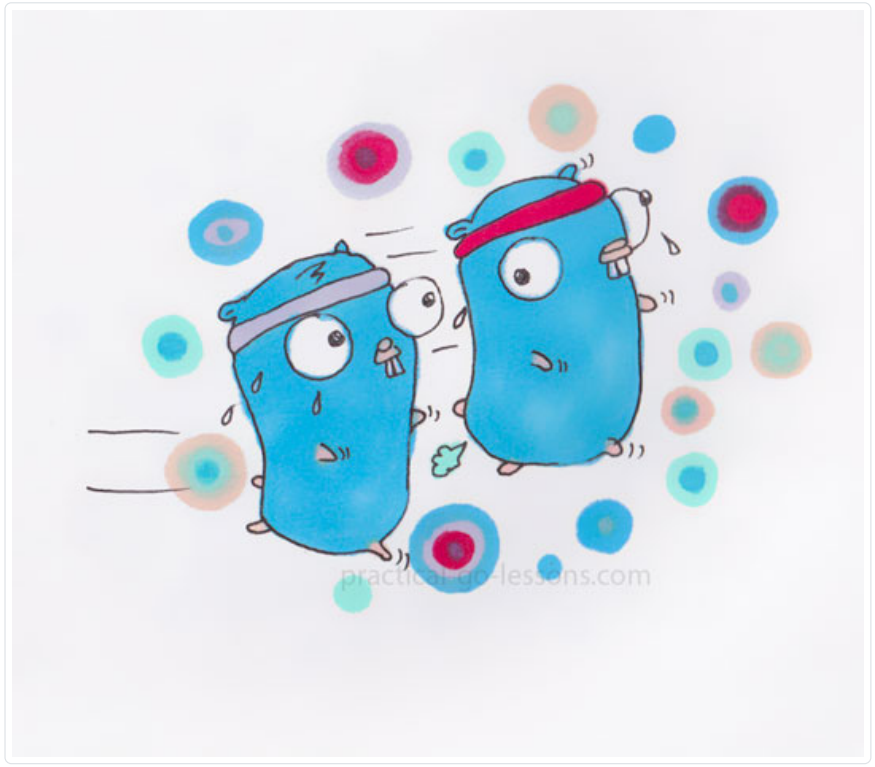
Chapter 34: Benchmarks



1 What will you learn in this chapter?

- What is a benchmark?
- How to write a benchmark.
- How to read the results of a benchmark.

2 Technical concepts covered

- Benchmark
- Solver
- Memory allocations (dynamic and static)

3 Introduction

A problem may have different solutions.

Let’s take an example: you have lost your keys, and you want to open the door of your house. This problem has several solutions. You can :

- Call somebody from your entourage that has a spare key
- Call a locksmith to open your door
- Go back to where you were and look for your keys; if not found in 3 hours, use solution 1 or 2.
- Break a window and enter your house

Those solutions will have the same result; your door will be open. But if you are reasonable, you can rank those solutions in terms of cost or time. Solution 2 and 4 will cost you money. Solution 3 (look for your keys) will probably cost you more time. But what if you just forget the keys in your car parked 5 minutes away? Clearly, in that case, solution three will cost you less than expected.

By examining all the different possible solutions and test them in your imagination, you are making a benchmark.

4 What is a benchmark

A benchmark is a tool to compare systems and components [@institute1990ieee]. The objective of designing and running a benchmark is to find the best solving strategy (called solver).

A solver is usually a method.

To choose the best solver, a rule has to be defined. During the benchmark, execution statistics are gathered (the computation time, the number of affectations, the number of function calls ...). With the help of those statistics, we can choose a decision rule.

There is no such thing as a general rule. Rules might differ depending on your needs;

for instance, if you want to select the program with less CPU usage, you have to focus only on these statistics. If you design a program that runs on devices with very small memory available, you might focus on the memory usage statistics to choose the best solver.

5 How to write a benchmark

We will compare two algorithms to concatenate strings. The first step is to create the two functions that will implement the two solutions :

```
// benchmark/basic/bench.go
package basic

import (
    "bytes"
    "strings"
)

func ConcatenateBuffer(first string, second string) string {
    var buffer bytes.Buffer
    buffer.WriteString(first)
    buffer.WriteString(second)
    return buffer.String()
}

func ConcatenateJoin(first string, second string) string {
    return strings.Join([]string{first, second}, "")
}
```

Both functions concatenate two strings. They use two different methods. The first function `ConcatenateBuffer` will use a buffer (from the `buffer` package). The second function is a wrapper of the function `Join` from the package `strings`. We want to know which approach is the best.

