十三章

This is the third and last chapter that is entirely focused on Lua and uses only a few functions of the World of Warcraft API. I will show you a few useful tips and tricks you can use in your addons. This chapter also covers the coroutine standard library, which hasn’t been discussed yet as it is quite complicated and rarely used in addons (but it is still very powerful).

I will also show you how a few things in Lua work internally, including strings and tables. It is important to understand how these elements work if you want to use them in a smart way. Every data structure has its strengths and weaknesses, so it is important that you choose the right one in your addon. Choosing the right data structure also allows you to optimize your code to run faster or with less memory usage. However, optimization is a really overhyped topic in World of Warcraft addons. You will see a lot of completely unnecessary or wrong (meaning the resulting code is actually slower) optimizations when reading addon code.

Memory usage especially is a completely overhyped topic. Many people claim that addons with high memory usage slow down their game and cause lag spikes. What they describe as “high memory usage” often means 1 megabyte or more, which is ridiculous to worry about, as you probably have 2 GB of memory or more, and World of Warcraft alone rarely needs more than a gigabyte. This leaves you with 1 gigabyte for your OS, background applications, and your addons. I mentioned this topic in Chapter 6, when we discussed the garbage collectors of Lua 5.0 and Lua 5.1. World of Warcraft used Lua 5.0 before the Burning Crusade, and it had a really bad garbage collector for an application like a game. Such a gar\_bage collector is described as a “stop the world” collector; it stops the execution of the script while it collects garbage. But the new Lua 5.1 garbage collector is really fast, and you should not worry about it too much.

As you begin writing WoW addons in Lua, you won’t need many of the optimization tips and tricks I’m showing you here. However, you will need a deeper understanding of Lua as your coding skills develop and you write bigger and more complex addons

This chapter shows you when you should think about performance, how you can mea\_sure it, and what can be optimized. I’m also showing you a lot of small tricks that can make your life easier. The first thing we need to discuss is measuring the performance of an addon or any Lua script.

Lua provides only a few functions that allow you to measure its performance, while World of Warcraft has several advanced profiling functions

There are basically two resources that can be used by a Lua script: CPU time and memory. However, saving CPU cycles and saving memory are often two mutually exclusive aims. Many optimizations that reduce the required CPU time do so by using more memory, for example as a cache. You will have to trade off CPU usage against memory usage. But in most cases you should opt for less CPU usage and more memory usage, as memory is cheap and you usually have enough of it.

You can get the CPU time that was used by a specific Lua script by calling the function os.clock(), which is not available in World of Warcraft. This function returns the CPU time used by the currently executing Lua script, and you can use it to quickly get the time required to execute a certain script outside the game. The following example illustrates this:

The result depends on the speed of your computer.

Another way to get the execution time is to use an external program that starts the Lua script and prints the time required to run it. This can be done with the simple program time in Linux and Unix by calling time lua5.1 myScript.lua. Windows does not come with such a program, but that is not a problem, as os.clock() is sufficient for us here.

The os.clock() function always returns the total execution time of the whole script. You can use the following idiom around a block of code to measure its execution time independently from the rest of your script:

Note Although I will not include the line print(os.clock()) at the end of every example in this chapter; you can add it wherever you want to measure the speed.

The other important resource you have to work with is the available memory. You’ve already seen how we can track the memory usage of a Lua script, when we discussed the garbage collector in Chapter 6. The function collectgarbage(option) returns the amount of memory (in KB) currently in use by Lua if option is set to “count”

The following code prints the size of an empty table in bytes by comparing the memory usage before and after creating the table. The resulting value is multiplied by 1024 to get a value in bytes, which is more meaningful here, as an empty table is relatively small:

The following code prints the size of an empty table in bytes by comparing the memory usage before and after creating the table. The resulting value is multiplied by 1024 to get a value in bytes, which is more meaningful here, as an empty table is relatively small:

World of Warcraft provides a few advanced features that can be used to profile the performance of your addon. There are advanced functions available for memory and CPU profiling.

Note Most of the examples in this chapter can be tested with a normal Lua interpreter without using World of Warcraft. I will note when an example works only in World of Warcraft because it uses WoW-specific functions

The CPU profiling functions of World of Warcraft are quite powerful; they track almost everything. They can tell you the amount of CPU time used by a specific function or by all event handlers of a given frame. These profiling features are disabled by default because they come with a significant performance hit. Execute the following slash command to enable the profiling:

You have to reload your UI before this change takes effect. But don’t forget to disable this feature by executing the following command after you are done with profiling:

Once profiling is enabled, there are a few functions you can call to profile your code:

ResetCPUUsage():Resets the profiling statistics

UpdateAddOnCPUUsage():Updates the data that is retrieved by all of the following functions. You have to call this function before retrieving statistics with the functions that follow in this list

GetAddOnCPUUsage(addon): Gets the CPU time used by addon in milliseconds until the last call to UpdateAddOnCPUUsage(). The addon argument can represent the name of the addon as either a string or a number.

GetFunctionCPUUsage(func, includeSubs): Retrieves the CPU time in milliseconds that was used by func. The includeSubs argument can be set to false to ignore the time needed by functions called by func; the default value is true, meaning that all following function calls are included in the statistics

GetFrameCPUUsage(frame, includeChildren): Returns the CPU time in milliseconds used by all event handlers of frame. The includeChildren argument can be set to false to ignore the script handlers of the frame’s children; the default value is true, meaning that the CPU usage of the children is included in the result.

GetEventCPUUsage(event) : Gets the CPU time used by all event handlers that handle event.

Throughout this chapter you will see a lot of short examples that compare the performance of different solutions for the same problem. I will often tell you the time required by the scripts on my computer (a laptop with a Core 2 Duo with 2.00 GHz CPU). You can then easily see which script is faster. But I will also use percentage values from time to time, which can cause a lot of confusion.

