for more than 5 years.

Many-To-Many Relationships

Continuing with the topic of relationships, this chapter is dedicated to the *many-to-many* type, which as its name implies, is used when it is not possible to identify any of the sides as a "one" side.

In this chapter you are going to learn how to implement a many-to-many relationship between the products table and a new countries table.

What Is a Many-To-Many Relationship?

The best way to understand the purpose of many-to-many relationships is with an actual example. Start a Python shell and obtain a list of countries, removing duplicates with the distinct() clause. Here is how to do it:

```
>>> from db import Session >>> from models import Product >>> from sqlalchemy import select
>>> session = Session()
>>> q = select(Product.country).order_by(Product.country).distinct()
>>> session.scalars(q).all()
['Australia', 'Belgium', 'Brazil', 'Bulgaria', 'China', 'Croatia', 'Czechoslovakia', 'East Germany', 'France',
'Germany', 'Hong Kong', 'Hungary', 'Japan', 'Netherlands',
'New Zealand', 'Norway', 'Portugal', 'Portugal/Poland', 'Romania', 'Serbia', 'Sweden',
'Taiwan', 'UK', 'USA', 'USA/UK/Portugal', 'USSR']
```

Looking through the list of countries, you can probably recognize two problematic entries. It appears that some products were made jointly by the USA, UK and Portugal, and others were made by Portugal and Poland. From these two occurrences it now seems that the format of the country column in the CSV data file should be interpreted as a list of countries separated by slashes and not as a single country.

This presents a new challenge, because there are products that need to be associated with a list of countries, and most countries will very likely have a list of associated products.

In a standard one-to-many relationship, the "many" side adds a foreign key that points to the "one" side, and this is sufficient to establish the relationship. When trying to create a relationship between products and

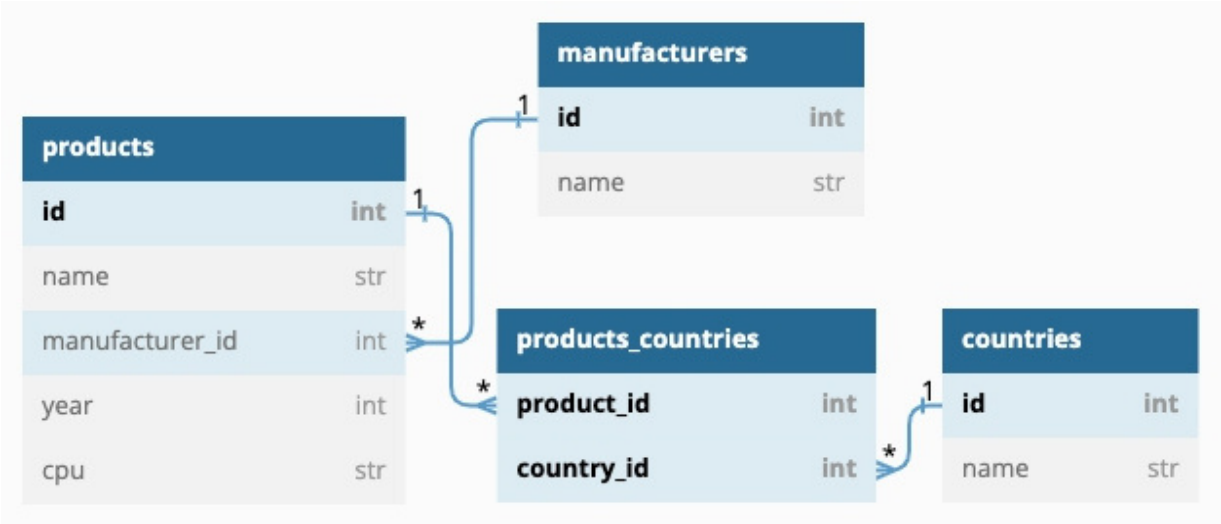
countries, a foreign key to a new countries table in the products table does not work, because that would only allow a single country to be linked to each product. A foreign key to the product in the countries table also fails to work, as that would limit each country to be associated with a single product.

It seems in a many-to-many relationship there is no place to insert a foreign key, right?

How Many-To-Many Relationships Work

The trick to be able to implement a many-to-many relationship requires some out-of-the-box thinking. Instead of a single one-to-many relationship, two one-to-many relationships are needed to fully represent these complex relationships.

Since it is impossible to build a direct many-to-many relationship between the two tables, a third table, called the *join table* is added. Then each side establishes a one-to-many relationship to the join table, which means that foreign keys referencing the two tables are added to it. Below is a diagram of the database structure with the many-to-many relationship between products and countries added.



Here the products_countries table is the join table, which records the foreign keys of each pair of associated product and country. A common naming convention for join tables is to use the names of the two entities that are part of the relationship.

As an example, for a product that was created jointly by three countries,

there's going to be three entries in the join table that have the product_id foreign key set to the product, each having country_id set to one of the countries. And coming from the other side, for a country in which seven products have been created there's going to be that number of entries with country_id set to the country, each linking it to one of the products.

A Simple Many-To-Many Relationship Implementation

Managing all the foreign keys in the join table of a many-to-many relationship might seem like a nightmare, but here once again SQLAlchemy does most of the hard work.

The first step in implementing this relationship is creating the join table.

This table is going to be managed by SQLAlchemy, so all the behavior provided by the Declarative Base class from SQLAlchemy is unnecessary. Instead, this table can be created using the Table class from SQLAlchemy Core. To make it easy to reference this table in the Product and the yet to be created Country models the join table should be added above the model classes in *models.py*.

Listing 15 *models.py*: Products and countries join table

```
from sqlalchemy import Table, Column

ProductCountry = Table(
    'products_countries',
    Model.metadata,
    Column('product_id', ForeignKey('products.id'), primary_key=True,
           nullable=False),
    Column('country_id', ForeignKey('countries.id'), primary_key=True,
           nullable=False),
)
```

The Table constructor takes the table name as first argument, followed by the database metadata instance. The remaining arguments are two Column instances for the foreign keys. Because tables use the Column() constructor instead of typing hints, the nullable=False option is added to make values for these columns required.

Unlike all the other tables derived from models, this table does not have an id primary key, and instead declares the two foreign keys as primary keys. When multiple columns are designated as primary keys, SQLAlchemy creates a *composite* primary key. By definition, primary keys are required to be unique, so for this relationship the join table will not allow two rows that have identical foreign key values.

The updated Product model and the new Country model are shown below:

Listing 16 *models.py*: Many-to-many relationship between products and countries

```
class Product(Model):
    __tablename__ = 'products'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(64), index=True, unique=True)
    manufacturer_id: Mapped[int] = mapped_column(
        ForeignKey('manufacturers.id'), index=True)
    year: Mapped[int] = mapped_column(index=True)
    cpu: Mapped[Optional[str]] = mapped_column(String(32))


    manufacturer: Mapped['Manufacturer'] = relationship(
        back_populates='products')
    countries: Mapped[list['Country']] = relationship(
        secondary=ProductCountry, back_populates='products')

    def __repr__(self):
        return f'Product({self.id}, "{self.name}")'


class Country(Model):
    __tablename__ = 'countries'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(32), index=True, unique=True)

    products: Mapped[list['Product']] = relationship(
        secondary=ProductCountry, back_populates='countries')

    def __repr__(self):
        return f'Country({self.id}, "{self.name}")'
```

There are two changes in the Product model. The country column has been removed, since now countries will be stored in a separate table.

Also, similar to the manufacturer attribute representing the one-to-many relationship, a countries relationship attribute is added to have access to the entities in the other side of the many-to-many relationship, with similar semantics to a list.

The secondary argument to relationship() tells SQLAlchemy that this relationship is supported by a secondary table (the join table). Note that the join table is referenced directly by its name, so it has to be defined above the model classes. The secondary option configures the relationship to work as a many-to-many, with SQLAlchemy automatically adding and removing items from the join table as needed.

The new Country model is very similar to the Manufacturer model, with just a primary key and a name attribute. The products relationship on this model represents the reverse view of the many-to-many relationship, and is also initialized with the join table in the secondary argument.

As before, the back_populates options in the two relationships reference each other, so that SQLAlchemy knows that they are two sides of the same relationship.

Product Importer Script Updates

It's time to make another update to the product importer script, so that countries are properly stored as standalone entities linked to products. The updated script is shown below.

Listing 17 *import_products.py*: Import products, manufacturers and countries from CSV file

```
import csv
from db import Model, Session, engine
from models import Product, Manufacturer, Country


def main():
    Model.metadata.drop_all(engine)  # warning: this deletes all data!
    Model.metadata.create_all(engine)

    with Session() as session:
        with session.begin():
            with open('products.csv') as f:
                reader = csv.DictReader(f)
                all_manufacturers = {}
                all_countries = {}
```

```
            for row in reader:
                row['year'] = int(row['year'])

                manufacturer = row.pop('manufacturer')
                countries = row.pop('country').split('/')
                p = Product(**row)

                if manufacturer not in all_manufacturers:
                    m = Manufacturer(name=manufacturer)
                    session.add(m)
                    all_manufacturers[manufacturer] = m
                all_manufacturers[manufacturer].products.append(p)

                for country in countries:
                    if country not in all_countries:
                        c = Country(name=country)
                        session.add(c)
                        all_countries[country] = c
                    all_countries[country].products.append(p)


if __name__ == '__main__':
    main()
```

The main() function still performs a drop_all() followed by a create_all() as a quick way to reset the database, but keep in mind that recreating the entire database from scratch to make changes gets less viable the more tables there are in the database. Soon you will learn about how to improve this using database migration scripts.

Following a similar pattern to the importing of the manufacturers, now there is also an all_countries dictionary that will maintain the Country instances that are added.

In this version, both the manufacturer and country attributes of the row dictionary that is imported for each line of the CSV file are removed. The remaining items in row are used to initialize the Product instance.

The way the countries attribute is extracted is unusual. Instead of store the string value of this field, the string is converted into a list of country names using split('/'). Recall that the CSV file uses / as a separator when a product belongs to multiple countries.

After the Product instance and manufacturer are created, a loop runs over the list of countries the product belongs to, and the product is added to each country with a familiar append() method on the products relationship of the Country model. It should be noted that given that this is a many-to-

many relationship, there are two equivalent ways to link a product with a country. In the script above, the product is appended to the country:

```
                all_countries[country].products.append(p)
```

If preferred, the two entities can be linked from the other side, by appending the country to the product:

```
                p.countries.append(all_countries[country])
```

Either way, this will tell SQLAlchemy to add an entry to the join table with the foreign keys of the two entities.

Many-To-Many Relationship Queries

As you surely can guess, the products and countries relationship objects added to the Country and Product models respectively also simplify the use of the many-to-many relationship in queries.

To try some queries, begin by starting a Python session and importing the needed components:

```
>>> from db import Session
>>> from models import Product, Manufacturer, Country
>>> from sqlalchemy import select, func
>>> session = Session()
```

Here is how to get a product and its countries:

```
>>> p = session.scalar(
      select(Product)
         .where(Product.name == 'Timex Sinclair 1000'))
>>> p
Product(138, "Timex Sinclair 1000")
>>> p.countries
[Country(1, "UK"), Country(3, "USA"), Country(22, "Portugal")]
```

Here the countries relationship uses the default lazy loader, so it implicitly runs a query to get the list of countries when the attribute is accessed for the first time.

Similarly, a country can report its products:

```
>>> c = session.scalar(
      select(Country)
```

```
        .where(Country.name == 'Portugal'))
>>> c
Country(22, "Portugal")
>>> c.products
[Product(138, "Timex Sinclair 1000"), Product(139, "Timex Sinclair 1500"),
Product(140, "Timex Sinclair 2048"), Product(141, "Timex Computer 2048"),
Product(142, "Timex Computer 2068"), Product(143, "Komputer 2086")]
```

Moving on to something more complex, here is a query that returns all the products that have multiple countries, along with how many countries each has:

```
>>> country_count = func.count(Country.id).label(None)
>>> q = (select(Product, country_count)
        .join(Product.countries)
        .group_by(Product)
        .having(country_count >= 2)
        .order_by(Product.name))
>>> session.execute(q).all()
[(Product(143, "Komputer 2086"), 2), (Product(142, "Timex Computer 2068"), 3),
(Product(138, "Timex Sinclair 1000"), 3), (Product(139, "Timex Sinclair 1500"), 3),
(Product(140, "Timex Sinclair 2048"), 3)]
```

This query uses similar techniques as those you've learned when working with the manufacturer one-to-many relationship. The query returns two values per result row, the product and the count of countries. The latter is created with a label and stored in the country_count variable, so that it can be used in the select() and having() clauses without repetition.

To count the countries it is necessary to join products with countries.

Grouping these results by product collapses the results back to a product per row, but now the second result runs the count() aggregation function and replaces the list of countries with how many there are. The having() clause filters the grouped results and leaves only those that have two or more countries.

The join() method in this query is interesting, because a many-to-many relationship cannot be queried with a single join. In fact, it is not possible in SQL to join the products and countries tables directly, since there are no common attributes in them that can be used.

In reality, a many-to-many relationship requires a two-step join. First the products table is joined with the products_countries join table, and then products_countries is joined with countries. SQLAlchemy does some invisible work here to make this join work.

If you are curious about the internals, below you can see the SQL generated by this query, including the two joins required by the many-to-many relationship:

```
>>> print(q)
SELECT products.id, products.name, products.manufacturer_id, products.year, products.cpu,
count(*) AS count_1
FROM products
JOIN products_countries AS products_countries_1 ON products.id =
products_countries_1.product_id
JOIN countries ON countries.id = products_countries_1.country_id
GROUP BY products.id, products.name, products.manufacturer_id, products.year, products.cpu
HAVING count(*) >= :param_1 ORDER BY products.name
```

The one-to-many and many-to-many relationships can be used together, and this opens the door to even more interesting queries. The next query obtains the list of manufacturers that operate in the UK:

```
>>> q = (select(Manufacturer)          .join(Manufacturer.products)
        .join(Product.countries)
                                          .where(Country.name    ==    'UK')
        .order_by(Manufacturer.name)
        .distinct())
>>>  session.scalars(q).all()  [Manufacturer(1,  "Acorn  Computers  Ltd"),
Manufacturer(2, "Amstrad"), ...,
Manufacturer(70, "Timex Sinclair")]
```

The goal of this query is to return a list of manufacturers, so that is the only model that is added to the select() statement. But the query needs access to countries, and countries have no direct connection with manufacturers. The only solution is to navigate the available relationships until a connection is achieved. In this case, first manufacturers are joined with their products, and then products are joined with their countries. The result of this chain of joins is that the query now has access to all the valid (manufacturer, product, country) triplets, and can add filters on any of these, for example to only keep the triplets with UK as the country.

The distinct() clause is necessary in this case because many of those

triplets that have the country set to UK are going to have the same manufacturer, and unless told otherwise, the database will return all of them as rows, leading to duplicate results.

Grouping also works across a chain of relationships. The next query gets a

list of manufacturers that operate in more than one country, along with the country count:

```
>>> country_count = func.count(Country.id.distinct()).label(None)
>>> q = (select(Manufacturer, country_count)
         .join(Manufacturer.products)
         .join(Product.countries)
         .group_by(Manufacturer)
         .having(country_count >= 2))
>>> session.execute(q).all()
[(Manufacturer(70, "Timex Sinclair"), 4)]
```

The country_count aggregation in this case needs to be expanded with a distinct() clause, because a simple count of rows would include duplicate entries that result from the joins.

If you fail to see why joins generate duplicates, an example should help clarify it. Let's assume that company Acme has two products A and B, both made in the USA. The chained join used in the query above would generate the following triplets for this company:

Manufacturer	Product	Country
Acme	A	USA
Acme	B	USA

The group_by(Manufacturer) clause in the query would cause these two rows to be collapsed into one, since both have Acme as the manufacturer. A plain func.count() function given as the second value in the query would count the number of rows that were merged, which in this example is 2, even though there is a single country at play. The distinct() clause asks the database to remove duplicates from the count.

Deleting From Many-To-Many Relationships

Many-to-many relationships that are configured with the secondary option have the advantage that SQLAlchemy does all the maintenance work on the join table, and this extends to deletions. When an entity is deleted, SQLAlchemy finds all the entities on the other side of the relationship that linked to it and removes those links.

The following example deletes a country, which makes that country automatically disappear from products that had it listed:

```
>>> c = session.get(Country, 22)
>>> c
Country(22, "Portugal")
>>> p = session.get(Product, 138)
>>> p
Product(138, "Timex Sinclair 1000")
>>> p.countries
[Country(1, "UK"), Country(3, "USA"), Country(22, "Portugal")]
>>> session.delete(c)
>>> session.commit()
>>> p.countries
[Country(1, "UK"), Country(3, "USA")]
```

It is also possible to detach entities that are connected through a many-to-many relationship, without deleting any of the entities themselves. The detachment can be issued from either side of the relationship using list semantics:

```
>>> c = session.get(Country, 1)
>>> c
Country(1, "UK")
>>> p = session.get(Product, 138)
>>> p
Product(138, "Timex Sinclair 1000")
>>> p.countries
[Country(1, "UK"), Country(3, "USA")]
>>> c in p.countries
True
>>> p.countries.remove(c)
>>> session.commit()
>>> p.countries
[Country(3, "USA")]
>>> c in p.countries
False
```

The above example removes a country from a product. The same result can be obtained by removing the product from the country:

```
>>> c.products.remove(p)
```

The one-to-many relationship between manufacturers and their products was configured with a non-nullable foreign key to force all products to have a manufacturer linked, effectively making it impossible for a product to exist without having a manufacturer. With a many-to-many relationship there is no automatic way to enforce that every entity has at least one link to an entity on the other side. The example below removes the only country associated with a product, leaving the product without any related countries:

```
>>> c = session.get(Country, 1)
>>> c
Country(1, "UK")
>>> p = session.get(Product, 1)
>>> p
Product(1, "Acorn Atom")
>>> p.countries
[Country(1, "UK")]
```

```
>>> c.products.remove(p)
>>> session.commit()
>>> p.countries
[]
```

For a many-to-many relationship for which it is desired that there always is at least one linked entity the application must manually perform validation checks and prevent the last link from being removed.

Database Migrations

The chapters that follow are going to introduce more tables and relationships. Before increasing the complexity of the database it would be a good idea to set up a robust mechanism to make updates, since the drop_all() and create_all() functions used until this point are very limited in that they require all the data to be re-imported.

Introduction to Alembic

In this section you are going to learn how to use Alembic, the database migration tool that is part of the SQLAlchemy family. Install this package in your virtual environment with pip:

```
(venv) $ pip install alembic
```

The first step to enable database migrations in your project is to create a migration repository with the alembic init command:

```
(venv) $ alembic init migrations
```

The migrations argument is the name of a subdirectory that is created under the project directory, where database migration scripts will be stored. In addition to this subdirectory, an *alembic.ini* file is created in the project directory. For a project that is under source control, the *alembic.ini* file and the entire contents of the *migrations* subdirectory should be treated as source code and maintained along with the application's source files.

The files that are created by the alembic init command do not have any initial awareness of what database the project is using. To point Alembic at the project's database, a few simple configuration changes need to be made.

There isn't a single way to do this, but for the RetroFun project the most convenient option is to edit the *env.py* file located inside *migrations*.

Open *migrations/env.py* in your code editor. At the top of the file, import engine and Model from *db.py* and the models:

Listing 18 *migrations/env.py*: Import Model and engine

```
from db import Model, engine
import models
```

Then locate the line that reads:

```
target_metadata = None
```

Replace this line with the following code:

Listing 19 *migrations/env.py*: Configure the project's database into Alembic

```
target_metadata = Model.metadata
config.set_main_option("sqlalchemy.url", engine.url.render_as_string(
    hide_password=False))
```

The target_metadata variable is where Alembic expects the metadata instance used by the application. For SQLAlchemy ORM, this instance exists as a metadata attribute of the Model declarative base class.

The second line inserts a value for the sqlalchemy.url option in the Alembic configuration object. This option is where Alembic goes to obtain the URL of the database. Since the application creates an engine instance with this URL, the most convenient (though maybe not the most efficient) is to get the URL from this object.

If you open the *alembic.ini* file, you will see that this same sqlalchemy.url option is initialized with a placeholder URL. If you prefer, you can enter the URL in this file instead of editing with config.set_main_option() in *env.py*. Both set the same configuration variable, but doing it in *env.py* as shown above has the benefit that the database URL does not need to be written in two different places.

There is one more change to make, which is important when using the SQLite database. Scroll down on *env.py* until you find the run_migrations_online() function, close to the end of the file. This function makes a context.configure() function call that looks like this:

```
    context.configure(
        connection=connection, target_metadata=target_metadata
    )
```

This call can be used to configure Alembic, and in particular its migration generator. The change shown below enables the render_as_batch option.

Listing 20 *migrations/env.py*: Configure the migration generator

```
    context.configure(
        connection=connection, target_metadata=target_metadata,
        render_as_batch=True,
    )
```

The render_as_batch option is needed to address migration limitations of the SQLite database. When this option is enabled, if an unsupported migration operation is requested on a SQLite database, Alembic will transparently create a new table with the change, move all the data over and finally delete the outdated table. For other databases this option does not have any effect, so it does not hurt to enable it too, but if you prefer, it is also okay to omit it.

You may have noticed that the import models statement added near the top of *env.py* appears to be unnecessary, since none of the model classes are referenced directly. If you are using a code checker, it will likely flag this import as unnecessary. Regardless of your code checking tool tells you, this import statement needs to be in the file. SQLAlchemy only learns about the models in your application when they are imported. If the models are not imported, then these models will not exist in the memory of the Python process, and SQLAlchemy and Alembic will not know about them.

Create a Migration Script

Alembic uses the concept of *migration scripts* to track changes that are made to the database. A migration script contains Python code that makes changes to the live database, without removing any tables or data.

Ready to generate your first database migration? Alembic can auto-generate a migration script by comparing the model classes against their corresponding database tables. For example, if the table referenced by a model class does not exist in the database, Alembic decides that this is a new table that needs to be created in the database, matching the definitions in the model class. If a table that exists in the database is not referenced by any model class in the application, then it decides that the table has to be deleted, and generates the code to do so in the migration script. The goal for the generated migration script is always to make any necessary changes in the database so that it reflects the state of the models.

To generate an initial migration, it is necessary that the database is completely empty, since this is what will make Alembic generate a migration that creates the products, manufacturers, countries and products_countries tables. Open a Python shell, and call the drop_all() function one last time:

```
>>> from db import Model, engine
>>> import models
>>> Model.metadata.drop_all(engine)
```

The import models line in the above code is there for the same reason it was included in the *env.py* script, which is to let SQLAlchemy know what the models are.

After running the above statements, the database is going to be completely empty. Now you can generate the first database migration script with the following command:

```
(venv) $ alembic revision --autogenerate -m "products, manufacturers, countries"
```

Alembic is going to print some logs to the terminal indicating that it has detected new tables and new indexes. Then at the bottom, it will show the name of the generated migration script, which will have the format *{code}_products_manufacturers_countries.py*, where {code} is a unique code that identifies the migration. The rest of the name is created from the description given in the -m option.

The generated migration script is a Python module that has two functions upgrade() and downgrade(). The upgrade() function applies changes to

make the database match the models, while the downgrade() function reverts these changes. Each migration script will have these two functions, making it possible for Alembic to make a chain of upgrades, or a chain of downgrades by calling the corresponding functions in the correct order.

