```
asyncio.run(main())
```

When we run this, we'll notice the result from our first fetch, and after 2 seconds, we'll see two timeout errors. We'll also see that two fetches are still running, giving output similar to the following:

```
starting <function main at 0x109c7c430> with args () {}
200
We got a timeout error!
We got a timeout error!
finished <function main at 0x109c7c430> in 2.0055 second(s)
<Task pending name='Task-2' coro=<fetch_status_code()>>
<Task pending name='Task-1' coro=<main>>
<Task pending name='Task-4' coro=<fetch_status_code()>>
```

`as_completed` works well for getting results as fast as possible but has drawbacks. The first is that while we get results as they come in, there isn't any way to easily see which coroutine or task we're awaiting as the order is completely nondeterministic. If we don't care about order, this may be fine, but if we need to associate the results to the requests somehow, we're left with a challenge.

The second is that with timeouts, while we will correctly throw an exception and move on, any tasks created will still be running in the background. Since it's hard to figure out which tasks are still running if we want to cancel them, we have another challenge. If these are problems we need to deal with, then we'll need some finer-grained knowledge of which awaitables are finished, and which are not. To handle this situation, asyncio provides another API function called `wait`.

ner-grained control with wait

One of the drawbacks of both `gather` and `as_completed` is that there is no easy way to cancel tasks that were already running when we saw an exception. This might be okay in many situations, but imagine a use case in which we make several coroutine calls and if the first one fails, the rest will as well. An example of this would be passing in an invalid parameter to a web request or reaching an API rate limit. This has the potential to cause performance issues because we'll consume more resources by having more tasks than we need. Another drawback we noted with `as_completed` is that, as the iteration order is nondeterministic, it is challenging to keep track of exactly which task had completed.

`wait` in asyncio is similar to `gather` `wait` that offers more specific control to handle these situations. This method has several options to choose from depending on when we want our results. In addition, this method returns two sets: a set of tasks that are finished with either a result or an exception, and a set of tasks that are still running. This function also allows us to specify a timeout that behaves differently from how other API methods operate; it does not throw exceptions. When needed, this function can solve some of the issues we noted with the other asyncio API functions we've used so far.

The basic signature of `wait` is a list of awaitable objects, followed by an optional timeout and an optional `return_when` string. This string has a few predefined values that we'll examine: `ALL_COMPLETED`, `FIRST_EXCEPTION` and `FIRST_COMPLETED`. It defaults to `ALL_COMPLETED`. While as of this writing, `wait` takes a list of awaitables, it will change in future versions of Python to only accept `task` objects. We'll see why at the end of this section, but for these code samples, as this is best practice, we'll wrap all coroutines in tasks.

Vaiting for all tasks to complete

This option is the default behavior if `return_when` is not specified, and it is the closest in behavior to `asyncio.gather`, though it has a few differences. As implied, using this option will wait for all tasks to finish before returning. Let's adapt this to our example of making multiple web requests concurrently to learn how this function works.

Listing 4.10 Examining the default behavior of `wait`

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from util import async_timed
from chapter_04 import fetch_status


@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        fetchers = \
            [asyncio.create_task(fetch_status(session, 'http
             asyncio.create_task(fetch_status(session, 'http
        done, pending = await asyncio.wait(fetchers)

        print(f'Done task count: {len(done)}')
        print(f'Pending task count: {len(pending)}')

        for done_task in done:
            result = await done_task
            print(result)
```

```
asyncio.run(main())
```

In the preceding listing, we run two web requests concurrently by passing a list of coroutines to `wait`. When we `await` `wait` it will return two sets once all requests finish: one set of all tasks that are complete and one set of the tasks that are still running. The `done` set contains all tasks that finished either successfully or with exceptions. The `pending` set contains all tasks that have not finished yet. In this instance, since we are using the `ALL_COMPLETED` option the `pending` set will always be zero, since `asyncio.wait` won't return until everything is completed. This will give us the following output:

```
starting <function main at 0x10124b160> with args () {}
Done task count: 2
Pending task count: 0
200
200
finished <function main at 0x10124b160> in 0.4642 second(s)
```

If one of our requests throws an exception, it won't be thrown at the `asyncio.wait` call in the same way that `asyncio.gather` did. In this instance, we'll get both the `done` and `pending` sets as before, but we won't see an exception until we `await` the task in `done` that failed.

With this paradigm, we have a few options on how to handle exceptions. We can use `await` and let the exception throw, we can use `await` and wrap it in a `try except` block to handle the exception, or we can use the `task.result ()` and `task.exception()` methods. We can safely call these methods since our tasks in the `done` set are guaranteed to be completed tasks; if they were not calling these methods, it would then produce an exception.

Let’s say that we don’t want to throw an exception and have our application crash. Instead, we’d like to print the task’s result if we have it and log an error if there was an exception. In this case, using the methods on the `Task` object is an appropriate solution. Let’s see how to use these two `Task` methods to handle exceptions.

Listing 4.11 Exceptions with `wait`

```
import asyncio
import logging

@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        good_request = fetch_status(session, 'https:/ / www
        bad_request = fetch_status(session, 'python:/ /bad')

        fetchers = [asyncio.create_task(good_request),
                    asyncio.create_task(bad_request)]

        done, pending = await asyncio.wait(fetchers)

        print(f'Done task count: {len(done)}')
        print(f'Pending task count: {len(pending)}')

        for done_task in done:
            # result = await done_task will throw an excepti
            if done_task.exception() is None:
```

```
                print(done_task.result())
            else:
                logging.error("Request got an exception",
                              exc_info=done_task.exception()


asyncio.run(main())
```

Using `done_task.exception()` will check to see if we have an exception. If we don't, then we can proceed to get the result from `done_task` with the `result` method. It would also be safe to do `result` `=` `await` `done_task` here, although it might throw an exception, which may not be what we want. If the exception is not `None`, then we know that the awaitable had an exception, and we can handle that as desired. Here we just print out the exception's stack trace. Running this will yield output similar to the following (we've removed the verbose traceback for brevity):

```
starting <function main at 0x10401f1f0> with args () {}
Done task count: 2
Pending task count: 0
200
finished <function main at 0x10401f1f0> in 0.123866796493530
ERROR:root:Request got an exception
Traceback (most recent call last):
AssertionError
```

Vatching for exceptions

The drawbacks of `ALL_COMPLETED` are like the drawbacks we saw with `gather`. We could have any number of exceptions while we wait for other coroutines to complete, which we won't see until all tasks complete. This could be an issue if, because of one exception, we'd like to cancel other running requests. We may also want to immediately handle any errors to ensure responsiveness and continue waiting for other coroutines to complete.

To support these use cases, `wait` supports the `FIRST_EXCEPTION` option. When we use this option, we'll get two different behaviors, depending on whether any of our tasks throw exceptions.

No exceptions from any awaitables

If we have no exceptions from any of our tasks, then this option is equivalent to `ALL_COMPLETED`. We'll wait for all tasks to finish and then the `done` set will contain all finished tasks and the `pending` set will be empty.

One or more exception from a task

If any task throws an exception, `wait` will immediately return once that exception is thrown. The `done` set will have any coroutines that finished successfully alongside any coroutines with exceptions. The `done` set is, at minimum, guaranteed to have one failed task in this case but may have successfully completed tasks. The `pending` set may be empty, but it may also have tasks that are still running. We can then use this `pending` set to manage the currently running tasks as we desire.

To illustrate how `wait` behaves in these scenarios, look at what happens when we have a couple of long-running web requests we'd like to cancel when one coroutine fails immediately with an exception.

Listing 4.12 Canceling running requests on an exception

```
import aiohttp
import asyncio
import logging
from chapter_04 import  fetch_status
from util import async_timed


@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        fetchers = \
            [asyncio.create_task(fetch_status(session, 'pyth
             asyncio.create_task(fetch_status(session, 'http
                                                  .com', delay=3
             asyncio.create_task(fetch_status(session, 'http
                                                  .com', delay=3

        done, pending = await asyncio.wait(fetchers, return_

        print(f'Done task count: {len(done)}')
        print(f'Pending task count: {len(pending)}')
        for done_task in done:
            if done_task.exception() is None:
                print(done_task.result())
            else:
                logging.error("Request got an exception",
                              exc_info=done_task.exception()
```

```
        for pending_task in pending:
            pending_task.cancel()


asyncio.run(main())
```

In the preceding listing, we make one bad request and two good ones; each lasts 3 seconds. When we await our `wait` statement, we return almost immediately since our bad request errors out right away. Then we loop through the `done` tasks. In this instance, we'll have only one in the `done` set since our first request ended immediately with an exception. For this, we'll execute the branch that prints the exception.

The `pending` set will have two elements, as we have two requests that take roughly 3 seconds each to run and our first request failed almost instantly. Since we want to stop these from running, we can call the `cancel` method on them. This will give us the following output:

```
starting <function main at 0x105cfd280> with args () {}
Done task count: 1
Pending task count: 2
finished <function main at 0x105cfd280> in 0.0044 second(s)
ERROR:root:Request got an exception
```

NOTE Our application took almost no time to run, as we quickly reacted to the fact that one of our requests threw an exception; the power of using this option is we achieve fail fast behavior, quickly reacting to any issues that arise.

Processing results as they complete

Both `ALL_COMPLETED` and `FIRST_EXCEPTION` have the drawback that, in the case where coroutines are successful and don't throw an exception, we must wait for all coroutines to complete. Depending on the use case, this may be acceptable, but if we're in a situation where we want to respond to a coroutine as soon as it completes successfully, we are out of luck.

In the instance in which we want to react to a result as soon as it completes, we could use `as_completed`; however, the issue with `as_completed` is there is no easy way to see which tasks are remaining and which tasks have completed. We get them only one at a time through an iterator.

The good news is that the `return_when` parameter accepts a `FIRST_COMPLETED` option. This option will make the `wait` coroutine return as soon as it has at least one result. This can either be a coroutine that failed or one that ran successfully. We can then either cancel the other running coroutines or adjust which ones to keep running, depending on our use case. Let's use this option to make a few web requests and process whichever one finishes first.

Listing 4.13 Processing as they complete

```
import asyncio
import aiohttp
from util import async_timed
from chapter_04 import fetch_status


@async_timed()
async def main():
```

```
    async with aiohttp.ClientSession() as session:
        url = 'https:/ / www .example .com'
        fetchers = [asyncio.create_task(fetch_status(session
                    asyncio.create_task(fetch_status(session
                    asyncio.create_task(fetch_status(session

        done, pending = await asyncio.wait(fetchers, return_

        print(f'Done task count: {len(done)}')
        print(f'Pending task count: {len(pending)}')

        for done_task in done:
            print(await done_task)


asyncio.run(main())
```

In the preceding listing, we start three requests concurrently. Our `wait` coroutine will return as soon as any of these requests completes. This means that `done` will have one complete request, and `pending` will contain anything still running, giving us the following output:

```
starting <function main at 0x10222f1f0> with args () {}
Done task count: 1
Pending task count: 2
200
finished <function main at 0x10222f1f0> in 0.1138 second(s)
```

These requests can complete at nearly the same time, so we could also see output that says two or three tasks are done. Try running this listing a few times to see how the result varies.

This approach lets us respond right away when our first task completes. What if we want to process the rest of the results as they come in like `as_completed`? The above example can be adopted easily to loop on the `pending` tasks until they are empty. This will give us behavior similar to `as_completed`, with the benefit that at each step we know exactly which tasks have finished and which are still running.

Listing 4.14 Processing all results as they come in

```
import asyncio
import aiohttp
from chapter_04 import fetch_status
from util import async_timed


@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        url = 'https:/ / www .example .com'
        pending = [asyncio.create_task(fetch_status(session,
                   asyncio.create_task(fetch_status(session,
                   asyncio.create_task(fetch_status(session,

        while pending:
            done, pending = await asyncio.wait(pending, retu

            print(f'Done task count: {len(done)}')
            print(f'Pending task count: {len(pending)}')
```

```
            for done_task in done:
                print(await done_task)

asyncio.run(main())
```

In the preceding listing, we create a set named `pending` that we initialize to the coroutines we want to run. We loop while we have items in the `pending` set and call `wait` with that set on each iteration. Once we have a result from `wait`, we update the `done` and `pending` sets and then print out any `done` tasks. This will give us behavior similar to `as_completed` with the difference being we have better insight into which tasks are done and which tasks are still running. Running this, we'll see the following output:

```
starting <function main at 0x10d1671f0> with args () {}
Done task count: 1
Pending task count: 2
200
Done task count: 1
Pending task count: 1
200
Done task count: 1
Pending task count: 0
200
finished <function main at 0x10d1671f0> in 0.1153 second(s)
```

Since the request function may complete quickly, such that all requests complete at the same time, it's not impossible that we see output similar to this as well:

```
starting <function main at 0x1100f11f0> with args () {}
Done task count: 3
Pending task count: 0
200
200
200
finished <function main at 0x1100f11f0> in 0.1304 second(s)
```

Handling timeouts

In addition to allowing us finer-grained control on how we wait for coroutines to complete, `wait` also allows us to set timeouts to specify how long we want for all awaitables to complete. To enable this, we can set the `timeout` parameter with the maximum number of seconds desired. If we've exceeded this timeout, `wait` will return both the `done` and `pending` task set. There are a couple of differences in how timeouts behave in `wait` as compared to what we have seen thus far with `wait_for` and `as_completed`.

Coroutines are not canceled

When we used `wait_for`, if our coroutine timed out it would automatically request cancellation for us. This is not the case with `wait`; it behaves closer to what we saw with `gather` and `as_completed`. In the case we want to cancel coroutines due to a timeout, we must explicitly loop over the tasks and cancel them.

Timeout errors are not raised

`wait` does not rely on exceptions in the event of timeouts as do `wait_for` and `as_completed`. Instead, if the timeout occurs the `wait` returns all tasks done and all tasks that are still pending up to that point when the timeout occurred.

For example, let's examine a case where two requests complete quickly and one takes a few seconds. We'll use a timeout of 1 second with `wait` to understand what happens when we have tasks that take longer than the timeout. For the `return_when` parameter, we'll use the default value of `ALL_COMPLETED`.

Listing 4.15 Using timeouts with `wait`

```
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        url = 'https:/ / example .com'
        fetchers = [asyncio.create_task(fetch_status(session
                    asyncio.create_task(fetch_status(session
                    asyncio.create_task(fetch_status(session

        done, pending = await asyncio.wait(fetchers, timeout

        print(f'Done task count: {len(done)}')
        print(f'Pending task count: {len(pending)}')
        for done_task in done:
            result = await done_task
            print(result)


asyncio.run(main())
```

Running the preceding listing, our `wait` call will return our `done` and `pending` sets after 1 second. In the `done` set we'll see our two fast requests, as they finished within 1 second. Our slow request is still running and is, therefore, in the `pending` set. We then `await` the `done` tasks to extract out their return values. We also could have canceled the `pending` task if we so desired. Running this code, we will see the following output:

```
starting <function main at 0x11c68dd30> with args () {}
Done task count: 2
Pending task count: 1
200
200
finished <function main at 0x11c68dd30> in 1.0022 second(s)
```

Note that, as before, our tasks in the `pending` set are not canceled and will continue to run despite the timeout. If we have a use case where we want to terminate the `pending` tasks, we'll need to explicitly loop through the `pending` set and call `cancel` on each task.

Why wrap everything in a task?

At the start of this section, we mentioned that it is best practice to wrap the coroutines we pass into `wait` in tasks. Why is this? Let's go back to our previous timeout example and change it a little bit. Let's say that we have requests to two different web APIs that we'll call API A and API B. Both can be slow, but our application can run without the result from API B, so it is just a "nice to have." Since we'd like a responsive application, we set a timeout of 1 second for the requests to complete. If the request to API B is still pending after that timeout, we cancel it and move on. Let's see what happens if we implement this without wrapping the requests in tasks.

Listing 4.16 Canceling a slow request

```
import asyncio
import aiohttp
from chapter_04 import fetch_status


async def main():
    async with aiohttp.ClientSession() as session:
        api_a = fetch_status(session, 'https:/ / www .exampl
        api_b = fetch_status(session, 'https:/ / www .exampl

        done, pending = await asyncio.wait([api_a, api_b], t

        for task in pending:
            if task is api_b:
                print('API B too slow, cancelling')
                task.cancel()