Benchmarks are living next to the unit tests. A benchmark is a function located in a test file. Its name must begin with **Benchmark**. The benchmark functions have the following signature

```
func BenchmarkXXX(b *testing.B) {
}
```

This function takes as parameter a pointer to a type struct `testing.B` . This type struct has only one property exported: `N` . Which represents the number of iterations to run. The benchmark will not just run the function one time but several time to gather reliable data about the execution of the benchmarked function. That's why benchmark functions always encapsulate this kind of for loop :

```
for i := 0; i < b.N; i++ {
    // execute the function here
}
```

You can see that the loop start at 0 and will stop when `b.N` is reached. Do not put a value instead of `b.N` . The benchmark package will run the benchmark once and then decide if it should continue to run it. The value of `N` is adjusted to reach a desirable level of reliability (we will go deep on that later in the chapter). Let's see our two benchmarks :

```
// benchmark/basic/bench_test.go

var result string
func BenchmarkConcatenateBuffer(b *testing.B) {
    var s string
    for i := 0; i < b.N; i++ {
        s = ConcatenateBuffer("test2","test3")
    }
    result = s
}

func BenchmarkConcatenateJoin(b *testing.B) {
    var s string
    for i := 0; i < b.N; i++ {
        s = ConcatenateJoin("test2","test3")
    }
    result = s
}
```

We first create a `result` variable. This variable is just here to avoid compiler optimization (a tip given by Dave Cheney in a blog post: https://dave.cheney.net /2013/06/30/how-to-write-benchmarks-in-go). We will save the results of our benchmarks in this variable.

Then we define our two benchmark functions `BenchmarkConcatenateBuffer` and

`BenchmarkConcatenateJoin`. Note that they have very similar constructions. The concatenation result is stored into a variable `s`. Then we define a for loop, and inside it, we are executing the function we want to bench.

The arguments are fixed; we test the function under the same conditions.

6 How to run benchmarks

To run benchmarks, we use the same go test command :

```
$ go test -bench=.
```

This command will output :

```
goos: darwin
goarch: amd64
pkg: go_book/benchmark
BenchmarkConcatenateBuffer-8    20000000                98.9 ns/op
BenchmarkConcatenateJoin-8      30000000                56.1 ns/op
PASS
ok      go_book/benchmark       3.833s
```

We will see in the next section how to interpret the test results.

The previous command will run all the benchmarks of the package.

6.1 Run only one benchmark

To run only the ConcatenateBuffer benchmark, you can use the following command :

```
$ go test -bench ConcatenateBuffer
```

The previous command is a shorthand for :

```
$ go test -test.bench ConcatenateBuffer
```

6.2 Run with code (without CLI)

The testing package exposes public methods to run a benchmark. Let's take an example :

```
// benchmark/without-cli/main.go
package main

import (
    "bytes"
    "fmt"
    "testing"
)

func main() {
    res := testing.Benchmark(BenchmarkConcatenateBuffer)
    fmt.Printf("Memory allocations : %d \n", res.MemAllocs)
    fmt.Printf("Number of bytes allocated: %d \n", res.Bytes)
    fmt.Printf("Number of run: %d \n", res.N)
    fmt.Printf("Time taken: %s \n", res.T)
}

// ..
func BenchmarkConcatenateBuffer(b *testing.B) {
    //..
}
```

The function `testing.Benchmark` is waiting for a valid benchmark function, ie. a variable of type `func(b *testing.B)`. Remember that in Go functions are the first-class citizens and can be passed to other functions.

The Benchmark function returns a variable of type BenchmarkResult :

```
// standard library
// src/testing/benchmark.go (v1.11.4)
type BenchmarkResult struct {
    N         int           // The number of iterations.
    T         time.Duration // The total time taken.
    Bytes     int64         // Bytes processed in one iteration.
    MemAllocs uint64        // The total number of memory allocations.
    MemBytes  uint64        // The total number of bytes allocated.
}
```

7 Benchmark flags

-cpu

Benchmarks are executed by default with GOMAXPROCS processors. To have a reliable benchmark, I suggest you control this value; it should be equal to the number

of processors of the targeted machine.

You must pass a regular expression to this flag. It will launch the benchmark functions which names match the regular expression.

For instance, the command :

```
$ go test -bench .
```

Will launch **all** the benchmarks.

```
$ go test -bench Join
```

Will launch all the benchmark functions that contain the string `"Join"`. In the example `BenchmarkConcatenateJoin` will be launched but not `BenchmarkConcatenateBuffer`

-benchtime
This flag allows you to control your benchmarks' execution time. You have to pass a duration string (ex: `3s`). The system will parse the duration and execute benchmarks for the specified amount of time. It means that you can increase/decrease the time that the benchmark will take

Example : Let's run the benchmark named `BenchmarkConcatenateJoin` for 5 seconds :

```
$ go test -bench BenchmarkConcatenateJoin -benchtime 5s
goos: darwin
goarch: amd64
pkg: go_book/benchmark
BenchmarkConcatenateJoin-8      100000000               56.9 ns/op
PASS
ok      go_book/benchmark       5.760s
```

Will display in the result the memory allocation statistics. This flag is boolean; it's set to `false` by default. Just add it to the command line to activate this feature.