Note

The method Alembic uses to auto-generate migration scripts is extremely convenient, but is not foolproof, so it is always best to review auto-generated scripts to make sure they are correct.

Database Upgrade

The alembic revision command creates a migration script, but it does not execute the script. The RetroFun database is, at this point, still empty. The next command runs the migration script:

```
(venv) $ alembic upgrade head
```

With this command, the changes in the upgrade() function are executed against the configured database, and now all the tables and indexes are created similarly to the create_all() function, but with a more robust solution that will also be able to apply partial changes going forward.

The alembic command has many sub-commands including downgrade to undo a database migration, current to show which migration the database is at, history to see the list of migrations, and more. Feel free to review the Alembic documentation to learn more about them.

A Migration-Aware Product Importer

The database migration does not have any knowledge of the data that the application wants to store in the tables, so it created all the tables without any data in them. The *import_products.py* script can now be used to insert the products, manufacturers and countries, but the drop_all() and create_all() function calls at the start of the main() function need to be removed, since these conflict with Alembic, which is now in charge of creating and maintaining the database structure.

Instead of dropping and recreating the tables, the importer script will now attempt to delete all the rows in all the tables, so that it can import them again from the CSV file. Here is the updated script:

Listing 21 *import_products.py*: Import data without recreating tables

```
import csv
from sqlalchemy import delete
from db import Session
from models import Product, Manufacturer, Country, ProductCountry


def main():
    with Session() as session:
        with session.begin():
            session.execute(delete(ProductCountry))
            session.execute(delete(Product))
            session.execute(delete(Manufacturer))
            session.execute(delete(Country))

    with Session() as session:
        with session.begin():
            with open('products.csv') as f:
                reader = csv.DictReader(f)
                all_manufacturers = {}
                all_countries = {}

                for row in reader:
                    row['year'] = int(row['year'])

                    manufacturer = row.pop('manufacturer')
                    countries = row.pop('country').split('/')
                    p = Product(**row)

                    if manufacturer not in all_manufacturers:
                        m = Manufacturer(name=manufacturer)
                        session.add(m)
                        all_manufacturers[manufacturer] = m
                    all_manufacturers[manufacturer].products.append(p)

                    for country in countries:
                        if country not in all_countries:
                            c = Country(name=country)
                            session.add(c)
                            all_countries[country] = c
                        all_countries[country].products.append(p)


if __name__ == '__main__':
    main()
```

The first session block in the main() function makes use of the delete() function from SQLAlchemy to delete all the entities in the products, manufacturers and countries tables plus the products_cou table. The rest of the script did not change from the previous version.

You can now run the importer script to load the CSV data one last time:

```
(venv) $ python import_products.py
```

Exercises

Ready to create some queries on your own? Write queries that generate:

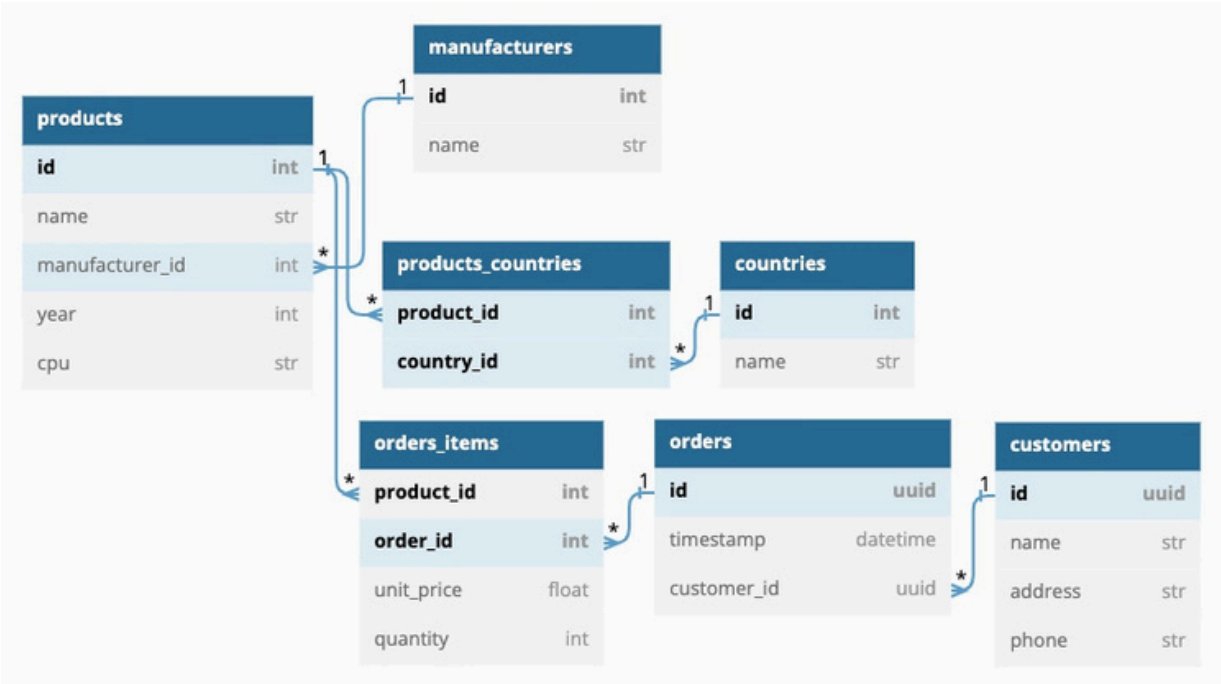
1. Products that were made in UK or USA.
2. Products not made in UK or USA. Products that were made in UK and/or USA jointly with other countries should be included in the query results.
3. Countries with products based on the Z80 CPU or any of its clones.
4. Countries that had products made in the 1970s in alphabetical order.
5. The 5 countries with the most products. If there is a tie, the query should pick the countries in alphabetical order.
6. Manufacturers that have more than 3 products in UK or USA.
7. Manufacturers that have products in more than one country.
8. Products made jointly in UK and USA.

Advanced Many-To-Many Relationships

You have now learned the design blocks used in relational databases. Sometimes, however, these building blocks have to be "tweaked" a bit to achieve a desired goal. This chapter is dedicated to exploring a very useful variation on the many-to-many relationship.

Many-To-Many Relationships with Additional Data

In this section you are going to add the RetroFun ordering subsystem to the database, which includes customers and orders. The next diagram shows how the new tables and relationships fit in the existing database.



The new customers and orders tables have a one-to-many relationship between them, with customers on the "one" side. The products and `orders`

tables have a many-to-many relationship, with the orders_items table as the join table. But the join table in this relationship has additional columns besides the two foreign keys.

This join table in this relationship effectively represents each line item in an order, with references to the order and the product. But this isn't sufficient, the relationship also needs to define a quantity and a unit price for the sale.

Having additional data in the join table complicates things. The many-to-many relationship between products and countries did not have any extra columns, and that allowed SQLAlchemy to take full control to insert or delete entries from this table as needed. When there are additional columns, how would SQLAlchemy know what to write in those extra columns when it needs to insert a new item?

Many-to-many relationships with extra columns in the join table need to use a less automatic workflow with SQLAlchemy, because the application has to provide values for the additional columns that are part of the relationship when two entities are linked.

As a first step to implement orders for RetroFun, the application can be expanded with the Customer and Order models, and a one-to-many relationship between them. These are added at the bottom of *models.py*.

Listing 22 *models.py*: Orders and customers

```
from datetime import datetime
from uuid import UUID, uuid4
from sqlalchemy.orm WriteOnlyMapped

# ...

class Order(Model):
    __tablename__ = 'orders'

    id: Mapped[UUID] = mapped_column(default=uuid4, primary_key=True)
    timestamp: Mapped[datetime] = mapped_column(default=datetime.utcnow,
                                    index=True)
    customer_id: Mapped[UUID] = mapped_column(ForeignKey('customers.id'),
                                    index=True)

    customer: Mapped['Customer'] = relationship(back_populates='orders')

    def __repr__(self):
        return f'Order({self.id.hex})'


class Customer(Model):
    __tablename__ = 'customers'
```

```
    id: Mapped[UUID] = mapped_column(default=uuid4, primary_key=True)
    name: Mapped[str] = mapped_column(String(64), index=True, unique=True)
    address: Mapped[Optional[str]] = mapped_column(String(128))
    phone: Mapped[Optional[str]] = mapped_column(String(32))

    orders: WriteOnlyMapped['Order'] = relationship(back_populates='customer')

    def __repr__(self):
        return f'Customer({self.id.hex}, "{self.name}")'
```

There are a few new things in these models. The following sections cover them one by one.

UUID Primary Keys

The most important difference in these two new models is that their id columns are defined with the UUID type instead of the int keys used in all the previous models.

Note

Support for the UUID type was introduced in SQLAlchemy 2.0. The documentation includes an implementation of a custom UUID type that can be used with older releases.

The problem with the auto-incrementing integer primary keys used earlier is that when they are included in URLs or emails, they indirectly allow people to estimate the size of the database tables they reference. Most business will probably prefer to keep the number of customers or orders they have private, so using integer keys for these tables is not a good idea.

A way to avoid giving away this type of information is to switch away from numeric primary keys. The new models define id as a UUID, which is a 16 byte binary sequence. There are several types of UUIDs and one in particular, called UUID4, is a good choice for primary keys. In case you are not familiar with UUID4 support in Python, the following Python session shows how to generate them and print them:

```
>>> from uuid import uuid4
>>> id = uuid4()
>>> id.bytes
b'kF\xf8\xe9o\x94MM\xad\xfe\x15\xe1\xeb\xb1\xd0\xac'
>>> id.hex
'6b46f8e96f944d4dadfe15e1ebb1d0ac'
```

Unfortunately SQLAlchemy can only manage integer primary keys automatically. To avoid having to remember to create and assign a UUID4 to each new Order or Customer instance that is added, the id columns in these two models are given a default argument, which automatically assigns values to them if the application doesn't.

The default argument can be set to a constant value or to a callable, such as a function. If it is set to a callable, SQLAlchemy will call it every time it needs to give a default value to a column. The id columns above pass the uuid4 function as default, so that each new item gets its own newly generated UUID4. When passing functions as default column values it is important to remember to not include the () after the function name. SQLAlchemy needs the reference to the function itself, so that it can call it when a value needs to be generated.

The __repr__() methods in both classes use the hex attribute of the UUID object to decode the binary UUID4 into a printable hexadecimal string for debugging, as shown in the Python example above.

The one-to-many relationship between customers and orders is established by adding a foreign key on the "many" side, which in this case is the Order model. This is the customer_id column, which is also given a UUID type to match the type of Customer.id.

Date and Time Columns

The Order model features a timestamp column declared with a datetime type. In the same way as with the UUID type, SQLAlchemy automatically maps Python datetime objects to the appropriate type in the database.

The timestamp column also has a default argument. Passing datetime.utcnow as a default will make SQLAlchemy call datetime.utcnow() each time a new order is added to the database. This effectively means that the timestamp column can be left unassigned when adding an order, and the current date and time will be automatically set during the commit operation. Once again the () are not included in the

default argument so that SQLAlchemy receives a reference to the function, without calling it.

When working with date and time values in a database it is important to maintain a uniform timezone. The most sane approach is to store all timestamps in the UTC timezone, and for that reason the default is set to utcnow instead of now. The application can convert UTC timestamps retrieved from the database to the timezone of the client when they are presented to the user. For web applications this is often done in the web browser, which has access to the user's timezone.

Write-Only Relationships

The relationship attributes on these models are configured similarly to the relationship between manufacturers and products. The Order.customer relationship uses the default lazy evaluation, but the Customer.orders relationship has a new definition:

```
    orders: WriteOnlyMapped['Order'] = relationship(back_populates='customer')
```

The WriteOnlyMapped typing hint is used here to define a relationship configured with the lazy='write_only' option. When typing isn't used or desired, the lazy argument can be given in the relationship() call.

Why is this relationship different? In general, lazy evaluation of relationships makes sense only in certain cases. Relationships that load a "one" side, such as Order.customer that may only be casually used are perfect to evaluate lazily, and so are those that are expected to have a short number of elements, such as the Product.countries relationship from the previous chapter.

The reason why Customer.orders is defined as a write_only relationship is that a lazily evaluated one isn't very useful for it. This could potentially be a long relationship for repeat customers, and getting the entire collection of orders from a customer as a list is unlikely to be very useful. What would work for this relationship is to have a way to request only some of the customer's orders, such as the most recent one, or the orders made in the

last month. But a lazily loaded relationship does not provide a way to define any filters that can configure a range of elements to retrieve.

What makes the write_only loader convenient is that it does not attempt to load the relationship, it just generates a query object that you can execute yourself, possibly after adding filters, sorting or any other option you may need. The relationship object also has add(), add_all() and remove() methods that can be used to add or remove elements from the relationship.

How does a write_only relationship work in practice? You will see this relationship in action later in this chapter.

Association Object Pattern

The next step is to create the many-to-many relationship between the Order and Product models, which will define the contents of each order. For a simple many-to-many relationship with extra columns the join table can be created as a Table instance and given to SQLAlchemy to manage. Because this relationship needs extra data, the join table is created as a Model subclass, to allow the application to manage the additional columns. SQLAlchemy calls this alternative method to define a many-to-many relationship the Association Object Pattern.

Below you can see the new join table, which is added at the bottom of *models.py*.

Listing 23 *models.py*: Order item model

```
class OrderItem(Model):
    __tablename__ = 'orders_items'

    product_id: Mapped[int] = mapped_column(ForeignKey('products.id'),
                                primary_key=True)
    order_id: Mapped[UUID] = mapped_column(ForeignKey('orders.id'),
                                primary_key=True)
    unit_price: Mapped[float]
    quantity: Mapped[int]
```

The primary key of the OrderItem model is made up of the two foreign key columns, similar to what was done in the simpler many-to-many relationship of the previous chapter. By not having the Optional typing hint, both keys are required (non-nullable in database jargon), which means

that if a product or an order are deleted, any entries in this table referencing the deleted item must be removed.

The model class has two additional columns to store the unit price and quantity of the ordered product, which are necessary to have all the information associated with the purchase.

Now comes the interesting part. The simpler many-to-many relationship between products and countries used the secondary argument to relationship on both models, and this was enough for SQLAlchemy to know what to do. Unfortunately, secondary cannot be used in this case because the application cannot give up control of the join table because of the extra columns.

The solution that makes this work can be derived from the fact that a many-to-many relationship can be thought of as two one-to-many relationships, or more accurately, as a one-to-many relationship between the first model and the join table, and a many-to-one relationship between the join table and the second model. It turns out that if you think about the two halves of this relationship separately, then there is no need to rely on the secondary argument.

Below you can see how the two sides of the many-to-many relationship can be implemented using one-to-many relationship objects.

Listing 24 *models.py*: Decomposed many-to-many relationship

```
class Product(Model):
    # ...
    order_items: WriteOnlyMapped['OrderItem'] = relationship(
        back_populates='product')
    # ...

class Order(Model):
    # ...
    order_items: Mapped[list['OrderItem']] = relationship(
        back_populates='order')
    # ...

class OrderItem(Model):
    # ...
    product: Mapped['Product'] = relationship(back_populates='order_items')
    order: Mapped['Order'] = relationship(back_populates='order_items')
    # ...
```

The result is that four relationship attributes are added, two for each one-to-many half of the many-to-many relationship. An Order instance can use its order_items relationship attribute to get the list of line items included in the order. Each element in the list returned by this relationship is going to be an instance of the OrderItem join table model, which provides access to the product, unit price and quantity.

Looking at the relationship from the product side, the order_items relationship attribute for a product represents the list of purchases, also instances of OrderItem, each providing a reference to the order (which in turn links to the customer), the unit price and quantity. Because a product is likely to have been sold many times, this part of the relationship is defined with the write_only loader, which would allow the application to query this list with filters, sorting options and pagination.

A New Database Migration

Now that the changes that implement customers and orders in *models.py* are complete, it is time to migrate the database, so that it receives these changes. Create a second database migration with the following command:

```
(venv) $ alembic revision --autogenerate -m "customers and orders"
```

After making sure that the generated migration script includes the three new tables, apply it to the database:

```
(venv) $ alembic upgrade head
```

At this point the database should be ready to accept customers and orders.

How to Create an Order

Many-to-many relationships with additional columns are extremely powerful, but they have the downside that they require more work compared to the simpler kind, since the join table has to be managed by the application.

How do you create an order with this solution? Here are the steps:

- First, create a Customer instance, or load an existing one if this is a repeat customer.
- Next, create an Order instance and associate it with the customer, either by passing the customer argument in the constructor, or by calling add() on the Customer.orders relationship.
- For each line item in the order, create an OrderItem instance with the product, unit price and quantity, and append it to the order's order_items relationship.

If you prefer to see this in terms of actual code, the following session writes an order with two items in it:

```
>>> # import all the necessary things and create a session
>>> from models import Product, Customer, Order,
OrderItem >>> from db import Session
>>> session = Session()


>>> # create a new customer
>>> c = Customer(name='Jane Smith')

>>> # create a new order, add it to the customer and to the database session
>>> o = Order()
>>> c.orders.add(o)
>>> session.add(o)

>>> # add the first line item in the order: product #45 for $45.50
>>> p1 = session.get(Product, 45)
>>> o.order_items.append(OrderItem(product=p1, unit_price=45.5, quantity=1))

>>> # add the second line item: 2 of product #82 for $37 each
>>> p2 = session.get(Product, 82)
>>> o.order_items.append(OrderItem(product=p2, unit_price=37, quantity=2))

>>> # write the order (along with the customer and order items) to the database
>>> session.commit()

>>> # check the UUID and the timestamp defaults assigned to the new order
>>> o.id
UUID('a73c6aad-8ba9-4550-ac2f-1fcc9285cddc')
>>> o.timestamp
datetime.datetime(2023, 2, 24, 19, 52, 47, 293727)
```

Something that is worth noticing is that the methods used to add and remove items from a relationship are not always the same. For relationships that present themselves as standard Python lists, the append() and remove() methods are used. You can see how the Order.order_items relationship is used with append() in the example above.

Relationships that use the write_only loader do not follow list semantics because the related items are never loaded directly. These relationships are

of type `WriteOnlyCollection`, and have `add()` and `delete()` methods. The above example uses `add()` on the `Customer.orders` relationship.

Deletions

As in previous relationships, it is important to understand what happens when an entity is deleted. If a Product or Order entity that is referenced in at least one OrderItem entry is deleted, SQLAlchemy will attempt to write a NULL in the invalidated foreign key of the OrderItem instance. Both foreign keys in this model are intentionally defined as required (non-nullable), so this attempt will fail. This was not a concern with the products and countries relationship because the secondary option of the relationship ensures that elements are removed from the join table when one of the linked entities is deleted.

There are two options to deal with this problem on many-to-many relationships that do not make use of the secondary option. One possible solution is to implement a "delete" cascade similar to the one configured in the Manufacturer.products relationship, which would automatically delete orphaned OrderItem entries, similarly to what SQLAlchemy does on secondary type relationships. Another option is to assume that products or orders cannot be deleted when there are OrderItem instances that reference them.

The second option is essentially a "do nothing" option, because SQLAlchemy returns an error when a product or an order are deleted. Making the foreign keys in the join table required is a convenient method to ensure that products or orders can only be deleted if there are no OrderItem instances pointing to them. To be able to delete an order, the application first needs to delete all the OrderItem entities associated with it, and only then the Order entity can be removed. For this relationship, this is the approach that will be taken.

Can OrderItem instances be deleted? Since no other entity in the database has foreign keys pointing to them, these entities can be deleted safely without any risk of database integrity errors. The OrderItem entities can be deleted from either side of the relationship, by removing the instances from

the corresponding `order_items` relationship in the `Order` or `Product` models. Here is an example:

```
>>> # this assumes "oi" has the OrderItem instance to delete,

>>> # delete the OrderItem from an Order instance "o"
>>> o.order_items.remove(oi)
>>> session.commit()

>>> # delete the OrderItem from a Product instance "p"
>>> p.order_items.delete(oi)
>>> session.commit()
```

An Order Importer Script

To make it possible to run queries and experiment with a database full of orders, the following script imports a large number of randomly generated orders from a CSV file. Copy the following code to a file named *import_orders.py* in the project directory.

Listing 25 *import_orders.py*: Import customers and orders from a CSV file

```
import csv
from datetime import datetime
from sqlalchemy import select, delete
from db import Session
from models import Product, Customer, Order, OrderItem


def main():
    with Session() as session:
        with session.begin():
            session.execute(delete(OrderItem))
            session.execute(delete(Order))
            session.execute(delete(Customer))

    with Session() as session:
        with session.begin():
            with open('orders.csv') as f:
                reader = csv.DictReader(f)
                all_customers = {}
                all_products = {}

                for row in reader:
                    if row['name'] not in all_customers:
                        c = Customer(name=row['name'], address=row['address'],
                                     phone=row['phone'])
                        all_customers[row['name']] = c
                    o = Order(
                        timestamp=datetime.strptime(row['timestamp'],
                                                    '%Y-%m-%d %H:%M:%S'))
                    all_customers[row['name']].orders.add(o)
                    session.add(o)

                    product = all_products.get(row['product1'])
                    if product is None:
                        product = session.scalar(select(Product).where(
                            Product.name == row['product1']))
                        all_products[row['product1']] = product
                    o.order_items.append(OrderItem(
                        product=product,
```

```
                unit_price=float(row['unit_price1']),
                quantity=int(row['quantity1'])))

            if row['product2']:
                product = all_products.get(row['product2'])
                if product is None:
                    product = session.scalar(select(Product).where(
                        Product.name == row['product2']))
                    all_products[row['product2']] = product
                o.order_items.append(OrderItem(
                    product=product,
                    unit_price=float(row['unit_price2']),
                    quantity=int(row['quantity2'])))

            if row['product3']:
                product = all_products.get(row['product3'])
                if product is None:
                    product = session.scalar(select(Product).where(
                        Product.name == row['product3']))
                    all_products[row['product3']] = product
                o.order_items.append(OrderItem(
                    product=product,
                    unit_price=float(row['unit_price3']),
                    quantity=int(row['quantity3'])))


if __name__ == '__main__':
    main()
```

A lot of the techniques used in this script are similar to those used in the product importer script. The first session block deletes all orders and customers to start from clean tables. Then in the second session block the CSV file is read and processed row by row.

Each line in the data file has information for an order, with the following fields:

- name: the customer's name.
- address: the customer's address.
- phone: the customer's phone number.
- timestamp: the date and time of the order.
- product1, unit_price1 and quantity1: the first line item in this order.
- `product2`, `unit_price2` and `quantity2`: the second line item in this order, or a "0" value for `quantity2` to indicate that there is no second line item in this order.

- product3, unit_price3 and quantity3: the third line item in this order, or a "0" value for quantity3 to indicate that there is no third line item in this order.

The all_customers dictionary keeps track of the Customer instances that are created as orders are being read from the file, while the all_products dictionary maintains a cache of products that were loaded from the database, to reduce the number of queries issued.

An Order instance is created and linked to the customer. The timestamp attribute is explicitly set to the date and time imported from the CSV file, to prevent the default clause in this model from setting the timestamps of all the orders to the time at which the script is running.

To complete the order, one, two or three OrderItem instances are created and appended to it. For each order item, the product is loaded from the database with a query if it hasn't been seen before, or from the all_products cache when it is found there.

Before you run this script, a copy of the *orders.csv* file must be placed in the project directory. You can download this file from the book's GitHub repository.

