ecuting queries with asyncpg

To run queries against our database, we'll first need to connect to our Postgres instance and create the database directly outside of Python. We can create the database by executing the following statement once connected to the database as the default Postgres user:

```
CREATE DATABASE products;
```

You can execute this via the command line by running `sudo -u postgres psql -c "CREATE TABLE products;"`. In the next examples, we'll assume you have executed this statement, as we'll connect to the products database directly.

Now that we've created our products database, we'll connect to it and execute our `create` statements. The connection class has a coroutine called `execute` that we can use to run our create statements one by one. This coroutine returns a string representing the status of the query that Postgres returned. Let's take the statements we created in the last section and execute them.

Listing 5.3 Using an `execute` coroutine to run `create` statements

```
import asyncpg
import asyncio


async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
```

```
                                        database='products',
                                        password='password')
    statements = [CREATE_BRAND_TABLE,
                  CREATE_PRODUCT_TABLE,
                  CREATE_PRODUCT_COLOR_TABLE,
                  CREATE_PRODUCT_SIZE_TABLE,
                  CREATE_SKU_TABLE,
                  SIZE_INSERT,
                  COLOR_INSERT]

    print('Creating the product database...')
    for statement in statements:
        status = await connection.execute(statement)
        print(status)
    print('Finished creating the product database!')
    await connection.close()


asyncio.run(main())
```

We first create a connection to our products database similarly to what we did in our first example, the difference being that here we connect to the products database. Once we have this connection, we then start to execute our `CREATE TABLE` statements one at a time with `connection.execute()`. Note that `execute()` is a coroutine, so to run our SQL we need to `await` the call. Assuming everything worked properly, the status of each `execute` statement should be `CREATE TABLE`, and each `insert` statement should be `INSERT 0 1`. Finally, we close the connection to the product database. Note that in this example we await each SQL

statement in a `for` loop, which ensures that we run the `INSERT` statements synchronously. Since some tables depend on others, we can't run them concurrently.

These statements don't have any results associated with them, so let's insert a few pieces of data and run some simple select queries. We'll first insert a few brands and then query them to ensure we've inserted them properly. We can insert data with the `execute` coroutine, as before, and we can run a query with the `fetch` coroutine.

Listing 5.4 Inserting and selecting brands

```
import asyncpg
import asyncio
from asyncpg import Record
from typing import List
async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    await connection.execute("INSERT INTO brand VALUES(DEFAU
    await connection.execute("INSERT INTO brand VALUES(DEFAU

    brand_query = 'SELECT brand_id, brand_name FROM brand'
    results: List[Record] = await connection.fetch(brand_que

    for brand in results:
        print(f'id: {brand["brand_id"]}, name: {brand["brand

    await connection.close()
```

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

We first insert two brands into the `brand` table. Once we've done this, we use `connection .fetch` to get all brands from our brand table. Once this query has finished, we will have all results in memory in the `results` variable. Each result will be an `asyncpg` `Record` object. These objects act similarly to dictionaries; they allow us to access data by passing in a column name with subscript syntax. Executing this will give us the following output:

```
id: 1, name: Levis
id: 2, name: Seven
```

In this example, we fetch all data for our query into a list. If we wanted to fetch a single result, we could call `connection.fetchrow()`, which will return a single record from the query. The default asyncpg connection will pull all results from our query into memory, so for the time being there is no performance difference between `fetchrow` and `fetch`. Later in this chapter, we'll see how to use streaming result sets with cursors. These will only pull a few results into memory at a time, which is a useful technique for times when queries may return large amounts of data.

These examples run queries one after another, and we could have had similar performance by using a non-asyncio database driver. However, since we're now returning coroutines, we can use the asyncio API methods we learned in chapter 4 to execute queries concurrently.

ecuting queries concurrently with connection pools

The true benefit of asyncio for I/O-bound operations is the ability to run multiple tasks concurrently. Queries independent from one another that we need to make repeatedly are good examples of where we can apply concurrency to make our application perform better. To demonstrate this, let's pretend that we're a successful e-commerce storefront. Our company carries 100,000 SKUs for 1,000 distinct brands.

We'll also pretend that we sell our items through partners. These partners make requests for thousands of products at a given time through a batch process we have built. Running all these queries sequentially could be slow, so we'd like to create an application that executes these queries concurrently to ensure a speedy experience.