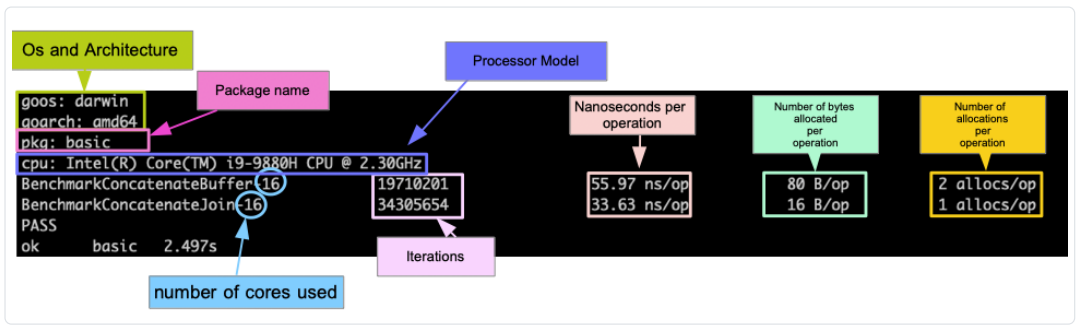
Example: We can run benchmarks with memory statistics with the following command :

```
$ go test -bench . -benchmem
goos: darwin
goarch: amd64
pkg: go_book/benchmark
BenchmarkConcatenateBuffer-8    20000000               105 ns/op
128 B/op          2 allocs/op
BenchmarkConcatenateJoin-8      30000000                60.2 ns/op
16 B/op           1 allocs/op
PASS
ok      go_book/benchmark       4.093s
```

Note that two additional columns are printed into the benchmark results. In the next section, we will see how to read those stats.

8 How to read benchmark results

I find that benchmarks results are difficult to read. We will go through each kind of statistic. For each statistic, we will try to give actionable advice...



Benchmark results output[fig:Benchmark-results-output]

In the figure 1 you can see the standard output of the following command :

```
$ go test -bench . -benchmem
```

Here we are running all the benchmarks of the current packages with memory statistics. The benchmark result contains the following statistics :

- The first elements to print in the benchmark result are the two **Go env variables** `GOOS` **and** `GOARCH`. You know them already, but they are useful to compare benchmark results.

- **Duration** : This is the total time taken to execute the benchmarks
- **The number of iterations (second column)** : Remember that inside every benchmark function, there is a for loop. This number represents the number of time the for loop has run to obtain the statistics. You can increase the number of iterations by using the -benchtime flag to increase the benchmark duration. It's not the total number of iteration executed by the benchmark.
- **Nanoseconds per operation (third column)** : it gives you an idea of how fast on **average** your solver run. In our example the `ConcatenateBuffer` function takes on **average** 55.97 nanoseconds to run. Whereas the `ConcatenateJoin` function takes on average 33.63 nanoseconds to run. The fastest function is `ConcatenateJoin` in the context of our benchmark.
- **Number of cores (appended to the name of the benchmark function)** a benchmark result is relative to the system that runs it. That's why it's important to know how many cores are used to run it. In our case, the benchmark is run with eight cores. You can adapt the number of cores to use for running the benchmark by using the flag -cpu. By default, it takes the maximum number of cores available.
- **Number of bytes allocated per operation (fourth column)** : This column is present only if you add the flag **-benchmem.** This will give you an idea about the memory consumption of your solvers. If your focus is to improve memory usage, then you should focus on this statistics.
- **Number of allocations per operation (fifth column)**: the name of this stat speaks for itself. This is the average number of memory allocations per run. In the section [sec:Detect-memory-allocations] we will see how to detect memory allocation to improve your code.

9 Detect memory allocations[sec:Detect-memory-allocations]

Go has a debug mode that allows you to print numerous and highly valuable information about your program's performance. Memory allocation is an important variable to understand how a program performs. They are roughly two types of memory allocations :

- **Static** : memory is allocated when the program is started. In C, it happens when you create a global or a static variable. This memory is freed when the program stops. It's only allocated one time
- **Dynamic** : In a program, everything is not known when the program is compiled or start. The behavior of the program can vary in function of the user input for instance. Imagine a program that computes highly complex mathematical operations, the memory needed by the program will depend on the input and can vary drastically (making an addition do not require a lot of memory whereas getting the result of !10000 requires a lot more space). This is why programs need to allocate memory dynamically when they run.