Run the following command to execute the script and import all the orders:

```
(venv) $ python import_orders.py
```

This script should take a few seconds to import the orders.

Queries

With a database full of customers and orders, it is now possible to create a large variety of interesting queries. To begin, import all the needed classes and functions into a brand-new Python shell, and also create a database session:

```
>>> from sqlalchemy import select, func
>>> from db import Session
>>> from models import Product, Customer, Order, OrderItem
>>> session = Session()
```

Let's try out the new write_only relationship. The next query grabs a customer by name, then accesses the orders relationship attribute:

```
>>> c = session.scalar(
      select(Customer)
         .where(Customer.name == 'John Butler'))
>>> c
Customer(14a4d407bca54a69a0118715b664032a, "John Butler")
>>> c.orders
<sqlalchemy.orm.writeonly.WriteOnlyCollection object at 0x10ae1be20>
```

As you see, this type of relationship does not present itself as a list. But this WriteOnlyCollection object has a select() method that returns a query that retrieves the related objects when it is executed:

```
>>> session.scalars(c.orders.select()).all()
[Order(2971e3d105aa4394822c227da3f4a743), ..., Order(6e5214f6af744d02bea74a6228dec725)]
```

You can craft the same query by hand, but by having the write_only relationship this query is generated by SQLAlchemy. And because this is a query, it can be extended with additional clauses, unlike the list-type relationships you've seen before. Here are some examples of that:

```
>>> # sort the orders from newer to older
>>> session.scalars(
      c.orders.select()
         .order_by(Order.timestamp.desc())
   ).all()
[Order(4cbd2174ee6a4f52bc89a65ff74942d2), ..., Order(6e5214f6af744d02bea74a6228dec725)]

>>> # get one order at most
>>> session.scalar(
      c.orders.select()
         .limit(1))
Order(2971e3d105aa4394822c227da3f4a743)
```

Note that all these queries that print orders will show different primary key values on your own system, since the UUIDs are randomly generated when the orders are imported.

How many customers and orders are there in the system? The following queries obtain these counts:

```
>>> session.scalar(select(func.count(Customer.id)))
2754
>>> session.scalar(select(func.count(Order.id)))
4728
```

Each OrderItem instance contains the product's unit price and the quantity for a line item of an order, but they do not include the total price for the

item, which needs to be calculated. The next query lists the three highest order item amounts, along with the product ordered.

```
>>> item_total = (OrderItem.unit_price * OrderItem.quantity).label(None)
>>> q = (select(item_total, Product)
        .join(Product.order_items)
        .order_by(item_total.desc())
        .limit(3))
>>> session.execute(q).all()
[(385.95000000000005, Product(127, "ZX Spectrum")), (283.16, Product(127, "ZX Spectrum")),
(259.98, Product(127, "ZX Spectrum"))]
```

To solve this query the price calculation needs to be made by multiplying the product's unit price by the quantity ordered. This is expressed by the item_total variable, which stores a labeled version of this calculation.

Note that the amounts returned by the query above may have very small variations depending on which database and database version you use. This happens due to the inaccurate nature of floating point math.

The requirements for this query are to list the order item's total price and the product, so those are the two arguments given to select(). Since these two arguments come from different tables in the database, a join is required. The Product.order_items argument to join() tells SQLAlchemy that this query will join the left-side entity, which is Product, with the right-side entity, which is OrderItem. It would be equivalent to use Order.order_items, the reverse relationship, and then the position of the tables in the join will be swapped, but the results would be the same.

The most interesting part of this query is that SQLAlchemy understands that the multiplication of the two columns stored in item_total is meant to be executed in the query instead of in the Python process. This can cause some confusion, as it is a somewhat "magical" behavior of SQLAlchemy. The columns attributes have their own custom implementation of the mathematical operators that do not really perform any calculations but instead transfer the operations to the SQL query, so that they are executed by the database.

If you are curious to see what is the SQL that is generated from this query, go ahead and print the query:

```
>>> print(q)
SELECT orders_items.unit_price * orders_items.quantity AS anon_1,
```

```
  products.id, products.name, products.manufacturer_id, products.year, products.cpu
FROM products JOIN orders_items ON products.id = orders_items.product_id
  ORDER BY anon_1 DESC LIMIT :param_1
```

This was an interesting query with some new challenges, but in reality it isn't very useful to look at individual order items, because these are part of an order, and orders can have many order items that were purchased together. A much more useful query would consider the total sale price of an order, adding all the order items. The next query finds the three most expensive orders, considering all their items combined. This may appear to be much harder to do, but the query is surprisingly similar to the one above:

```
>>> order_total = func.sum(OrderItem.unit_price * OrderItem.quantity).label(None)
>>> q = (select(Order, order_total)
          .join(Order.order_items)
          .group_by(Order)
          .order_by(order_total.desc())
          .limit(3))
>>> session.execute(q).all()
[(Order(a3e5d5187a7d420a8086dec947721a1c), 463.99),
(Order(4b659023464b43688f4eb49cc19cc787), 461.51),
(Order(8731df42c5fb45e7a90232d67dab3f9a), 443.3)]
```

The trick to make this query work is to use grouping, along with the sum() aggregation function. The query retrieves orders joined with their order items, and the results are grouped by the order, so that all the order items belonging to an order are collapsed into one result that can be aggregated.

The item_total calculation used in the previous query is replaced with order_total here, which applies the same multiplication to each order item, but given that the items in this query are grouped, they can be added together with the sum() function to obtain the order's grand total.

Note that depending on which database you are using and due to variations in the database drivers, the results of the func.sum() aggregation function may be returned as decimal objects, which represent numbers more accurately than the standard floating point arithmetic.

The next query finds the five products that sold the most units:

```
>>> units = func.sum(OrderItem.quantity).label(None)
>>> q = (select(Product, units)
          .join(Product.order_items)
          .group_by(Product)
          .order_by(units.desc())
          .limit(5))
>>> session.execute(q).all()
[(Product(41, "Commodore 64"), 2023), (Product(48, "Amiga"), 1578),
```

```
(Product(127, "ZX Spectrum"), 1004), (Product(16, "Apple II"), 600),
(Product(2, "BBC Micro"), 209)]
```

For this query, a labeled expression is created on the sum of the quantities of all the grouped order items. The query retrieves products joined with order items and groups the items by product.

Here once again you may see the sums reported as instances of the Decimal class from Python.

The one thing that is missing in the queries above is date ranges. Normally a business wants to calculate their sales statistics during a specific period such as a month or a quarter. The query above that returns the highest priced orders can be constrained to operate within a date range with an added between() condition in a where() clause. Here is how to calculate the top 3 orders in November 2022:

```
>>> from datetime import datetime >>> order_total =
func.sum(OrderItem.unit_price * OrderItem.quantity).label(None)
>>> q = (select(Order, order_total)
        .join(Order.order_items)          .where(Order.timestamp.between(
            datetime(2022, 11, 1), datetime(2022, 12, 1)))
        .group_by(Order)          .order_by(order_total.desc())
        .limit(3))
>>>  session.execute(q).all()  [(Order(2cfeb68e0bed4fe7b0f3bbe707f194ee),
335.09000000000003),
(Order(5b02a2f26aa8499a96da008d3cff99f0), 318.48),
(Order(d53530d88c9f4f45b960bdddf1b89a40), 305.57)]
```

For the query that calculated the five best-selling products, there is a small complication when adding a date range, because that query does not use the Order model, which has the order timestamps. To be able to filter by date, this query needs an additional join between OrderItem and Order, which effectively means that the full many-to-many relationship between products and orders will be used.

```
>>> units = func.sum(OrderItem.quantity).label(None) >>> q =
(select(Product, units)
        .join(Product.order_items)
                                                .join(OrderItem.order)
        .where(Order.timestamp.between(
            datetime(2022, 11, 1), datetime(2022, 12, 1)))
        .group_by(Product)          .order_by(units.desc())
        .limit(5))
>>> session.execute(q).all() [(Product(41, "Commodore 64"), 157),
(Product(48, "Amiga"), 139),
(Product(127, "ZX Spectrum"), 65), (Product(16, "Apple II"), 46),
(Product(2, "BBC Micro"), 23)]
```

Note how this many-to-many relationship, which was built using the association object pattern, needs explicit joins for its two legs, while the simpler relationship between products and countries that is based on the relationship's secondary argument SQLAlchemy automatically issues the two joins to the database from a single join() clause. This is another small convenience that is lost when manually managing the many-to-many relationship.

Many of the previous queries used a between() filter on the order timestamp to constrain the results to a particular period of time. Another common query pattern when working with timestamps is to obtain results grouped by some unit of time such as day, month, quarter or year. This is harder to do because the timestamps need to be transformed into something that can be used in a group_by() clause, so that all the results from each interval can be aggregated.

The following query extracts the year and the month from order timestamps using the extract() function, and then groups by them to calculate the total number of units sold monthly during the year 2022.

```
>>> month = func.extract('month', Order.timestamp).label(None) >>>
year = func.extract('year', Order.timestamp).label(None) >>> units =
func.sum(OrderItem.quantity).label(None)
>>> q = (select(year, month, units)
        .join(OrderItem)
        .where(Order.timestamp.between(
                        datetime(2022, 1, 1), datetime(2023, 1, 1)))
        .group_by(year, month)
        .order_by(year, month))
>>> session.execute(q).all()
[(2022, 1, 505), (2022, 2, 426), (2022, 3, 525), ..., (2022, 12, 564)]
```

The extract() function accepts a unit such as day, week, month, quarter or year as first argument followed by a datetime column, and it returns the requested date or time component. The above example extracts the year and the month as individual result values, and then uses a compound group_by() clause that groups by both. Results are then sorted in ascending order by these same two values.

As mentioned earlier, with some databases results that are obtained through a calculation or function are returned as Decimal objects. The results printed in the above example came from SQLite, which uses standard int

and float numbers. When using MySQL, you will get the same results, but the sums are decimal objects:

```
[(2022, 1, Decimal('505')), (2022, 2, Decimal('426')), ..., (2022, 12, Decimal('564'))]
```

When running the same query with PostgreSQL, the results come back in yet another format:

```
[(Decimal('2022'), Decimal('1'), 505), (Decimal('2022'), Decimal('2'), 426), ...,
Decimal('12'), 564)]
```

Here the results from the extract() function calls are decimal objects, while the sums are returned as integers. These are minor implementation differences between the database engines. When you receive a Decimal object you can convert it to a primitive integer type with the int() function:

```
>>> from decimal import Decimal
>>> int(Decimal('2022'))
2022
```

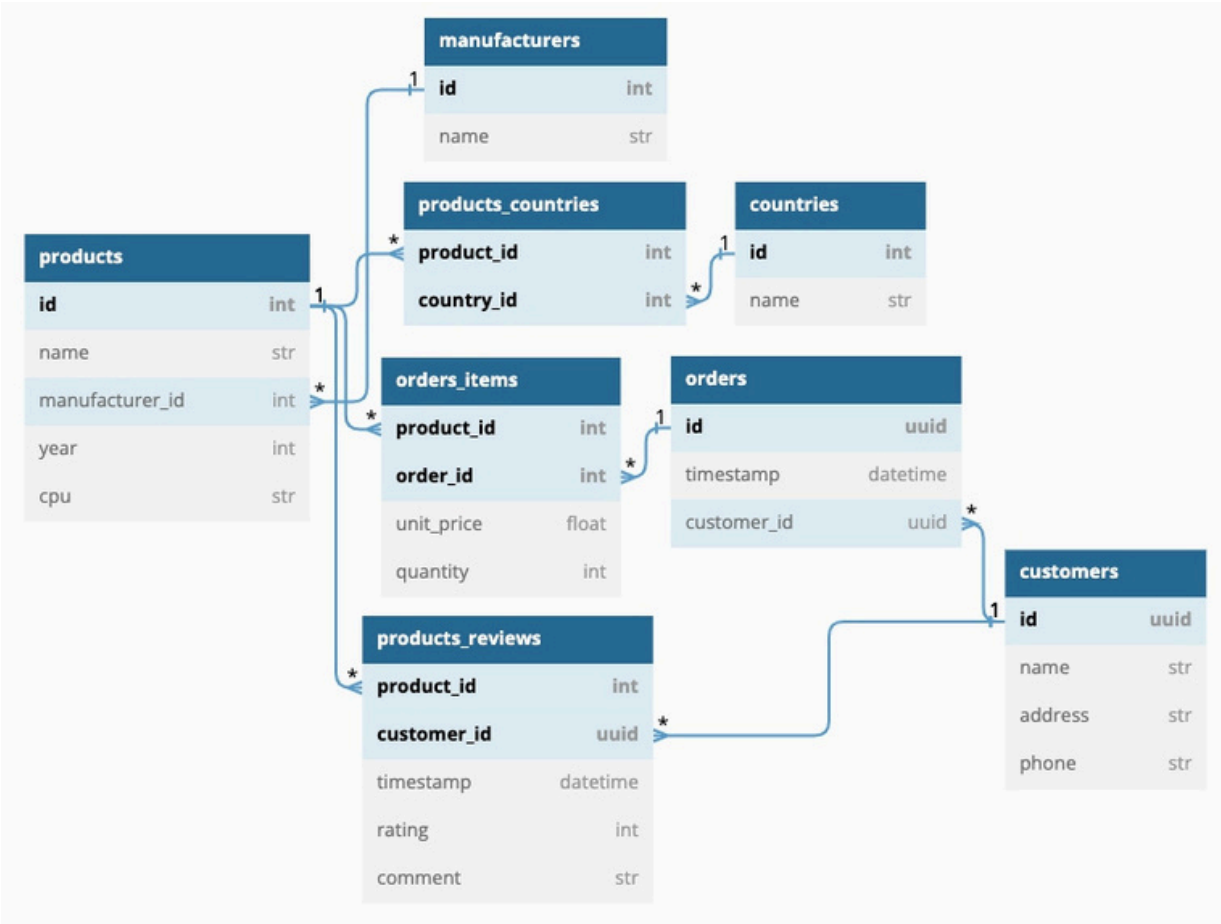
You can use the float() conversion function to convert a decimal object to floating point, but keep in mind that some precision may be lost in the conversion.

More Aggregation Techniques

The next feature you are going to add to the RetroFun database is to support customer written product reviews. A customer review consists of a star rating between 1 and 5 stars, plus an optional written comment.

Thinking about the implementation, a customer review ties a customer to a product, so there is going to be a new relationship between these two models. To decide which type of relationship is appropriate you have to think about the cardinality in each side of the relationship. Given that it is expected that a customer should be able to review many products, and a product can be reviewed by many customers, the relationship type that fits this problem is the many-to-many once again.

The following diagram shows a new product_reviews join table connects products and customers.



Should this use the simple many-to-many style from the previous chapter or the more advanced one used above for the orders? The relationship is going to need additional storage for the star rating and the review text, so the advanced solution based on the association object pattern is the correct choice.

Here is the model that represents the join table of this relationship, plus new relationship attributes in the Product and Customer models:

Listing 26 *models.py*: Decomposed many-to-many relationship

```
from sqlalchemy import Text


class Product(Model):
    # ...
    reviews: WriteOnlyMapped['ProductReview'] = relationship(
        back_populates='product')
    # ...


class Customer(Model):
```

```
    # ...
    product_reviews: WriteOnlyMapped['ProductReview'] = relationship(
        back_populates='customer')
    # ...


class ProductReview(Model):
    __tablename__ = 'products_reviews'

    product_id: Mapped[int] = mapped_column(ForeignKey('products.id'),
                                            primary_key=True)
    customer_id: Mapped[UUID] = mapped_column(ForeignKey('customers.id'),
                                              primary_key=True)
    timestamp: Mapped[datetime] = mapped_column(default=datetime.utcnow,
                                                index=True)
    rating: Mapped[int]
    comment: Mapped[Optional[str]] = mapped_column(Text)

    product: Mapped['Product'] = relationship(back_populates='reviews')
    customer: Mapped['Customer'] = relationship(
        back_populates='product_reviews')
```

This relationship is constructed in a way that is very similar to the one between products and orders. The join table is written as a Model subclass, and it contains three additional columns: one that stores the timestamp for the review (with a default option that automatically assigns the current time), the numeric rating and a comment. The timestamp and the star rating are required or non-nullable. The comment text is optional, and is given a type of Text instead of String, which SQLAlchemy uses for a possibly long block of text with unspecified maximum length.

Given that the list of product reviews can be large, the relationship attributes in the Product and Customer models are both configured with the write_only loader. The relationship attributes in the join table map to single entities, so those are set to the default lazy loading mechanism.

The commands to generate a new database migration with these changes and apply it to the database are:

```
(venv) $ alembic revision --autogenerate -m "product reviews"
(venv) $ alembic upgrade head
```

As with products and orders, it is useful to have some data in the database to be able to test queries. To that end, the following script imports a batch of product reviews from a CSV file.

Listing 27 *import_reviews.py*: Import reviews from CSV file

```
import csv
from datetime import datetime
from sqlalchemy import select, delete
```

```
from db import Session
from models import Product, Customer, ProductReview


def main():
    with Session() as session:
        with session.begin():
            session.execute(delete(ProductReview))

    with Session() as session:
        with session.begin():
            with open('reviews.csv') as f:
                reader = csv.DictReader(f)

                for row in reader:
                    c = session.scalar(select(Customer).where(
                        Customer.name == row['customer']))
                    p = session.scalar(select(Product).where(
                        Product.name == row['product']))
                    r = ProductReview(
                        customer=c,
                        product=p,
                        timestamp=datetime.strptime(row['timestamp'],
                                                    '%Y-%m-%d %H:%M:%S'),
                        rating=int(row['rating']),
                        comment=row['comment'] or None)
                    session.add(r)


if __name__ == '__main__':
    main()
```

This importer is simpler than the previous ones because it is importing data into a single table, the join table represented by the ProductReview model.

For this script, instead of using the add() method of the write_only relationship, the customer and product instances are assigned directly into each new ProductReview object, which achieves the same result.

The *reviews.csv* file must be in the project directory for the above script to work. You can download this file from the book's ~~GitHub repository~~.

Run the next command to import all the reviews:

```
(venv) $ python import_orders.py
```

Queries

Now it is time to start a Python shell and run some queries. Start by importing all the necessary functions and classes, and creating a database session:

```
>>> from sqlalchemy import select, func
>>> from db import Session
```

```
>>> from models import Product, Customer, ProductReview
>>> session = Session()
```

The first query calculates the average of all customer star ratings:

```
>>> q = select(func.avg(ProductReview.rating))
>>> session.scalar(q)
3.7731384829505914
```

This query uses the avg() aggregation function, which calculates an average of all the input results. Remember that depending on the database that you use the result may be provided as a Decimal object instead of a plain number.

Something more useful is to calculate the average rating for a product. The next query obtains the rating for the ZX Spectrum home computer:

```
>>> p = session.scalar(
      select(Product)
          .where(Product.name == 'ZX Spectrum'))
>>> q = (select(func.avg(ProductReview.rating))
          .where(ProductReview.product == p))
>>> session.scalar(q)
4.0
```

Note how in this query the where() clause uses the ProductReview.product relationship to create the condition. The ORM module of SQLAlchemy converts this high-level condition to a more basic and equivalent one using the ProductReview.product_id foreign key that can be included in the SQL query.

Another interesting option is to generate a report with the average ratings of all products. The next query does that:

```
>>> product_rating = func.avg(ProductReview.rating).label(None)
>>> q = (select(Product, product_rating)
          .join(Product.reviews)
          .group_by(Product)
          .order_by(product_rating.desc(), Product.name))
>>> session.execute(q).all()
[(Product(19, "Apple IIc Plus"), 5.0), ..., (Product(138, "Timex Sinclair 1000"), 1.0)]
```

This query uses the avg() function along with grouping, so now the averages apply to the groups instead of the entire result set. The Product model is joined with ProductReview, so results from both can be requested in the select() portion of the query and in the order_by() clause, where

the results are ordered first by rating from highest to lowest, and then alphabetically by product name.

As you recall, the comment column in the ProductReview model is optional.

The next script generates a list of products with the percentage of its reviews that do not have a written comment.

```
>>> no_comment_percent = (
      100 - 100 * func.count(ProductReview.comment) / func.count(ProductReview.rating)
   ).label(None)
>>> q = (select(Product.name, no_comment_percent)
         .join(ProductReview.product)
         .group_by(Product)
         .order_by(no_comment_percent.desc(), Product.name))
>>> session.execute(q).all()
[('464 Plus', 100.0), ('Acorn Atom', 100.0), ..., ('ZX81', 0.0)]
```

The no_comment_percent labeled expression calculates the percentage of blank reviews using a clever trick. The count() function applied to the ProductReview.comment column will return the number of columns in each group that are not NULL, and the same function applied to the ProductReview.rating column will return the total count of reviews in each product group, because the rating is required and will have a value for every item. To calculate the percentage of blank reviews the percentage of written reviews is calculated as 100.0 * comments / total, and then this amount is subtracted from 100 to get the percentage of the blank reviews. SQLAlchemy translates the multiplications, divisions and subtractions performed on these columns into the SQL query, so that the calculation is executed by the database.

The query selects the product name and the calculated percentage. It groups the rows by product so that the aggregation expression calculates percentages separately for each product. The results are finally sorted first by the percentage numbers in descending order, and then by product name.

Exercises

It's time to practice on your own. Write queries that return:

1. Orders above $300 in descending ordered by the sale amount from highest to lowest.

2. Orders that include one or more ZX81 computers.

3. Orders that include a product made by Amstrad.

4. Orders made on the 25th of December 2022 with two or more line items.

5. Customers with their first and last order date and time. Hint: the min() and max() functions can help with this query.

6. The top 5 manufacturers that had the most sale amounts, sorted by those amounts in descending order.

7. Products, their average star rating and their review count, sorted by review count in descending order.

8. Products and their average star rating, but only counting reviews that include a written comment.

9. Average star rating for the Commodore 64 computer in each month of 2022.

10. Customers with the minimum and maximum star rating they gave to a product, sorted alphabetically by customer name.

11. Manufacturers with their average star rating, sorted from highest to lowest rating.

12. Product countries with their average star rating, sorted from highest to lowest rating.

A Page Analytics Solution

The goal of this chapter is to use the concepts you have learned to build a web traffic analytics solution. This will serve as reinforcement of the techniques demonstrated in previous chapters as well as an example of a more complex and realistic database design.

Part 1: Blog Articles and Authors

Like many companies in the real world, RetroFun has a blog, in which authors post articles intended to promote sales. In this section you will expand the database to keep track of the articles that are published in the company blog, with the purpose of later tracking web traffic to them.

The code block that follows adds models for blog articles and authors.

Listing 28 *models.py*: Blog articles and authors

```
class Product(Model):
    # ...
    blog_articles: WriteOnlyMapped['BlogArticle'] = relationship(
        back_populates='product')
    # ...


class BlogArticle(Model):
    __tablename__ = 'blog_articles'

    id: Mapped[int] = mapped_column(primary_key=True)
    title: Mapped[str] = mapped_column(String(128), index=True)
    author_id: Mapped[int] = mapped_column(ForeignKey('blog_authors.id'),
                                           index=True)
    product_id: Mapped[Optional[int]] = mapped_column(
        ForeignKey('products.id'), index=True)
    timestamp: Mapped[datetime] = mapped_column(default=datetime.utcnow,
                                        index=True)

    author: Mapped['BlogAuthor'] = relationship(back_populates='articles')
    product: Mapped[Optional['Product']] = relationship(
        back_populates='blog_articles')

    def __repr__(self):
        return f'BlogArticle({self.id}, "{self.title}")'


class BlogAuthor(Model):
    __tablename__ = 'blog_authors'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(64), index=True)

    articles: WriteOnlyMapped['BlogArticle'] = relationship(
      back_populates='author')

    def __repr__(self):
        return f'BlogAuthor({self.id}, "{self.name}")'
```

The BlogArticle model maintains a title, foreign keys to an author and a product, and the publication date. The BlogAuthor model just stores the author's name. For these two models the primary keys are standard auto-incrementing integers.