asyncio.run(main())
```

We'd expect for this code to print out `API B is too slow and cancelling`, but what happens if we don't see this message at all? This can happen because when we call `wait` with just coroutines they are automatically wrapped in tasks, and the `done` and `pending` sets returned are those tasks that `wait` created for us. This means that we can't do any comparisons to see which specific task is in the `pending` set such as `if task is api_b`, since we'll be comparing a `task` object, we have no access to with a coroutine. However, if we wrap `fetch_status` in a `task`, `wait` won't create any new objects, and the comparison `if task is api_b` will work as we expect. In this case, we're correctly comparing two `task` objects.

ary

- We've learned how to use and create our own asynchronous context managers. These are special classes that allow us to asynchronously acquire resources and then release them, even if an exception occurred. These let us clean up any resources we may have acquired in a non-verbose manner and are useful when working with HTTP sessions as well as database connections. We can use them with the special `async with` syntax.
- We can use the aiohttp library to make asynchronous web requests. aiohttp is a web client and server that uses non-blocking sockets. With the web client, we can execute multiple web requests concurrently in a way that does not block the event loop.
- The `asyncio.gather` function lets us run multiple coroutines concurrently and wait for them to complete. This function will return once all awaitables we pass into it have completed. If we want to keep track of any errors that happen, we can set `return_exeptions` to `True`. This will return the results of awaitables that completed successfully alongside any exceptions we received.
- We can use the `as_completed` function to process results of a list of awaitables as soon as they complete. This will give us an iterator of futures that we can loop over. As soon as a coroutine or task has finished, we'll be able to access the result and process it.

- If we want to run multiple tasks concurrently but want to be able to understand which tasks are done and which are still running, we can use `wait`. This function also allows us greater control on when it returns results. When it returns, we get a set of tasks that have finished and set of tasks that are still running. We can then cancel any tasks we wish or do any other awaiting we need.

5 Non-blocking database drivers

This chapter covers

- Running asyncio friendly database queries with asyncpg
- Creating database connection pools running multiple SQL queries concurrently
- Managing asynchronous database transactions
- Using asynchronous generators to stream query results

Chapter 4 explored making non-blocking web requests with the aiohttp library, and it also addressed using several different asyncio API methods for running these requests concurrently. With the combination of the asyncio APIs and the aiohttp library, we can run multiple long-running web requests concurrently, leading to an improvement in our application's runtime. The concepts we learned in chapter 4 do not apply only to web requests; they also apply to running SQL queries and can improve the performance of database-intensive applications.

Much like web requests, we'll need to use an asyncio-friendly library since typical SQL libraries block the main thread, and therefore the event loop, until a result is retrieved. In this chapter, we'll learn more about asynchronous database access with the asyncpg library. We'll first create a simple schema to keep track of products for an e-commerce storefront that we'll then use to run queries against asynchronously. We'll then look at how to manage transactions and rollbacks within our database, as well as setting up connection pooling.

roducing asyncpg

As we've mentioned earlier, our existing blocking libraries won't work seamlessly with coroutines. To run queries concurrently against a database, we'll need to use an asyncio-friendly library that uses non-blocking sockets. To do this, we'll use a library called *asyncpg*, which will let us asynchronously connect to Postgres databases and run queries against them.

In this chapter we'll focus on Postgres databases, but what we learn here will also be applicable to MySQL and other databases as well. The creators of aiohttp have also created the *aiomysql* library, which can connect and run queries against a MySQL database. While there are some differences, the APIs are similar, and the knowledge is transferable. It is worth noting that the asyncpg library did not implement the Python database API specification defined in PEP-249 (available at https://www.python.org/dev/peps/pep-0249). This was a conscious choice on the part of the library implementors, since a concurrent implementation is inherently different from a synchronous one. The creators of aiomysql, however, took a different route and do implement PEP-249, so this library's API will feel familiar to those who have used synchronous database drivers in Python.

The current documentation for asynpg is available at https://magicstack.github.io/asyncpg/current/. Now that we've learned a little about the driver we'll be using, let's connect to our first database.

nnecting to a Postgres database

To get started with asyncpg, we'll use a real-world scenario of creating a product database for an e-commerce storefront. We'll use this example database throughout the chapter to demonstrate database problems in this domain that we might need to solve.

The first thing for getting started creating our product database and running queries is establishing a database connection. For this section and the rest of the chapter, we'll assume that you have a Postgres database running on your local machine on the default port of 5432, and we'll assume the default user `postgres` has a password of `'password'`.

WARNING We'll be hardcoding the password in these code examples for transparency and learning purposes; but note you should *never* hardcode a password in your code as this violates security principles. Always store passwords in environment variables or some other configuration mechanism.

You can download and install a copy of Postgres from https://www.postgresql.org/download/; just choose the appropriate operating system you're working on. You may also consider using the Docker Postgres image; more information can be found at https://hub.docker.com/_/postgres/.

Once we have our database set up, we'll install the asyncpg library. We'll use `pip3` to do this, and we'll install the latest version at the time of writing, `0.0.23`:

```
pip3 install -Iv asyncpg==0.23.0
```

Once installed, we can now import the library and establish a connection to our database. asyncpg provides this with the `asyncpg.connect` function. Let's use this to connect and print out the database version number.

Listing 5.1 Connecting to a Postgres database as the default user

```
import asyncpg
import asyncio

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='postgres',
                                       password='password')
    version = connection.get_server_version()
    print(f'Connected! Postgres version is {version}')
    await connection.close()

asyncio.run(main())
```

In the preceding listing, we create a connection to our Postres instance as the default `postgres` user and the default `postgres` database. Assuming our Postgres instance is up and running, we should see something like `"Connected! Postgres version is ServerVersion(major=12, minor=0, micro=3, releaselevel='final' serial=0)"` displayed on our console, indicating we've successfully connected to our database. Finally, we close the connection to the database with `await connection.close()`.

Now we've connected, but nothing is currently stored in our database. The next step is to create a product schema that we can interact with. In creating this schema, we'll learn how to execute basic queries with asyncpg.

fining a database schema

To begin running queries against our database, we'll need to create a database schema. We're going to select a simple schema that we'll call `products`, modeling real-world products that an online storefront might have in stock. Let's define a few different entities that we can then turn into tables in our database.

Brand

A *brand* is a manufacturer of many distinct products. For instance, Ford is a brand that produces many different models of cars (e.g., Ford F150, Ford Fiesta, etc.).

Product

A *product* is associated with one brand and there is a one-to-many relationship between brands and products. For simplicity, in our product database, a product will just have a product name. In the Ford example, a product is a compact car called the *Fiesta*; the brand is *Ford*. In addition, each product in our database will come in multiple sizes and colors. We'll define the available sizes and colors as SKUs.

SKU

SKU stands for *stock keeping unit*. A SKU represents a distinct item that a storefront has for sale. For instance, *jeans* may be a product for sale and a SKU might be *Jeans, size: medium, color: blue;* or *jeans, size: small, color: black*. There is a one-to-many relationship between a product and a SKU.

Product size

A product can come in multiple sizes. For this example, we'll consider that there are only three sizes available: small, medium, and large. Each SKU has one product size associated with it, so there is a one-to-many relationship between product sizes and SKUs.

Product color

A product can come in multiple colors. For this example, we'll say our inventory consists of only two colors: black and blue. There is a one-to-many relationship between product color and SKUs.

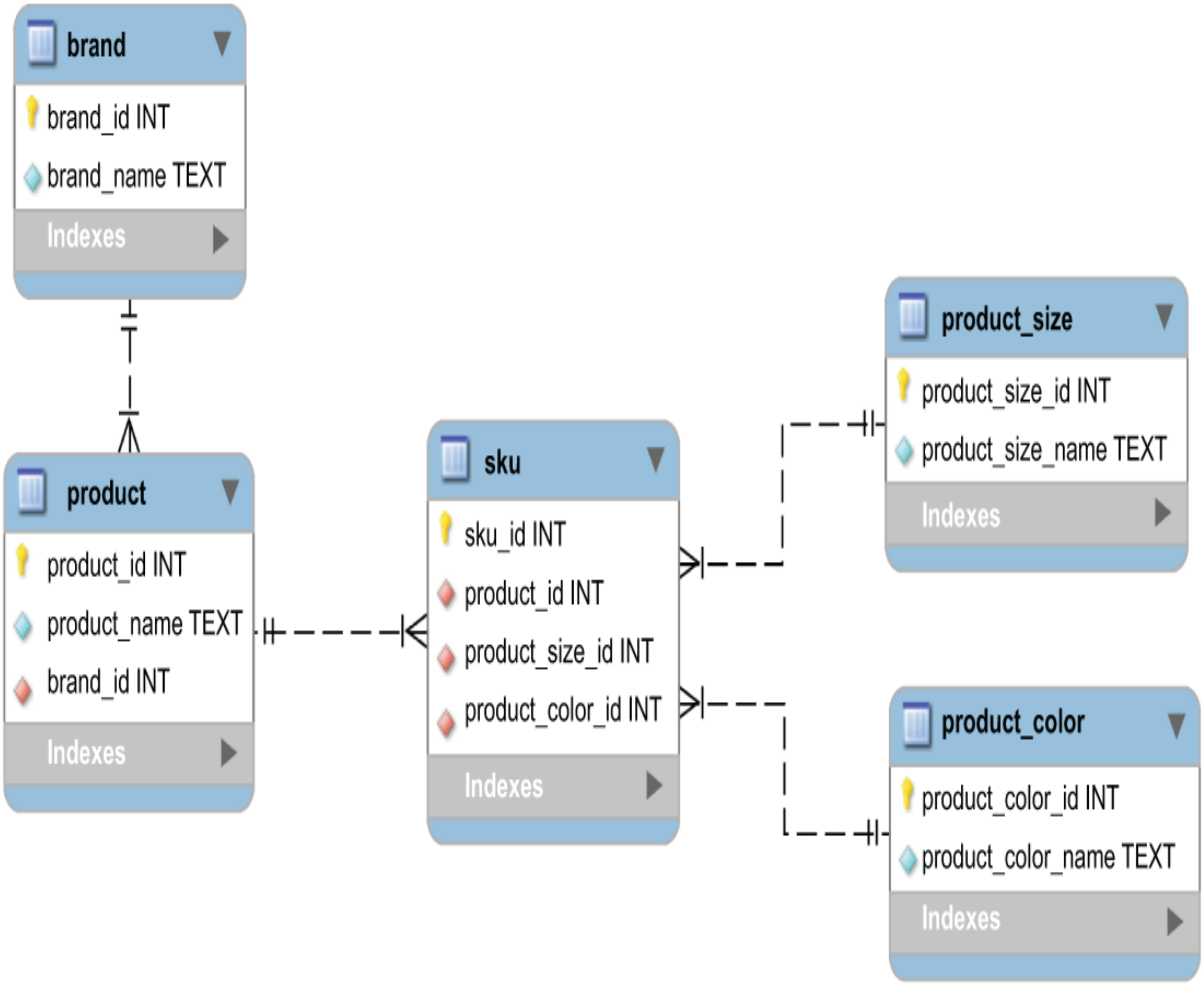

Figure 5.1 The entity diagram for the products database

Putting this all together, we'll be modeling a database schema, as shown in figure 5.1.

Now, let's define some variables with the SQL we'll need to create this schema. Using asyncpg, we'll execute these statements to create our product database. Since our sizes and colors are known ahead of time, we'll also insert a few records into the `product_size` and `product_color` tables. We'll reference these variables in the upcoming code listings, so we don't need to repeat lengthy SQL create statements.

Listing 5.2 Product schema table create statements