We will focus on dynamic memory allocation. We will use the variable `GODEBUG` to output the memory allocations that are done by our two functions.

The first thing to do is to create a sample application that will call our two functions :

```
package main

// inports

func main() {
    basic.ConcatenateBuffer("first","second")
    basic.ConcatenateJoin("first","second")
}
```

This application will call our two functions (that are part of the package `basic` with import path `go_book/benchmark/basic` **).** Then we compile our program :

```
$ go build -o allocDetect main.go
```

Note that the `-o` flag is used to give a specific name to our binary. Here we choose to name is `allocDetect` you could have named it something else, of course.

Then we can run our binary with the `GODEBUG` variable set :

```
$ GODEBUG=allocfreetrace=1 ./allocDetect &>> trace.log
```

`GODEBUG` is an environment variable that accepts a list of key-value pairs. Here we tell the go runtime to generate a stack trace for each allocation and free. Then we add `"&>> trace.log"` to redirect both the standard output and the standard error to the

file trace.log. It will create this file if it does not exist, and if it exists, logs will be appended to it.

Inside our trace.log I have 1034 lines of text, consisting of stack traces. How to exploit them? If we refer to the documentation, each program's memory allocation will generate a stack trace.

We can search by hand into this file to see where the allocation append. But we can use the two commands `cat` and `grep` :

```
$ cat trace.log | grep -n /path/to/the/package/basic/bench.go
```

Here we are first printing the content of the trace.log file with "cat trace.log", then we are asking `grep` to search into this file for the string `"/path/to/the/package/basic/bench.go"` (the string `"/path/to/the/package/basic/bench.go"` needs to be changed by the path of the package file that you want to analyze)

There is a pipe (`|`) between the two commands `cat trace.log` and `grep -n /path/to/the/package/basic/bench.go` . The pipe is used to chain commands. The output of the first command is the the input of the second command, the whole command is forming a pipeline.

Here is the output :

```
988:    /path/to/the/package/basic/bench.go:9 +0x31 fp=0xc000044758
sp=0xc000044710 pc=0x1055c81
1005:   /path/to/the/package/basic/bench.go:12 +0xca fp=0xc000044758
sp=0xc000044710 pc=0x1055d1a
1028:   /path/to/the/package/basic/bench.go:16 +0x7e fp=0xc00008af58
sp=0xc00008aef0 pc=0x1055dde
```

The path has been found three times in the trace.log on lines 988, 1005, and 1028 (the line numbers are returned by grep because we added the flag -n). Just next to the path string, you have the line number that caused allocation in /path/to/the/package/basic /bench.go.

The next set is to analyze your code to see where memory allocation happens and how you can avoid it. In the ConcatenateBuffer function, line two caused a memory allocation. The creation of the buffer :

```
var buffer bytes.Buffer
```

And the call to the String method :

```
buffer.String()
```

The complete list of debug options is available here: https://golang.org/pkg/runtime /#hdr-Environment_Variables

10 Benchmark with variable input

In the previous sections, we have written benchmarks where the input remains stable. This approach is sufficient for most use cases. But you might need to understand how your function behaves when its arguments change.

We will use the method `Run` defined in the `testing` package. The receiver of this method is a pointer to a `testing.B` variable.

If we want to go deeper in the analysis, we can test our two functions with variable-length strings. We will use length that are powers of two :

- 2
- 16
- 128
- 1024
- 8192
- 65536
- 524288
- 4194304
- 16777216
- 134217728

The first step is to put those integers into a slice named **lengths.**

```
lengths :=
[]int{2,16,128,1024,8192,65536,524288,4194304,16777216,134217728}
```

With a for range loop we iterate over those numbers. At each iteration, we create two random strings.

```
for _, l := range lengths {
    first := generateRandomString(l)
    second := generateRandomString(l)

}
```

Once those two strings are created, we can use them as input for our two benchmarked functions.

We will create two sub benchmarks. Sub benchmarks are defined with the help of the Run method. They must be defined into a classical benchmark function. We will name this wrapping function “BenchmarkConcatenation” :

```
// benchmark/variable-input/bench_test.go

func BenchmarkConcatenation(b *testing.B){
    var s string
    lengths :=
[]int{2,16,128,1024,8192,65536,524288,4194304,16777216,134217728}
    for _, l := range lengths {
        first := generateRandomString(l)
        second := generateRandomString(l)

    }
}
```

Inside the for loop, we will call the `b.Run` method twice (`b.Run` will create a sub-benchmark). First, we benchmark the `ConcatenateJoin` function :

```
b.Run(fmt.Sprintf("ConcatenateJoin-%d",l), func(b *testing.B) {
    for i := 0; i < b.N; i++ {
        s = ConcatenateJoin(first, second)
    }
    result = s
})
```

And the second time with ConcatenateBuffer :

```
b.Run(fmt.Sprintf("ConcatenateBuffer-%d",l), func(b *testing.B) {
    for i := 0; i < b.N; i++ {
        s = ConcatenateBuffer(first, second)
    }
    result = s
})
```

Note that the run function takes two arguments :

- A **name**, that will be displayed in the benchmark results
- A **function** which represents the sub benchmark. It must take as argument a pointer to a testing.B variable.