There are two new one-to-many relationships introduced with this change. One is from authors to articles, managed by the BlogAuthor.articles and BlogArticle.author attributes, and the other is from products to articles, managed by Product.articles and BlogArticle.product.

The BlogArticle.product_id foreign key is the first in this project that has been defined as optional, meaning that a blog article is not required to be linked to a product. The idea is that the company blog is going to feature a variety of articles, some specifically related to a product (such as a review), while others that are more general. The BlogArticle.product relationship object is also typed as optional, because its value will be None when product_id is not set.

Part 2: Blog Sessions and Views

In this second phase of this implementation, blog users and sessions are introduced. Each visitor to the blog will be considered a "blog user", regardless of being a known customer or an anonymous visitor.

The assumption is that the RetroFun website will use cookies or a similar mechanism to keep track of its visitors. Each time a user enters the blog a new session will be created for the user, and all the blog articles that the user views during that visit will be recorded under that session. The next code block shows the definition of the BlogUser and BlogSession models.

Listing 29 *models.py*: Blog users and sessions

```
class Customer(Model):
    # ...
    blog_users: WriteOnlyMapped['BlogUser'] = relationship(
        back_populates='customer')
    # ...


class BlogUser(Model):
    __tablename__ = 'blog_users'

    id: Mapped[UUID] = mapped_column(default=uuid4, primary_key=True)
    customer_id: Mapped[Optional[UUID]] = mapped_column(
        ForeignKey('customers.id'), index=True)

    customer: Mapped[Optional['Customer']] = relationship(
        back_populates='blog_users')
    sessions: WriteOnlyMapped['BlogSession'] = relationship(
        back_populates='user')

    def __repr__(self):
        return f'BlogUser({self.id.hex})'


class BlogSession(Model):
    __tablename__ = 'blog_sessions'

    id: Mapped[UUID] = mapped_column(default=uuid4, primary_key=True)
    user_id: Mapped[UUID] = mapped_column(ForeignKey('blog_users.id'),
                                          index=True)

    user: Mapped['BlogUser'] = relationship(back_populates='sessions')

    def __repr__(self):
        return f'BlogSession({self.id.hex})'
```

Both users and sessions use UUID primary keys, because in a web application these identifiers may be stored in cookies that may be potentially visible to visitors. As explained previously, using auto-

incrementing numeric identifiers is not recommended in cases where public exposure may give away the size of the underlying database tables.

Two one-to-many relationships are also introduced. One is between blog users and blog sessions, and the other between customers and blog users. The thinking behind these two relationships is based on the assumption that the RetroFun website will be able to "remember" users across different visits. A possible implementation for the user tracking logic could be as follows:

- When a visitor enters the website for the first time, a new blog user is created and its identifier is stored in a cookie on the client's browser. A blog session is also started, and linked to the blog user.
- When a visitor enters the website and a blog user cookie is found, only a new session is created, and linked to the blog user found in the cookie.
- On any page visit to the blog, if the user is also logged in to the RetroFun website as a customer, a link between the customer and the blog user is stored.

The BlogUser.customer_id foreign key that supports the link between customers and blog users is defined as optional, because many blog users will not be customers. There is also the possibility that a customer visits the blog without being logged in, and in that case they will not be recognized. The important aspect of this solution is that the RetroFun website should make an effort to match blog users to customers, as this will allow for more interesting reports to be generated, as you will see later.

It may not be immediately clear why the relationship between customers and blog users is defined as a one-to-many, instead of a one-to-one. The reason is that a person will not always be represented by a single blog user in the RetroFun database. For example, when an anonymous visitor opens the website on their phone and later on their laptop, there will be two different blog users created. If later this user makes a purchase and becomes a customer, the intention is that eventually this customer will be linked to

both blog users, so that their behavior both on the phone and the laptop can be analyzed.

What's left to do is to create an association between blog articles and sessions, which would record which articles were viewed under a session by a user, so in simpler words, a table that record page views. The RetroFun website would insert an entry into this table whenever a blog user visits a blog article. Thinking about this new table, it is clear that it is a join table for a many-to-many relationship between articles and sessions, because an article can be viewed in many sessions, and a session can have many articles viewed.

Listing 30 *models.py*: Blog views

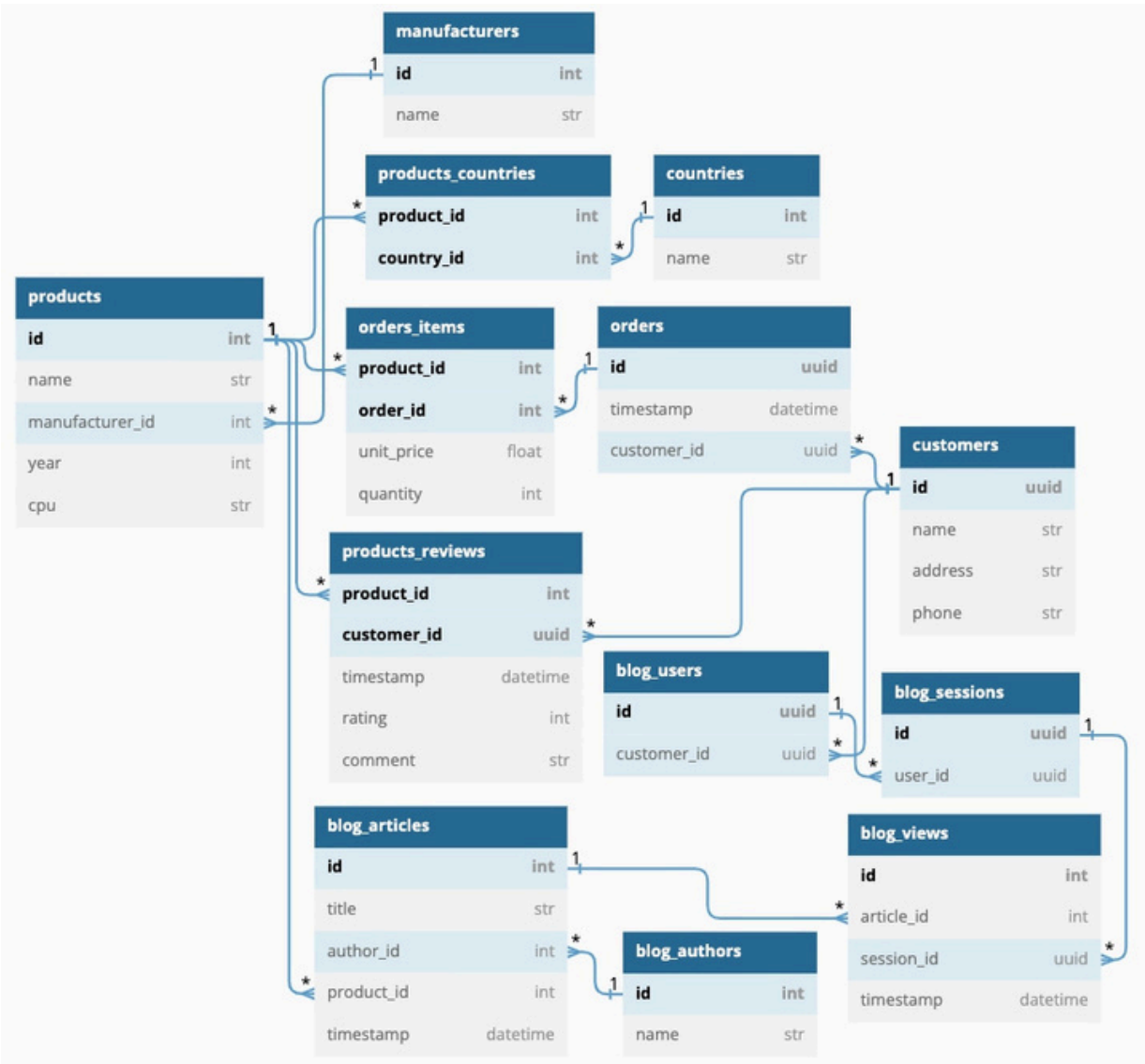
```
class BlogArticle(Model):
    # ...
    views: WriteOnlyMapped['BlogView'] = relationship(back_populates='article')
    # ...


class BlogSession(Model):
    # ...
    views: WriteOnlyMapped['BlogView'] = relationship(back_populates='session')
    # ...


class BlogView(Model):
    __tablename__ = 'blog_views'

    id: Mapped[int] = mapped_column(primary_key=True)
    article_id: Mapped[int] = mapped_column(ForeignKey('blog_articles.id'))
    session_id: Mapped[UUID] = mapped_column(ForeignKey('blog_sessions.id'))
    timestamp: Mapped[datetime] = mapped_column(default=datetime.utcnow,
                                    index=True)

    article: Mapped['BlogArticle'] = relationship(back_populates='views')
    session: Mapped['BlogSession'] = relationship(back_populates='views')
```

For this relationship, it is useful to store the date and time of the page view, which means that the advanced association object style has to be used.

This relationship presents an interesting problem that did not exist in any of the previous many-to-many relationships. A user may view a given article two or more times, all in the context of a single web session, for example, by refreshing the page in the browser, which means it should be possible to have two or more BlogView entries that have the same article and session foreign keys.

As you recall, many-to-many relationships set the two foreign keys as a compound primary key for the join table, which prevents duplicate records. Preventing duplicates is useful in many situations, but for this particular relationship duplicates should be allowed so that all page views can be accurately counted. The tweak that is made to allow the duplicates is to use a standard numeric auto-incremented id as primary key instead of a compound primary key made up of the two foreign keys.

To help you keep track of the current database structure, here is a diagram showing all the tables and relationships up to this point.



Believe it or not, this is all it takes to have storage for a basic page analytics solution. Now it's time to migrate the database so that all these changes are recorded and applied:

```
(venv) $ alembic revision --autogenerate -m "blog integration"
(venv) $ alembic upgrade head
```

Importer Scripts

As with all previous tables, it is useful to add some data so that it is possible to experiment with queries. Here is a script that imports articles and authors from a CSV file:

Listing 31 *import_articles.py*: Article importer script

```
import csv
from datetime import datetime
from sqlalchemy import select, delete
from db import Session
from models import BlogArticle, BlogAuthor, Product, BlogView, BlogSession, \
    BlogUser


def main():
    with Session() as session:
        with session.begin():
            session.execute(delete(BlogView))
            session.execute(delete(BlogSession))
            session.execute(delete(BlogUser))
            session.execute(delete(BlogArticle))
            session.execute(delete(BlogAuthor))

    with Session() as session:
        with session.begin():
            all_authors = {}
            all_products = {}

            with open('articles.csv') as f:
                reader = csv.DictReader(f)

                for row in reader:
                    author = all_authors.get(row['author'])
                    if author is None:
                        author = BlogAuthor(name=row['author'])
                        all_authors[author.name] = author

                    product = None
                    if row['product']:
                        product = all_products.get(row['product'])
                        if product is None:
                            product = session.scalar(select(Product).where(
                                Product.name == row['product']))
                            all_products[product.name] = product

                    article = BlogArticle(
                        title=row['title'],
                        author=author,
                        product=product,
                        timestamp=datetime.strptime(
                            row['timestamp'], '%Y-%m-%d %H:%M:%S'
```

```
                ),
            )
            session.add(article)


if __name__ == '__main__':
    main()
```

This script uses the same techniques used in previous importers to create BlogArticle and BlogAuthor entries in the database, so it should be self-explanatory.

The *articles.csv* file referenced in the script must be copied to the project directory. You can download this file from the book's GitHub repository. . Note that the article titles and author names used in this data file were created with a fake data generator, so they are not real.

Run the following command to execute the script and import the articles and authors into the database:

```
(venv) $ python import_articles.py
```

The next script imports page views, along with blog users and sessions. Here is the code for it:

Listing 32 *import_views.py*: Blog page views importer script

```
import csv
from datetime import datetime
from uuid import UUID
from sqlalchemy import select, delete
from db import Session
from models import BlogArticle, BlogUser, BlogView, BlogSession, Customer


def main():
    with Session() as session:
        with session.begin():
            session.execute(delete(BlogView))
            session.execute(delete(BlogSession))
            session.execute(delete(BlogUser))

    with Session() as session:
        all_articles = {}
        all_customers = {}
        all_blog_users = {}
        all_blog_sessions = {}

        with open('views.csv') as f:
            reader = csv.DictReader(f)

            i = 0
            for row in reader:
                user = all_blog_users.get(row['user'])
                if user is None:
                    customer = None
                    if row['customer']:
```

```
                customer = all_customers.get(row['customer'])
                if customer is None:
                    customer = session.scalar(select(Customer).where(
                        Customer.name == row['customer']))
                    all_customers[customer.name] = customer

                user_id = UUID(row['user'])
                user = BlogUser(id=user_id, customer=customer)
                session.add(user)
                all_blog_users[row['user']] = user

            blog_session = all_blog_sessions.get(row['session'])
            if blog_session is None:
                session_id = UUID(row['session'])
                blog_session = BlogSession(id=session_id, user=user)
                session.add(blog_session)
                all_blog_sessions[row['session']] = blog_session

            article = all_articles.get(row['title'])
            if article is None:
                article = session.scalar(select(BlogArticle).where(
                    BlogArticle.title == row['title']))
                all_articles[article.title] = article

            view = BlogView(
                article=article,
                session=blog_session,
                timestamp=datetime.strptime(
                    row['timestamp'], '%Y-%m-%d %H:%M:%S'),
            )
            session.add(view)

            i += 1
            if i % 100 == 0:
                print(i)
                session.commit()
        print(i)
        session.commit()


if __name__ == '__main__':
    main()
```

This importer uses the database session in a way that is different from the previous ones. The reason for the change is that a page view table is likely to hold a much larger volume of data than other tables. To reflect this reality, the CSV file with example data is significantly larger, enough that accumulating all the imported data into a single database session that is committed at the end is highly impractical.

Instead of relying on the session.begin() context manager that commits at the end, this importer keeps a count of imported rows and issues explicit commits every 100 rows. The logic that achieves this uses a counter. Here is the code specific to this isolated from the rest:

```
        i = 0
        for row in reader:

            # ... import the row

            i += 1
            if i % 100 == 0:
```

```
                print(i)
                session.commit()
        print(i)
        session.commit()
```

The print(i) statement will print 100, 200, etc. to the terminal, to show progress. The second print and commit at the bottom ensure that the final set of rows that were imported right before the loop exited are also stored.

Download the *views.csv* file from the book's GitHub repository. As mentioned above, this is a fairly large data file (about 19MB), so it may take some time to download depending on your connection speed.

Run the importer as follows:

```
(venv) $ python import_views.py
```

Once the import process starts you will start seeing multiples of 100 printing to the terminal. This is a fairly large data file, so it will take a few minutes for the script to go through the entire CSV file, which has about 138,000 rows.

Page Analytics Queries

With all this new information in the database, there are a number of very interesting queries that can be made. Open a new Python session, import all the required dependencies, and create a new database session:

```
>>> from datetime import datetime
>>> from sqlalchemy import select, func
>>> from db import Session
>>> from models import BlogArticle, BlogView, Product
>>> session = Session()
```

To start with the obvious, you may want to know the total number of pages viewed over a specific period of time. The next query calculates total page views in November 2022:

```
>>> q = (select(func.count(BlogView.id))
        .where(BlogView.timestamp.between(
            datetime(2022, 11, 1), datetime(2022, 12, 1))))
>>> session.scalar(q)
4034
```

The query that follows shows the ranking of blog articles from most to least viewed, also for the month of November 2022:

```
>>> page_views = func.count(BlogView.id).label(None)
>>> q = (select(BlogArticle.title, page_views)
          .join(BlogArticle.views)
          .where(BlogView.timestamp.between(
              datetime(2022, 11, 1), datetime(2022, 12, 1)))
          .group_by(BlogArticle)
          .order_by(page_views.desc(), BlogArticle.title))
>>> session.execute(q).all()
[('Boy itself fish traditional', 57), ..., , ('Still defense foreign social', 1)]
```

If you were to add all the views reported by this query, the total would come up to 4034, which makes perfect sense since this and the previous queries are retrieving exactly the same page views, just organized in different ways.

As you recall, articles can be associated with a specific product in the BlogArticle.product relationship. This makes it possible to navigate across relationships and get a report of page views associated with each product:

```
>>> page_views = func.count(BlogView.id).label(None)
>>> q = (select(Product.name, page_views)
          .join(Product.blog_articles)
          .join(BlogArticle.views)
          .where(BlogView.timestamp.between(
              datetime(2022, 11, 1), datetime(2022, 12, 1)))
          .group_by(Product)
          .order_by(page_views.desc()))
>>> session.execute(q).all()
[('ZX Spectrum', 1096), ('Commodore 64', 1056), ('Apple II', 349),
('TRS-80 Color Computer', 349), ('Amiga', 301), ('BBC Micro', 180), ('TI-99/4A', 133),
('Commodore 128', 77)]
```

Now this is an interesting query. If you add up all the page views the total is 3541, not 4034 as in the previous queries. Can you guess why?

The BlogArticle.product_id foreign key was configured as optional (or "nullable", in database jargon) column. The page views for articles without a product association are not included in this report, because the `join(Product.blog_articles)` clause pairs Product instances a BlogArticle instances where the product matches in both. The BlogArticle instances that have product_id (and consequently also the product relationship) set to None will not match anything on the Product side and will be omitted from the join.

A join that only includes matching rows from the two tables is said to be an *inner join*. This is the default join that SQLAlchemy's join() method uses, and the only type used in this book so far. But inner joins are not the only type of join that can be used.

Another way to join two tables is with an *outer join*, which also includes entities on each side of the relationship that do not match anything on the other side. Outer joins come in three types:

- Full outer join: include unmatched entities of the left and right tables
- Left outer join: include unmatched entities of the left table only
- Right outer join: include unmatched entities of the right table only

What does this all mean, exactly? The results of a full outer join between blog articles and products are going to combine three different types of records:

- Matching blog article and product pairs (these are the results returned by the default inner join query)
- Blog articles that have no matching product (the product will be None in these results)
- Products that have no matching blog articles (the blog article will be None in these results)

Note

Unfortunately support for outer joins is not uniform across databases.

SQLite supports all the outer join types, but full and right outer joins were added in release 3.39.0 from 2022, which isn't widely deployed still. An error will be returned if these types of joins are attempted when using previous releases.

MySQL supports left and right outer joins, but as of April 2023 it does not support full outer joins.

PostgreSQL supports all the outer join types.

SQLAlchemy only implements full and left outer joins. When a right outer join is desired, the tables must be swapped so that a left outer join can be used.

Changing the first join of the previous query to a full outer join will ensure that all the blog articles (which are on the right side in that join) are retrieved, and not just those that can be matched against a product. Then the next join against BlogView will not drop any page views. The only change to convert the default inner join to a full outer join is to add a full=True argument to the join() clause:

```
>>> q = (select(Product.name, page_views)
        .join(Product.blog_articles, full=True)
        .join(BlogArticle.views)
        .where(BlogView.timestamp.between(
            datetime(2022, 11, 1), datetime(2022, 12, 1)))
        .group_by(Product)
        .order_by(page_views.desc()))
>>> session.execute(q).all()
[('ZX Spectrum', 1096), ('Commodore 64', 1056), (None, 493), ('Apple II', 349),
('TRS-80 Color Computer', 349), ('Amiga', 301), ('BBC Micro', 180), ('TI-99/4A', 133),
('Commodore 128', 77)]
```

Note how these new results include an item with the product set to None containing the 493 page views that were missed in the previous report.

This query could have also used a right outer join, which might even be more efficient, but as noted above, SQLAlchemy does not currently have support for this join type. It was also noted that full outer joins are not implemented by all databases, so depending on your database choice the above query may fail with a database error. In particular, this query would not work with MySQL or with older versions of SQLite.

What can be done when full outer joins are not available? Luckily, this query does not require a full outer join but just a right outer join. The trick is to reverse the join direction and then use a left outer join, which is generated with the isouter=True argument added to the join() clause. Here is how to do it:

```
>>> q = (select(Product.name, page_views)
        .join(BlogArticle.product, isouter=True)
        .join(BlogArticle.views)
        .where(BlogView.timestamp.between(
            datetime(2022, 11, 1), datetime(2022, 12, 1)))
```

```
        .group_by(Product)
        .order_by(page_views.desc()))
>>> session.execute(q).all()
[('ZX Spectrum', 1096), ('Commodore 64', 1056), (None, 493), ('Apple II', 349),
('TRS-80 Color Computer', 349), ('Amiga', 301), ('BBC Micro', 180), ('TI-99/4A', 133),
('Commodore 128', 77)]
```

In this version of the query, the first join switches from the `Product.blog_articles` relationship to `BlogArticle.product`. T attributes represent the relationship between products and blog articles from the two sides, so this change effectively reverses the direction of the join, putting blog articles on the left side and products on the right, which makes it possible to use a left outer join to retrieve blog articles with no product matching.

Isn't it interesting that the full and right outer join queries above return the exact same results? When using a full outer join there should have been more data in the results, right? The results include the page view counts for blog articles that are associated with products, and the page views for blog articles that have no associated product. But when using the full outer join, the results should have also included all the products that have no associated blog articles, which should have appeared with zero page views. Why are those missing?

To understand this you have to review the rest of the full outer join version of this query. After the full outer join is performed, there is another join with the BlogView entities, and this join is a default inner join. This second join matches pairs of products and articles that resulted from the full outer join to BlogView records, using the BlogArticle.views relationship as the matching column. Because this is an inner join, all the (Product, None) pairs returned by the first join are discarded, since the article portion of the pair is None and will never match anything on the BlogView side.

If the intention is to preserve those products that have no blog views, then the second join must also be upgraded to a full outer join, which will ensure that all the (Product, BlogArticle, BlogView) triplets in which both the article and the view are None are kept in the results.

But if the second join is changed to a full outer join, there are going to be some results with a None for the BlogView entity. The where() clause in this query uses BlogView.timestamp, so this has to be updated to allow not only

the page views that are in the period of interest but also those that are `None`. This can be done with the or_() function. Here is the final query:

```
>>> from sqlalchemy import or_ >>> q = (select(Product.name, page_views)
        .join(Product.blog_articles, full=True)
        .join(BlogArticle.views, full=True)
        .where(or_(
            BlogView.timestamp == None,
            BlogView.timestamp.between( datetime(2022, 11, 1), datetime(2022, 12,
1))))
        .group_by(Product)
        .order_by(page_views.desc(), Product.name))
>>> session.execute(q).all()
[('ZX Spectrum', 1096), ('Commodore 64', 1056), (None, 493), ..., ('ZX80', 0), ('ZX81', 0)]
```

And this query returns a report of all the products, with their page view counts for the given period, including page views for generic articles not associated with a product and products with no articles written about them or with no blog views. But of course, this complete version requires a full outer join, which isn't available in some databases.