For example, what does “100% faster” mean? Let’s say we have two scripts, A and B, and the former runs faster than the latter. I will say that script A runs x% faster than script B if script A runs in x% less time than B. 100% faster would thus mean that script A runs in no time at all.

Another important convention I’ll use in this chapter is the big O notation. This notation provides us with a way to describe the resource usage (CPU or memory) of an algorithm based on the size of the input value. I’ll spare you the definition as it is highly mathematical, but let’s see how this works without too much math.

The big O notation looks like this: o(f(n)), where f(n) is replaced with a function that describes the growth of the required CPU time or memory usage, and n is the size of the input, for example the size of a table or the length of a string. This means that an O(1) function does not become slower (requiring more memory) when you use it on larger input values. A simple example of this is the following function:

This function does not become slower if you use a larger table t, because accessing a key is internally an O(1) operation (you will see how table indexing works in detail later). The following function goes through a table and looks for a specific value; it is therefore O(n) as it grows in line with the size of the table t (because it has to iterate over the whole table):

Note that an O(n) operation is not necessarily slower than an O(1) operation on certain input values. For example, the following completely pointless function runs in O(1) but is still very slow:

The performance of this function is obviously poor. But it does not get worse when you use a bigger table t, which means it is O(1). The O(n) function earlier is probably faster for all input values you ever might use, but it is slower than this O(1) for really huge tables. And it keeps getting slower as the table grows. This illustrates that the big O notation only tells us how fast the required time grows, not how much time is actually required

Also interesting is the following function, which takes a sorted array filled with numbers. It checks whether a given number is in this array but it is faster than the previous O(n) function as it uses the so-called binary search algorithm, which runs in O(log n):

The function is more complicated than iterating over the table, but it is faster. Feel free to test it with the techniques I mentioned earlier. The binSearch function first looks at the element in the middle of the array and continues looking for the value in the left part of the array if the value in the middle is larger than val. Otherwise it looks in the right part of the array, as we assume that the array is sorted. The function continues doing this until the searched interval is empty or the value is found

Another example that will help you get a feeling for the meaning of the big O notation is the following function. It implements the bubble sort algorithm, which sorts a table in the time O(n2):

You might have guessed that O(n log n) lies between O(n) and O(n2 ). An example of such a function is table.sort(t, compare). It can sort a table in O(n log n), as it internally uses quick\_sort. It is actually not strictly O(n log n); it might run in O(n2 ) in the worst case, but it is very unlikely that this happens for huge tables. O(n log n) is the average required time for quicksort. There is another sorting algorithm that guarantees to run in O(n log n): merge sort. But it is still slower than quicksort for almost all tables, as a single step is more expensive than in quicksort. Most programming languages thus use quicksort. I’m not showing quicksort here, as you don’t need to worry about sorting tables in Lua; you can just use table.sort.

You now know how you can measure the performance of your code, and we’ve discussed the necessary conventions. But what should be optimized and what not?

The results of optimizing your code without thinking about it first are that you’ll prob\_ably waste your time, you’ll make your code harder to understand, and you might do it wrong and lose performance in the end. This means that you should always measure the performance of your script to see if it needs optimization; 99% of all addons simply don’t need to be optimized

All of the following optimization tips should only be applied to code that is crucial for the performance of your addon. An example of this would be an OnUpdate handler that is used by more than one frame (like the OnUpdate handler of the bar objects in our first version of CooldownMonitor). Another example is the COMBAT\_LOG\_EVENT\_UNFILTERED event, which can be called really often. I measured peaks of up to 800 events per second in certain boss fights; this means your event handler is called even more often than an OnUpdate handler. Do not waste your time with optimizing a function that is called only occasionally. Always measure performance before and after optimization to see if the optimization really works

Let’s see what can be optimized in an addon. A good example of something that can be a real performance hog is the string

Strings seem to be innocuous, and you probably didn’t expect that a simple operation like the concatenation of two strings can slow down your script. You need to know how strings work internally in order to understand the problems you can run into.

Strings seem to be a value type, meaning that they seem to be passed as values (rather than references) to a function. The function can then modify the string, and the caller doesn’t notice it. The following example illustrates this behavior:

This prints test because strings behave as if they were passed as values, which means the modification of the string in test does not affect the original string.

But a string is internally always passed as a reference; you just don’t notice this implementation detail because strings are immutable. There is no way to change an existing string; you can only generate new strings. This means that the statement t = t..”foo” creates a new string that consists of the concatenation of t and “foo” and stores it in the local variable t.

Every string is internally kept in the string pool (think of it as a huge hash table with all strings), and every string is unique. The following happens when you try to create a new string: Lua builds the string, checks to see if it is already in the string pool, and inserts it if it doesn’t exist. You then get a reference to this string in the string pool

There is nothing new so far; I described this behavior in Chapter 2. But what does it mean for us? Strings are subject to garbage collection, which means an unused string doesn’t vanish immediately when you remove the last reference to it. So if you have an extremely long string (let’s say 50 million characters long, or 50 MB memory usage) and append a short string (for example, “\n” for a new line), you will have this string twice in your memory until the original string is collected.

Let’s start with an example. Imagine you have a function that accepts CHAT\_MSG\_ADDON events with strings sent by other players. But the message you want to receive may exceed the 255-character limit of chat messages, so your sender function splits the outgoing message into multiple strings and the receiver function concatenates them.