We customize the name of the benchmark. We append at the end of the name the value of `l` (which represents the number of characters of the two concatenated strings). This customization is necessary to improve the readability of the results.

The second argument is a very classical benchmark function: a for loop that will iterate from `1` to `b.N` . Inside this, for loop, you finally find the call to the benchmarked function. We save the result of the function to avoid compiler optimization.

To run this benchmark, you can use the same command as before :

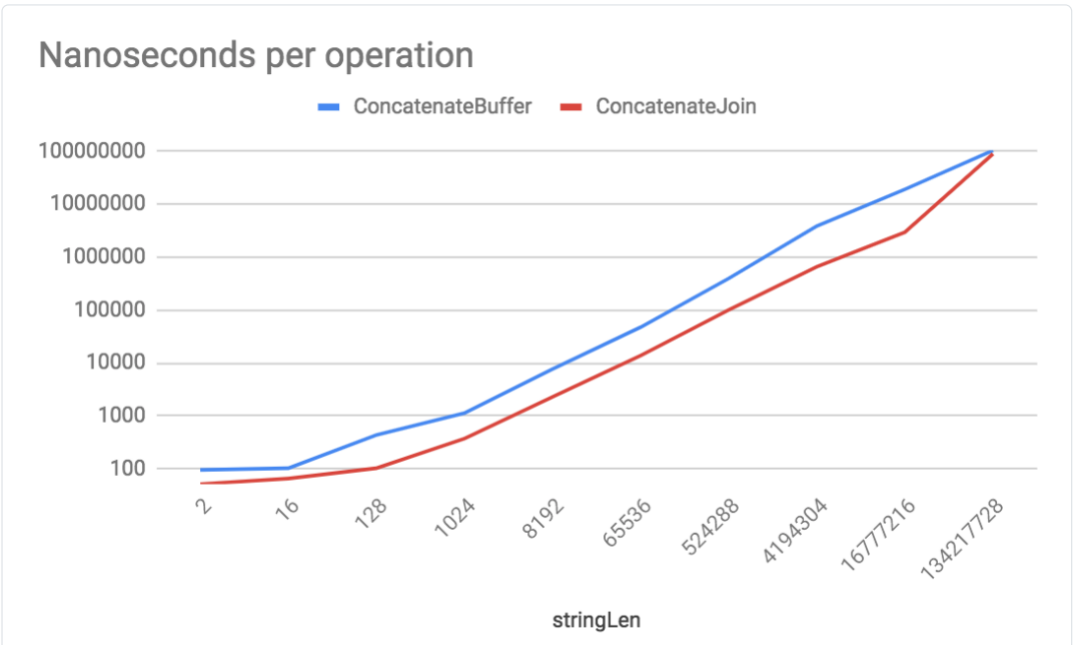
```
$ go test -bench BenchmarkConcatenation -benchmem
goos: darwin
goarch: amd64
pkg: go_book/benchmark/variableInput
BenchmarkConcatenation/ConcatenateJoin-2-8              30000000
51.2 ns/op             4 B/op          1 allocs/op
BenchmarkConcatenation/ConcatenateBuffer-2-8            20000000
93.0 ns/op           116 B/op          2 allocs/op
BenchmarkConcatenation/ConcatenateJoin-16-8             20000000
62.5 ns/op            32 B/op          1 allocs/op
BenchmarkConcatenation/ConcatenateBuffer-16-8           20000000
103 ns/op            144 B/op          2 allocs/op
//...
ok      go_book/benchmark/variableInput 33.975s
```

This is a partial output. I did not copy all the standard output.