Generating this last query using only left outer joins is difficult, because no matter which way the joins are configured there is always going to be one side of unmatched entities that is not going to come back with the results. A common solution to simulate full outer joins is to run two queries instead of one. The left outer join query used above can be used to get the list of products with their page views, plus the page views for articles without a product assignment. Then a second query can be used to get the list of products that had no blog views, either because they have no content in the blog or because their content hasn't been viewed by anyone. Here is a query that retrieves these:

```
>>> q2 = (select(Product.name, page_views)
        .join(Product.blog_articles, isouter=True)
        .join(BlogArticle.views, isouter=True)
        .where(or_( BlogView.timestamp == None,
            BlogView.timestamp.between(
                datetime(2022, 11, 1), datetime(2022, 12, 1))))
        .group_by(Product)
        .having(page_views == 0)
        .order_by(Product.name))
>>> session.execute(q2).all()
[('464 Plus', 0), ('6128 Plus', 0), ..., ('ZX80', 0), ('ZX81', 0)]
```

For this query the products are joined with the blog articles, and the resulting pairs are joined with the blog views. Both joins are left outer joins, which means that products that have no matching blog articles or no

matching blog views in the period of interest will be kept in the results. The having() clause discards any results that have non-zero page views, since those were already captured by the first query.

You can now combine the results from the two queries in Python, or if you prefer, you can use the union() function from SQLAlchemy to have this merge done by the database. Below you can see how to write two queries q1 and q2, which get consolidated into q using the union operator:

```
>>> from sqlalchemy import union >>> q1 = (select(Product.name, page_views)
        .join(BlogArticle.product, isouter=True)
        .join(BlogArticle.views)
        .where(BlogView.timestamp.between(
           datetime(2022, 11, 1), datetime(2022, 12, 1)))
        .group_by(Product)) >>> q2 = (select(Product.name, page_views)
        .join(Product.blog_articles, isouter=True)
        .join(BlogArticle.views, isouter=True)
        .where(or_(
                BlogView.timestamp == None,                    BlogView.timestamp.between(
             datetime(2022, 11, 1), datetime(2022, 12, 1))))
        .group_by(Product)
        .having(page_views == 0))
>>> q = union(q1, q2).order_by(page_views.desc(), Product.name)
>>> session.execute(q).all() [('ZX Spectrum', 1096), ('Commodore 64', 1056), (None, 493), ...,
('ZX80', 0), ('ZX81', 0)]
```

Once you work with outer joins you will find lots of queries that are improved by upgrading inner joins to one of the outer join types. For example, the query that returns page views by article presented earlier in this chapter did not include the articles that received no page views in the results, because the inner join between blog articles and blog views discarded those articles. Changing this query to use a left outer join preserves the articles without views. Here is the updated query:

```
>>> page_views = func.count(BlogView.id).label(None) >>> q =
(select(BlogArticle.title, page_views)
        .join(BlogArticle.views, isouter=True)
        .where(or_(
           BlogView.timestamp == None,
            BlogView.timestamp.between(                datetime(2022,
11, 1), datetime(2022, 12, 1))))          .group_by(BlogArticle)
        .order_by(page_views.desc()))
>>> session.execute(q).all()
[..., ('Prepare culture part budget star organization there', 0)]]
```

The updated query reveals that there was only one article in November 2022 that did not receive page views, appearing at the bottom of the list of results with a count of zero views.

To make this query work, not only the join was changed to a left outer join, but also the where() clause was expanded to accept BlogView results that are None, as in the previous query.

Part 3: Multi-Language Blog Articles

Like many companies, RetroFun is interested in selling internationally. To that effect, it creates original blog content in other languages besides English, and it also has a team dedicated to translate successful English blog posts to other languages.

To make the web analytics project even more useful, in this third and last phase of the project you will learn how to expand the database to keep track of the language in which each article is written, and also which articles are translations of other articles instead of original pieces of content. This will lead to many more interesting reports that can be extracted from the data.

The first change adds a Language model, plus a one-to-many relationship between languages and blog articles.

Listing 33 *models.py*: Blog article languages

```
class BlogArticle(Model):     # ...
    language_id: Mapped[Optional[int]] = mapped_column(
        ForeignKey('languages.id'), index=True)     # ...
    language: Mapped[Optional['Language']] = relationship(
        back_populates='blog_articles')     # ...




class Language(Model):
    __tablename__ = 'languages'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(32), index=True, unique=True)

    blog_articles: WriteOnlyMapped['BlogArticle'] = relationship(
        back_populates='language')

    def __repr__(self):
        return f'Language({self.id}, "{self.name}")'
```

This new one-to-many relationship is very much like previous ones, so it does not present any challenge.

Tracking a relationship between original content and their translations does seem tricky, however. Which relationship type works for this use case? An original article can have many translations, and each translated article can

only have one original source. This suggests this is going to be a one-to-many relationship.

What are the two entities of the relationship? Very clearly the "one" side of the relationship is going to be original articles, which are instances of the BlogArticle model. What about the "many" side? These are also articles, right? This is the first time in this project where a relationship has the same entity, BlogArticle in this case, on both sides. This is called a *self-referential relationship*.

With SQLAlchemy, self-referential relationships need some additional configuration. Below are the changes to the BlogArticle model to add this relationship.

Listing 34 *models.py*: Blog article translations

```
class BlogArticle(Model):
    # ...
    translation_of_id: Mapped[Optional[int]] = mapped_column(
        ForeignKey('blog_articles.id'), index=True)
    # ...
    translation_of: Mapped[Optional['BlogArticle']] = relationship(
        remote_side=id, back_populates='translations')
    translations: Mapped[list['BlogArticle']] = relationship(
        back_populates='translation_of')
    # ...
```

The translation_of_id foreign key is defined in the same way as other foreign keys, with the only difference that it references the primary key in the same table.

The two relationship attributes that represent the sides of the relationship now have to be added to the same model class, and this requires some care. SQLAlchemy cannot easily figure out which of the two relationship attributes is which in a self-referential relationship, so the remote_side argument is added to the relationship() definition that references the "one" side to remove the ambiguity. In this case, the translation_of relationship has its remote_side argument set to the id primary key, and this is enough for SQLAlchemy to understand that this relationship points to the "one" side and consequently the other relationship is the list with the "many" side.

The following diagram shows the database table diagram after the new translation_of_id foreign key column is added to the blog_articles table. This is the final version of the RetroFun database.

These changes can now be incorporated in a database migration:

```
(venv) $ alembic revision --autogenerate -m "multi-language support"
(venv) $ alembic upgrade head
```

The *articles.csv* data file that was used earlier to import articles and authors already includes language and translation_of columns, which the *import_articles.py* script ignored. With the new multi-language support, a second pass through this file can import this additional information. The *import_languages.py* script shown below does this.

Listing 35 *import_languages.py*: Import language and translation relationships

```
import csv
from sqlalchemy import select
from db import Session
from models import BlogArticle, Language


def main():
    with Session() as session:
        with session.begin():
            all_articles = {}
            all_languages = {}

            with open('articles.csv') as f:
                reader = csv.DictReader(f)

                for row in reader:
                    article = all_articles.get(row['title'])
                    if article is None:
                        article = session.scalar(select(BlogArticle).where(
                            BlogArticle.title == row['title']))
                        all_articles[article.title] = article

                    language = all_languages.get(row['language'])
                    if language is None:
                        language = session.scalar(select(Language).where(
                            Language.name == row['language']))
                        if language is None:
                            language = Language(name=row['language'])
                            session.add(language)
                        all_languages[language.name] = language
                    article.language = language

                    if row['translation_of']:
                        translation_of = all_articles.get(
                            row['translation_of'])
                        if translation_of is None:
                            translation_of = session.scalar(select(
                                BlogArticle).where(BlogArticle.title ==
                                                   row['translation_of']))
                            all_articles[article.title] = article
                        article.translation_of = translation_of


if __name__ == '__main__':
    main()
```

The script reads the rows of *articles.csv*, only looking at the article's title, language and translation_of columns. No articles are inserted in this script, articles are directly loaded from the database by their title, because the assumption is that the *import_articles.py* script has already imported all of them.

For the language support, the script assigns the corresponding Language instance to the BlogArticle.language attribute, creating new Language instances when a language appears for the first time. As before, the all_languages dictionary keeps a cache of all the languages added so far for convenience.

The translation_of column of the CSV file is empty for original articles,

so the script first checks if there is a value for this column. When a value exists it means that the article is translated, and the value of this field is the title of the original article. The translated article is then assigned the original in the translation_of self-referential relationship.

Run the script to incorporate the language and translation relationships:

```
(venv) $ python import_languages.py
```

Language Queries

The language support adds one more dimension to the queries that can be generated. Open a new Python session and import the usual components needed to experiment:

```
>>> from datetime import datetime
>>> from sqlalchemy import select, func
>>> from models import Language, BlogArticle, BlogView
>>> from db import Session
>>> session = Session()
```

First, here is an easy query that returns the number of articles in each language:

```
>>> q = (select(Language, func.count(BlogArticle.id))
        .join(Language.blog_articles)
        .group_by(Language)
        .order_by(Language.name))
>>> session.execute(q).all()
[(Language(1, "English"), 108), (Language(3, "French"), 25), (Language(2, "German"), 21),
(Language(4, "Italian"), 13), (Language(6, "Portuguese"), 25), (Language(5, "Spanish"), 17)]
```

These counts are for all the articles, regardless of being original or a translation. The next two queries generate counts for originals and for translations separately:

```
>>> q = (select(Language, func.count(BlogArticle.id))
        .join(Language.blog_articles)
```

```
        .where(BlogArticle.translation_of == None)
        .group_by(Language)
        .order_by(Language.name))
>>> session.execute(q).all()
[(Language(1, "English"), 108), (Language(3, "French"), 11), (Language(2, "German"), 6),
(Language(4, "Italian"), 5), (Language(6, "Portuguese"), 7), (Language(5, "Spanish"), 6)]
```

```
>>> q = (select(Language, func.count(BlogArticle.id))
        .join(Language.blog_articles)
        .where(BlogArticle.translation_of != None)
        .group_by(Language)
        .order_by(Language.name))
>>> session.execute(q).all()
[(Language(3, "French"), 14), (Language(2, "German"), 15), (Language(4, "Italian"), 8),
(Language(6, "Portuguese"), 18), (Language(5, "Spanish"), 11)]
```

These queries use the BlogArticle.translation_of relationship to filter the original content from the translations or vice versa because original articles always have this attribute set to None.

Note

When writing expressions for SQLAlchemy that compare against None, the comparison has to be done with the == and != operators. Many Python developers would prefer to use is None or is not None, but SQLAlchemy cannot translate these into SQL expressions.

The next query is more tricky. The goal is to generate a report of original articles, each with the number of available translations. The solution is to query for all the pairs of original and translated articles, then group by the original articles and apply a count function to the translation.

```
>>> from sqlalchemy.orm import aliased
>>> TranslatedBlogArticle = aliased(BlogArticle)
>>> article_count = func.count(TranslatedBlogArticle.id).label(None)
>>> q = (select(BlogArticle, article_count)
        .join(TranslatedBlogArticle.translation_of)
        .group_by(BlogArticle)
        .order_by(article_count.desc(), BlogArticle.title))
>>> session.execute(q).all()
[(BlogArticle(63, "Business seven ability cup church similar itself"), 3), ...,
(BlogArticle(1, "Within across act song"), 1)]
```

This query uses the aliased function, which you haven't seen before. To be able to pair articles with their translations, a join needs to be made on the self-referential translation_of relationship. But having the same table on the two sides of a relationship creates a complication, because when both sides have the same table name it is not possible to independently refer to the left or right sides. SQL solves the ambiguity of this situation with

aliases. Giving one of the sides a new name makes it possible to work with two instances of the same table as if they were different. The TranslatedBlogArticle alias created above represents the left-side of the relationship when looking at it as a many-to-one relationship from translated articles into their originals.

So now there is TranslatedBlogArticle and BlogArticle, and a join is made between them. The join(TranslatedBlogArticle.transla expression creates the join with the aliased table on the left, and the original on the right.

To be able to count the translations, the TranslatedBlogArticle instances are aggregated with the count() function. The counting expression is given a label, so that it can be reused in the order_by() clause, as done several times before.

This database design is extremely flexible and allows for even more complex and interesting queries. Let's say that the company wants to have a report of page views per article similar to those generated earlier, but with the additional complication that only original articles should be listed, with the aggregated page view counts that include their translations. For consistency with previous queries this is also going to cover page views in November 2022.

As you recall, the query that returned page views per article joined the BlogArticle and BlogView models, then grouped by BlogArticle and used the count() aggregation function to return how many rows were in each group. To be able to include page views of translated articles along with the original article, it is necessary to have a column in every result row that references the original article, and then this column can be used to group the results.

The BlogArticle.translation_of relationship has the reference to the original article, but for the original articles themselves this relationship is set to None. What this query needs is a column that works as a conditional: when the article is original, it should be a reference to that same article, but when the article is translated the reference should be to the parent article.

The SQL language provides a conditional that can be used to solve this problem, the CASE construct, which in SQLAlchemy is available through the case() function. Here is a labeled column definition that uses this function:

```
>>> from sqlalchemy import case
>>> original_id = case(
      (BlogArticle.translation_of == None, BlogArticle.id),
      else_=BlogArticle.translation_of_id).label(None)
```

The case() function accepts one or more tuples as arguments. Each tuple has a condition in the first element, and a value in the second. The result of the case expression takes the value from the first tuple for which the condition evaluates to True. The else_ argument provides a value to use when none of the tuples have a condition that evaluates to True.

In the above definition, case() has a single condition that checks for the translation_of relationship being None, which indicates that the article is an original. In that case, the value that will be assigned to that column is the id of the article. When the condition is False, the else_ argument provides an alternative value of the column from the translation_of_id attribute, which has the id of the parent article.

Note

As much as possible it is preferred to work with ORM entities, but the values used in the case() function cannot be set to model entities, and for that reason primary key identifiers are used instead.

Now this original_id column can be used in a group_by() clause instead of the blog article that was used before.

```
>>> page_views = func.count(BlogView.id).label(None)
>>> q = (select(original_id, page_views)
         .join(BlogArticle.views)
         .where(BlogView.timestamp.between(
            datetime(2022, 11, 1), datetime(2022, 12, 1)))
         .group_by(original_id)
         .order_by(page_views.desc()))
>>> session.execute(q).all()
[(171, 136), (76, 107), (98, 99), ..., (112, 2), (31, 2), (201, 1)]
```

And unfortunately this isn't the expected result, right? The query groups by original_id values, which are numeric primary key values assigned to BlogArticle instances. Using primitive values was required to be able to

use the case() function, but now it would be ideal to convert these numbers back to the entities they represent.

A nice little trick that can solve this problem is to join the original_id column with an aliased instance of BlogArticle, which would associate each number with its corresponding BlogArticle entity:

```
>>> OriginalBlogArticle = aliased(BlogArticle) >>> q =
(select(OriginalBlogArticle, page_views) .join(BlogArticle.views)
        .join(OriginalBlogArticle, original_id == OriginalBlogArticle.id)
        .where(BlogView.timestamp.between(
            datetime(2022, 11, 1), datetime(2022, 12, 1)))
        .group_by(OriginalBlogArticle) .order_by(page_views.desc()))
>>> session.execute(q).all()
[(BlogArticle(171, "Our activity public responsibility represent"), 136), ...,
(BlogArticle(201, "Exist they particular important note kitchen current"), 1)]
```

Here a second instance of the BlogArticle model is created as an alias, and stored with the name OriginalBlogArticle. Then an additional join() clause is added to the query to join the original_id values with it.

The join() call is the first that does not have a relationship in its first argument. When a relationship is given, SQLAlchemy can determine all the parameters of the join from it. In this case there is no relationship defined between the original_id values and the OriginalBlogArticle aliased model, so SQLAlchemy needs more details on how this join has to be carried out. This alternative format of the join() clause takes the right-side entity of the join as first argument, and a join condition as second argument. The join condition that was used above ensures that for each value of original_id the blog article with the same identifier is matched.

This query could be expanded to also return how many articles were considered for each of the results. For an article that has no translations a 1 would be returned, but for an article with some translations you would know how many articles were aggregated into the page view results.

To do this, a third column needs to be added to the query that counts the number of articles in each group. You have seen that the count() function counts rows, so using count(BlogArticle.id) would return the same result as the page_counts label, since both would count different columns of the same rows, which represent page views and not blog articles. Adding

the distinct() method to the count eliminates the duplication and returns the correct count of articles:

```
>>> q = (select(          OriginalBlogArticle,          page_views,
        func.count(BlogArticle.id.distinct())
      )
      .join(BlogArticle.views)
                  .join(OriginalBlogArticle,  original_id  ==  OriginalBlogArticle.id)
      .where(BlogView.timestamp.between(
         datetime(2022, 11, 1), datetime(2022, 12, 1)))
      .group_by(OriginalBlogArticle)
      .order_by(page_views.desc()))
>>> session.execute(q).all() [(BlogArticle(171, "Our activity public responsibility
represent"), 136, 4), ..., (BlogArticle(201, "Exist they particular important note
kitchen current"), 1, 1)]
```

Exercises

Do you want to practice with some more queries? Write queries that return:

1. Blog posts that have received more than 40 page views in March 2020.
2. Blog article with the largest number of translations. In case of a tie, the article that comes first alphabetically should be returned.
3. Page views in March 2022, categorized by language.
4. Page views by article, only considering content in German.
5. Monthly page views between January and December 2022.
6. Daily page views in February 2022.

Asynchronous SQLAlchemy

Starting with release 1.4, SQLAlchemy includes support for asynchronous programming with the asyncio package, for both the Core and ORM modules. This is an exciting improvement that brings the power of SQLAlchemy to modern applications such as those written with the FastAPI web framework.

In this chapter you are going to learn how the asynchronous support in SQLAlchemy works by adapting all the work done in previous chapters to the asynchronous model.

How is Async Different?

The asynchronous programming paradigm introduces some differences in the execution model of an application.

One of the aspects that make asynchronous programming difficult is what has been brilliantly described as function coloring, which is a metaphor for the limitations Python and other languages have in regard to the mixing of synchronous and asynchronous code.

In short, function coloring means that asynchronous applications should avoid long-running synchronous functions because these are *blocking* and prevent concurrency. As a result of this limitation, a web application that uses SQLAlchemy needs asynchronous code in all of its layers, which from top to bottom may include:

- Web server
- Web framework
- Route logic
- SQLAlchemy session

- SQLAlchemy engine
- Database driver

While it is obvious that an asynchronous web application needs a server and a framework that are also asynchronous, the requirement extends to the lower layers as well. That means that any application functions that use SQLAlchemy asynchronously must be async functions, and the session and engine objects have to be replaced with asynchronous equivalents. Finally, the database driver must also be designed to work asynchronously.

Another important difference is related to implicit database activity.

SQLAlchemy ORM is a high-level database framework that sometimes decides on its own to issue database queries. The best example of this are the relationship attributes that are configured with the default lazy loader, which implicitly run a database query to obtain the results the first time they are accessed.

These implicit behaviors cannot exist in an asynchronous application due to the function coloring limitations of the asynchronous model. All asynchronous database activity must happen inside functions that are asynchronous.

Asynchronous Database Drivers

Starting from the bottom of the stack, to be able to use SQLAlchemy in an asynchronous application you must use a compatible database driver. None of the regular database drivers mentioned earlier in the book can be used asynchronously.

The following sections discuss what options are available for the three major open-source databases. If you are not using any of these databases, you can find your database dialect in the SQLAlchemy documentation, where you can look for asynchronous driver options.

SQLite

The sqlite module that comes with the Python interpreter does not support the asynchronous model. To be able to work with SQLite from asynchronous code, SQLAlchemy supports the aiosqlite third-party package, which needs to be installed into your virtual environment as follows:

```
(venv) $ pip install aiosqlite
```

The database connection URL given to SQLAlchemy needs to be modified to reflect the use of this driver. As you recall, database URLs specify the dialect and the driver in the scheme portion of the URL separated by a + sign. Below is an example URL for the aiosqlite driver:

```
DATABASE_URL=sqlite+aiosqlite:///retrofun.sqlite
```

MySQL

When using MySQL or MariaDB, SQLAlchemy 2.0 supports two asynchronous drivers: aiomysql and asyncmy.

You have to install the chosen package into your virtual environment. For example, here is how to install aiomysql:

```
(venv) $ pip install aiomysql
```

Then the dialect portion of the database connection URL must be changed to reflect the driver in use. Example:

```
DATABASE_URL=mysql+aiomysql://retrofun:my-password@localhost:3306/retrofun
```

PostgreSQL

For PostgreSQL, the asyncpg driver is currently the only asynchronous option.

As with all the other databases, the package needs to be installed:

```
(venv) $ pip install asyncpg
```

Your database connection URL must have `asyncpg` in the dialect part. For example:

```
DATABASE_URL=postgresql+asyncpg://retrofun:my-password@localhost:5432/retrofun
```

Engines, Metadata and Sessions

SQLAlchemy comes with an asynchronous extension that provides alternative engine and session objects. These have the same interfaces as the regular ones you used in previous chapters, but its methods are awaitable.

Below you can see a version of *db.py* that is appropriate for an asynchronous application. If you intend to run this code on your computer, you can create a new project directory for the asynchronous code, so that you can keep the code from previous chapters available in case you need it.

Listing 36 *db.py*: Asynchronous engine, metadata and sessions

```
import os
from dotenv import load_dotenv
from sqlalchemy import MetaData
from sqlalchemy.ext.asyncio import create_async_engine, async_sessionmaker
from sqlalchemy.orm import DeclarativeBase


class Model(DeclarativeBase):
    metadata = MetaData(naming_convention={
        "ix": "ix_%(column_0_label)s",
        "uq": "uq_%(table_name)s_%(column_0_name)s",
        "ck": "ck_%(table_name)s_%(constraint_name)s",
        "fk": "fk_%(table_name)s_%(column_0_name)s_%(referred_table_name)s",
        "pk": "pk_%(table_name)s",
    })


load_dotenv()

engine = create_async_engine(os.environ['DATABASE_URL'])
Session = async_sessionmaker(engine, expire_on_commit=False)
```

As you see, there aren't that many differences from the synchronous version. Instead of create_engine() this version uses `create_async_engine()`, and instead of sessionmaker it uses `async_sessionmaker`.

The only other difference is the expire_on_commit=False option configured for the session. This disables a default SQLAlchemy behavior

that marks models as *expired* after the session is committed. Models that are marked as expired are implicitly refreshed from a database query when any of its attributes are accessed again. Since implicit database activity cannot occur in an asynchronous application, expired objects should not be used. The expire_on_commit=False option makes sure no models will ever be marked as expired as a result of a commit.

The disadvantage of not having expired models is that when using long-lived sessions that are committed several times, models will be assumed to always be updated and will never be refreshed from the database. In a situation where the database can be modified by different processes this can cause the long-lived session to end up with stale models. To avoid this problem the application can use shorter sessions, or it can also manually *expunge* objects from the session and load them again to ensure their freshness. The session.expunge() and session.expunge_all() can be used to remove models from the session as necessary, and the session.refresh() can be used to explicitly update an object from the database.

One interesting aspect of the asynchronous support is that the MetaData instance does not have an asynchronous version. This is of particular importance when the create_all() and drop_all() functions are used, because these do not have awaitable versions. SQLAlchemy provides a run_sync() method that can be used to run synchronous database code such as these functions and await them, as shown below:

```
async with engine.begin() as connection:
    await connection.run_sync(Model.metadata.drop_all)
    await connection.run_sync(Model.metadata.create_all)
```

Relationships Loaders

For the most part, model definitions do not need to change for an asynchronous application. The one are that needs to be carefully checked is the configuration of the relationship loaders.

