```
def positive_integers(until: int):
    for integer in range(until):
        yield integer


positive_iterator = positive_integers(2)

print(next(positive_iterator))
print(next(positive_iterator))
```

In the preceding listing, we create a function which takes an integer that we want to count to. We then start a loop until our specified end integer. Then, at each iteration of the loop, we `yield` the next integer in the sequence. When we call `positive_integers(2)`, we don't return an entire list or even run the loop in our method. In fact, if we check the type of `positive_iterator`, we'll get `<class 'generator'>.`

We then use the `next` utility function to iterate over our generator. Each time we call `next`, this will trigger one iteration of the `for` loop in `positive_integers`, giving us the result of the `yield` statement per each iteration. Thus, the code in listing 5.13 will print `0` and `1` to the console. Instead of using `next,` we could have used a `for` loop with our generator to loop through all values in our generator.

This works for synchronous methods, but what if we wanted to use coroutines to generate a sequence of values asynchronously? Using our database example, what if we wanted to generate a sequence of rows that we "lazily" get from our database? We can do this with Python's asynchronous generators and special `async for` syntax. To demonstrate a simple asynchronous generator, let's start with our positive integer

example but introduce a call to a coroutine that takes a few seconds to complete. We'll use the `delay` function from chapter 2 for this.

Listing 5.14 A simple asynchronous generator

```
import asyncio
from util import delay, async_timed


async def positive_integers_async(until: int):
    for integer in range(1, until):
        await delay(integer)
        yield integer


@async_timed()
async def main():
    async_generator = positive_integers_async(3)
    print(type(async_generator))
    async for number in async_generator:
        print(f'Got number {number}')


asyncio.run(main())
```

Running the preceding listing, we'll see the type is no longer a plain generator but `<class 'async_generator'>`, an asynchronous generator. An asynchronous generator differs from a regular generator in that, instead of generating plain Python objects as elements, it generates coroutines that we can then await until we get a result. Because of this, our normal for loops and `next` functions won't work with

these types of generators. Instead, we have a special syntax, `async for`, to deal with these types of generators. In this example, we use this syntax to iterate over `positive_integers_async`.

This code will print the numbers 1 and 2, waiting 1 second before returning the first number and 2 seconds before returning the second. Note that this is not running the coroutines generated concurrently; instead, it is generating and awaiting them one at a time in a series.

Using asynchronous generators with a streaming cursor

The concept of asynchronous generators pairs nicely with the concept of a `streaming database cursor`. Using these generators, we'll be able to fetch one row at a time with a simple `for` loop-like syntax. To perform streaming with asyncpg, we'll first need to start a transaction, as Postgres requires this to use cursors. Once we've started a transaction, we can then call the `cursor` method on the `Connection` class to obtain a cursor. When we call the `cursor` method, we'll pass in the query we'd like to stream. This method will return an asynchronous generator that we can use to stream results one at a time.

To get familiar with how to do this, let's run a query to get all products from our database with a cursor. We'll then use `async for` syntax to fetch elements one at a time from our result set.

Listing 5.15 Streaming results one by one

```
import asyncpg
import asyncio
import asyncpg
```

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

    query = 'SELECT product_id, product_name FROM product'
    async with connection.transaction():
        async for product in connection.cursor(query):
            print(product)

    await connection.close()


asyncio.run(main())
```

The preceding listing will print all our products out one at a time. Despite having put 1,000 products in this table, we'll only pull a few into memory at a time. At the time of writing, to cut down on network traffic the cursor defaults to prefetching 50 records at a time. We can change this behavior by setting the `prefetch` parameter with however many elements we'd like to prefetch.

We can also use these cursors to skip around our result set and fetch an arbitrary number of rows at a time. Let's see how to do this by getting a few records from the middle of the query we just used.

Listing 5.16 Moving the cursor and fetching records

```
import asyncpg
import asyncio


async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    async with connection.transaction():
        query = 'SELECT product_id, product_name from produc
        cursor = await connection.cursor(query)
        await cursor.forward(500)
        products = await cursor.fetch(100)
        for product in products:
            print(product)

    await connection.close()


asyncio.run(main())
```

❶ Create a cursor for the query.

❷ Move the cursor forward 500 records.

❸ Get the next 100 records.

The code in the preceding listing will first create a cursor for our query. Note that we use this in an `await` statement like a coroutine instead of an asynchronous generator; this is because in asyncpg a cursor is both an asynchronous generator *and* an awaitable. For the most part, this is similar to using an async generator, but there is a difference in prefetch behavior when creating a cursor this way. Using this method, we cannot set a prefetch value. Doing so would raise an `InterfaceError`.

Once we have the cursor, we use its `forward` coroutine method to move forward in the result set. This will effectively skip the first 500 records in our product table. Once we've moved our cursor forward, we then fetch the next 100 products and print them each out to the console.

These types of cursors are non-scrollable by default, meaning we can only advance forward in the result set. If you want to use scrollable cursors that can move both forwards and backwards, you'll need to execute the SQL to do so manually using `DECLARE ... SCROLL CURSOR` (you can read more on how to do this in the Postgres documentation at https://www.postgresql.org/docs/current/plpgsql-cursors.html).

Both techniques are useful if we have a really large result set and don't want to have the entire set residing in memory. The `async for` loops we saw in listing 5.16 are useful for looping over the entire set, while creating a cursor and using the `fetch` coroutine method is useful for fetching a chunk of records or skipping a set of records.

However, what if we only want to retrieve a fixed set of elements at a time with prefetching and still use an `async for` loop? We could add a counter in our `async for` loop and break out after we've seen a certain number of elements, but that isn't particularly reusable if we need to do this often in our code. What we can do

to make this easier is build our own async generator. We'll call this generator `take`. This generator will take an async generator and the number of elements we wish to extract. Let's investigate creating this and grabbing the first five elements from a result set.

Listing 5.17 Getting a specific number of elements with an asynchronous generator