We can generate a graph from this data to understand the results better. We will redirect the output to a file for further processing :

```
$ go test -bench BenchmarkConcatenation -benchmem &>>
benchmarkConcatenation.log
```

Then we can parse the benchmarkConcatenation.log file to generate a table and draw a graph :



Mean execution time (ns/op) in function of string length (log-lin plot)[fig:Mean-execution-time-log]

10.0.0.1 Logarithmic scale

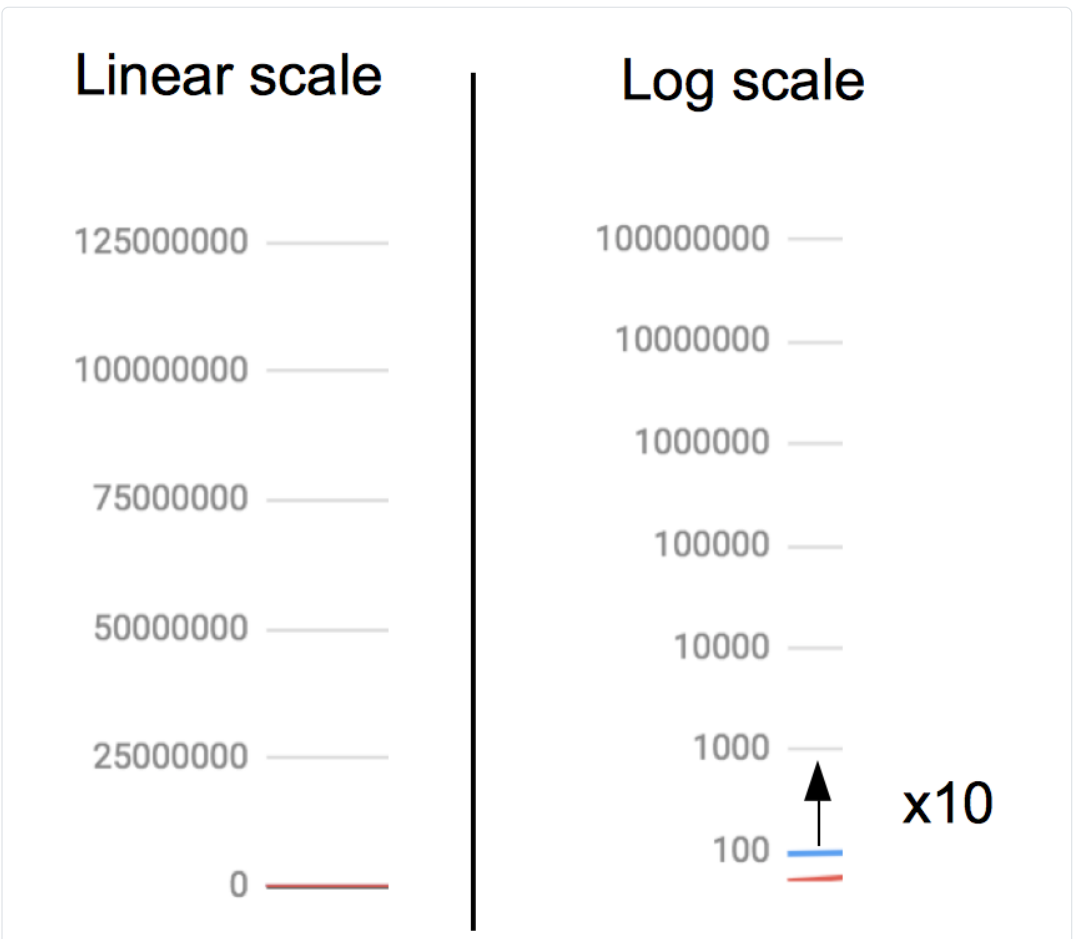
The figure 2 represented the data on a log-lin plot. A log-lin plot is a graph on which the vertical axis is logarithmic and the horizontal has a logarithmic scale. You might be unfamiliar with that approach (if you know it already, you can skip this section).

A logarithmic scale is used when the data range is big. In statistics, the range is the difference between the largest and the smallest value. For the dataset :

```
{2,16,128,1024,8192,65536,524288,4194304,16777216,134217728}
```

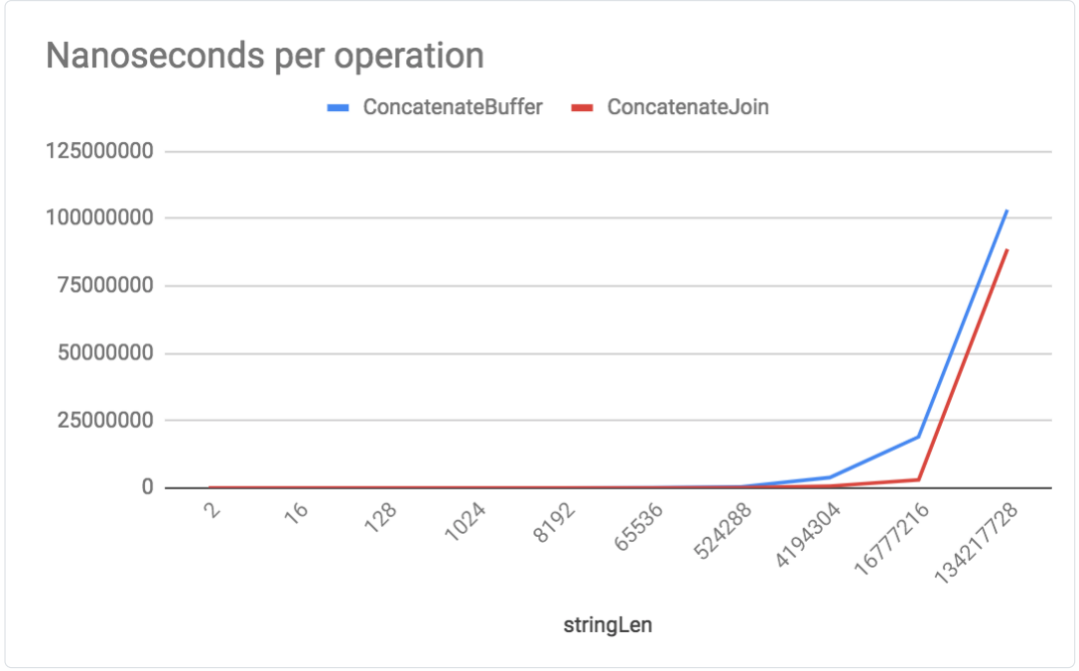
the range is $134217728 - 2 = 134217726$. This is big.