The following code shows two functions that could be used in such an addon, syncHandler and receive. The receive function is called every time the addon receives a part of a multipart message; the prefix “Foo-End” indicates the end of the message. The function concatenates and stores all partial messages in the table received with the sending player as key, to prevent problems when multiple players send you messages at the same time. Then receive calls syncHandler with the fully concatenated message as its argument when the last part of a message is received

It is very unlikely that you will ever need to send a lot of data through to another player, so let’s say the maximum number of messages you expect to receive here is 10 and each one has a size of 245 characters, which gives us 10 characters for the prefix to identify the addon

Add the following code after the functions to simulate the CHAT\_MSG\_ADDON events with test data:

This code simply creates 10 messages and passes them to receive. Try to execute the code and you will see that it runs extremely fast, and you probably don’t expect it to be a problem. But let’s examine what happens when you run the code

It concatenates the first message with the second one, so that you now have the original message (245 bytes) and the concatenated message (490 bytes) in your memory. It then con\_catenates this 490-byte message again with the 24-byte message; you now have three strings: the original string, the first two strings concatenated, and the first three strings concatenated (735 bytes)

The overall size of the generated strings is now already 1470 bytes, and it gets worse and worse with each additional message. The memory that is moved around if we concatenate n messages is 245 \_ (1 + 2 + … + n) = 245 \_ n \_ (n + 1) / 2 bytes. Thus the example code that simply concatenates ten messages already uses 13 MB of memory. A real addon would even require slightly more memory, as the received sync messages would differ. Our example uses the same content for every test message, so it doesn’t generate a new 245-byte string in every step. But this is only a small amount of memory required compared to the concatenation

Note that the garbage collector will collect unused strings during the concatenation so that the whole memory isn’t allocated at the same time. But this can really slow your script down, especially because the required memory grows by O(n2 ).

Imagine that something goes wrong and you receive a lot of synchronization messages without a Foo-End message. Or even worse, what happens if a malicious player in your raid sends you faked messages all the time to crash your game?

Sending 10,000 messages can be done relatively quickly. Let’s test what would happen if someone tries to do this. Change the line saying for i = 1, 10 do to the following to simulate 10,000 messages:

We can calculate the memory that is moved around during this concatenation with the formula from earlier: 245×10,000×(10,000 + 1) / 2 = 12,251,225,000 bytes. That is over 12 gigabytes. Running this script now takes about 30 seconds on my laptop as Lua requires a tremendous amount of memory and the garbage collector is working all the time. You can watch the process lua5.1.exe in your Task Manager while you execute the script. This will give you an impression of how Lua has to shuffle memory around to concatenate all those strings

Figure 13-1 shows the memory graph in my Task Manager while executing the script with 10,000 messages. You can clearly see that the garbage collector runs interleaved with your script; it is started every few execution steps and collects a few objects that are considered to be garbage.

This problem with string concatenation is not restricted to Lua; many languages (most notably Java and C#) have a similar way of dealing with strings and thus the same problems. The solution in many languages is a StringBuilder object that internally uses a smarter way of concatenating the strings. Lua doesn’t have such an object, but there is still a simple solution to this problem

There is one function that can be used to concatenate multiple strings without quadratically increasing memory usage: table.concat(tb1, delimiter). This function takes a table that has only strings in its array parts and concatenates them (separated by delimiter, which is optional). The World of Warcraft API provides a similar function that can also be used: string.join(delimiter, ...). It works on multiple string arguments instead of a table, but we will use table.concat as it is more convenient for our example.

We can modify the function receive to build a table containing all received strings; this table is then concatenated when the last message is received:

You can test it with 10,000 or even 100,000 messages; it runs extremely fast. Table 13-1 compares the performance of the old version, which simply concatenated the strings, with the new receive function, which uses table.concat

The asterisks mean that concatenating strings fails with a “not enough memory” error message on my laptop (32 bit OS with 2 GB memory) when I try to use 20,000 or more messages. You might go farther with more memory or with a 64-bit version of Lua, but the required time still grows extremely fast. Tuning the garbage collector settings to be more aggressive will also help, but it does not solve the actual problem.

What we learn from this is that one should be careful with string concatenations. Using a concentration operator in a loop is a particularly bad idea in most cases; try to use table.concat or string.join whenever possible

This optimization is specific to World of Warcraft and helps you to avoid string garbage. Imagine you have a FontString object in a unit frame that displays the health of the unit. You now register the UNIT\_HEALTH event and use an event handler like the following:

The handler is now called every time the health of the watched unit changes, which can be as often as every frame. Imagine you use this handler in a raid frame mod and you are in a 40-man battleground like Alterac Valley. The handler is probably called about 20 to 40 times every frame if the group is currently in combat. Every call creates a new string from the format string with about 20 characters.

What happens to these strings? They are passed to the SetText method and never used again by Lua, only internally by the frame. This means that they are now garbage and will be collected sooner or later. It seems to be not much; only about 600 bytes of garbage if 30 raid frames are updated. But every font string that is often updated (think of your combat log or the text displayed on timers) generates these garbage strings, which are passed to a function and then dropped. All of these frames together can quickly generate a lot of garbage, which keeps the garbage collector busy and slows down the game.

You actually already know the solution to this problem, as I mentioned it earlier in this book when we wrote the addon CooldownMonitor. All frames that display text like FontStrings have the method SetFormattedText, which basically has the following functionality:

The only difference is that it does not call string.format in Lua; it formats the string internally and displays it without generating a Lua string. This can save a lot of garbage if it is used consistently.

This simple trick can safely be used every time you have to format a string and then display it. It is one of the easiest optimizations in World of Warcraft. Another really easy optimization is using local variables instead of global variables.