```
import asyncpg
import asyncio


async def take(generator, to_take: int):
    item_count = 0
    async for item in generator:
        if item_count > to_take - 1:
            return
        item_count = item_count + 1
        yield item


async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    async with connection.transaction():
        query = 'SELECT product_id, product_name from produc
        product_generator = connection.cursor(query)
        async for product in take(product_generator, 5):
            print(product)
```

```
        print('Got the first five products!')

    await connection.close()


asyncio.run(main())
```

Our `take` async generator keeps track of how many items we've seen so far with `item_count`. We then enter an `async_for` loop and `yield` each record that we see. Once we `yield`, we check `item_count` to see if we have yielded the number of items the caller requested. If we have, we `return`, which ends the async generator. In our main coroutine, we can then use `take` within a normal async for loop. In this example, we use it to ask for the first five elements from the cursor, giving us the following output:

```
<Record product_id=1 product_name='among paper foot see shoe
<Record product_id=2 product_name='major wait half speech la
<Record product_id=3 product_name='war area speak listen hor
<Record product_id=4 product_name='smell proper force road h
<Record product_id=5 product_name='ship many dog fine surfac
Got the first five products!
```

While we've defined this in code ourselves, an open source library, *aiostream*, has this functionality and more for processing asynchronous generators. You can view the documentation for this library at aiostream.readthedocs.io.

ary

In this chapter, we've learned the basics around creating and selecting records in Postgres using an asynchronous database connection. You should now be able to take this knowledge and create concurrent database clients.

- We've learned how to use asyncpg to connect to a Postgres database.
- We've learned how to use various asyncpg coroutines to create tables, insert records, and execute single queries.
- We've learned how to create a connection pool with asyncpg. This allows us to run multiple queries concurrently with asyncio's API methods such as `gather`. Using this we can potentially speed up our applications by running our queries in tandem.
- We've learned how to manage transactions with asyncpg. Transactions allow us to roll back any changes we make to a database as the result of a failure, keeping our database in a consistent state even when something unexpected happens.
- We've learned how to create asynchronous generators and how to use them for streaming database connections. We can use these two concepts together to work with large data sets that can't fit in memory all at once.

6 Handling CPU-bound work

This chapter covers

- The multiprocessing library
- Creating process pools to handle CPU-bound work
- Using `async` and `await` to manage CPU-bound work
- Using MapReduce to solve a CPU-intensive problem with asyncio
- Handling shared data between multiple processes with locks
- Improving the performance of work with both CPU- and I/O-bound operations

Until now, we've been focused on performance gains we can get with asyncio when running I/O-bound work concurrently. Running I/O-bound work is asyncio's bread and butter, and with the way we've written code so far, we need to be careful not to run any CPU-bound code in our coroutines. This seems like it severely limits asyncio, but the library is more versatile than just handling I/O-bound work.

asyncio has an API for interoperating with Python's multiprocessing library. This lets us use async `await` syntax as well as asyncio APIs with multiple processes. Using this, we can get the benefits of the asyncio library even when using CPU-bound code. This allows us to achieve performance gains for CPU-intensive work, such as mathematical computations or data processing, letting us sidestep the global interpreter lock and take full advantage of a multicore machine.

In this chapter, we'll first learn about the multiprocessing module to become familiar with the concept of executing multiple processes. We'll then learn about *process pool executors* and how we can hook them into asyncio. We'll then take this knowledge and use it to solve a CPU-intensive problem with MapReduce. We'll also learn about

managing shared states amongst multiple processes, and we'll introduce the concept of locking to avoid concurrency bugs. Finally, we'll look at how to use multiprocessing to improve the performance of an application that is both I/O- and CPU-bound as we saw in chapter 5.

roducing the multiprocessing library

In chapter 1, we introduced the global interpreter lock. The global interpreter lock prevents more than one piece of Python bytecode from running in parallel. This means that for anything other than I/O-bound tasks, excluding some small exceptions, using multithreading won't provide any performance benefits, the way it would in languages such as Java and C++. It seems like we might be stuck with no solution for our parallelizable CPU-bound work in Python, but this is where the multiprocessing library provides a solution.

Instead of our parent process spawning threads to parallelize things, we instead spawn subprocesses to handle our work. Each subprocess will have its own Python interpreter and be subject to the GIL, but instead of one interpreter we'll have several, each with its own GIL. Assuming we run on a machine with multiple CPU cores, this means that we can parallelize any CPU-bound workload effectively. Even if we have more processes than cores, our operating system will use preemptive multitasking to allow our multiple tasks to run concurrently. This setup is both concurrent *and* parallel.

To get started with the multiprocessing library, let's start by running a couple of functions in parallel. We'll use a very simple CPU-bound function that counts from zero to a large number to examine how the API works as well as the performance benefits.

Listing 6.1 Two parallel processes with multiprocessing