In that case, it's recommended to use a logarithmic scale and not a linear scale. Instead of having a scale where one millimeter always represent the same value, with a logarithmic scale, the value of a mark on a scale is equal to the the previous mark multiplied by a constant. This constant can vary, but it's usually 10. The figure 3 shows the difference between the log and the linear scale.



Linear scale vs. logarithmic scale[fig:Linear-scale-vs.log]

One axis can have a log scale and the other a linear scale. This type of graph is called "log-lin plots". If both axes have a logarithmic scale, it's called a log-log plot. Compare the figure 4 and the figure 2. Which plot is better?



Lin-Lin plot[fig:Lin-Lin-plot]

10.0.0.2 Parse the benchmark results

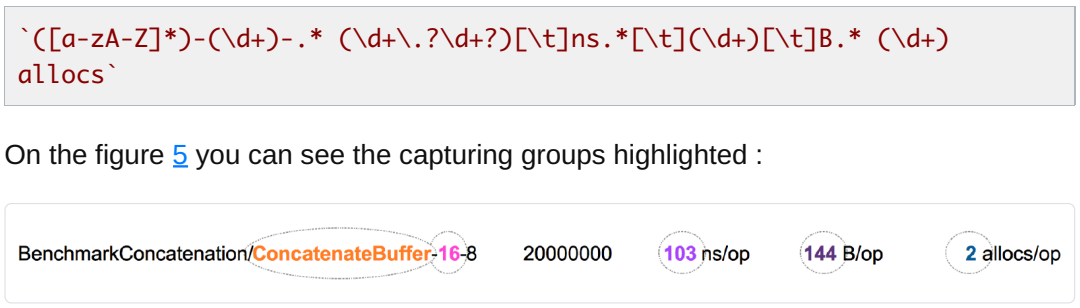
Unfortunately, Go has no internal tool to generate this kind of graph. I had to manually parse the standard benchmark output to get the data. Here is the script I used :

```
package main

import (
    "fmt"
    "io/ioutil"
    "regexp"
)

func main() {
    b, err := ioutil.ReadFile("/path/to/benchmarkConcatenation.log")
    if err!= nil {
        panic(err)
    }
    benchmarkResult := string(b)
    regexBench := regexp.MustCompile(`([a-zA-Z]*)-(\d+)-.* (\d+\.?
\d+?)[\t]ns.*[\t](\d+)[\t]B.* (\d+) allocs`)
    matches := regexBench.FindAllStringSubmatch(benchmarkResult,-1)

fmt.Println("benchmarkedFunction,stringLen,nsPerOp,bytesPerOp,mallocs")
    for _, m := range matches {
        fmt.Printf("%s,%s,%s,%s,%s\n",m[1],m[2],m[3],m[4],m[5])
    }
}
```

I used the following regular expression with five capturing groups to retrieve the benchmark data :

```
`([a-zA-Z]*)-(\d+)-.* (\d+\.?\d+?)[\t]ns.*[\t](\d+)[\t]B.* (\d+)
allocs`
```

On the figure 5 you can see the capturing groups highlighted :



Regex capturing groups highlighted[fig:Regex-capturing-groups]

- The first group captures the name of the function benchmarked (which is stored into `m[1]`)
- The second group captures the length of the string (`m[2]`)
- The third group captures the nanoseconds per operations (`m[3]`)
- The fourth is the number of bytes in memory per operation (`m[4]`)
- The final group represent the number of allocations (`m[5]`)

The variable matches is a two dimensional slice of strings : `[][]string` . `matches[0]` represent the first benchmark and `matches[0][1]` the name of the function benchmarked.