I have already mentioned several times that one should prefer local variables over global ones. But why? Well, one obvious reason is that your code will be more readable if a variable is only accessed and changed in a specific block of code. Moreover, a global variable can be accessed and changed from anywhere, so you can never be sure what other file or addon also modifies a given global variable

But there is another reason you should prefer local variables over global ones: accessing a local variable is about 30% faster. We can test this with a tight loop (a loop with a single instruction to measure the performance of this instruction) that counts from 1 to 10^8. The following code uses a global variable to do this

It runs quite slowly; Lua needs 13.1 seconds on my laptop for this loop. Recall how global variables work in Lua: all global variables are stored in a huge hash table called the global environment, \_G. You will see how tables work in detail later in this chapter. A local variable is a “real” variable, which means that its name is just an identifier for the compiler and not a string. Lua knows where a local variable is stored in your memory, while it needs to calculate the position of a global variable.

The following version uses a local variable instead of a global one, so it should be faster:

The simple change of adding the keyword local makes the code about 78% faster (so that it takes only about 2.8 seconds). Note that the results can vary slightly for you, as they depend on a lot of factors.

Another interesting test is to declare the used local variable outside of the function that uses it. This means we create a closure of a function that references this local variable. Such a local variable is also called an upvalue in Lua. Here’s an example:

One might expect that there is no difference between a local variable and an upvalue. But there is a huge difference; this code runs in 4.6 seconds. It is still 65% faster than using a global variable. But using a “real” local variable is 39% faster than an upvalue.

Accessing a global variable is still incredibly fast. The huge speed gains we saw in the last section are not so huge in a real application. They were so high in the previous examples simply because the only thing the examples did was access variables. Let’s see an example that calls the functions math.cos and math.sin.

Accessing these two functions takes even longer than accessing a simple global variable, as each call first gets the global variable math, which is a table. Lua then needs to get the fields cos and sin from this hash table; the time needed for this is comparable to accessing a global variable, as global variables are technically also entries in a hash table. Note that the size of a hash table does not affect the time needed to get an entry from it (we will see how this works later in this chapter).

This runs in 31.5 seconds on my computer. Let’s store the functions in local variables. This reduces the four table/global variable accesses to two local variable accesses:

The code now runs in 23 seconds, only 27% faster. You can also test upvalues, and you’ll find that there is barely a difference.

But there is a difference, and this optimization can be applied easily. It is always a good idea to save references to functions or tables that are used in frequently called code in local variables. You will see this optimization in a lot of files; they often begin with code looking like the following. This optimization technique is often called caching, as the local variables basically act as cache for the global functions.

But one important question is left unanswered: why is accessing a global variable so slow? Global variables are stored in a table (\_G), so we need to understand how tables work in order to understand what happens when we access a global variable

Tables are divided into two parts: the array part and the hash part. The array part is relatively simple, so we will start with it.

An array is a very simple data structure that uses positive integers as indices; these indices are used to calculate the position of an entry in your memory. This means we first need to look at how the memory is organized internally.

Your memory is divided into many small cells, each of which is 8 bits (1 byte) wide, and you can only access these cells. The cells are numbered, and the number of a cell is its address. The size of your memory on 32-bit systems is limited to 2^32 bytes, or 4 gigabytes, as the maximum number of bits that can be used for an address is 32.

We luckily don’t need to worry about this when programming in Lua, as Lua already does all the hard work for us. We cannot tell Lua to read the memory at a given address. But we can get the memory address of some Lua objects, like tables, as the default \_\_toString() method prints it:

Here, 0059AC90 is the base memory address of t on my system; you will probably get a different memory address. Lua internally stores various items of information about the table at this memory address. One of these items is the memory address of the array part of the table, which can be in a completely different part of your memory (the same applies to the hash part). I will refer to this address as the array base address from now on.

The interesting question is what happens when we now try to access a field in the table, like t[1]. Lua first checks whether the index is in the range of the array, as arrays internally have a fixed size, even though they seem to be dynamically sized. (We will further discuss sizes of arrays later in this section.) Lua checks the hash table part of the array if the requested index is not in the range of the array part.

If it is in the array part, Lua first subtracts one from the index to get a zero-based index, which is used internally. This zero-based index is now multiplied by 16 and added to the array base address. The calculated memory address holds an object that represents an array entry, which is always 16 bytes wide.

The following illustration shows how such an array t looks in your memory. The table header is at the memory address that is returned by tostring(t); it is slightly simplified here, as it is internally 32 bytes wide and consists of more than just its array and hash address. Discussing everything that is stored in a table header would go too far here and is not important for understanding how tables work. The array address holds the position of t[1], the first entry in the array part. A short calculation shows that this is correct:

This explains why accessing an array is comparatively fast and does not get slower as you add more elements to it. Lua can always calculate the position of every element in the array. One obvious conclusion we can draw from this behavior is that the size of an array must be fixed internally. Imagine what would happen if it was not fixed and you tried to insert a value somewhere. The memory at the calculated position is not necessarily part of the array, and you might overwrite something important, causing your script to crash. This is not the case, so we can see that the size of the array is fixed.

The initial size of an array depends on the table constructor used. A simple {} creates an empty array part, while {1} creates an array with one slot. You might have guessed that {1, 2} creates two slots, but you probably didn’t guess that {1, 2, 3} creates an array part with four entries, of which one will stay empty.

The reason for this is that arrays always double their size once they reach their limits (powers of two). The following code illustrates this behavior:

Resizing an array is with O(n) a comparatively expensive operation, compared to accessing or changing a value. Lua needs to find a new place in your memory with enough consecutive free space for all entries and then copy the old array to it. You don’t want to do this all the time when you have a large array and add new values to it. We can test the performance of different table constructors with a simple loop:

This code runs in 8 seconds on my laptop as the table is initially created with 8 entries. The following code creates an empty table and adds the entries afterward:

This code, which does the same thing, requires 31 seconds to run, as it resizes the table three times in every execution of the loop’s body

There is a third way to create such a table, by explicitly using the integer indices in the table expression:

You might expect this to run as fast as the first version, but it takes 13 seconds. The reason for this is that Lua does not recognize it as an array in the first place. It mistakes the indices for hash table entries and generates a table with an eight-slot hash table part. However, it is still faster than adding the entries outside of the constructor, as Lua internally fills and resizes the array without the overhead of switching between Lua code and the internal C code

This discussion of array size leads us to an old issue we encountered in the Chapter 4 example SimpleTimingLib, which used arrays to store arguments that were later passed a function with unpack(). The problem was that one might pass nil as an argument to a function, which raised the question of whether arrays can store nil values. The answer I gave you there was yes, but only under certain circumstances.