```
import time
from multiprocessing import Process


def count(count_to: int) -> int:
    start = time.time()
    counter = 0
    while counter < count_to:
        counter = counter + 1
    end = time.time()
    print(f'Finished counting to {count_to} in {end-start}')
    return counter


if __name__ == "__main__":
    start_time = time.time()

    to_one_hundred_million = Process(target=count, args=(100
    to_two_hundred_million = Process(target=count, args=(200

    to_one_hundred_million.start()
    to_two_hundred_million.start()

    to_one_hundred_million.join()
    to_two_hundred_million.join()

    end_time = time.time()
    print(f'Completed in {end_time-start_time}')
```

❶ Create a process to run the countdown function.

❷ Start the process. This method returns instantly.

❸ Wait for the process to finish. This method blocks until the process is done.

In the preceding listing, we create a simple count function which takes an integer and loops one by one until we count to the integer we pass in. We then create two processes, one to count to 100,000,000 and one to count to 200,000,000. The `Process` class takes in two arguments, a `target` which is the function name we wish to run in the process and `args` representing a tuple of arguments we wish to pass to the function. We then call the `start` method on each process. This method returns instantly and will start running the process. In this example we start both processes one after another. We then call the `join` method on each process. This will cause our main process to block until each process has finished. Without this, our program would exit almost instantly and terminate the subprocesses, as nothing would be waiting for their completion. Listing 6.1 runs both count functions concurrently; assuming we're running on a machine with at least two CPU cores, we should see a speedup. When this code runs on a 2.5 GHz 8-core machine, we achieve the following results:

```
Finished counting down from 100000000 in 5.3844
Finished counting down from 200000000 in 10.6265
Completed in 10.8586
```

In total, our countdown functions took a bit over 16 seconds, but our application finished in just under 11 seconds. This gives us a time savings over running sequentially of about 5 seconds. Of course, the results you see when you run this will

be highly variable depending on your machine, but you should see something directionally equivalent to this.

Notice the addition of `if __name__ == "__main__":` to our application where we haven't before. This is a quirk of the multiprocessing library; if you don't add this you may receive the following error: `An attempt has been made to start a new process before the current process has finished its bootstrapping phase`. The reason this happens is to prevent others who may import your code from accidentally launching multiple processes.

This gives us a decent performance gain; however, it is awkward because we must call `start` and `join` for each process we start. We also don't know which process will complete first; if we want to do something like `asyncio.as_completed` and process results as they finish, we're out of luck. The `join` method also does not return the value our target function returns; in fact, currently there is no way to get the value our function returns without using shared inter-process memory!

This API will work for simple cases, but it clearly won't work if we have functions where we want to get the return value out or want to process results as soon as they come in. Luckily, process pools provide a way for us to deal with this.

ing process pools

In the previous example, we manually created processes and called their `start` and `join` methods to run and wait for them. We identified several issues with this approach, from code quality to not having the ability to access the results our process returned. The multiprocessing module has an API that lets us deal with this issue, called *process pools*.

Process pools are a concept similar to the connection pools that we saw in chapter 5. The difference in this case is that, instead of a collection of connections to a database, we create a collection of Python processes that we can use to run functions in parallel. When we have a CPU-bound function we wish to run in a process, we ask the pool directly to run it for us. Behind the scenes, this will execute this function in an available process, running it and returning the return value of that function. To see how a process pool works, let's create a simple one and run a few "hello world"-style functions with it.

Listing 6.2 Creating a process pool

```
from multiprocessing import Pool


def say_hello(name: str) -> str:
    return f'Hi there, {name}'


if __name__ == "__main__":
    with Pool() as process_pool:
        hi_jeff = process_pool.apply(say_hello, args=('Jeff'
        hi_john = process_pool.apply(say_hello, args=('John'
        print(hi_jeff)
        print(hi_john)
```

1 Create a new process pool.

2 Run say_hello with the argument 'Jeff ' in a separate process and get the result.

In the preceding listing, we create a process pool using `with Pool() as process_pool`. This is a context manager because once we are done with the pool, we need to appropriately shut down the Python processes we created. If we don't do this, we run the risk of leaking processes, which can cause resource-utilization issues. When we instantiate this pool, it will automatically create Python processes equal to the number of CPU cores on the machine you're running on. You can determine the number of CPU cores you have in Python by running the `multiprocessing.cpu_count()` function. You can set the `processes` argument to any integer you'd like when you call `Pool()`. The default value is usually a good starting point.

Next, we use the `apply` method of the process pool to run our `say_hello` function in a separate process. This method looks similar to what we did previously with the `Process` class, where we passed in a target function and a tuple of arguments. The difference here is that we don't need to start the process or call `join` on it ourselves. We also get back the return value of our function, which we couldn't do in the previous example. Running this code, you should see the following printed out:

```
Hi there, Jeff
Hi there, John
```

This works, but there is a problem. The `apply` method blocks until our function completes. That means that, if each call to `say_hello` took 10 seconds, our entire program's run time would be about 20 seconds because we've run things sequentially, negating the point of running in parallel. We can solve this problem by using the process pool's `apply_async` method.

Using asynchronous results

In the previous example, each call to `apply` blocked until our function completed. This doesn't work if we want to build a truly parallel workflow. To work around this, we can use the `apply_async` method instead. This method returns an `AsyncResult` instantly and will start running the process in the background. Once we have an `AsyncResult`, we can use its `get` method to block and obtain the results of our function call. Let's take our `say_hello` example and adapt it to use asynchronous results.

Listing 6.3 Using async results with process pools

```
from multiprocessing import Pool


def say_hello(name: str) -> str:
    return f'Hi there, {name}'


if __name__ == "__main__":
    with Pool() as process_pool:
        hi_jeff = process_pool.apply_async(say_hello, args=(
        hi_john = process_pool.apply_async(say_hello, args=(
        print(hi_jeff.get())
        print(hi_john.get())
```

When we call `apply_async`, our two calls to `say_hello` start instantly in separate processes. Then, when we call the `get` method, our parent process will block until each process returns a value. This lets things run concurrently, but what if

`hi_jeff` took 10 seconds, but `hi_john` only took one? In this case, since we call `get` on `hi_jeff` first, our program would block for 10 seconds before printing our `hi_john` message even though we were ready after only 1 second. If we want to respond to things as soon as they finish, we're left with an issue. What we really want is something like asyncio's `as_completed` in this instance. Next, let's see how to use process pool executors with asyncio, so we can address this issue.

ing process pool executors with asyncio

We've seen how to use process pools to run CPU-intensive operations concurrently. These pools are good for simple use cases, but Python offers an abstraction on top of multiprocessing's process pools in the `concurrent.futures` module. This module contains *executors* for both processes and threads that can be used on their own but also interoperate with asyncio. To get started, we'll learn the basics of `ProcessPoolExecutor`, which is similar to `ProcessPool`. Then, we'll see how to hook this into asyncio, so we can use the power of its API functions, such as `gather`.

ntroducing process pool executors

Python's process pool API is strongly coupled to processes, but multiprocessing is one of two ways to implement preemptive multitasking, the other being multithreading. What if we need to easily change the way in which we handle concurrency, seamlessly switching between processes and threads? If we want a design like this, we need to build an abstraction that encompasses the core of distributing work to a pool of resources that does not care if those resources are processes, threads, or some other construct.

The `concurrent.futures` module provides this abstraction for us with the `Executor` abstract class. This class defines two methods for running work asynchronously. The first is `submit`, which will take a callable and return a `Future` (note that this is not the same as asyncio futures but is part of the `concurrent.futures` module)—this is equivalent to the `Pool.apply_async` method we saw in the last section. The second is `map`. This method will take a callable and a list of function arguments and then execute each argument in the list asynchronously. It returns an iterator of the results of our calls similarly to `asyncio.as_completed` in that results are available once they complete. `Executor` has two concrete implementations: `ProcessPoolExecutor` and `ThreadPoolExecutor`. Since we're using multiple processes to handle CPU-bound work, we'll focus on `ProcessPoolExecutor`. In chapter 7, we'll examine threads with `ThreadPoolExecutor.` To learn how a `ProcessPoolExecutor` works, we'll reuse our count example with a few small numbers and a few large numbers to show how results come in.

Listing 6.4 Process pool executors

```
import time
from concurrent.futures import ProcessPoolExecutor


def count(count_to: int) -> int:
    start = time.time()
    counter = 0
    while counter < count_to:
        counter = counter + 1
    end = time.time()
    print(f'Finished counting to {count_to} in {end - start}
```

```
    return counter


if __name__ == "__main__":
    with ProcessPoolExecutor() as process_pool:
        numbers = [1, 3, 5, 22, 100000000]
        for result in process_pool.map(count, numbers):
            print(result)
```

Much like before, we create a `ProcessPoolExecutor` in `context manager`. The number of resources also defaults to the number of CPU cores our machine has, as process pools did. We then use `process_pool.map` with our `count` function and a list of numbers that we want to count to.

When we run this, we'll see that our calls to countdown with a low number will finish quickly and be printed out nearly instantly. Our call with `100000000` will, however, take much longer and will be printed out after the few small numbers, giving us the following output:

```
Finished counting down from 1 in 9.5367e-07
Finished counting down from 3 in 9.5367e-07
Finished counting down from 5 in 9.5367e-07
Finished counting down from 22 in 3.0994e-06
1
3
5
22
Finished counting down from 100000000 in 5.2097
100000000
```

While it seems that this works the same as `asyncio.as_completed`, the order of iteration is deterministic based on the order we passed in the `numbers` list. This means that if `100000000` was our first number, we'd be stuck waiting for that call to finish before we could print out the other results that completed earlier. This means we aren't quite as responsive as `asyncio.as_completed`.

rocess pool executors with the asyncio event loop

Now that we've know the basics of how process pool executors work, let's see how to hook them into the asyncio `event` loop. This will let us use the API functions such as `gather` and `as_completed` that we learned of in chapter 4 to manage multiple processes.

Creating a process pool executor to use with asyncio is no different from what we just learned; that is, we create one in within a context manager. Once we have a pool, we can use a special method on the asyncio event loop called `run_in_executor`. This method will take in a callable alongside an executor (which can be either a thread pool or process pool) and will run that callable inside the pool. It then returns an awaitable, which we can use in an `await` statement or pass into an API function such as `gather`.

Let's implement our previous count example with a process pool executor. We'll submit multiple count tasks to the executor and wait for them all to finish with `gather`. `run_in_executor` only takes a callable and does not allow us to supply function arguments; so, to get around this, we'll use partial function application to build countdown calls with 0 arguments.

What is partial function application?

Partial function application is implemented in the `functools` module. Partial application takes a function that accepts some arguments and turns it into a function that accepts fewer arguments. It does this by "freezing" some arguments that we supply. As an example, our count function takes one argument. We can turn it into a function with 0 arguments by using `functools.partial` with the parameter we want to call it with. If we want to have a call to `count(42)` but pass in no arguments we can say `call_with_42 = functools.partial(count, 42)` that we can then call as `call_ with_42()`.

Listing 6.5 Process pool executors with asyncio

```
import asyncio
from asyncio.events import AbstractEventLoop
from concurrent.futures import ProcessPoolExecutor
from functools import partial
from typing import List


def count(count_to: int) -> int:
    counter = 0
    while counter < count_to:
        counter = counter + 1
    return counter


async def main():
    with ProcessPoolExecutor() as process_pool:
        loop: AbstractEventLoop = asyncio.get_running_loop()
        nums = [1, 3, 5, 22, 100000000]
        calls: List[partial[int]] = [partial(count, num) for
        call_coros = []
```

```
        for call in calls:
            call_coros.append(loop.run_in_executor(process_p

        results = await asyncio.gather(*call_coros)

        for result in results:
            print(result)