Since this is an example, and we don't have 100,000 SKUs on hand, we'll start by creating a fake product and SKU records in our database. We'll randomly generate 100,000 SKUs for random brands and products, and we'll use this data set as a basis for running our queries.

nserting random SKUs into the product database

Since we don't want to list brands, products, and SKUs ourselves, we'll randomly generate them. We'll pick random names from a list of the 1,000 most frequently occurring English words. For the sake of this example, we'll assume we have a text file that contains these words, called `common_words.txt`. You can download a copy of this file from the book's GitHub data repository at https://github.com/concurrency-in-python-with-asyncio/data.

The first thing we'll want to do is insert our brands, since our product table depends on `brand_id` as a foreign key. We'll use the `connection.executemany`

coroutine to write parameterized SQL to insert these brands. This will allow us to write one SQL query and pass in a list of parameters we want to insert instead of having to create one `INSERT` statement for each brand.

The `executemany` coroutine takes in one SQL statement and a list of tuples with values we'd like to insert. We can parameterize the SQL statement by using `$1, $2 ... $N` syntax. Each number after the dollar sign represents the index of the tuple we'd like to use in the SQL statement. For instance, if we have a query we write as `"INSERT INTO table VALUES($1, $2)"` and a list of tuples `[('a', 'b'), ('c', 'd')]` this would execute two inserts for us:

```
INSERT INTO table ('a', 'b')
INSERT INTO table ('c', 'd')
```

We'll first generate a list of 100 random brand names from our list of common words. We'll return this as a list of tuples each with one value inside of it, so we can use this in the `executemany` coroutine. Once we've created this list, it's a matter of passing a parameterized `INSERT` statement alongside this list of tuples.

Listing 5.5 Inserting random brands

```
import asyncpg
import asyncio
from typing import List, Tuple, Union
from random import sample


def load_common_words() -> List[str]:
    with open('common_words.txt') as common_words:
```

```
        return common_words.readlines()
def generate_brand_names(words: List[str]) -> List[Tuple[Uni
    return [(words[index],) for index in sample(range(100),


async def insert_brands(common_words, connection) -> int:
    brands = generate_brand_names(common_words)
    insert_brands = "INSERT INTO brand VALUES(DEFAULT, $1)"
    return await connection.executemany(insert_brands, brand


async def main():
    common_words = load_common_words()
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    await insert_brands(common_words, connection)


asyncio.run(main())
```

Internally, `executemany` will loop through our brands list and generate one `INSERT` statement per each brand. Then it will execute all those `insert` statements at once. This method of parameterization will also prevent us from SQL injection attacks, as the input data is sanitized. Once we run this, we should have 100 brands in our system with random names.

Now that we've seen how to insert random brands, let's use the same technique to insert products and SKUs. For products, we'll create a description of 10 random words and a random brand ID. For SKUs, we'll pick a random size, color, and product. We'll assume that our brand ID starts at 1 and ends at 100.

Listing 5.6 Inserting random products and SKUs

```
import asyncio
import asyncpg
from random import randint, sample
from typing import List, Tuple
from chapter_05.listing_5_5 import load_common_words


def gen_products(common_words: List[str],
                 brand_id_start: int,
                 brand_id_end: int,
                 products_to_create: int) -> List[Tuple[str,
    products = []
    for _ in range(products_to_create):
        description = [common_words[index] for index in samp
        brand_id = randint(brand_id_start, brand_id_end)
        products.append((" ".join(description), brand_id))
    return products


def gen_skus(product_id_start: int,
             product_id_end: int,
             skus_to_create: int) -> List[Tuple[int, int, in
    skus = []
    for _ in range(skus_to_create):
```

```
        product_id = randint(product_id_start, product_id_en
        size_id = randint(1, 3)
        color_id = randint(1, 2)
        skus.append((product_id, size_id, color_id))
    return skus


async def main():
    common_words = load_common_words()
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')

    product_tuples = gen_products(common_words,
                                  brand_id_start=1,
                                  brand_id_end=100,
                                  products_to_create=1000)
    await connection.executemany("INSERT INTO product VALUES
                                 product_tuples)

    sku_tuples = gen_skus(product_id_start=1,
                          product_id_end=1000,
                          skus_to_create=100000)
    await connection.executemany("INSERT INTO sku VALUES(DEF
                                 sku_tuples)

    await connection.close()
```