It depends on the size of the array and the number of nil values. Lua tries to balance memory usage and access speed when using integer indices. Having an array with 128 entries just because you want to use the entries 1 and 65 obviously doesn’t make sense at all. But having a 4-slot array when you want to use the entries 1 and 3 does make sense and is possible.

Lua uses the array part if we explicitly tell the compiler to do so in the table constructor. The following example illustrates this:

Note that an array can never have trailing nil values; they will always be cut off. This also works when creating arrays from the vararg parameter ....

You might wonder why SimpleTimingLib sometimes failed with relatively few nil values even though it used the vararg to create the table. The reason is that modifying the table, for example by using the hash table part of it, causes Lua to optimize the array by moving values into the hash table part.

The array part will then be used only if more than 50% of the slots in the array part are occupied. We can test it with a simple function that adds a dummy field to the hash table part of the array. Our SimpleTimingLib used the hash table to store the function to be called and the time when the task is due:

This illustrates the “more than 50%” restriction that applies as soon as you start using the hash table part. However, this restriction is not really a problem for our timing library, as you rarely need to pass multiple nil values to a function. And even if you have to do so, most functions also accept false instead, which can be used without problems in an array.

A hash table is a more sophisticated data structure than a simple array. It has the ability to associate arbitrary values like strings with other values and is therefore also called an associative array. What makes hash tables especially interesting is that reading or setting a value in them is an O(1) operation. One might think that an associative array internally simply iterates over all entries until it finds the requested one, which would be an O(n) operation. But that is not the case. Lua can calculate the memory address of a given entry in the hash table, and you will see how this works as we build our own hash table.

I’ll outline how a hash table works before we create the example table

Arrays simply store values; hash tables also have to store the key, which can also be an arbitrary Lua value. Hash table entries are also called key/value pairs. You might have guessed the memory usage of such a key/value pair: 32 bytes, as arrays use 16 bytes.

A hash table internally uses an array with a fixed size to store its key/value pairs. The array starts with no slots for key/value pairs and grows by the same rules as the array part of the table

The position of a key/value pair in this array can be calculated from the key by using a hash function, which converts an arbitrary object into a number called the hash code (or simply hash) of the object. This hash is then used as the index for the key/value pair in the array. The simple calculation hash % #array (with array being the array used for the key/value pairs) ensures that this index is guaranteed to be inside the bounds of the array.

Note that this hash is not unique; there might be multiple different objects with the same hash, which leads to a collision in the hash table. A good hash function ensures that the number of collisions in the table is kept low, but you cannot avoid them completely. This means that we need a way to resolve these conflicts

There are several ways of resolving conflicts; one of the simplest is using a linked list of key/value pairs in the slot where multiple keys collide. New colliding keys will simply be appended to the list, and the get operation that retrieves a key/value pair from the hash table will iterate over the list and check if the requested key is in one of the key/value pairs.

The worst case is that all keys in your table have the same hash code, which degrades the hash table. It is then a linked list that is traversed every time you add a new entry or read an entry from it which is an O(n) operation. But it is extremely unlikely that this will happen when using a good hash function like the Lua hash function.

That was the theory; let’s now create a simple hash table.

We will create a hash table object in Lua to get a better understanding of hash tables. Note that the performance of the hash table we create here will be very poor compared to the original Lua hash tables, especially because we are using Lua hash tables with two entries (three entries in case of a collision) to represent key/value pairs in Lua. The only purpose of this example is to demonstrate how hash tables work.

Our hash table object will behave like a normal Lua hash table. We can do this by using metamethods to overload the table-access operations. The only difference is the performance, which as noted will be very bad; one reason for this is that it is written in Lua and not in C, like the hash tables provided by the language. Another reason is that the hash table will even internally use Lua hash tables to represent key/value pairs. Our example also uses a very simple hash function; this will lead to a lot of collisions, as a good hash function is somewhat complicated. You can read the Lua source code (file ltable.c) if you understand C and want to see how the Lua hash tables work in detail

Let’s start with creating a simple hash function. We will be using a function that simply uses tostring on the value and then calculates the sum of all characters’ byte values in the string. This can be done with two simple functions:

You might be tempted to use a recursive function for sum, but recursion is usually slower than a loop. Let’s assume for a second that we really want to use this hash table in an addon; that means the hash function is a performance-critical part of the implementation. So using recursion to sum up a few numbers is off the table here

Let’s now build the actual hash table, which usually defines three operations: a get operation to retrieve the value that is associated with a given key, put to store a new key/value pair in the table or to update an existing one, and delete to remove an existing key. We don’t have to define methods for this; we can use the metamethods \_\_index for the get operation and \_\_newindex for put and delete

Let’s start by creating the \_\_index method. We will use the field size to store the current maximum size of the array to emulate a static sized array. kvPairs holds the array with all the key/value pairs that are currently stored in our hash table. The method calculates the position of the requested key and then checks all key/value pairs stored in this bucket (an entry in the array holding the key/value pairs) which is just one pair unless there have been collisions. We have to add 1 to the calculated position because Lua arrays are 1-based; the position 0 would be in the hash table part of the array, and we don’t want that.