if __name__ == "__main__":
    asyncio.run(main())
```

❶ Create a partially applied function for countdown with its argument.

❷ Submit each call to the process pool and append it to a list.

❸ Wait for all results to finish.

We first create a process pool executor, as we did before. Once we have this, we get the asyncio event loop, since `run_in_executor` is a method on the `AbstractEventLoop`. We then partially apply each number in `nums` to the count function, since we can't call count directly. Once we have `count` function calls, then we can submit them to the executor. We loop over these calls, calling `loop.run_in_executor` for each partially applied count function and keep track of the awaitable it returns in `call_coros`. We then take this list and wait for everything to finish with `asyncio.gather`.

If we had wanted, we could also use `asyncio.as_completed` to get the results from the subprocesses as they completed. This would solve the problem we saw

earlier with process pool's `map` method, where if we had a task it took a long time.

We've now seen all we need to start using process pools with asyncio. Next, let's look at how to improve the performance of a real-world problem with multiprocessing and asyncio.

ving a problem with MapReduce using asyncio

To understand the type of problem we can solve with MapReduce, we'll introduce a hypothetical real-world problem. We'll then take that understanding and use it to solve a similar problem with a large, freely available data set.

Going back to our example of an e-commerce storefront from chapter 5, we'll pretend our site receives a lot of text data through our customer support portal's *questions and concerns* field. Since our site is successful, this data set of customer feedback is multiple terabytes in size and growing every day.

To better understand the common issues our users are facing, we've been tasked to find the most frequently used words in this data set. A simple solution would be to use a single process to loop through each comment and keep track of how many times each word occurs. This will work, but since our data is large, going through this in serial could take a long time. Is there a faster way we could approach this type of problem?

This is the exact kind of problem that MapReduce can solve. The MapReduce programming model solves a problem by first partitioning up a large data set into smaller chunks. We can then solve our problem for that smaller subset of data instead of entire set—this is known as *mapping*, as we "map" our data to a partial result.

Once the problem for each subset is solved, we can then combine the results into a final answer. This step is known as *reducing*, as we "reduce" multiple answers into one. Counting the frequency of words in a large text data set is a canonical MapReduce problem. If we have a large enough dataset, splitting it into smaller chunks can yield performance benefits as each map operation can run in parallel, as shown in figure 6.1.

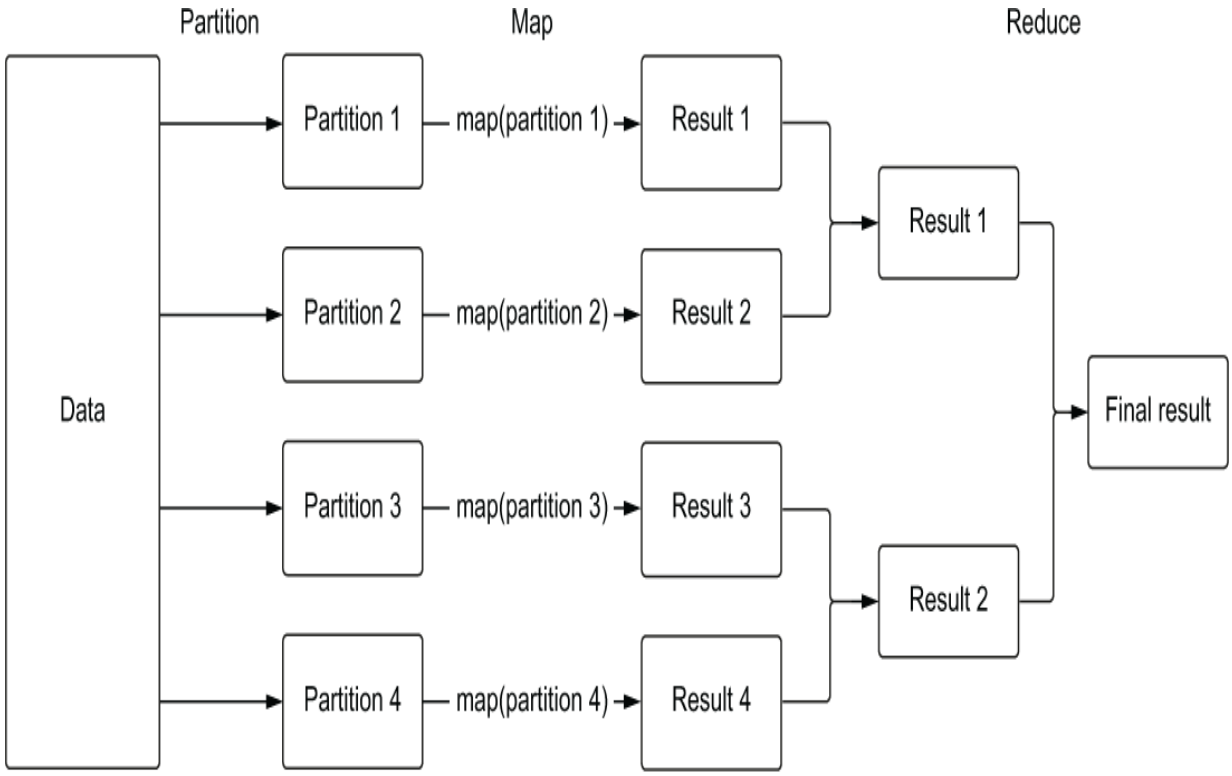

Figure 6.1 A large set of data is split into partitions, then a map function produces intermediate results. These intermediate results are combined into a result.

Systems such as Hadoop and Spark exist to perform MapReduce operations in a cluster of computers for truly large datasets. However, many smaller workloads can be processed on one computer with multiprocessing. In this section, we'll see how to implement a MapReduce workflow with multiprocessing to find how frequently certain words have appeared in literature since the year 1500.

A simple MapReduce example

To fully understand how MapReduce works, let's walk through a concrete example. Let's say we have text data on each line of a file. For this example, we'll pretend we have four lines to handle:

```
I know what I know.
I know that I know.
I don't know that much.
They don't know much.
```

We'd like to count how many times each distinct word occurs in this data set. This example is small enough that we could solve it with a simple `for` loop, but let's approach it using a MapReduce model.

First, we need to partition this data set into smaller chunks. For simplicity, we'll define a smaller chunk as one line of text. Next, we need to define the mapping operation. Since we want to count word frequencies, we'll split the line of text on a space. This will get us an array of each individual word in the string. We can then loop over this, keeping track of each distinct word in the line of text in a dictionary.

Finally, we need to define a *reduce* operation. This will take one or more results from our map operation and combine them into an answer. In this example, we need to take two dictionaries from our map operation and combine them into one. If a word exists in both dictionaries, we add their word counts together; if not, we copy over the word count to the result dictionary. Once we've defined these operations, we can then run our `map` operation on each individual line of text and our `reduce` operation on each pair of results from our map. Let's see how to do this example in code with the four lines of text we introduced earlier.

Listing 6.6 Single-threaded MapReduce

```
import functools
from typing import Dict


def map_frequency(text: str) -> Dict[str, int]:
    words = text.split(' ')
    frequencies = {}
    for word in words:
        if word in frequencies:
            frequencies[word] = frequencies[word] + 1
        else:
            frequencies[word] = 1
    return frequencies


def merge_dictionaries(first: Dict[str, int],
                       second: Dict[str, int]) -> Dict[str,
    merged = first
    for key in second:
        if key in merged:
            merged[key] = merged[key] + second[key]
        else:
            merged[key] = second[key]
    return merged


lines = ["I know what I know",
         "I know that I know",
         "I don't know much",
```