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

When we run this listing, we should have a database with 1,000 products and 100,000 SKUs. Depending on your machine, this may take several seconds to run. With a few joins, we can now query all available SKUs for a particular product. Let's see what this query would look like for `product id 100`:

```
product_query = \
"""
SELECT
p.product_id,
p.product_name,
p.brand_id,
s.sku_id,
pc.product_color_name,
ps.product_size_name
FROM product as p
JOIN sku as s on s.product_id = p.product_id
JOIN product_color as pc on pc.product_color_id = s.product_
JOIN product_size as ps on ps.product_size_id = s.product_si
WHERE p.product_id = 100"""
```

When we execute this query, we'll get one row for each SKU for a product, and we'll also get the proper English name for size and color instead of an ID. Assuming we have a lot of product IDs we'd like to query at a given time, this provides us a good opportunity to apply concurrency. We may naively try to apply `asyncio.gather` with our existing connection like so:

```
async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    print('Creating the product database...')
    queries = [connection.execute(product_query),
               connection.execute(product_query)]
    results = await asyncio.gather(*queries)
```

However, if we run this we'll be greeted with an error:

```
RuntimeError: readexactly() called while another coroutine i
```

Why is this? In the SQL world, one connection means one socket connection to our database. Since we have only one connection and we're trying to read the results of multiple queries concurrently, we experience an error. We can resolve this by creating multiple connections to our database and executing one query per connection. Since creating connections is resource expensive, caching them so we can access them when needed makes sense. This is commonly known as a *connection pool*.

Creating a connection pool to run queries concurrently

Since we can only run one query per connection at a time, we need a mechanism for creating and managing multiple connections. A connection pool does just that. You can think of a connection pool as a cache of existing connections to a database

instance. They contain a finite number of connections that we can access when we need to run a query.

Using connection pools, we *acquire* connections when we need to run a query. Acquiring a connection means we ask the pool, “Does the pool currently have any connections available? If so, give me one so I can run my queries.” Connection pools facilitate the reuse of these connections to execute queries. In other words, once a connection is acquired from the pool to run a query and that query finishes, we return or “release” it to the pool for others to use. This is important because establishing a connection with a database is time-expensive. If we had to create a new connection for every query we wanted to run, our application’s performance would quickly degrade.

Since the connection pool has a finite number of connections, we could be waiting for some time for one to become available, as other connections may be in use. This means connection acquisition is an operation that may take time to complete. If we only have 10 connections in the pool, each of which is in use, and we ask for another, we’ll need to wait until 1 of the 10 connections becomes available for our query to execute.

To illustrate how this works in terms of asyncio, let’s imagine we have a connection pool with two connections. Let’s also imagine we have three coroutines and each runs a query. We’ll run these three coroutines concurrently as tasks. With a connection pool set up this way, the first two coroutines that attempt to run their queries will acquire the two available connections and start running their queries. While this is happening, the third coroutine will be waiting for a connection to become available. When either one of the first two coroutines finishes running its query, it will release

its connection and return it to the pool. This lets the third coroutine acquire it and start using it to run its query (figure 5.2).