10.0.0.3 Bits of Advice

- Take into consideration the time variable (ns/op) and the memory usage metrics.
- Choose the variable (or the mix of variables) that is (are) coherent with your objectives.
- Use the logarithmic scale on your graphs when appropriate (large range of data)

11 Common error: b.N as argument

The time taken by the benchmarked function should not increase when the value of b.N increase. Your function's input should not depend on the b.N number. Otherwise, your benchmark results will not be significant.

Let's take an example :

```
func BenchmarkConcatenateBuffer(b *testing.B) {
    var s string
    for i := 0; i < b.N; i++ {
        s =
ConcatenateBuffer(generateRandomString(b.N),generateRandomString(b.N))
    }
    result = s
}
```

Here we have modified the input of ConcatenateBuffer. Instead of two fixed string, we use a random string generator named generateRandomString. This function will generate a pseudo-random string with the help of the math/rand package.

Let's see the result of our benchmark :

```
BenchmarkConcatenateBuffer-8   30000        138583 ns/op      319600
B/op    8 allocs/op
```

The final number of operations is only 30.000, and it takes an average of 138,583 nanoseconds per operation.

Those results are very different from the one we collected with two fixed strings: 100 nanoseconds per operation.

12 Test yourself

12.1 Questions

1. What is the standard header of a benchmark function (name and signature)?
2. Where are benchmark functions located in the source code?
3. What is the command to use to run a specific benchmark?
4. Which flag can you use to display memory allocation statistics?
5. True or False ? The statistics ns/op is the function's total time to execute during the benchmark run.
6. How to create a benchmark with variable input?

12.2 Answers

1. What is the standard header of a benchmark function (name and signature)?
    1. func BenchmarkNameHere(b *testing.B)
2. Where are benchmark functions located in the source code?
    1. They are placed next to unit tests.
3. What is the command to use to run a specific benchmark?
    1. If you have somewhere in your code the benchmark function `func BenchmarkConcatenateBuffer(b *testing.B)`
    2. go test -bench ConcatenateBuffer
    3. the string `"ConcatenateBuffer"` is a regular expression
4. Which flag can you use to display memory allocation statistics?
    1. go test -bench . -benchmem
5. True or False ? The statistics ns/op is the function's total time to execute during the benchmark run.
    1. False
    2. This is the **average** time taken by the function.
6. How to create a benchmark with variable input?
    1. Inside your benchmark function, you can create "sub-benchmarks" with `b.Run`

13 Key Takeaways

- The objective of designing and running a benchmark is to find the best solving strategy (called solver).
- The term “best” should be adapted for your needs.
  - Do you want the fastest solver?
  - Do you want the solver that has the lowest memory footprint?
  - A combination of both?
- To create a benchmark, write a function with the following header : `func BenchmarkNameHere(b *testing.B)`
- Benchmark functions are placed in unit test files. Here is an example benchmark :

```
func BenchmarkConcatenateJoin(b *testing.B) {
    var s string
    for i := 0; i < b.N; i++ {
        s = ConcatenateJoin("test2", "test3")
    }
    result = s
}
```

- To run all benchmarks in a module, use the command : `$ go test -bench=.`
- To run a specific benchmark, use this command : `$ go test -bench ConcatenateBuffer`
- To display memory statistics, add the flag “benchmem” : `go test -bench . -benchmem`
- With memory statistics set to ON, you can get the number of bytes allocated per operation and the number of allocations per operation.
- The env variable `GODEBUG` allows you to debug program runtime (listing memory allocations, for instance)
- A benchmark function can have sub-benchmarks. This is practical to bench the function against different inputs :

```
func BenchmarkConcatenation(b *testing.B) {
    var s string
    lengths := []int{2, 16, 128, 1024, 8192, 65536, 524288, 4194304,
16777216, 134217728}
    for _, l := range lengths {
        first := generateRandomString(l)
        second := generateRandomString(l)
        b.Run(fmt.Sprintf("ConcatenateJoin-%d", l), func(b *testing.B)
{
            for i := 0; i < b.N; i++ {
                s = ConcatenateJoin(first, second)
            }
            result = s
        })
        b.Run(fmt.Sprintf("ConcatenateBuffer-%d", l), func(b
*testing.B) {
            for i := 0; i < b.N; i++ {
                s = ConcatenateBuffer(first, second)
            }
            result = s
        })
    }
}
```

Bibliography

- [institute1990ieee] Electrical, Institute of, and Electronics Engineers. 1990. “IEEE Standard Glossary of Software Engineering Terminology: Approved September 28, 1990, IEEE Standards Board.” In. Inst. of Electrical; Electronics Engineers.

Previous

Application Configuration

Next

Build an HTTP Client

Table of contents