```
          "They don't know much"]

mapped_results = [map_frequency(line) for line in lines]

for result in mapped_results:
    print(result)

print(functools.reduce(merge_dictionaries, mapped_results))
```

❶ If we have the word in our frequency dictionary, add one to the count.

❷ If we do not have the word in our frequency dictionary, set its count to one.

❸ If the word is in both dictionaries, combine frequency counts.

❹ If the word is not in both dictionaries, copy over the frequency count.

❺ For each line of text, perform our map operation.

❻ Reduce all our intermediate frequency counts into one result.

For each line of text, we apply our `map` operation, giving us the frequency count for each line of text. Once we have these mapped partial results, we can begin to combine them. We use our merge function `merge_dictionaries` along with the `functools .reduce` function. This will take our intermediate results and add them together into one result, giving us the following output:

```
Mapped results:
{'I': 2, 'know': 2, 'what': 1}
{'I': 2, 'know': 2, 'that': 1}
```

```
{'I': 1, "don't": 1, 'know': 1, 'much': 1}
{'They': 1, "don't": 1, 'know': 1, 'much': 1}

Final Result:
{'I': 5, 'know': 6, 'what': 1, 'that': 1, "don't": 2, 'much'
```

Now that we understand the basics of MapReduce with a sample problem, we'll see how to apply this to a real-world data set where multiprocessing can yield performance improvements.

The Google Books Ngram dataset

We'll need a sufficiently large set of data to process to see the benefits of MapReduce with multiprocessing. If our dataset is too small, we'll see no benefit from MapReduce and will likely see performance degradation from the overhead of managing processes. A data set of a few uncompressed should be enough for us to show a meaningful performance gain.

The Google Books Ngram dataset is a sufficiently large data set for this purpose. To understand what this data set is, we'll first define what an n-gram is.

An *n-gram* is a concept from natural language processing and is a phrase of N words from a sample of given text. The phrase "the fast dog" has six n-grams. Three 1-grams or *unigrams*, each of one single word (*the*, *fast*, and *dog*), two 2-grams or *digrams* (*the fast* and *fast dog*), and one 3-gram or *trigram* (*the fast dog*).

The Google Books Ngram dataset is a scan of n-grams from a set of over 8,000,000 books, going back to the year 1500, comprising more than six percent of all books published. It counts the number of times a distinct n-gram appears in text, grouped by

the year it appears. This data set has everything from unigrams to 5-grams in tab-separated format. Each line of this data set has an n-gram, the year when it was seen, the number of times it was seen, and how many books it occurred in. Let's look at the first few entries in the unigram dataset for the word *aardvark*:

```
Aardvark   1822   2   1
Aardvark   1824   3   1
Aardvark   1827   10  7
```

This means that in the year 1822, the word *aardvark* appeared twice in one book. Then, in 1827, the word *aardvark* appeared ten times in seven different books. The dataset has many more entries for *aardvark* (for example, aardvark occurred 1,200 times in 2007), demonstrating the upwards trajectory of aardvarks in literature over the years.

For the sake of this example, we'll count the occurrences of single words (unigrams) for words that start with *a*. This dataset is approximately 1.8 GB in size. We'll aggregate this to the number of times each word has been seen in literature since 1500. We'll use this to answer the question, "How many times has the word *aardvark* appeared in literature since the year 1500?" The relevant file we want to work with is downloadable at https://storage.googleapis.com/books/ngrams/books/googlebooks-eng-all-1gram-20120701-a.gz or at https://mattfowler.io/data/googlebooks-eng-all-1gram-20120701-a.gz. You can also download any other part of the dataset from http:// storage.googleapis.com/books/ngrams/books/datasetsv2.html.

Mapping and reducing with asyncio

To have a baseline to compare to, let's first write a synchronous version to count the frequencies of words. We'll then use this frequency dictionary to answer the question, "How many times has the word *aardvark* appeared in literature since 1500?" We'll first load the entire contents of the dataset into memory. Then we can use a dictionary to keep track of a mapping of words to the total time they have occurred. For each line of our file, if the word on that line is in our dictionary, we add to the count in our dictionary with the count for that word. If it is not, we add the word and the count on that line to the dictionary.

Listing 6.7 Counting frequencies of words that start with *a*

```
import time

freqs = {}

with open('googlebooks-eng-all-1gram-20120701-a', encoding='
    lines = f.readlines()

    start = time.time()

    for line in lines:
        data = line.split('\t')
        word = data[0]
        count = int(data[2])
        if word in freqs:
            freqs[word] = freqs[word] + count
        else:
            freqs[word] = count
```