This function demonstrates why retrieving a value from the hash table is an O(1) operation if the hash function is good. The position is calculated and the operation is then a simple array access, which is an O(1) operation because the actual memory address can be calculated.

The \_\_newindex method is slightly more complicated than \_\_index as it needs to check for collisions and resolve them. We also need an additional attribute in our hash table object: usedSlots, which holds the number of currently occupied slots. The put operation calls the function resize, which doubles the size of the hash table if we use more slots than we have.

One thing you might notice is that setting a field to nil does not remove it from the table. It just sets its value to nil, but the key/value pair still exists and the table doesn’t shrink. Regular Lua hash tables show the same behavior; the only way to really remove something from them is to trigger a resize. This resize operation has to calculate the new positions (rehash the table); this means it has to loop over all entries in the array, so this is a good place to drop the nil values from it. Place the following function above the \_\_newindex metamethod, as it is required there:

Note that our table can never actually shrink, only grow. Real Lua tables can also shrink when a rehash is triggered, but a rehash can only be triggered by adding entries, not by deleting them

Our hash table is nearly finished. The last thing we need before we can test it is a constructor:

The simplest test we can run is using a few keys that do not collide and check to confirm that we can read them:

The hash code of a is 97, b is 98, and so on. This means that consecutive characters do not collide with our hash algorithm. We know that t behaves like a normal table, so let’s test whether it is really using our hash table implementation, by adding the following code:

The rawget function performs a raw table access without invoking the metamethod \_\_index. We can now try to read the key/value pairs from the subtable kvPairs. Note that accessing t.kvPairs does not invoke \_\_index, as this metamethod is only called if rawget returns nil. (The same is true for \_\_newindex.)

The first entry contains the key d as:

One might expect “a” to be the first entry in this hash table because it was the first one we added to it, but you cannot control the order of elements in a hash table and you cannot sort a hash table. We already know that from normal Lua hash tables, where the iterator function next returned the key/value pairs in an order that seemed to be random. We can now explain why: next just iterates over the array that holds the key/value pairs and follows all linked lists in case of collisions, so the order of hash entries might even change when the table is resized

Another interesting test is provoking a collision, which is really easy with our bad hash function. For example, the keys 1 and “1” collide as the hash calls tostring on the key:

We can also see the collision by looking into the array kvPairs, where both entries are stored in the second bucket:

One last test for our hash table is checking if the rehash works properly. Let’s start with a hash table with the two colliding entries “a” and “c”:

This actually already triggers the first rehash when the second entry is added. This is why “c” is in the second entry of the list and not in the first, as you might expect, because colliding entries are added to the beginning of the list. But let’s test what happens if we add another entry, which resizes the table to 4. We see that “a” and “c” no longer collide if the table is resized to 4:

Everything seems to work just fine in our hash table. We can now use this knowledge to see what we can optimize when using tables.

The simplest way to optimize tables is to avoid creating them unnecessarily. A good example of this is a typical constructor for an object:

This creates a new metatable for each object that is created. That is not necessary, as all objects have the same metatable. It is better to define the metatable outside the function as we have done throughout the book

This was a very obvious optimization. But there are more sophisticated tricks that do require understanding the internal details of hash tables. One of them is creating tables with the appropriate size. Imagine you have a table storing an object that starts with four entries, but you know that there will be a fifth entry added later. Let’s write a simple demonstration of this:

The problem is that the fifth entry triggers a rehash of the table, and the code therefore runs quite slowly, taking 18 seconds on my laptop. We can create a table with a hash table of the size 8 by adding a dummy entry and setting it to nil in the table constructor:

This avoids the rehash and is 33% faster, as it runs in only 12 seconds.

Let’s see more optimization with a real-world example: our library SimpleTimingLib desperately needs optimization, as we did not consider performance in its current version

The current implementation of SimpleTimingLib is not very smart. The OnUpdate handler performs a task that runs in O(n), which is not acceptable if you want to use this library in addons that schedule a lot of tasks.

Those of you who are familiar with this topic might suggest using a so-called priority queue as the data structure. You’re probably already thinking of a binary search tree like an AVL tree or a heap such as a binary heap or a Fibonacci heap; these are theoretically extremely efficient solutions for our problem. However, such a data structure is very sophisticated and difficult for beginning programmers to understand. So we are not going to see it here. I did consider explaining a simple priority queue (a binary heap); it would be relatively short with about 50 lines of code. However, the explanation would certainly be very long and complicated. I therefore decided to skip this here as it would be beyond the scope of the book. If you are interested in this topic, you might want to read a book about algorithms and data structures. However, we can still improve SimpleTimingLib here; the difference in performance between the solution I’ll present here and a proper priority queue is minimal for most use cases

We will see a very simple but effective solution here: a linked list that stores all entries. This seems to be very similar to our current solution using an array. But we will use a simple trick to speed the code up: we will keep the array sorted when inserting new values. The OnUpdate handler only needs to look at the first entry in this list if it is sorted. Thus it runs in O(1), while the new insert operation runs in O(n)

I’ve tested all of the data structures just mentioned for the scheduler used in Deadly Boss Mods. A proper priority queue, which does both the schedule (insert) operation and the extract min operation (removing the timer with the lowest time value) in O(log n), provides a clear advantage over a simple linked list. But I used millions of simultaneous timers in this test, which is not a realistic scenario for either DBM or SimpleTimingLib. For example, Deadly Boss Mods makes extensive use of its scheduling library, but there are usually only four to eight timers running at a given time in a complex boss fight. The advantage is then extremely small as a lot of the time is not wasted in the O(n) insert operation but in tasks like building the table that holds the function arguments.