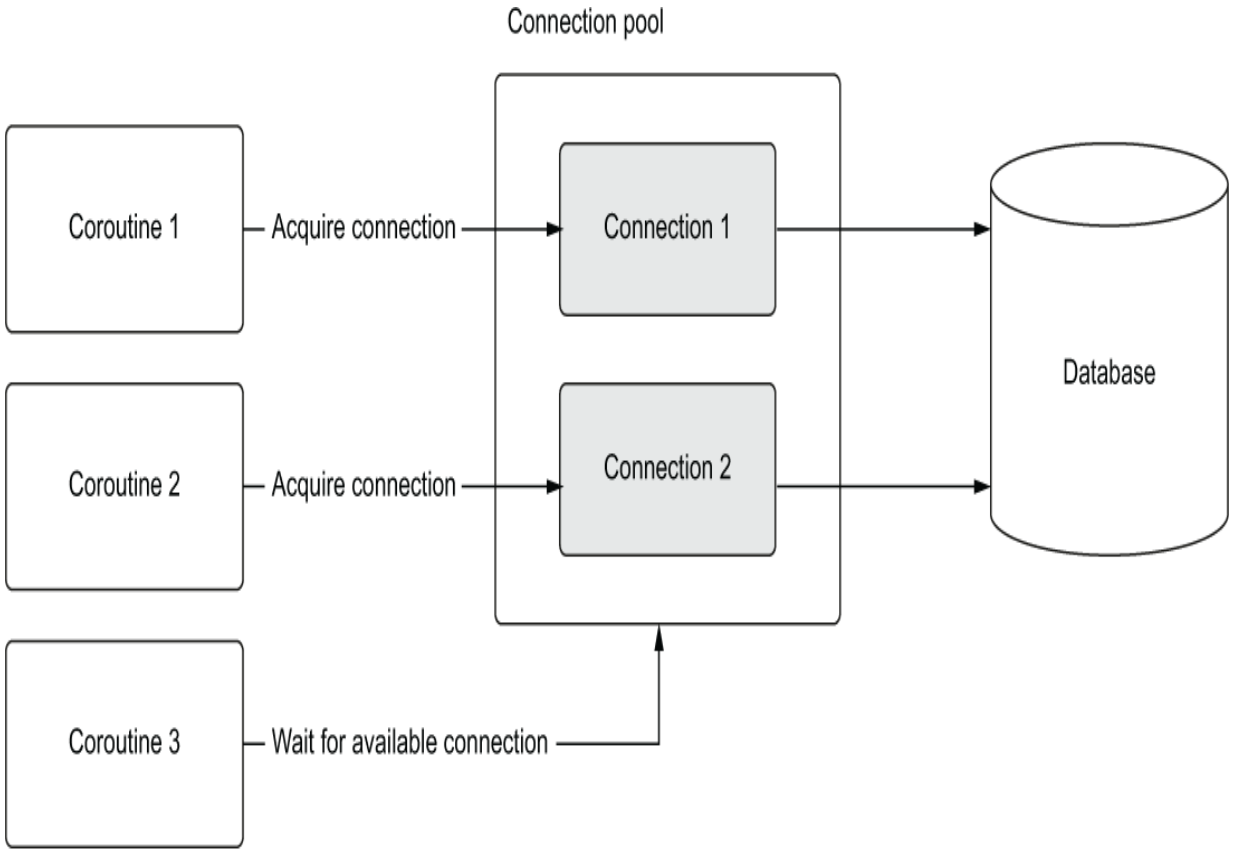

Figure 5.2 Coroutines 1 and 2 acquire connections to run their queries while Coroutine 3 waits for a connection. Once either Coroutine 1 or 2 finishes, Coroutine 3 will be able to use the newly released connection and will be able to execute its query.

In this model, we can have at most two queries running concurrently. Normally, the connection pool will be a bit bigger to enable more concurrency. For our examples, we'll use a connection pool of six, but the actual number you want to use is dependent on the hardware your database and your application run on. In this case, you'll need to benchmark which connection pool size works best. Keep in mind, bigger is not always better, but that's a much larger topic.

Now that we understand how connection pools work, how do we create one with asyncpg? asyncpg exposes a coroutine named `create_pool` to accomplish this. We use this instead of the `connect` function we used earlier to establish a connection to our database. When we call `create_pool`, we'll specify the number of connections we wish to create in the pool. We'll do this with the `min_size` and `max_size` parameters. The `min_size` parameter specifies the minimum number of connections in our connection pool. This means that once we set up our pool, we are guaranteed to have this number of connections inside of it already established. The `max_size` parameter specifies the maximum number of connections we want in our pool, determining the maximum number of connections we can have. If we don't have enough connections available, the pool will create a new one for us if the new connection won't cause the pool size to be above the value set in `max_size`. For our first example, we'll set both these values to six. This guarantees we always have six connections available.

asyncpg pools are *asynchronous context managers*, meaning that we must use `async with` syntax to create a pool. Once we've established a pool, we can acquire connections using the `acquire` coroutine. This coroutine will suspend execution until we have a connection available. Once we do, we can then use that connection to execute whatever SQL query we'd like. Acquiring a connection is also an async context manager that returns the connection to the pool when we are done with it, so we'll need to use `async with` syntax just like we did when we created the pool. Using this, we can rewrite our code to run several queries concurrently.