You have seen that many of the relationship() attributes in the model classes use a lazy loading mechanism that queries the relationship from the

database the first time an attribute is accessed. You have also seen that the lazy argument, the options() query clause, and the WriteOnlyMapped typing hint can all be used to change this behavior. The default lazy behavior, which maps to lazy='select' or options(lazyload(...)), is incompatible with asynchronous applications, so it has to be changed to a loader with a more predictable behavior.

But what to use instead? Here is once again the table that shows all the available loaders, categorized by when the database is accessed:

When	Loaders
Lazy load	select (default), dynamic (legacy)
Eager load	joined, selectin, subquery, immediate
Explicit load	write_only (SQLAlchemy 2.0 and up), noload, raise, raise_on_sql

The lazy loaders are out, so the main choice you have to make for each relationship object is if it should be loaded eagerly along with its parent model or explicitly only when and if needed. Once you decide which of the two makes most sense you can look at the different options each method offers. Some relationships were already changed from the lazy default to write_only, and those do not need to change since this loader never issues implicit database queries.

A safe choice is to change all the lazy loading relationships to lazy='raise', so that they raise an error if SQLAlchemy would need to lazy load them. With this in place, the application can explicitly pass one of the eager loaders in an options() clause as an explicit override when a relationship needs to be loaded.

Another option is to pick appropriate loaders for all relationships to avoid any possibility of relationships being lazy loaded. This is how all the relationships in the RetroFun database will be changed to avoid implicit database queries:

- The "one" side relationships will use the joined eager loader. If the relationship is not optional, the innerjoin=True option will be added to tell SQLAlchemy to use an inner join, which is often more efficient than the default left outer join this loader uses.
- The "many" relationships that use the write_only loader will not be changed, as these are compatible with asynchronous code.
- The remaining "many" relationships will use the selectin eager loader, which often performs better than joined when there are multiple items in the relationship.

The following code block shows all the updates that need to be made to the relationships. No other changes need to be made in *models.py* beyond that. If you intend to try the asynchronous solution, copy the *models.py* from the previous chapter to the directory where you are building the asynchronous project and edit the relationships as shown below.

Listing 37 *models.py*: Asynchronous-friendly relationships

```
# ...

class Product(Model):
    # ...
    manufacturer: Mapped['Manufacturer'] = relationship(
        lazy='joined', innerjoin=True, back_populates='products')
    countries: Mapped[list['Country']] = relationship(
        lazy='selectin', secondary=ProductCountry, back_populates='products')
    order_items: WriteOnlyMapped['OrderItem'] = relationship(
        back_populates='product')
    product_reviews: WriteOnlyMapped['ProductReview'] = relationship(
        back_populates='product')
    blog_articles: WriteOnlyMapped['BlogArticle'] = relationship(
        back_populates='product')
    # ...


class Manufacturer(Model):
    # ...
    products: Mapped[list['Product']] = relationship(
        lazy='selectin', cascade='all, delete-orphan',
        back_populates='manufacturer')
    # ...


class Country(Model):
    # ...
    products: Mapped[list['Product']] = relationship(
        lazy='selectin', secondary=ProductCountry,
        back_populates='countries')
    # ...


class Order(Model):
    # ...
    customer: Mapped['Customer'] = relationship(
```

```
        lazy='joined', innerjoin=True, back_populates='orders')
    order_items: Mapped[list['OrderItem']] = relationship(
        lazy='selectin', back_populates='order')
    # ...


class Customer(Model):
    # ...
    orders: WriteOnlyMapped['Order'] = relationship(back_populates='customer')
    product_reviews: WriteOnlyMapped['ProductReview'] = relationship(
        back_populates='customer')
    blog_users: WriteOnlyMapped['BlogUser'] = relationship(
        back_populates='customer')
    # ...


class OrderItem(Model):
    # ...
    product: Mapped['Product'] = relationship(
        lazy='joined', innerjoin=True, back_populates='order_items')
    order: Mapped['Order'] = relationship(
        lazy='joined', innerjoin=True, back_populates='order_items')
    # ...


class ProductReview(Model):
    # ...
    product: Mapped['Product'] = relationship(
        lazy='joined', innerjoin=True, back_populates='product_reviews')
    customer: Mapped['Customer'] = relationship(
        lazy='joined', innerjoin=True, back_populates='product_reviews')
    # ...


class BlogArticle(Model):
    # ...
    author: Mapped['BlogAuthor'] = relationship(
        lazy='joined', innerjoin=True, back_populates='articles')
    product: Mapped[Optional['Product']] = relationship(
        lazy='joined', back_populates='blog_articles')
    views: WriteOnlyMapped['BlogView'] = relationship(back_populates='article')
    language: Mapped[Optional['Language']] = relationship(
        lazy='joined', back_populates='blog_articles')
    translation_of: Mapped[Optional['BlogArticle']] = relationship(
        lazy='joined', remote_side=id, back_populates='translations')
    translations: Mapped[list['BlogArticle']] = relationship(
        lazy='selectin', back_populates='translation_of')
    # ...


class BlogAuthor(Model):
    # ...
    articles: WriteOnlyMapped['BlogArticle'] = relationship(
        back_populates='author')
    # ...


class BlogUser(Model):
    # ...
    customer: Mapped[Optional['Customer']] = relationship(
        lazy='joined', back_populates='blog_users')
    sessions: WriteOnlyMapped['BlogSession'] = relationship(
        back_populates='user')
    # ...


class BlogSession(Model):
    # ...
    user: Mapped['BlogUser'] = relationship(
        lazy='joined', innerjoin=True, back_populates='sessions')
    views: WriteOnlyMapped['BlogView'] = relationship(back_populates='session')
    # ...
```

```
class BlogView(Model):
    # ...
    article: Mapped['BlogArticle'] = relationship(
        lazy='joined', innerjoin=True, back_populates='views')
    session: Mapped['BlogSession'] = relationship(
        lazy='joined', innerjoin=True, back_populates='views')
    # ...


class Language(Model):
    # ...
    blog_articles: WriteOnlyMapped['BlogArticle'] = relationship(
        back_populates='language')
    # ...
```

Alembic Configuration

Database migrations is another area that requires some minimal changes when switching to the asynchronous programming model. Alembic uses the concept of *templates* to generate the contents of the migration repositories that it creates with the init command, in particular the *env.py* and *alembic.ini* files. The default Alembic template, which you used for the RetroFun database in previous chapters, assumes your database engine and driver are synchronous.

Alembic ships with an asynchronous template that can be used when initializing a migration repository. The command below creates the repository with this template. If you want to try this command, make sure you have the asynchronous versions of *db.py* and *models.py* in a separate directory that does not have a migration repository created.

```
(venv) $ alembic init -t async migrations
```

The resulting *env.py* file in the *migrations* subdirectory will have a few minor differences with the one based on the default template.

As you've done before, this file needs to be edited so that Alembic knows about the project's database. The changes are similar to those made in the synchronous version. First, add the imports at the top:

Listing 38 *migrations/env.py*: Alembic imports

```
from db import Model, engine
import models
```

Then find the line that initializes the target_metadata variable and enter the following code in its place:

Listing 39 *migrations/env.py*: Configure the project's database into Alembic

```
target_metadata = Model.metadata
config.set_main_option("sqlalchemy.url", engine.url.render_as_string(
    hide_password=False))
```

The last change is to enable the batch migration mode. This is especially important if you are using SQLite because this database has limited migration capabilities on its own, but it can be safely enabled for all databases. Find the context.configure() call in the do_run_migrations() function, and make sure it includes the `render_as_batch=True` option.

Listing 40 *migrations/env.py*: Configure batch mode

```
def do_run_migrations(connection: Connection) -> None:
    context.configure(connection=connection, target_metadata=target_metadata,
                      run_as_batch=True)

    with context.begin_transaction():
        context.run_migrations()
```

Now Alembic is fully configured, and you should be able to generate an initial database migration. Before you run the following command, make sure your *.env* file has the DATABASE_URL variable configured with an asynchronous database driver as shown above, and also that you have configured a brand-new database and not the same one you've used before.

```
(venv) $ alembic revision --autogenerate -m "initial migration"
```

This command will scan your database models and compare them with the still empty database, so the initial database migration is going to include all the tables, indexes and constraints that map to these models.

Now that the migration script is in place, the database can be migrated with it:

```
(venv) $ alembic upgrade head
```

Implicit I/O After a Session is Flushed

With the changes made above to the models, almost all implicit database activity is now disabled. There is one remaining implicit situation that occurs when a session that has new objects is flushed.

In the context of a session, a flush() operation writes all the outstanding changes that were accumulated in the session to the underlying database transaction, so that they are known to the database. Flushes are often issued automatically by SQLAlchemy because sessions have their autoflush option enabled by default, which issues a flush() call before a database query to ensure that the query includes results from the session that have not been committed yet.

For the most part, a flush() call does not cause issues, but there is one particular situation in which it does. If the session has new objects that have just been added, and these objects have list-style relationship attributes that have not been initialized, then when these objects are flushed the uninitialized relationships will be marked as not loaded, which means that the next time they are accessed a lazy load attempt will be made on them.

This problem is somewhat obscure, so it may be hard to understand how it can affect an application. If you have made all the updates for asynchronous compatibility, you can trigger this error easily in a Python session to understand it better. Start an async-friendly Python session with the following command:

```
(venv) $ python -m asyncio
```

The difference between this and just running python is that with this command it is possible to use the await keyword directly from the prompt. A regular Python session only allows await inside functions declared with `async def`.

Here is a simple demonstration of the error after flush:

```
>>> from db import Session
>>> from models import Customer, Order
>>> session = Session()
>>> c = Customer(name='Susan')  # order_items has not been initialized explicitly
>>> o = Order(customer=c)
>>> session.add(o)
>>> o.order_items  # no error before flush
[]
>>> await session.flush()  # flush marks the order_items relationship as unloaded
```

```
>>> o.order_items  # error after flush!
Traceback ...
```

There are several ways to avoid lazy loading of relationships after the session is flushed:

- Use the write_only loading mechanism for all the list-style relationships. This loader requires the application to load the relationship explicitly, so it will never trigger a lazy load operation.
- Use the raise loader for all the list-style relationships and override this loader through an options() clause when the relationship needs to be loaded. This solution does not solve the problem, but it will alert you if the application ever attempts to lazy load a relationship with an error that is less cryptic than the one above.
- Disable the autoflush option in the session, even though this may produce unexpected query results that do not include outstanding changes in the session, because without the flush these would not be known to the database until a commit() is issued. If you are interested in trying this out, here is how you can reconfigure the asynchronous session to not issue flushes before queries:

```
session = async_sessionmaker(engine, expire_on_commit=False, autoflush=False)
```

- Ensure that all the list-style relationships are initialized to a value before the session is flushed. This will make SQLAlchemy flush the relationship as well, and preserve their value after the flush.

These solutions all have their pros and cons, so you should evaluate which one provides the most value for your application. The last proposed solution is the one that imposes the least amount of restrictions, as it can work with the autoflush option enabled, while also allowing the relationship loaders that use list semantics. So that is the solution that will be implemented for the asynchronous version of the RetroFun database.

The simplest option to initialize a relationship before the session is flushed is to do it explicitly. Continuing with the above example, here is how to create an Order model instance and initialize its Order.order_items relationship:

```
>>> await session.rollback()  # clear the errored session state from above
>>> o = Order(customer=c, order_items=[])  # order_items is given an initial value
>>> session.add(o)
>>> await session.flush()
>>> o.order_items  # the initial value is preserved after the flush
[]
```

To avoid having to remember to initialize relationships every time a new object is created, it is possible to expand the Model class to automatically initialize all list-based relationships to an empty list. An implementation of this idea is shown below.

Listing 41 *db.py*: Initialize all list relationships

```
from sqlalchemy import event, inspect

# ...

@event.listens_for(Model, "init", propagate=True)
def init_relationships(tgt, arg, kw):
    mapper = inspect(tgt.__class__)
    for arg in mapper.relationships:
        if arg.collection_class is None and arg.uselist:
            continue  # skip write-only and similar relationships
        if arg.key not in kw:
            kw.setdefault(
                arg.key, None if not arg.uselist else arg.collection_class())
```

The init_relationships() function needs to be added at the bottom of *db.py*. The @event.listens_for() decorator added to the function registers the function as a handler that is invoked by SQLAlchemy when the init event of Model occurs, which means that the function will be called every time a new Model instance is created. The propagate=True option extends this event handler to all subclasses of Model, effectively including this behavior in all the models defined by the application.

The body of the function uses SQLAlchemy's inspect() function to perform introspection on the model classes and find all those relationships that need to be initialized.

Note

The event and inspect features of SQLAlchemy are not covered in this book beyond the above example. If you are interested in learning more about them, you can find them in the official documentation:

- Events

- Runtime Inspection API

Import Scripts

Before you get to experience the asynchronous database by running some queries, it is necessary to import all the CSV data files, but to be able to do this the import scripts also have to be adapted to work as asynchronous applications.

The general structure of each import script has to change to use asyncio.

Here is how the scripts will be structured:

```
import asyncio

async def main():
    # ... import logic here

if __name__ == '__main__':
    asyncio.run(main())
```

Database sessions use asynchronous context managers, so `with` statements have to be changed to async with. Example:

```
    async with Session() as session:
        async with session.begin():
            # ... do database work here
```

Finally, queries and commits are now executed asynchronously, so they need to be awaited. This means that all the session.execute(), session.scalar() and session.commit() calls in these scripts have to be prepended with await.

After making these changes, the scripts will be fully compatible with the asyncio support in SQLAlchemy. If you don't want to copy the scripts from the previous chapter and adapt them yourself, you can find the asynchronous versions in the book's GitHub repository.

To import all the data you have to run all the importer scripts in order, as shown below:

```
(venv) $ python import_products.py
(venv) $ python import_orders.py
(venv) $ python import_reviews.py
(venv) $ python import_articles.py
```

```
(venv) $ python import_views.py
(venv) $ python import_languages.py
```

Queries

By now you probably have an idea of how to run many database queries, so what are the changes to run them asynchronously? The good news is that the queries themselves are constructed exactly as before. The query API does not need an asynchronous version because there are no long-running or blocking functions in it.

The `session.execute()`, `session.scalars()` and `session.scalar()` functions, however, have to be awaited as they run asynchronously. In addition, the asynchronous session offers two additional execution methods called `session.stream()` and `session.stream_scalars()` that ar demonstrated below.

Start by running a fresh asynchronous Python shell:

```
(venv) $ python -m asyncio
```

As discussed above, this will make the entire shell session run inside an asyncio loop, giving you the possibility of use await directly in the prompt, without having to create a wrapper function.

Now you can import all the needed symbols and manually start a database session:

```
>>> from sqlalchemy import select
>>> from db import Session
>>> from models import Product, Customer, Order
>>> session = Session()
```

Start by retrieving the "Commodore 64" product:

```
>>> c64 = await session.scalar(
      select(Product)
         .where(Product.name == 'Commodore 64'))
>>> c64
Product(41, "Commodore 64")
```

The `manufacturer` and `countries` relationships in the Product model were configured with the `joined` and `selectin` eager loaders respectively, so

they were automatically loaded when the query above was issued. This can be confirmed:

```
>>> c64.manufacturer
Manufacturer(14, "Commodore")
>>> c64.countries
[Country(3, "USA")]
```

Let's try to get the last customer in alphabetical order:

```
>>> c = await session.scalar(
      select(Customer)
         .order_by(Customer.name.desc())
         .limit(1))
>>> c
Customer(e084528681ab4cb7bf45413ad6c7ce45, "Zoe Bradley")
```

All the relationships in the Customer model use the write_only loader. As you've seen in previous chapters, to get the items in the relationship, the select query returned by the relationship attribute has to be manually executed. The next example gets the last two orders from this customer:

```
>>> r = await session.scalars(
      c.orders.select()
         .order_by(Order.timestamp.desc())
         .limit(2))
>>> r.all()
[Order(eaf9c1386a514c9781bdd849f7e99787), Order(db2c90dcc4ae4072b12a58496f47f5cf)]
```

Here, the query to obtain the orders is returned by the select() method of the Customer.orders relationship attribute. Because this is a query object, it can be expanded with additional clauses before it is executed in the session, giving the most freedom in accessing relationships, especially if they can potentially have many elements.

Streamed Results

As you have seen, the execute() and scalars() methods return a results object that is a standard, non-asynchronous Python iterable, and this is the case also when using an asynchronous session.

When using standard Python the results object is very efficient, as it only loads one item at a time from the database. However, when using the asynchronous session SQLAlchemy is forced to retrieve the entire list of results from the database before returning the results, because asynchronous

activity is not possible inside a standard Python iterable. So these results are not efficient when using asynchronous code, especially for large queries.

The stream() and stream_scalars() methods were added to provide the same efficient iteration of results in the asynchronous session. These methods function like the original counterparts, with the only difference that they return an asynchronous version of the results object that supports Python's asynchronous iteration protocol. The last query above can be issued more efficiently as a stream:

```
>>> r = await session.stream_scalars(
      c.orders.select()
         .order_by(Order.timestamp.desc())
         .limit(2))
>>> [order async for order in r]
[Order(eaf9c1386a514c9781bdd849f7e99787), Order(db2c90dcc4ae4072b12a58496f47f5cf)]
```

Here you can see that the streamed results can be access inside an async for loop or list comprehension. The all() method is also available for cases that do not benefit from asynchronous iteration.

In general, you should use stream() instead of execute() when expecting many values per row, and stream_scalars() instead of scalars() for single value per row queries. The standard scalar(), scalar_one() and scalar_or_none() methods can be used safely in an asynchronous application.

SQLAlchemy and the Web

Whether you are building a traditional web application, or a web API that works alongside a web front end or smartphone app, SQLAlchemy is one of the best choices to add database support to a Python web server. In this chapter two example integrations with Flask and FastAPI will be demonstrated. These are two of the most popular Python web frameworks and should serve as examples even if you use another web framework.

General Integration Approach

If you are looking for the easiest method to integrate SQLAlchemy into a web application, then you should consider not using an integration at all.

Following the structure presented in this book, you can add the *db.py* and *models.py* modules to your application and then use the session context manager to include database functionality in every place where it is needed. This is, in fact, the technique that was used in all the importer scripts that were presented in previous chapters.

This approach is suitable not only for web applications but for any other types of Python applications, and it has the advantage that it does not require any additional dependencies or extensions.

SQLAlchemy Integration Techniques

While the no-integration option suggested in the previous section should work just fine with a lot of projects, you may prefer to use a solution that encapsulates and simplifies the database functions in a way that is convenient for your chosen web framework. The sections that follow discuss some implementation details that should be considered.

Disambiguation of SQLAlchemy Imports

One aspect of SQLAlchemy that is sometimes tedious is that it exports many classes and functions. Consider the SQLAlchemy imports used in the *models.py* module from the previous chapter:

```
from sqlalchemy import String, Text, ForeignKey, Table, Column
from sqlalchemy.orm import Mapped, WriteOnlyMapped, mapped_column, relationship
```

If you need to run database queries, then you have to import Session, the select() function and a few additional symbols, depending on your needs.

Having these long lists of imports in an application has two disadvantages.

First, it is time-consuming to maintain these import lists at the top of every module that needs to use the database, but more importantly, some imports with fairly generic names such as select may collide with symbols from other dependencies or from the application itself.

A useful technique that addresses these two concerns is to *namespace* all the imports by only importing the parent modules. For a SQLAlchemy ORM application there are usually two parent modules, sqlalchemy and sqlalchemy.orm, so these can be imported directly:

```
import sqlalchemy
import sqlalchemy.orm
```

When doing this, all the symbols can be accessed through their parent, for example `sqlalchemy.select` or `sqlalchemy.orm.relationship`. To avoid the long names, the imports can be renamed to shorter prefixes:

```
import sqlalchemy as sa
import sqlalchemy.orm as so
```

Now the symbols are prefixed with `sa.` and `so.` for SQLAlchemy Core and ORM respectively. Here is how the `Order` model looks when using this style:

```
class Product(Model):
    __tablename__ = 'products'

    id: so.Mapped[int] = so.mapped_column(primary_key=True)
    name: so.Mapped[str] = so.mapped_column(
        sa.String(64), index=True, unique=True)
    manufacturer_id: so.Mapped[int] = so.mapped_column(
        sa.ForeignKey('manufacturers.id'), index=True)
    year: so.Mapped[int] = so.mapped_column(index=True)
    cpu: so.Mapped[Optional[str]] = so.mapped_column(sa.String(32))

    manufacturer: so.Mapped['Manufacturer'] = so.relationship(
```

```
        back_populates='products')      countries: so.Mapped[list['Country']] =
so.relationship(
      secondary=ProductCountry, back_populates='products')
   order_items: so.WriteOnlyMapped['OrderItem'] = so.relationship(
      back_populates='product')
    product_reviews: so.WriteOnlyMapped['ProductReview'] = so.relationship(
                        back_populates='product')                  blog_articles:
so.WriteOnlyMapped['BlogArticle'] = so.relationship(
      back_populates='product')


   def __repr__(self):
      return f'Product({self.id}, "{self.name}")'
```

Model Serialization

A need that is specific to web applications and web-based APIs is to send models to clients that request them. To be sent over the network, these entities have to be *serialized*, which is a process that converts the Python model instance from its internal binary representation to a string or byte sequence that can be transmitted and then reconstructed on the other side.

The most commonly used serialization format is JavaScript Object Notation or JSON (pronounced "Jason"). Here is how a Product entity from the RetroFun database might look once serialized to the JSON format:

```
{      "id":  41,      "name":
"Commodore 64",
 "manufacturer": {
   "id": 14,
   "name": "Commodore"
 },   "countries": [
   {
     "id": 3,
     "name": "USA"
   }  ],   "year": 1982,
 "cpu": "6510"
}
```

The JSON format is similar in syntax to Python dictionaries, lists and primitive types such as integers and strings. In fact, the json module from the Python standard library can render a Python dictionary with the above structure to the corresponding JSON serialized representation, which is returned as a string that can be sent in the response to a client.

The reverse process to serialization is called *deserialization*. In the case of JSON, it is carried out by a JSON decoder. There are JSON decoders for all languages and technology stacks, so when the client receives a response

string from the server with the above contents, it can decode it into a format that is convenient. In the case of JavaScript running in the browser, the JSON.parse() function converts a JSON payload into a structure based on JavaScript objects, arrays and primitive types.

Looking at the structure of the above example more closely, you will find that this particular representation not only includes a product, but also some of its relationships, namely the manufacturer and countries of origin, which have their own JSON representations embedded into the parent entity.

Building a serialization system that can easily generate responses that recursively include relationships is actually not too difficult. The basic idea is to add a to_dict() method to each model class that returns a dictionary version of the object that can be serialized to the JSON format.

Below you can see the implementation of the Product model's to_dict() method as an example:

```
class Product(Model):
    # ...

    def to_dict(self):
        return {
            'id': self.id,
            'name': self.name,
            'manufacturer': self.manufacturer.to_dict(),
            'year': self.year,
            'cpu': self.cpu,
            'countries': [country.to_dict() for country in self.countries],
        }
```

This example shows how related objects are embedded into the representation of the parent object by calling their own to_dict() methods, which ensures that the logic to serialize an entity is kept in a single place.