But a small advantage is still an advantage, and DBM therefore uses a binary heap as the priority queue for its scheduling. You can read the well-commented code in the file DBM-Core.lua (search for “Scheduler” to find it) if you are curious about how such a data structure works. But beware: it is not discussed here for the reason that it is beyond the scope of a beginner’s book so the code might seem to be very sophisticated.

Enough theory; let’s get started with implementing our linked list in SimpleTimingLib.

We basically moved the hard work from the OnUpdate handler to the function that inserts the entry, which is quite a smart solution. OnUpdate is called very frequently, while the insert function is only called from time to time by another addon

All of the following code blocks are updates for the embedded library SimpleTimingLib-1.0, which means you must replace the respective functions. It might also be a good idea to increase the major version of this library to SimpleTimingLib-1.1.

The first thing we change is the array that currently stores all entries; we will replace it with a linked list and therefore initialize the variable tasks with nil instead of an empty table. Replace this:

The schedule function now creates an ascending sorted linked list of all scheduled tasks:

You can easily see from the code that this operation is now O(n), as it might iterate over the whole linked list to find the right place to insert it. The following code shows the new OnUpdate handler:

It looks like a typical O(n) operation, as it seems to loop over the whole array. It might run in O(n) when all scheduled tasks are due in the current frame. However, we have to look at the overall performance, and even in worst case, that all tasks are executed in this frame, you will have an empty linked list in the next call to OnUpdate. In almost all calls, the function will look only at the first element, because it cancels the loop as soon as it encounters a task that is not due yet.

The unschedule function also needs to be adjusted, but we can’t really optimize it. The reason for this is that we have to look at all tasks to check whether they match the provided arguments:

SimpleTimingLib now looks slightly more complicated, but it will be faster if used by a lot of addons. The purpose of a library is to be used by multiple addons, and you have to assume that a library is used extensively by another addon. You don’t want to waste unnecessary performance here.

Another optimization that can easily be applied is recycling tables.

Recycling tables is often not worth the effort in Lua. The garbage collector works very efficiently and collecting tables is usually faster than emptying tables and refilling them. But World of Warcraft provides the function table.wipe(t), which quickly deletes all entries from a given table.

We could do the following: create a stack and push all tables that are no longer required on this stack after wiping them. The scheduling function could then try to pop a table from this stack or create a new one if that is not possible. However, there are a few problems:

We have to use a loop to insert all arguments into a recycled table. We are currently using simply {...}, and we earlier saw that this is faster than a loop.

Imagine that there are a lot of timers required for a short period of time. You would then have a lot of dead timer objects on your recycle stack, and it would never shrink back. On possible solution for this would be using a weak table to store all the recycled timers; the garbage collector would then collect all tables that are not recycled. But this comes with another disadvantage, discussed next.

It would not be possible to use a stack as a data structure, because the garbage collector would destroy it by collecting elements in the middle of it. The only solution is using a hash table and next to get the first element from it. But recall that in our hash table implementation, next would simply look for the first spot in the table that is not free, and this might take some time. This is especially problematic as the hash table can become comparatively big if there are a lot of timers started; recall that a hash table can only shrink if you add even more nil elements to it. You will see more about the performance of next on almost empty big tables in just a bit

It would be very hard to track the number of tables that are currently available for recycling, as the garbage collector will delete them in an unpredictable order and point in time. However, it would still be possible to track the number by using a finalizer metamethod, a function that is called when the object is collected. But this creates an even bigger overhead, as objects with finalizers require special treatment. I will tell you more about finalizers in the next section when we discuss userdata values.

The weak hash table solution still seems to be attractive in spite of the disadvantages, as you probably didn’t expect next to be such a performance hog. Let’s see how much worse next can be. The following code creates a simple example hash table with 50,000 entries

Let’s test a simple loop that removes all entries from the table:

It is as fast as one might expect it to be, about 0.03 seconds on my laptop. Now let’s try to use the following code instead. It also deletes all the entries from the table, but it uses a while loop and next(t, nil) to retrieve the first entry of the table until the table is completely empty:

This loop requires 5 seconds of CPU time on my laptop because next has to go over the whole table to find an entry at the end. This takes some time on a huge table with only a few entries, and our hash table that stores recycled tables can become such a table if there are a lot of timers in a short period (the hash table grows) followed by a period with only a few timers (the garbage collector removes most of them).

Recycling tables is in this case not worth doing, because of these disadvantages. You would increase the CPU usage to save a few kilobytes of memory. But your CPU is usually overloaded with work when playing World of Warcraft, while a few kilobytes of memory won’t make any difference.

We now have optimized a lot of tables, and we worked with strings earlier. But there are two more interesting data types we have not discussed at all yet: userdata values and threads Let’s first see how we can use userdata.

I told you in Chapter 2 that you cannot create userdata values from Lua, but this doesn’t mean we can’t use them. Now, what could we do with a userdata object? The most common use of this object is to represent an object that is provided by the Lua host, like World of Warcraft frames. All frames are represented by a table holding a userdata object. But the only thing we can do with this object is to pass it to functions provided by the API that know what this userdata object actually represents. This means we currently can’t do anything with userdata objects except use them as identifiers.

But userdata values can have metatables, and they even define two additional metamethods: \_\_len is called when you use the length operator # ­on the userdata object, and \_\_gc is a finalizer, executed when the userdata object is collected by the garbage collector. But there are still two problems: how can we create a userdata object, and how do we apply a metatable to it? setmetatable does not work on userdata values. There is no way to create a userdata value and apply a metatable to it according to the official documentation.

However, there is an undocumented function in Lua that creates a new userdata value (without any data attached to it) and a metatable attached to it. The name of this function is newproxy(mt), which already suggests its purpose: using it as a proxy.

A proxy object wraps a table and can track, modify, or block accesses to it. You saw a proxy table in Chapter 6, where I showed you a few useful tricks for debugging by using a proxy for the environment of your file. This proxy tracked and forwarded all accesses to the global environment; that is, all global variables.