Listing 5.7 Establishing a connection pool and running queries concurrently

```
import asyncio
import asyncpg
```

```
product_query = \
    """
SELECT
p.product_id,
p.product_name,
p.brand_id,
s.sku_id,
pc.product_color_name,
ps.product_size_name
FROM product as p
JOIN sku as s on s.product_id = p.product_id
JOIN product_color as pc on pc.product_color_id = s.product_
JOIN product_size as ps on ps.product_size_id = s.product_si
WHERE p.product_id = 100"""


async def query_product(pool):
    async with pool.acquire() as connection:
        return await connection.fetchrow(product_query)


async def main():
    async with asyncpg.create_pool(host='127.0.0.1',
                                   port=5432,
                                   user='postgres',
                                   password='password',
                                   database='products',
                                   min_size=6,
                                   max_size=6) as pool:

        await asyncio.gather(query_product(pool),
```

```
                                query_product(pool))          ❷


asyncio.run(main())
```

❶ Create a connection pool with six connections.

❷ Execute two product queries concurrently.

In the preceding listing, we first create a connection pool with six connections. We then create two query coroutine objects and schedule them to run concurrently with `asyncio.gather`. In our `query_product` coroutine, we first acquire a connection from the pool with `pool.acquire()`. This coroutine will then suspend running until a connection is available from the connection pool. We do this in an `async with` block; this will ensure that once we leave the block, the connection will be returned to the pool. This is important because if we don't do this we can run out of connections, and we'll end up with an application that hangs forever, waiting for a connection that will never become available. Once we've acquired a connection, we can then run our query as we did in previous examples.

We can expand this example to run 10,000 queries by creating 10,000 different query coroutine objects. To make this interesting, we'll write a version that runs the queries synchronously and compare how long things take.

Listing 5.8 Synchronous queries vs. concurrent

```
import asyncio
import asyncpg
from util import async_timed
```

```
product_query = \
    """
SELECT
p.product_id,
p.product_name,
p.brand_id,
s.sku_id,
pc.product_color_name,
ps.product_size_name
FROM product as p
JOIN sku as s on s.product_id = p.product_id
JOIN product_color as pc on pc.product_color_id = s.product_
JOIN product_size as ps on ps.product_size_id = s.product_si
WHERE p.product_id = 100"""


async def query_product(pool):
    async with pool.acquire() as connection:
        return await connection.fetchrow(product_query)

@async_timed()
async def query_products_synchronously(pool, queries):
    return [await query_product(pool) for _ in range(queries


@async_timed()
async def query_products_concurrently(pool, queries):
    queries = [query_product(pool) for _ in range(queries)]
    return await asyncio.gather(*queries)
```

```
async def main():
    async with asyncpg.create_pool(host='127.0.0.1',
                                   port=5432,
                                   user='postgres',
                                   password='password',
                                   database='products',
                                   min_size=6,
                                   max_size=6) as pool:
        await query_products_synchronously(pool, 10000)
        await query_products_concurrently(pool, 10000)

asyncio.run(main())
```

In `query_products_synchronously`, we put an `await` in a list comprehension, which will force each call to `query_product` to run sequentially. Then, in `query_products_ concurrently` we create a list of coroutines we want to run and then run them concurrently with `gather`. In the main coroutine, we then run our synchronous and concurrent version with 10,000 queries each. While the exact results can vary substantially based on your hardware, the concurrent version is nearly five times as fast as the serial version:

```
starting <function query_products_synchronously at 0x1219ea1
finished <function query_products_synchronously at 0x1219ea1
starting <function query_products_concurrently at 0x1219ea31
finished <function query_products_concurrently at 0x1219ea31
```

An improvement like this is significant, but there are still more improvements we can make if we need more throughput. Since our query is relatively fast, this code is a

mixture of CPU-bound in addition to I/O-bound. In chapter 6, we'll see how to squeeze even more performance out of this setup.

So far, we've seen how to insert data into our database assuming we don't have any failures. But what happens if we are in the middle of inserting products, and we get a failure? We don't want an inconsistent state in our database, so this is where database transactions come into play. Next, we'll see how to use asynchronous context managers to acquire and manage transactions.

ınaging transactions with asyncpg

Transactions are a core concept in many databases that satisfy the ACID (atomic, consistent, isolated, durable) properties. A *transaction* consists of one or more SQL statements that are executed as one atomic unit. If no errors occur when we execute the statements within a transaction, we *commit* the statements to the database, making any changes a permanent part of the database. If there are any errors, we *roll back* the statements, and it is as if none of them ever happened. In the context of our product database, we may need to roll back a set of updates if we attempt to insert a duplicate brand, or if we have violated a database constraint we've set.