Note

An alternative to the to_dict() serialization methods that can be useful with large and complex models is to use a specialized serialization library such as Marshmallow.

The Alchemical Package

When working with SQLAlchemy, there are a few objects that need to be created in every project:

- An Engine instance.
- A MetaData instance with an explicit naming convention for indexes and constraints.
- A Model declarative base class.
- A Session base class associated with the engine.

In the code examples presented in previous chapters all these objects were initialized in the *db.py* module, and this or a similar module would need to be included in every project that integrates SQLAlchemy.

The ~~Alchemical~~ package (created and maintained by the author of this book) attempts to simplify the use of SQLAlchemy by encapsulating all the above items into an Alchemical instance. You can install Alchemical with pip:

```
(venv) $ pip install alchemical
```

Consider the following example, which provides the same functionality as the *db.py* module:

```
import os
from dotenv import load_dotenv
from alchemical import Alchemical

load_dotenv()

db = Alchemical(os.environ['DATABASE_URL'])
```

The db object created from the Alchemical class contains all the SQLAlchemy items enumerated above. Here is how to access them:

- The engine instance is managed internally by the Alchemical object and is normally not referenced directly by the application. If necessary, it can be obtained by calling db.get_engine().

- The MetaData instance is also managed internally without the application needing to reference it. If necessary, it is available as `db.metadata`.
- The db.create_all() and db.drop_all() methods create and destroy the database tables respectively.
- The declarative base class is db.Model.
- The session base class is db.Session. To create a session and begin a transaction on it, a single context manager with db.begin() can be used instead of the two context managers required by SQLAlchemy.

The Alchemical class provides additional features:

- Ability to maintain connections to multiple databases, something that is hard to implement manually for projects that use the ORM module.
- Simplified support for Alembic database migrations.
- Asynchronous support, when imported from the alchemical.aio module.
- Integration with the Flask web framework, when imported from the `alchemical.flask` module.

An Example Web Application

The GitHub repository for this book includes a complete example of a small web application that presents a table of RetroFun orders with support for efficient pagination, sorting, searching and informational tooltips. Implementations for Flask and FastAPI are provided as examples of traditional and asynchronous integrations.

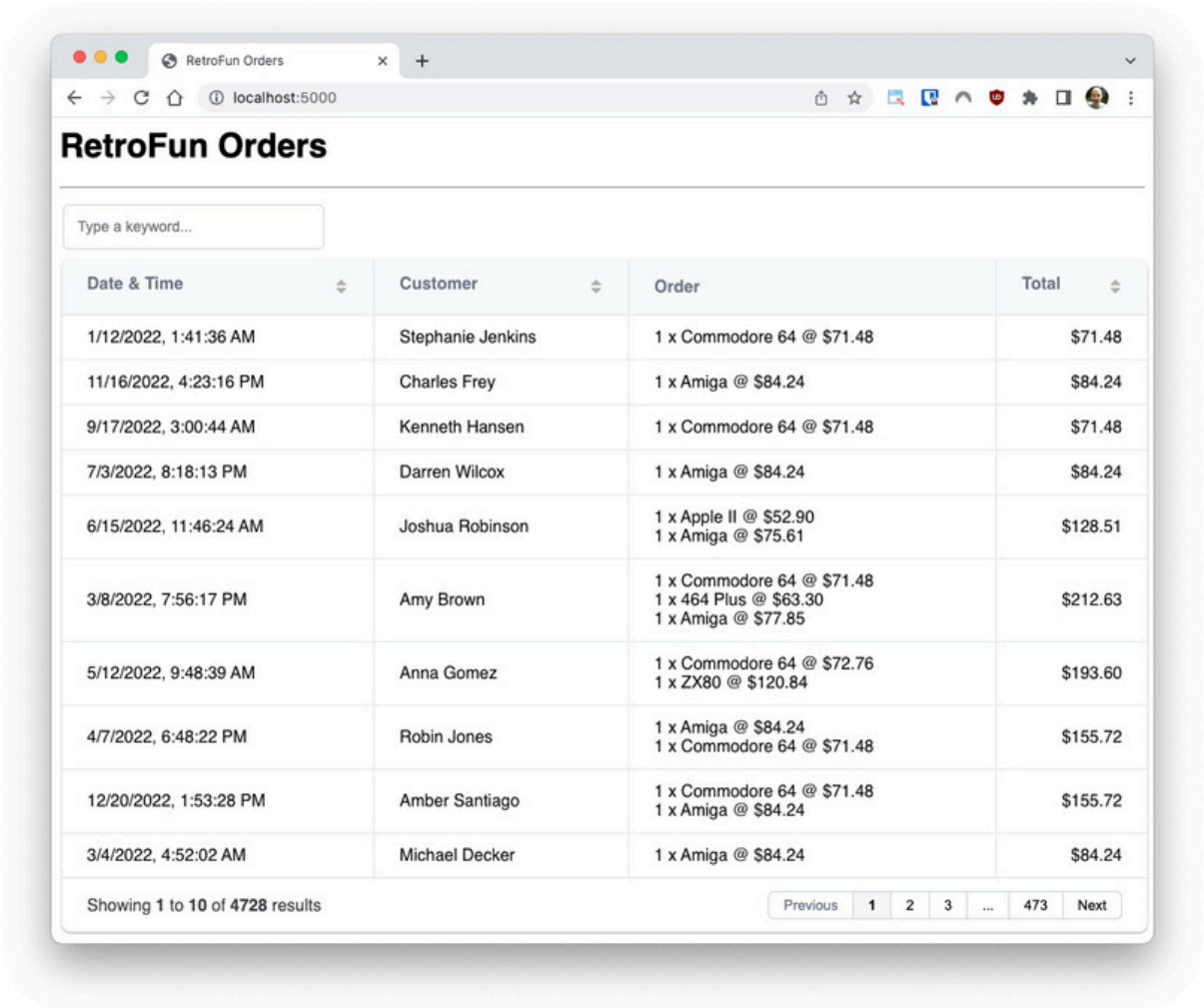


Table Front End

The table featured in the example applications uses the grid.js library. This table is configured in "server-side" mode, which means that it obtains data by making requests to the server. This is the most efficient way to configure the table, because only the data that needs to be displayed is requested. Whenever the user clicks on a pagination link, a column sorting header, or types something in the search box, a new request is made to the server to refresh the table with updated information.

Discussing the front end implementation details of this table falls outside the scope of this book, but you can inspect the client-side source code in the *index.html* file included with the two example applications.

What's important to know about the front end is that when the table needs to display new data, a request is issued to the server's */api/orders* endpoint with the following query string parameters:

- start: the 1-based index of the first element that needs to be displayed.
- length: the number of elements that the table needs.
- sort: a comma-separated list of orderings, where each item starts with a + or - for ascending or descending order, followed by the field name. For example, +customer,-total sorts the list by customer name in ascending order, and secondarily by order total in descending order.
- search: the search string typed by the user in the search field, or an empty string if there is no search requested. If a search string is given, only orders that have a matching customer name or product name are expected to be returned.

Here is an example request URL that the front would issue to get the third page of results with a page size of 10, with orders sorted by their total amount in ascending order and with a search string of Dylan:

```
http://domain.com/api/orders?start=21&length=10&sort=%2Btotal&search=Dylan
```

Note the value of the sort argument, which is %2Btotal. Certain characters in URLs have to be escaped, and the + is one of them. The encoding uses the hexadecimal ASCII code for the character, with a % prefix. A web framework such as Flask or FastAPI handle character escaping transparently, so normally the developer does not need to be concerned with this task.

The role of the */api.orders* endpoint in the back end is to accept these four query string parameters, execute a database query based on them and return the items requested using the following JSON structure:

```
{   "data": [     { ... order
... },
    { ... order ... },
    ...
 ],
```

```
  "total": <n>
}
```

The data section of the response must include an array of orders, each formatted according to its to_dict() serialization method. There should be up to length orders included in the response. The total field should include the total number of entries that satisfy the current search criteria, or the total number of orders when there is no search defined. This is so that the table can show a legend such as "Showing items 31 to 40 of 4798 results".

Database Queries

One of the most important parts of the back end is the logic that generates the queries that solve the request from the client. It's queries in plural, because the expected JSON payload needs one query for the data section of the response and another for total.

The query for the total is actually the simpler of the two. This query needs to calculate the count of orders that match the search string, or the total count of orders if there is no search string. Only the search query string argument is used for this query. The start, length and sort arguments do not have any effect on this calculation.

The total_orders() function shown below, which will be part of a *queries.py* module in the example applications, creates this query.

```
def total_orders(search):
    if not search:
        return sa.select(sa.func.count(Order.id))

    return (
        sa.select(sa.func.count(sa.distinct(Order.id)))
            .join(Order.customer)
            .join(Order.order_items)
            .join(OrderItem.product)
            .where(
                sa.or_(
                    Customer.name.ilike(f'%{search}%'),
                    Product.name.ilike(f'%{search}%'),
                )
            )
    )
```

There are two different implementations for this query, depending on the existence of a search string. When no search string was given, a simple

query that returns the total count of orders is used.

When there is a search string, the query is more complex. The select() portion still specifies a count, but this time unique orders must be counted, because the joins with customers and products can create duplicate results, as you have seen in many example queries.

Joining Order with Customer makes the customer names searchable in the query. To also be able to search product names, Order is joined with OrderItem, which in turn is joined with Product. Recall that OrderItem is the join table for the many-to-many relationship between orders and products.

With all the joins in place, the search is carried out with a where() clause that has two conditions combined with the "or" logical operator. The ilike() function is used to run a case-insensitive pattern search of the given search string on the Customer.name and Product.name columns.

The query that returns a page worth of orders is implemented in a paginated_orders() function, shown below.

```
def paginated_orders(start, length, sort, search):      # base query to retrieve
orders with their total amount      total = sa.func.sum(OrderItem.quantity *
OrderItem.unit_price).label(None)
   q = (
      sa.select(Order, total)
         .join(Order.customer)
         .join(Order.order_items)          .join(OrderItem.product)
         .group_by(Order)
         .distinct()
   )


   # add search filters
   if search:
      q = q.where(
         sa.or_(
            Customer.name.ilike(f'%{search}%'),
            Product.name.ilike(f'%{search}%'),
         )
      )

   # add sorting
   if sort:
      order = []
      for s in sort.split(','):
         direction = s[0]  # first character is either + or -
         name = s[1:]  # rest of the string is the column name
         if name == 'customer':
            column = Customer.name
         elif name == 'total':
            column = total
         else:
            column = getattr(Order, name)
         if direction == '-':
```

```
            column = column.desc()
        order.append(column)
    if not order:
        order = [Order.timestamp.desc()]
    q = q.order_by(*order)

# add pagination
q = q.offset(start).limit(length)

return q
```

This query takes significant more work to create. One interesting implementation choice in this function is that the query is built in four separate chunks that are appended instead of as a single chain of clauses following the select() function call.

The base query obtains orders along with their totals, in a way that is very similar to examples presented in earlier chapters. The join of Order with OrderItem is necessary to be able to calculate the total, and the joins with Customer and Product are done here preventively, to enable the search and sort options.

The second part of the query is conditional on the existence of a search string. This part appends a where() clause that is identical to the one used in the query that calculates the total number of orders.

The third part of the query is also conditional, and only used when there is a sort request. The sort string comes as a comma-separated list, so this section splits the value of sort into each part and then obtains each column to sort by. Supported columns are Customer.name, the total label, or else any of the primary columns of the Order model, of which only timestamp is used in this example. If the column name was given with a - as a prefix, then the desc() method is called on the sorting attribute to reverse the sort. The list of columns that were collected while parsing the sort string are then included in an order_by() clause that is appended to the query.

In the fourth and final section, the pagination offset() and limit() clauses are added, so that the correct range of results are retrieved.

It is important to note that these two functions just create the queries. The separation between creating and executing queries makes it possible to write these queries in a completely generic way that will work without changes in the Flask and FastAPI examples.

Endpoints

This application needs two endpoints. The root URL will return an HTML page that includes the front end JavaScript code. The front end will be configured to make requests to the */api/orders* endpoint when it needs to update the items that are displayed as a result of a user action such as clicking a pagination link.

The endpoints need to be coded according to the conventions set in place by the web framework you are using. Both Flask and FastAPI define endpoints as functions decorated with a route decorator. The syntax used by the two implementations is not identical, but it is fairly similar.

The handler for the root URL does not require any database access, as it just needs to return the HTML file with the front end code. The handler for the */api/orders* endpoint is where the core logic that drives the content of the table is defined. This endpoint must perform the following tasks:

- Obtain the start, length, sort and search parameters given by the client in the query string of the request URL.
- Pass the four parameters to the functions that generate the two database queries.
- Execute the two queries in a database session.
- Return a JSON response with the appropriate format including the results from the queries.

Flask Routes

The Flask version of the two endpoints, which are stored in the *routes.py* module, is shown below.

```
from flask import Blueprint, render_template, request
from .models import db
from . import queries

bp = Blueprint('routes', __name__)
```

```
@bp.route('/')
def index():
    return render_template('index.html')


@bp.route('/api/orders')
def get_orders():
    start = request.args.get('start')
    length = request.args.get('length')
    sort = request.args.get('sort')
    search = request.args.get('search')

    data_query = queries.paginated_orders(start, length, sort, search)
    total_query = queries.total_orders(search)

    orders = db.session.execute(data_query)
    data = [{**o[0].to_dict(), 'total': o[1]} for o in orders]
    return {
        'data': data,
        'total': db.session.scalar(total_query),
    }
```

In Flask, it is a common practice to define the routes of the application in *blueprints*. Each blueprint is then registered with the application instance to get its routes included. In this application, bp is the only blueprint, with the two endpoints described above implemented in the index() and get_orders() functions respectively.

The index() function uses Flask's render_template() function to return the HTML page. The JavaScript logic in this page is going to start sending requests to */api/orders* to feed the orders table.

The get_orders() function is where the table content is generated. First the four parameters are extracted from the query string, which Flask exposes in the request.args dictionary. The data_query and total_query database queries are generated by calling the functions described earlier.

With the Flask integration provided by Alchemical, db.session is a somewhat magical attribute that automatically starts a session the first time it is used. This is a common pattern that is used throughout Flask and many of its extensions, so the Flask integration of Alchemical uses it as well. For this reason there is no need to use the Session context manager to start a session. Alchemical closes the db.session object at the end of the request.

The rest of the get_orders() function executes the two queries via db.session.execute() and db.session.scalar() respectively, ar returns a dictionary that is formatted as required by the client. Flask

automatically renders dictionaries returned as responses to the JSON format.

The following fragment needs to be studied carefully to fully understand it:

```
    orders = db.session.execute(data_query)
    data = [{**o[0].to_dict(), 'total': o[1]} for o in orders]
```

The results from executing data_query are stored in the orders variable. This is a SQLAlchemy results object, which is an iterable. In the second line, a list comprehension iterates over the results and creates the data section of the JSON response. Each element in the list of results must be the serialized Order model, which can be obtained with the expression o[0].to_dict(). But this is insufficient, because the client expects a total attribute, which is not part of the Order model, to also be included in the order. This total is returned as the second value in each result row, so the returned dictionary for each order is assembled with all the data from the Order.to_dict() method, plus the total result.

FastAPI Routes

For FastAPI, the endpoints are stored in the *router.py* module of the application, which you can see next.

```
from fastapi import APIRouter
from fastapi.responses import FileResponse
from .models import db
from . import queries

router = APIRouter()


@router.get('/')
async def index():
    return FileResponse('retrofun/html/index.html')


@router.get('/api/orders')
async def get_orders(start: int, length: int, sort: str = '',
                     search: str = ''):
    data_query = queries.paginated_orders(start, length, sort, search)
    total_query = queries.total_orders(search)

    async with db.Session() as session:
        orders = await session.stream(data_query)
        data = [{**o[0].to_dict(), 'total': o[1]} async for o in orders]
        return {
            'data': data,
            'total': await session.scalar(total_query),
        }
```

What Flask calls a blueprint FastAPI calls an API router, but aside from the name differences both have the same purpose. Since FastAPI is an asynchronous framework, the handler functions are defined as async def functions.

The index() function is registered as the handle for the root URL of the application, and just returns the HTML file with the front end code.

The get_orders() function handles the */api/orders* endpoint. With FastAPI, query string parameters are defined as typed arguments to the function, so the four expected query parameters are included in the function declaration. The start and length parameters are always provided by the client, so there is no need to provide defaults for them, but for sort and search an empty string default is used, in case the client does not send these arguments.

The queries are generated by calling the two functions discussed earlier, exactly as it was done in the Flask version.

To run the queries, a session is started with a context manager. When using the Alchemical package, the base class for sessions is db.Session, with db being the Alchemical instance.

The queries in this version are issued with await, since the database is running in asynchronous mode. The stream() method of the session is used instead of execute(), so that the results are returned as an asynchronous iterator. The list of orders is transformed into the JSON list of orders using an asynchronous list comprehension, in the same way as it was done in the Flask application.

Flask Back End

All the interesting implementation details have already been discussed, so what is left is the boilerplate and glue code that ties all the parts together. The complete code for this application is available in the GitHub repository.

The project has the following structure:

```
- main.py            # The entry point of the application - config.py          # Flask
configuration variables - retrofun        # Python package with the application logic
  - __init__.py     # Package initialization (load environment variables)
  - app.py         # The application factory function
  - models.py       # The Alchemical database instance and models
  - queries.py      # The database queries that support the orders table pagination   -
routes.py      # A Flask blueprint with the two routes of the application
  - templates       # Flask templates directory
    - index.html    # The HTML page of the application
- migrations       # The Alembic migrations directory
- alembic.ini        # The Alembic configuration file - .flaskenv         # Flask-specific
environment variables - .env.template      # Template for the environment variables
needed
- requirements.txt  # the dependencies used by this application
```

The *retrofun* directory is a Python package with the application logic. The *retrofun/queries.py* and *retrofun/routes.py* in this package have been described in earlier sections of this chapter.

The *retrofun/models.py* module defines the Alchemical database instance db and all the models, which use the same definitions as in Chapter 6, extended with to_dict() methods in all the models.

This application has no *db.py* module with database initialization code, because when using Alchemical the database is initialized with a single line of code. Consequently, this initialization has been moved to *models.py*. This is how the database is now initialized in *models.py*:

```
from alchemical.flask import Alchemical

db = Alchemical()
```

The Alchemical class is imported from the alchemical.flask package, so that the Flask integration is used. The database URL is obtained from the ALCHEMICAL_DATABASE_URL entry in the Flask configuration, which is defined in the *config.py* module:

```
import os

ALCHEMICAL_DATABASE_URL = os.environ.get('DATABASE_URL')
```

The *retrofun/app.py* is where the Flask application instance is initialized, using the application factory pattern. The db database instance created in *models.py* functions as a standard Flask extension, so it is initialized in the factory function. Here is how this is done:

```
from flask import Flask
from .models import db
```

```
def create_app():
    app = Flask(__name__)
    app.config.from_object('config')

    db.init_app(app)

    from .routes import bp
    app.register_blueprint(bp)

    return app
```

The *main.py* module in the top-level directory of the project calls `create_app()`:

```
from retrofun.app import create_app

app = create_app()
```

When starting the application with the flask run command, the *.flaskenv* file configures *main.py* as the place where the application is defined, and also sets debug mode:

```
FLASK_APP=main.py
FLASK_DEBUG=true
```

When a WSGI production web server such as Gunicorn is used, the location of the application instance is given with the notation main:app. Here is how to start the web server with Gunicorn:

```
(venv) $ gunicorn -b :5000 main:app
```

The flask command automatically imports the variables defined in the *.env* file, but other web servers do not. The *retrofun/__init__.py* module calls the load_dotenv() function, in case the web server doesn't do it:

```
from dotenv import load_dotenv

load_dotenv()
```

The Alembic database migration repository was created by the Alchemical package, so it is slightly different from the one you created using the alembic init command. To create a migration repository with Alchemical, the following command is used:

```
(venv) $ python -m alchemical.alembic.cli init migrations
```

This is done so that the *alembic.ini* and *migrations/env.py* files are created to be compatible with Alchemical. When creating the repository in this way, the only configuration that needs to be done is to edit the alchemical_db value in the *alembic.ini* file:

```
alchemical_db = retrofun.models:db
```

The *requirements.txt* file lists all the dependencies that are necessary to run the application. You can install them with the following command:

```
(venv) $ pip install -r requirements.txt
```

The project does not include a *.env* file, because the contents of this file depend on the database that you would like to use with the application. A *.env.template* file is included to serve as a template for the real *.env* file. You should create a copy of this file with the name *.env* and then set the value of the DATABASE_URL variable to your database. This application is compatible with the database that you created in earlier chapters of this book, so you can test it out with the same database URL.

FastAPI Back End

The FastAPI example is also provided in the GitHub repository. .

This is the structure of this project:

```
- retrofun          # Python package with the application logic
  - __init__.py     # Package initialization (load environment variables)
  - app.py          # The FastAPI application instance
  - models.py       # The Alchemical database instance and models
  - queries.py      # The database queries that support the orders table pagination
  - router.py       # An API router with the two routes of the application
  - html            # HTML pages directory
    - index.html    # The HTML page for the application
- migrations        # The Alembic migrations directory
- alembic.ini       # The Alembic configuration file
- .env.template     # Template for the environment variables needed
- requirements.txt  # the dependencies used by this application
```

As in the Flask version, the *retrofun* directory is a Python package that contains the application logic. Also, similar to the Flask version, the *retrofunc/__init__.py* file imports the environment variables from the *.env* file.

The *retrofun/app.py* module initializes the FastAPI application and registers its routes.

```
from fastapi import FastAPI
from .router import router

app = FastAPI()
app.include_router(router)
```

The *retrofun/models.py* module defines the db database instance and includes all the models, which match the definitions used in Chapter 7 for an asynchronous application, with to_dict() methods added in each model class. Here is how the Alchemical database instance is defined for this application:

```
import os
from alchemical.aio import Alchemical

db = Alchemical(os.environ['DATABASE_URL'])
```

The asynchronous version of Alchemical is imported from the alchemical.aio package. This version makes all the necessary adjustments designed to prevent implicit database queries.

The *retrofun/queries.py* module is framework-agnostic, so it is exactly what was described earlier in this chapter, and also what was used for the Flask application. The *retrofun/router.py* module defines the two routes of the application, and has also been covered already.

As in the Flask version, the Alembic database migration repository was created to be compatible with the Alchemical package. Instead of using the alembic init command to create it, Alchemical provides its own method to create the repository, so that it can insert itself in the configuration.

```
(venv) $ python -m alchemical.alembic.cli init migrations
```

For a migration repository created with this command, the only configuration that needs to be made is to edit the `alchemical_db` variable in *alembic.ini* to point to the db database instance.

```
alchemical_db = retrofun.models:db
```

As in most Python projects, a *requirements.txt* file is included, with the list of all the dependencies that the application needs. These are installed with pip as follows:

```
(venv) $ pip install -r requirements.txt
```

The *.env* is not included in the code repository because its contents depend on the database you intend to use with the application. A *.env.template* file is included to serve as an example when creating a *.env* file. To configure the project, make a copy of the *.env.template* file with the name *.env*, and then set the value of the DATABASE_URL variable appropriately. This project uses the retrofun database that you created following this book, so you can point it at the same database, making sure you use an asynchronous database driver.

A Last Word

Congratulations on reaching the end of this book! As with most technical topics, learning SQLAlchemy does not end here, the journey will continue for you as it does for me.