The first argument mt of newproxy(mt) can be true to create a new, empty metatable for the new userdata value, or false to create it without a metatable. The third possible value for mt is another userdata object that already has a metatable, which will then be used for the new userdata object.

The following example shows a userdata proxy to track accesses to another table:

You might wonder where the advantage over using normal tables is. Well, the only difference is the memory usage. A table allocates 32 bytes of memory that is wasted if the only purpose of the table is forwarding accesses through a metatable. The userdata object does not allocate any unnecessary memory, but the advantage is still very minimal and often not worth the effort.

However, the default UI makes use of newproxy in a similar way in a lot of code related to secure templates and restricted frames. It is used there as a proxy that wraps frames in order to protect them from becoming tainted. The comments in the files suggest that the only reason they use userdata objects instead of tables is memory usage.

There are two additional metamethods available, as mentioned earlier. First, \_\_len is invoked when you use the length operator # on a userdata value. We can use this to create a wrapper for a hash table object that keeps track of the hash table’s size. We will use the metatable of the userdata object as storage for the contents of the hash table. The field \_\_size in the metatable will store the size of the hash table. Here’s the code:

We can test our hash table object by adding the following code

One disadvantage of using such a table is that accessing and setting values in it is slower than in a regular table. That is because you have an additional function call every time you access or change a value.

The second metamethod is the finalizer \_\_gc, which is invoked when the userdata value is deleted by the garbage collector. The reason such a metamethod exists is that a userdata value often stores data that cannot be accessed by Lua, and the garbage collector cannot clean up regular userdata values. A finalizer is usually not a Lua function but a function provided by the underlying API that cleans up the userdata value.

The use of this metamethod is somewhat limited for us, as we can only add Lua functions here. The following example simply prints a message, as I can’t think of a useful application for this metamethod:

There is no need to trigger the garbage collector by calling collectgarbage(“collect”) as the garbage collector also collects all objects with finalizers when the scripts ends. The reason for this behavior is that the userdata objects might have opened system resources like files, and the finalizer closes them. Lua therefore has to ensure that all finalizers are called, even if the script terminates with an error.

There is another feature of Lua we haven’t discussed yet, as you will rarely need it in World of Warcraft: the coroutine library.

This library is one of the Lua standard libraries, and I mentioned it at the beginning of the book. The library allows you to create coroutine objects, which are basically functions with one difference: they can yield and they can be resumed. Yielding in a coroutine is similar to returning in a function, but a coroutine will continue executing at the point where it yielded when it is resumed. A function cannot store the point where it returned and resume there

There can only be one coroutine running at a given time, as coroutines do not implement multithreading. A coroutine can therefore not be stopped from the outside; it has to yield and can then be resumed from the outside.

Let’s get an overview of the available functions that work with coroutines:

coroutine.create(func): Creates a new coroutine object from the given function func. This function is executed when the coroutine is resumed for the first time (that is, when started)

coroutine.resume(co, ...): Resumes the coroutine \_k and passes the arguments ... to it. This function returns true followed by the values passed to the yield function or return statement if the coroutine runs without errors, false followed by the error message otherwise.

coroutine.yield(...): Yields and activates the coroutine that called the currently active coroutine. All arguments passed to this function are returned as additional return values of the coroutine.resume call that resumed the currently active coroutine. yield returns the additional values that are passed to the resume function when the coroutine is resumed the next time.

coroutine.running(): Returns the coroutine that is currently running or nil if no coroutine is running.

coroutine.status(co): Returns the status of the coroutine \_k, which has the following possible values: running means that it is currently running; suspended means that it yielded or was not started yet; normal means that the coroutine is currently stopped because it resumed another coroutine; dead means the function the coroutine was created from has returned or an error occurred.

coroutine.wrap(func): Creates a coroutine from func and returns a wrapper function that calls coroutine.resume on the created coroutine every time the wrapper function is called. This wrapper function strips the first return value (which indicates occurred errors) and generates an error message if this value was false.

It looks quite complicated, so let’s see a simple example.

The following example creates a coroutine that wraps a function which basically consists of an infinite loop. This function yields in the body of the loop and passes a counter to the calling function:

This example was quite simple but you can imagine that coroutines can quickly become very complicated, especially when you have multiple coroutines that resume each other and pass arguments around.

However, it is very unlikely that you are ever going to need such complicated coroutine constructs in a World of Warcraft addon. But simple loops like in this example can be used if you need to execute a longer task and you want to distribute it over multiple frames to avoid a lag spike. You can simply define an OnUpdate handler that checks whether a given coroutine is still alive and executes it in this case. The coroutine then executes a few steps of the task and yields

This chapter was one of the toughest in this book, as it goes really deep into detail. Don’t worry if you didn’t understand every single detail. You might want to read this chapter a second time after you have written your first addon by yourself. You can then apply a few of the tips and tricks I presented here to improve your addon.

You learned how we can measure and improve the performance of our addons with a lot of small tricks, and you also learned a lot of details about how Lua internally works. We discussed in detail how strings and tables work, with examples for optimizing them.

We then improved our library SimpleTimingLib by applying the discussed optimization techniques. The insertion is now an O(n) operation and the OnUpdate handler is O(1); the older version was the other way around. This is better because the OnUpdate handler is called every frame, while the insertion is only called from time to time.

I then showed you two parts of the Lua API we hadn’t discussed earlier. One of them was the function newproxy(), a completely undocumented feature that can be quite useful as it allows you to create a userdata object. The last trick I showed you here was how we can use the coroutine standard library, which is rarely used in World of Warcraft addons. This library can be incredibly useful and powerful, but it is quite hard to use.