In asyncpg, the easiest way to deal with transactions is to use the `connection.transaction` asynchronous context manager to start them. Then, if there is an exception in the `async with` block, the transaction will automatically be rolled back. If everything executes successfully, it will be automatically committed. Let's look at how to create a transaction and execute two simple `insert` statements to add a couple of brands.

Listing 5.9 Creating a transaction

```
import asyncio
import asyncpg

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    async with connection.transaction():
        await connection.execute("INSERT INTO brand "
                                 "VALUES(DEFAULT, 'brand_1')
        await connection.execute("INSERT INTO brand "
                                 "VALUES(DEFAULT, 'brand_2')

    query = """SELECT brand_name FROM brand
                WHERE brand_name LIKE 'brand%'"""
    brands = await connection.fetch(query)
    print(brands)

    await connection.close()

asyncio.run(main())
```

❶ Start a database transaction.

❷ Select brands to ensure that our transaction was committed.

Assuming our transaction committed successfully, we should see `[<Record brand_ name='brand_1'>, <Record brand_name='brand_2'>]` printed

out to the console. This example assumes zero errors running the two `insert` statements, and everything was committed successfully. To demonstrate what happens when a rollback occurs, let's force a SQL error. To test this out, we'll try and insert two brands with the same primary `key` `id`. Our first insert will work successfully, but our second insert will raise a duplicate key error.

Listing 5.10 Handling an error in a transaction

```
import asyncio
import logging
import asyncpg

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    try:
        async with connection.transaction():
            insert_brand = "INSERT INTO brand VALUES(9999, '
            await connection.execute(insert_brand)
            await connection.execute(insert_brand)
    except Exception:
        logging.exception('Error while running transaction')
    finally:
        query = """SELECT brand_name FROM brand
                    WHERE brand_name LIKE 'big_%'"""
        brands = await connection.fetch(query)
        print(f'Query result was: {brands}')
```

```
        await connection.close()

asyncio.run(main())
```

❶ This insert statement will error because of a duplicate primary key.

❷ If we had an exception, log the error.

❸ Select the brands to ensure we didn't insert anything.

In the following code, our second insert statement throws an error. This leads to the following output:

```
ERROR:root:Error while running transaction
Traceback (most recent call last):
  File "listing_5_10.py", line 16, in main
    await connection.execute("INSERT INTO brand "
  File "asyncpg/connection.py", line 272, in execute
    return await self._protocol.query(query, timeout)
  File "asyncpg/protocol/protocol.pyx", line 316, in query
asyncpg.exceptions.UniqueViolationError: duplicate key value
DETAIL:  Key (brand_id)=(9999) already exists.
Query result was: []
```

We first retrieved an exception because we attempted to insert a duplicate key and then see that the result of our `select` statement was empty, indicating that we successfully rolled back the transaction.

Nested transactions

asyncpg also supports the concept of a *nested transaction* through a Postgres feature called *savepoints*. Savepoints are defined in Postgres with the `SAVEPOINT` command. When we define a savepoint, we can roll back to that savepoint and any queries executed after the savepoint will roll back, but any queries successfully executed before it will not roll back.

In asyncpg, we can create a savepoint by calling the `connection.transaction` context manager within an existing transaction. Then, if there is any error within this inner transaction it is rolled back, but the outer transaction is not affected. Let's try this out by inserting a brand in a transaction and then within a nested transaction attempting to insert a color that already exists in our database.

Listing 5.11 A nested transaction

```
import asyncio
import asyncpg
import logging


async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    async with connection.transaction():
        await connection.execute("INSERT INTO brand VALUES(D
```

```
        try:
            async with connection.transaction():
                await connection.execute("INSERT INTO produc
        except Exception as ex:
            logging.warning('Ignoring error inserting produc

    await connection.close()


asyncio.run(main())
```

When we run this code, our first `INSERT` statement runs successfully, since we don't have this brand in our database yet. Our second `INSERT` statement fails with a duplicate key error. Since this second insert statement is within a transaction and we catch and log the exception, despite the error our outer transaction is not rolled back, and the brand is properly inserted. If we did not have the nested transaction, the second insert statement would have also rolled back our brand insert.

Manually managing transactions

So far, we have used asynchronous context managers to handle committing and rolling back our transactions. Since this is less verbose than managing things ourselves, it is usually the best approach. That said, we may find ourselves in situations where we need to manually manage a transaction. For example, we may want to have custom code execute on rollback, or we may want to roll back on a condition other than an exception.