While I did my best to cover a wide variety of use cases and solutions, SQLAlchemy is a very large framework that can't possibly be covered entirely within the tutorial format of this book. The good news is that every little detail of this library is well covered in the official documentation.

I would like to make a special mention of three of the areas that I have not covered, in case you are interested in researching them on your own:

- Subqueries and Common Table Expressions (CTEs)

  Subqueries and CTEs are standard SQL features that provide two different approaches to create queries that can be issued recursively from other queries. SQLAlchemy ORM has support for both.

- Events

SQLAlchemy has a fairly sophisticated event subsystem that allows an application to be notified via callback functions when certain events occur. An event handler was added in the asynchronous version of this book's database, but there are many more ways to take advantage of this feature.

- ORM Extensions

  The ORM module has several optional extensions, but only the one that implements asynchronous support received coverage in this book. There are other useful extensions such as Automap (to generate model classes from database schemas), Association Proxy (to simplify navigating through multiple relationships) and Hybrid Attributes (to define model attributes that are evaluated as functions of other attributes) that are well worth investigating.

I wish you the best luck with your SQLAlchemy projects!

Solutions to the Exercises

Chapter 2

1. The first three products in alphabetical order built in the year 1983.

```
>>> q = (select(Product)
      .where(Product.year == 1983)
      .order_by(Product.name)
      .limit(3))
>>> session.scalars(q).all()
[Product(17, "Apple IIe"), Product(85, "Aquarius"), Product(26, "Atari 1200XL")]
```

2. Products that use the "Z80" CPU or any of its clones. Assume that all products based on this CPU have the word "Z80" in the cpu column.

```
>>> q = (select(Product)
      .where(Product.cpu.like('%Z80%')))
>>> session.scalars(q).all()
[ ... 63 results ... ]
```

3. Products that use either the "Z80" or the "6502" CPUs, or any of its clones, built before 1990, sorted alphabetically by name.

```
>>> q = (select(Product)        .where(
         or_(Product.cpu.like('%Z80%'), Product.cpu.like('%6502%')),
         Product.year < 1990)
      .order_by(Product.name))
>>> session.scalars(q).all() [ ... 90 results ... ]
```

4. The manufacturers that built products in the 1980s.

```
>>> q = (select(Product.manufacturer)
      .where(Product.year.between(1980, 1989))
      .distinct())
>>> session.scalars(q).all()
[ ... 65 results ... ]
```

5. Manufacturers whose names start with the letter "T", sorted alphabetically.

```
>>> q = (select(Product.manufacturer)
      .where(Product.manufacturer.like('T%'))
      .order_by(Product.manufacturer)
      .distinct())
>>> session.scalars(q).all()
['Tangerine Computer Systems', 'Technosys', 'Tesla', 'Texas Instruments',
'Thomson', 'Timex Sinclair', 'Tomy', 'Tsinghua University']
```

6. The first and last years in which products have been built in Croatia, along with the number of products built.

```
>>> q = select(
       func.min(Product.year), func.max(Product.year),
       func.count(Product.id)
    ).where(Product.country == 'Croatia')
>>> session.execute(q).first()
(1981, 1984, 4)
```

7. The number of products that were built each year. The results should start from the year with the most products, to the year with the least. Years in which no products were built do not need to be included.

```
>>> product_count = func.count(Product.id).label(None)
>>> q = (select(Product.year, product_count)
       .group_by(Product.year)
       .order_by(product_count.desc()))
>>> session.execute(q).all()
[(1983, 24), (1984, 21), (1985, 21), (1982, 17), (1986, 11), (1980, 10),
(1979, 9), (1977, 7), (1987, 6), (1981, 6), (1990, 5), (1989, 4), (1988, 2),
(1978, 2), (1969, 1), (1995, 1), (1992, 1), (1991, 1)]
```

8. The number of manufacturers in the United States (note that the country field for these products is set to USA)

```
>>> q = (select(func.count(Product.manufacturer.distinct()))
       .where(Product.country == 'USA'))
>>> session.scalar(q)
17
```

Chapter 3

1. The list of products made by IBM and Texas Instruments.

```
>>> q = (select(Product)
      .join(Product.manufacturer)
      .where(or_(
         Manufacturer.name == 'IBM',
         Manufacturer.name == 'Texas Instruments')))
>>> session.scalars(q).all()
[Product(75, "PCjr"), Product(76, "IBM PS/1"), Product(132, "TI-99/4"),
Product(133, "TI-99/4A")]
```

   Another solution using the `in_()` operator on the column:

```
>>> q = (select(Product)
      .join(Product.manufacturer)
      .where(Manufacturer.name.in_(['IBM', 'Texas Instruments'])))
```

2. Manufacturers that operate in Brazil.

```
>>> q = (select(Manufacturer)
      .join(Manufacturer.products)
      .where(Product.country == 'Brazil')
      .distinct())
>>> session.scalars(q).all()
[Manufacturer(32, "Gradiente"), Manufacturer(46, "Comércio de Componentes Eletrônicos"),
Manufacturer(47, "Microdigital Eletronica"), Manufacturer(59, "Prológica")]
```

   Another solution using `group_by()` instead of `distinct()`:

```
>>> q = (select(Manufacturer)
      .join(Manufacturer.products)
      .where(Product.country == 'Brazil')
      .group_by(Manufacturer))
```

3. Products that have a manufacturer that has the word "Research" in their name.

```
>>> q = (select(Product)
      .join(Product.manufacturer)
      .where(Manufacturer.name.like('%Research%')))
>>> session.scalars(q).all()
[Product(125, "ZX80"), Product(126, "ZX81"), Product(127, "ZX Spectrum"),
Product(128, "Sinclair QL")]
```

4. Manufacturers that made products based on the Z80 CPU or any of its clones.

```
>>> q = (select(Manufacturer)
      .join(Manufacturer.products)
      .where(Product.cpu.like('%Z80%'))
      .distinct())
>>> session.scalars(q).all()
[ ... 39 results ... ]
```

5. Manufacturers that made products that are not based on the 6502 CPU or any of its clones.

```
>>> q = (select(Manufacturer)
      .join(Manufacturer.products)
      .where(not_(Product.cpu.like('%8502%')))
      .distinct())
>>> session.scalars(q).all()
[ ... 76 results ... ]
```

6. Manufacturers and the year they went to market with their first product, sorted by the year.

```
>>> first_year = func.min(Product.year).label(None)
>>> q = (select(Manufacturer, first_year)
      .join(Manufacturer.products)
      .group_by(Manufacturer)
      .order_by(first_year))
>>> session.scalars(q).all()
[ ... 76 results ... ]
```

7. Manufacturers that have 3 to 5 products in the catalog.

```
>>> q = (select(Manufacturer)
      .join(Manufacturer.products)
      .group_by(Manufacturer)
      .having(func.count(Product.id).between(3, 5)))
>>> session.scalars(q).all()
[Manufacturer(54, "Tangerine Computer Systems"), Manufacturer(63, "Sinclair Research"),
Manufacturer(60, "VEB Robotron"), Manufacturer(62, "Sharp"),
Manufacturer(44, "Memotech"), Manufacturer(9, "Atari Corporation"),
Manufacturer(57, "Pravetz"), Manufacturer(20, "Didaktik"), Manufacturer(56, "Philips")]
```

8. Manufacturers that operated for more than 5 years.

```
>>> q = (select(Manufacturer)
      .join(Manufacturer.products)
      .group_by(Manufacturer)
      .having(func.max(Product.year) - func.min(Product.year) > 5))
>>> session.scalars(q).all()
[Manufacturer(34, "IBM"), Manufacturer(52, "Radio Shack"),
Manufacturer(14, "Commodore"), Manufacturer(2, "Amstrad"), Manufacturer(62, "Sharp"),
Manufacturer(9, "Atari Corporation"), Manufacturer(30, "Fujitsu"),
Manufacturer(1, "Acorn Computers Ltd"), Manufacturer(5, "Apple Computer"),
Manufacturer(8, "Atari, Inc.")]
```

Chapter 4

1. Products that were made in UK or USA.

```
>>> q = (select(Product)
      .join(Product.countries)
      .where(Country.name.in_(['UK', 'USA']))
      .distinct())
```

   Note: The distinct() clause is needed in this query to eliminate duplicates for products made jointly by the two countries.

2. Products not made in UK or USA. Products that were made in UK and/or USA jointly with other countries should be included in the query results.

```
>>> q = (select(Product)
      .join(Product.countries)
      .where(not_(Country.name.in_(['UK', 'USA'])))
      .distinct())
>>> session.scalars(q).all()
[ ... 70 results ... ]
```

   An alternative solution without the `in_()` operator:

```
>>> q = (select(Product)
      .join(Product.countries)
      .where(Country.name != 'UK', Country.name != 'USA')
      .distinct())
```

3. Countries with products based on the Z80 CPU or any of its clones.

```
>>> q = (select(Country)
      .join(Country.products)
      .where(Product.cpu.like('%Z80%'))
      .distinct())
>>> session.scalars(q).all()
[Country(11, "Japan"), Country(12, "Brazil"), Country(7, "Belgium"),
Country(24, "Hungary"), Country(16, "Australia"), Country(4, "Netherlands"),
Country(1, "UK"), Country(3, "USA"), Country(25, "Norway"),
Country(21, "East Germany"), Country(5, "Romania"), Country(22, "Portugal"),
Country(6, "Hong Kong"), Country(9, "USSR"), Country(14, "Sweden"),
Country(8, "Czechoslovakia"), Country(23, "Poland")]
```

4. Countries that had products made in the 1970s in alphabetical order.

```
>>> q = (select(Country)
      .join(Country.products)
      .where(Product.year.between(1970, 1979))
      .order_by(Country.name)
      .distinct())
>>> session.scalars(q).all()
[Country(11, "Japan"), Country(14, "Sweden"), Country(3, "USA")]
```

5. The 5 countries with the most products. If there is a tie, the query should select countries in alphabetical order.

```
>>> product_count = func.count(Product.id).label(None)
>>> q = (select(Country, product_count)
      .join(Country.products)
      .group_by(Country)
      .order_by(product_count.desc(), Country.name)
      .limit(5))
>>> session.execute(q).all()
[(Country(3, "USA"), 51), (Country(1, "UK"), 36), (Country(11, "Japan"), 12),
(Country(6, "Hong Kong"), 6), (Country(22, "Portugal"), 6)]
```

6. Manufacturers that have more than 3 products in UK or USA.

```
>>> product_count = func.count(Product.id.distinct()).label(None)
>>> q = (select(Manufacturer, product_count)
      .join(Manufacturer.products)
      .join(Product.countries)
      .where(Country.name.in_(['UK', 'USA']))
      .group_by(Manufacturer)
      .having(product_count > 3))
>>> session.execute(q).all()
[(Manufacturer(1, "Acorn Computers Ltd"), 6), (Manufacturer(2, "Amstrad"), 7),
(Manufacturer(5, "Apple Computer"), 6), (Manufacturer(8, "Atari, Inc."), 7),
(Manufacturer(14, "Commodore"), 10), (Manufacturer(52, "Radio Shack"), 6),
(Manufacturer(63, "Sinclair Research"), 4), (Manufacturer(70, "Timex Sinclair"), 4)]
```

7. Manufacturers that have products in more than one country.

```
>>> country_count = func.count(Country.id.distinct()).label(None)
>>> q = (select(Manufacturer, country_count)
      .join(Manufacturer.products)
      .join(Product.countries)
      .group_by(Manufacturer)
      .having(country_count > 1))
>>> session.execute(q).all()
[(Manufacturer(70, "Timex Sinclair"), 4)]
```

8. Products made jointly in UK and USA.

```
>>> q = (select(Product)
      .join(Product.countries)
      .where(Country.name.in_(['UK', 'USA']))
      .group_by(Product)
      .having(func.count(Country.id) > 1))
>>> session.execute(q).all()
[(Product(138, "Timex Sinclair 1000"),), (Product(139, "Timex Sinclair 1500"),),
(Product(140, "Timex Sinclair 2048"),), (Product(142, "Timex Computer 2068"),)]
```

Note: the trick that makes this query work is that the where() clause filters any products not made in the two countries of interest, so after grouping by product any product with a row count of two must have been linked to both countries.

Chapter 5

1. Orders above $300 in descending ordered by the sale amount from highest to lowest.

```
order_total = func.sum(OrderItem.unit_price * OrderItem.quantity).label(None)
q = (select(Order, order_total)
      .join(Order.order_items)
      .group_by(Order)
      .having(order_total > 300)
      .order_by(order_total.desc()))
>>> session.execute(q).all()
[ ... 50 results ... ]
```

2. Orders that include one or more ZX81 computers.

```
q = (select(Order)
      .join(Order.order_items)
      .join(OrderItem.product)
      .where(Product.name == 'ZX81'))
>>> session.scalars(q).all()
[ ... 3 results ... ]
```

   A possibly more efficient solution with two queries, but one less join:

```
>>> zx81 = session.scalar(
      select(Product)
         .where(Product.name == 'ZX81'))
>>> q = (select(Order)
      .join(Order.order_items)
      .where(OrderItem.product == zx81))
```

3. Orders that include a product made by Amstrad.

```
>>> q = (select(Order)
                        .join(Order.order_items)
      .join(OrderItem.product)
      .join(Product.manufacturer)
      .where(Manufacturer.name == 'Amstrad')
      .distinct())
>>> session.scalars(q).all() [ ... 30 results ... ]
```

   An alternative version with one less join:

```
>>> amstrad = session.scalar(
      select(Manufacturer)
         .where(Manufacturer.name == 'Amstrad'))
>>> q = (select(Order)
      .join(Order.order_items)
      .join(OrderItem.product)
      .where(Product.manufacturer == amstrad)
      .distinct())
```

Note: distinct() is necessary here to remove duplicates from orders that have two or more line items with products by this manufacturer.

4. Orders made on the 25th of December 2022 with two or more line items.

```
>>> q = (select(Order)
      .join(Order.order_items)
      .where(Order.timestamp.between(
         datetime(2022, 12, 25), datetime(2022, 12, 26)))
      .group_by(Order)
      .having(func.count(Order.id) >= 2))
>>> session.scalars(q).all()
[ ... 4 results ... ]
```

Note that technically the above query will also pick up orders made on the 26th of December at exactly 00:00:00.0, because the between() operator is inclusive of the start and end values. A more accurate (and lengthy) query can be built using the extract() function:

```
>>> q = (select(Order)
      .join(Order.order_items)
      .where(
         func.extract('day', Order.timestamp) == 25,
         func.extract('month', Order.timestamp) == 12,
         func.extract('year', Order.timestamp) == 2022)
      .group_by(Order)
      .having(func.count(Order.id) >= 2))
>>> session.scalars(q).all()
```

5. Customers with their first and last order date and time. Hint: the `min()` and max() functions can help with this query.

```
>>> q = (select(Customer, func.min(Order.timestamp),
func.max(Order.timestamp))
      .join(Customer.orders)
      .group_by(Customer))
>>> session.execute(q).all()
[ ... 2754 results ... ]
```

6. The top 5 manufacturers that had the most sale amounts, sorted by those amounts in descending order.

```
>>> order_total = func.sum(OrderItem.unit_price * OrderItem.quantity).label(None)
>>> q = (select(Manufacturer, order_total)
      .join(Manufacturer.products)
      .join(Product.order_items)
      .group_by(Manufacturer)
      .order_by(order_total.desc())
      .limit(5))
>>> session.execute(q).all()
[(Manufacturer(14, "Commodore"), 281666.6599999996),
(Manufacturer(63, "Sinclair Research"), 122582.61999999928),
(Manufacturer(5, "Apple Computer"), 34169.33000000025),
(Manufacturer(1, "Acorn Computers Ltd"), 14018.28000000003),
(Manufacturer(8, "Atari, Inc."), 3154.7399999999984)]
```

7. Products, their average star rating and their review count, sorted by review count in descending order.

```
>>> product_rating =
func.avg(ProductReview.rating).label(None) >>> review_count =
func.count(ProductReview.rating).label(None)
>>> q = (select(Product, product_rating, review_count)
      .join(Product.reviews)      .group_by(Product)
      .order_by(review_count.desc()))
>>> session.execute(q).all() [ ... 125 results ... ]
```

The solution above does not include products that do not have reviews. To include the missing products, the join must be upgraded to a left outer join:

```
>>> q = (select(Product, product_rating, review_count)
      .join(Product.reviews, isouter=True)
      .group_by(Product)
      .order_by(review_count.desc()))
>>> session.execute(q).all()
[ ... 149 results ... ]
```

8. Products and their average star rating, but only counting reviews that include a written comment.

```
>>> product_rating = func.avg(ProductReview.rating).label(None)
>>> q = (select(Product, product_rating)
      .join(Product.reviews)
      .where(ProductReview.comment != None)
      .group_by(Product))
>>> session.execute(q).all()
[ ... 70 results ... ]
```

9. Average star reviews for the Commodore 64 computer in each month of 2022.

```
>>> month = func.extract('month', ProductReview.timestamp).label(None)
>>> year = func.extract('year', ProductReview.timestamp).label(None)
>>> product_rating = func.avg(ProductReview.rating).label(None)
>>> q = (select(year, month, product_rating)
      .join(ProductReview.product)
      .where(Product.name == 'Commodore 64')
      .group_by(year, month)
      .order_by(year, month))
>>> session.execute(q).all()
[(2022, 1, 4.294117647058823), (2022, 2, 3.6551724137931036),
(2022, 3, 3.4375), (2022, 4, 3.975609756097561), (2022, 5, 3.317073170731707),
(2022, 6, 3.6774193548387095), (2022, 7, 3.606060606060606),
(2022, 8, 3.973684210526316), (2022, 9, 3.6666666666666665), (2022, 10, 3.9375),
(2022, 11, 3.8958333333333335), (2022, 12, 3.6)]
```

As in previous exercises, the join can be removed when the product is grabbed in advance:

```
>>> c64 = session.scalar(select(Product).where(Product.name == 'Commodore
64')) >>> q = (select(year, month, product_rating)
      .where(ProductReview.product == c64)
```

```
      .group_by(year, month)
      .order_by(year, month))
```

10. Customers with the minimum and maximum star rating they gave to a product, sorted alphabetically by customer name.

```
>>> min_rating =
func.min(ProductReview.rating).label(None) >>> max_rating
= func.max(ProductReview.rating).label(None)
>>> q = (select(Customer, min_rating, max_rating)
      .join(Customer.product_reviews)
      .group_by(Customer)
       .order_by(Customer.name)) >>> session.execute(q).all()
[ ... 931 results ... ]
```

11. Manufacturers with their average star rating, sorted from highest to lowest rating.

```
>>> product_rating = func.avg(ProductReview.rating).label(None)
>>> q = (select(Manufacturer, product_rating)
      .join(Manufacturer.products)
      .join(Product.reviews)
      .group_by(Manufacturer)
      .order_by(product_rating.desc()))
>>> session.execute(q).all()
[ ... 68 results ... ]
```

    The above solution only reports manufacturers that have at least one product rated. To include manufacturers without any rated products, the joins must be upgraded to left outer:

```
>>> q = (select(Manufacturer, product_rating)
      .join(Manufacturer.products, isouter=True)
      .join(Product.reviews, isouter=True)
      .group_by(Manufacturer)
      .order_by(product_rating.desc()))
>>> session.execute(q).all()
[ ... 76 results ... ]
```

12. Product countries with their average star rating, sorted from highest to lowest rating.

```
>>> product_rating = func.avg(ProductReview.rating).label(None)
>>> q = (select(Country, product_rating)
      .join(Country.products)
      .join(Product.reviews)
      .group_by(Country)
      .order_by(product_rating.desc()))
>>> session.execute(q).all()
[ ... 23 results ... ]
```

    As above, only countries with at least one rated product will be included. To add countries with no ratings, left outer joins must be used:

```
>>> q = (select(Country, product_rating)
      .join(Country.products, isouter=True)
```

```
        .join(Product.reviews, isouter=True)
        .group_by(Country)
        .order_by(product_rating.desc()))
>>> session.execute(q).all()
[ ... 25 results ... ]
```

Chapter 6

1. Blog posts that have received more than 40 page views in March 2020.

```
>>> q = (select(BlogArticle)
      .join(BlogArticle.views)
      .where(BlogView.timestamp.between(
         datetime(2020, 3, 1), datetime(2020, 4, 1)))
      .group_by(BlogArticle)
      .having(func.count(BlogView.id) > 50))
>>> session.execute(q).all()
[(BlogArticle(143, "Evening however issue"),)]
```

2. Blog article with the largest number of translations. In case of a tie, the article that comes first alphabetically should be returned.

```
>>> TranslatedBlogArticle = aliased(BlogArticle)
>>> q = (select(BlogArticle  , func.count(BlogArticle.id))
      .join(TranslatedBlogArticle.translation_of)
      .group_by(BlogArticle)
      .order_by(func.count(BlogArticle.id).desc(), BlogArticle.title)
      .limit(1))
>>> session.scalar(q)
BlogArticle(63, "Business seven ability cup church similar itself")
```

3. Page views in March 2022, categorized by language.

```
>>> page_views = func.count(BlogView.id).label(None) >>> q =
(select(Language, page_views)
      .join(Language.blog_articles)
      .join(BlogArticle.views)
      .where(BlogView.timestamp.between(
                  datetime(2022, 3, 1), datetime(2022, 4, 1)))
      .group_by(Language)       .order_by(page_views.desc()))
>>> session.execute(q).all()
[(Language(1, "English"), 2155), (Language(3, "French"), 512),
(Language(2, "German"), 417), (Language(6, "Portuguese"), 404),
(Language(5, "Spanish"), 305), (Language(4, "Italian"), 283)]
```

4. Page views by article, only considering content in German.

```
>>> page_views = func.count(BlogView.id).label(None)
>>> q = (select(BlogArticle, page_views)
      .join(BlogArticle.views)
      .join(BlogArticle.language)
      .where(Language.name == 'German')
      .group_by(BlogArticle)
      .order_by(page_views.desc()))
>>> session.execute(q).all()
[ ... 21 results ... ]
```

5. Monthly page views between January and December 2022.

```
>>> month = func.extract('month', BlogView.timestamp).label(None)
>>> year = func.extract('year', BlogView.timestamp).label(None)
>>> page_views = func.count(BlogView.id).label(None)
>>> q = (select(year, month, page_views)
```

```
      .where(BlogView.timestamp.between(
          datetime(2022, 1, 1), datetime(2023, 1, 1)))
      .group_by(year, month)
      .order_by(year, month))
>>> session.execute(q).all()
[(2022, 1, 3649), (2022, 2, 3287), (2022, 3, 4076), (2022, 4, 3820),
(2022, 5, 4034), (2022, 6, 3659), (2022, 7, 3900), (2022, 8, 3705),
(2022, 9, 3639), (2022, 10, 4066), (2022, 11, 4034), (2022, 12, 3925)]
```

6. Daily page views in February 2022.

```
>>> day = func.extract('day', BlogView.timestamp).label(None)
>>> month = func.extract('month', BlogView.timestamp).label(None)
>>> year = func.extract('year', BlogView.timestamp).label(None)
>>> page_views = func.count(BlogView.id).label(None)
>>> q = (select(year, month, day, page_views)
      .where(BlogView.timestamp.between(
          datetime(2022, 2, 1), datetime(2022, 3, 1)))
      .group_by(year, month, day)
      .order_by(year, month, day))
>>> session.execute(q).all()
[ ... 28 results ... ]
```

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

Your gateway to knowledge and culture. Accessible for everyone.