```
    end = time.time()
    print(f'{end-start:.4f}')
```

To test how long the CPU-bound operation takes, we'll time how long the frequency counting takes and won't include the length of time needed to load the file. For multiprocessing to be a viable solution, we need to run on a machine with sufficient CPU cores to make parallelization worth the effort. To see sufficient gains, we'll likely need a machine with more CPUs than most laptops have. To test on such a machine, we'll use a large Elastic Compute Cloud (EC2) instance on Amazon Web Servers (AWS).

AWS is a cloud computing service run by Amazon. AWS is a collection of cloud services that enable users to handle tasks from file storage to large-scale machine learning jobs—all without managing their own physical servers. One such service offered is EC2. Using this, you can rent a virtual machine in AWS to run any application you want, specifying how many CPU cores and memory you need on your virtual machine. You can learn more about AWS and EC2 at https://aws.amazon.com/ec2.

We'll test on a c5ad.8xlarge instance. At the time of writing, this machine has 32 CPU cores, 64 GB of RAM, and a solid-state drive, or SSD. On this instance, listing 6.7's script requires approximately 76 seconds. Let's see if we can do any better with multiprocessing and asyncio. If you run this on a machine with fewer CPU cores or other resources, your results may vary.

Our first step is to take our data set and partition it into a smaller set of chunks. Let's define a partition generator which can take our large list of data and grab chunks of

arbitrary size.

```
def partition(data: List,
              chunk_size: int) -> List:
    for i in range(0, len(data), chunk_size):
        yield data[i:i + chunk_size]
```

We can use this partition generator to create slices of data that are `chunk_size` long. We'll use this to generate the data to pass into our map functions, which we will then run in parallel. Next, let's define our map function. This is almost the same as our map function from the previous example, adjusted to work with our data set.

```
def map_frequencies(chunk: List[str]) -> Dict[str, int]:
    counter = {}
    for line in chunk:
        word, _, count, _ = line.split('\t')
        if counter.get(word):
            counter[word] = counter[word] + int(count)
        else:
            counter[word] = int(count)
    return counter
```

For now, we'll keep our reduce operation, as in the previous example. We now have all the blocks we need to parallelize our map operations. We'll create a process pool, partition our data into chunks, and for each partition run `map_frequencies` in a resource ("worker") on the pool. We have almost everything we need, but one question remains: what partition size should I use?

There isn't an easy answer for this. One rule of thumb is the *Goldilocks approach*; that is, the partition should not be too big or too small. The reason the partition size should not be small is that when we create our partitions they are serialized ("pickled") and sent to our worker processes, then the worker process unpickles them. The process of serializing and deserializing this data can take up a significant amount of time, quickly eating into any performance gains if we do it too often. For example, a chunk size of two would be a poor choice as we would have nearly 1,000,000 pickle and unpickle operations.

We also don't want the partition size to be too large; otherwise, we might not fully utilize the power of our machine. For example, if we have 10 CPU cores but only create two partitions, we're missing out on eight cores that could run workloads in parallel.

For this example, we'll chose a partition size of 60,000, as this seems to offer reasonable performance for the AWS machine we're using based on benchmarking. If you're considering this approach for your data processing task, you'll need to test out a few different partition sizes to find the one for your data and the machine you're running on, or develop a heuristic algorithm for determining the right partition size. We can now combine all these parts together with a process pool and the event loop's `run_in_executor` coroutine to parallelize our map operations.

Listing 6.8 Parallel MapReduce with process pools

```
import asyncio
import concurrent.futures
import functools
import time
from typing import Dict, List
```

```
def partition(data: List,
              chunk_size: int) -> List:
    for i in range(0, len(data), chunk_size):
        yield data[i:i + chunk_size]


def map_frequencies(chunk: List[str]) -> Dict[str, int]:
    counter = {}
    for line in chunk:
        word, _, count, _ = line.split('\t')
        if counter.get(word):
            counter[word] = counter[word] + int(count)
        else:
            counter[word] = int(count)
    return counter


def merge_dictionaries(first: Dict[str, int],
                       second: Dict[str, int]) -> Dict[str,
    merged = first
    for key in second:
        if key in merged:
            merged[key] = merged[key] + second[key]
        else:
            merged[key] = second[key]
    return merged


async def main(partition_size: int):
    with open('googlebooks-eng-all-1gram-20120701-a', encodi
```

```
            contents = f.readlines()
            loop = asyncio.get_running_loop()
            tasks = []
            start = time.time()
            with concurrent.futures.ProcessPoolExecutor() as poo
                for chunk in partition(contents, partition_size)
                    tasks.append(loop.run_in_executor(pool, func

                intermediate_results = await asyncio.gather(*tas
                final_result = functools.reduce(merge_dictionari

                print(f"Aardvark has appeared {final_result['Aar

                end = time.time()
                print(f'MapReduce took: {(end - start):.4f} seco


if __name__ == "__main__":
    asyncio.run(main(partition_size=60000))
```

❶ For each partition, run our map operation in a separate process.

❷ Wait for all map operations to complete.

❸ Reduce all our intermediate map results into a result.

In the `main` coroutine we create a process pool and partition the data. For each partition, we launch a `map_frequencies` function in a separate process. We then use `asyncio.gather` to wait for all intermediate dictionaries to finish. Once all our map operations are complete, we run our reduce operation to produce our result.