To manually manage a transaction, we can use the transaction manager returned by `connection.transaction` outside of a context manager. When we do this, we'll

manually need to call its `start` method to start a transaction and then `commit` on success and `rollback` on failure. Let's look at how to do this by rewriting our first example.

Listing 5.12 Manually managing a transaction

```
import asyncio
import asyncpg
from asyncpg.transaction import Transaction


async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    transaction: Transaction = connection.transaction()
    await transaction.start()
    try:
        await connection.execute("INSERT INTO brand "
                                 "VALUES(DEFAULT, 'brand_1')
        await connection.execute("INSERT INTO brand "
                                 "VALUES(DEFAULT, 'brand_2')
    except asyncpg.PostgresError:
        print('Errors, rolling back transaction!')
        await transaction.rollback()
    else:
        print('No errors, committing transaction!')
        await transaction.commit()
```

```
        query = """SELECT brand_name FROM brand
                    WHERE brand_name LIKE 'brand%'"""
        brands = await connection.fetch(query)
        print(brands)

        await connection.close()

    asyncio.run(main())
```

❶ Create a transaction instance.

❷ Start the transaction.

❸ If there was an exception, roll back.

❹ If there was no exception, commit.

We first start by creating a transaction with the same method call we used with async context manager syntax, but instead, we store the `Transaction` instance that this call returns. Think of this class as a manager for our transaction, since with this we'll be able to perform any commits and rollbacks we need. Once we have a transaction instance, we can then call the `start` coroutine. This will execute a query to start the transaction in Postgres. Then, within a `try` block we can execute any queries we'd like. In this case we insert two brands. If there were errors with any of those INSERT statements, we'll experience the `except` block and roll back the transaction by calling the `rollback` coroutine. If there were no errors, we call the `commit` coroutine, which will end the transaction and make any changes in our transaction permanent in the database.

Up until now we have been running our queries in a way that pulls all query results into memory at once. This makes sense for many applications, since many queries will return small result sets. However, we may have a situation where we are dealing with a large result set that may not fit in memory all at once. In these cases, we may want to stream results to avoid taxing our system's random access memory (RAM). Next, we'll explore how to do this with asyncpg and, along the way, introduce asynchronous generators.

ynchronous generators and streaming result sets

One drawback of the default `fetch` implementation asynpg provides is that it pulls all data from any query we execute into memory. This means that if we have a query that returns millions of rows, we'd attempt to transfer that entire set from the database to the requesting machine. Going back to our product database example, imagine we're even more successful and have billions of products available. It is highly likely that we'll have some queries that will return very large result sets, potentially hurting performance.

Of course, we could apply `LIMIT` statements to our query and paginate things, and this makes sense for many, if not most, applications. That said, there is overhead with this approach in that we are sending the same query multiple times, potentially creating extra stress on the database. If we find ourselves hampered by these issues, it can make sense to stream results for a particular query only as we need them. This will save on memory consumption at our application layer as well as save load on the database. However, it does come at the expense of making more round trips over the network to the database.

Postgres supports streaming query results through the concept of *cursors*. Consider a cursor as a pointer to where we currently are in iterating through a result set. When we get a single result from a streamed query, we advance the cursor to the next element, and so on, until we have no more results.

Using asyncpg, we can get a cursor directly from a connection which we can then use to execute a streaming query. Cursors in asyncpg use an asyncio feature we have not used yet called *asynchronous generators*. Asynchronous generators generate results asynchronously one by one, similarly to regular Python generators. They also allow us to use a special `for` loop style syntax to iterate over any results we get. To fully understand how this works, we'll first introduce asynchronous generators as well as `async for` syntax to loop these generators.

ntroducing asynchronous generators

Many developers will be familiar with generators from the synchronous Python world. Generators are an implementation of the iterator design pattern made famous in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by the "gang of four" (Addison-Wesley Professional, 1994). This pattern allows us to define sequences of data "lazily" and iterate through them one element at a time. This is useful for potentially large sequences of data, where we don't need to store everything in memory all at once.

A simple synchronous generator is a normal Python function which contains a `yield` statement instead of a `return` statement. For example, let's see how creating and using a generator returns positive integers, starting from zero until a specified end.

Listing 5.13 A synchronous generator