```
CREATE_BRAND_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS brand(
        brand_id SERIAL PRIMARY KEY,
        brand_name TEXT NOT NULL
    );"""

CREATE_PRODUCT_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS product(
        product_id SERIAL PRIMARY KEY,
        product_name TEXT NOT NULL,
        brand_id INT NOT NULL,
        FOREIGN KEY (brand_id) REFERENCES brand(brand_id)
    );"""

CREATE_PRODUCT_COLOR_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS product_color(
        product_color_id SERIAL PRIMARY KEY,
        product_color_name TEXT NOT NULL
    );"""

CREATE_PRODUCT_SIZE_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS product_size(
        product_size_id SERIAL PRIMARY KEY,
        product_size_name TEXT NOT NULL
    );"""

CREATE_SKU_TABLE = \
```

```
    """
    CREATE TABLE IF NOT EXISTS sku(
       sku_id SERIAL PRIMARY KEY,
       product_id INT NOT NULL,
       product_size_id INT NOT NULL,
       product_color_id INT NOT NULL,
       FOREIGN KEY (product_id)
       REFERENCES product(product_id),
       FOREIGN KEY (product_size_id)
       REFERENCES product_size(product_size_id),
       FOREIGN KEY (product_color_id)
       REFERENCES product_color(product_color_id)
    );"""

COLOR_INSERT = \
    """
    INSERT INTO product_color VALUES(1, 'Blue');
    INSERT INTO product_color VALUES(2, 'Black');
    """


SIZE_INSERT = \
    """
    INSERT INTO product_size VALUES(1, 'Small');
    INSERT INTO product_size VALUES(2, 'Medium');
    INSERT INTO product_size VALUES(3, 'Large');
    """
```

Now that we have the statements to create our tables and insert our sizes and colors, we need a way to run queries against them.