Web Scraping at Scale: Python vs Go

4th Aug, 2015

BY DATAWEAVE

Scaling is a common challenge when you have to deal with a lot of data everyday. This is one thing we face at **DataWeave** a lot and have managed to tackle it fairly successfully. Often there is no one best solution. You have to keep looking and testing to find out what suits you the best. One thing we do a lot in DW, is crawl and extract data from HTML pages to get meaningful information out of them. And by a lot, I mean in millions. Speed becomes an important factor here. For instance, if your script is taking one tenth of a second to process a single HTML page, it is not good enough as this will turn out to be a huge bottleneck when you are trying to scale your system.

We use Python to take care of all our day to day operations, and things are going smooth. But as I mentioned, there is no best solution, and we need to keep exploring. Thanks to Murthy, I became curious in **Go**. I knew about Go for a long time, probably since it came out but I never paid much attention to it. Last weekend, I started playing around with Go and liked it immediately. It seemed like a perfect marriage between C and Python (probably C++ too). Then I decided to do a small experiment. I wanted to see how Go performs against Python for scraping webpages at scale. The Regular Expression package of Go is as good as Python, so I didn't face much problem building a basic parser. The task of the parser is: I will feed it an apparel product webpage from an Indian online shopping website and it will give me the 'Title', 'MRP' (price), and 'Fabric' information of the apparel and the product 'Thumbnail URL'.The parser function looked something like this:

```
func parse(filename string) {
    bs, err := ioutil.ReadFile(filename)
    if err != nil {
        fmt.Println(err)
        return
    }

    data := string(bs)

    var title = regexp.MustCompile(`(?i)(?s)<div class="[^<]*?prd-brand-detail">(.*?)<
div class="mid-row mb5 full-width"`)
    var mrp = regexp.MustCompile(`(?i)(?s)\{"simple_price":(.*?),"simple_special_pric
e":(.*?)\}`)
    var thumbnail = regexp.MustCompile(`(?i)(?s)<meta property="og:image" content="(.
*?)"`)
    var fabric = regexp.MustCompile(`(?i)(?s)<td>Fabric</td>\s*<td[^>]*?>(.*?)</td>`)

    titleString := HTML(title.FindStringSubmatch(data)[1])
    fabricString := fabric.FindStringSubmatch(data)[1]
    mrpString := mrp.FindStringSubmatch(data)[1]
    thumbnailString := thumbnail.FindStringSubmatch(data)[1]
    fmt.Println(filename, titleString, fabricString, mrpString, thumbnailString)

}
```

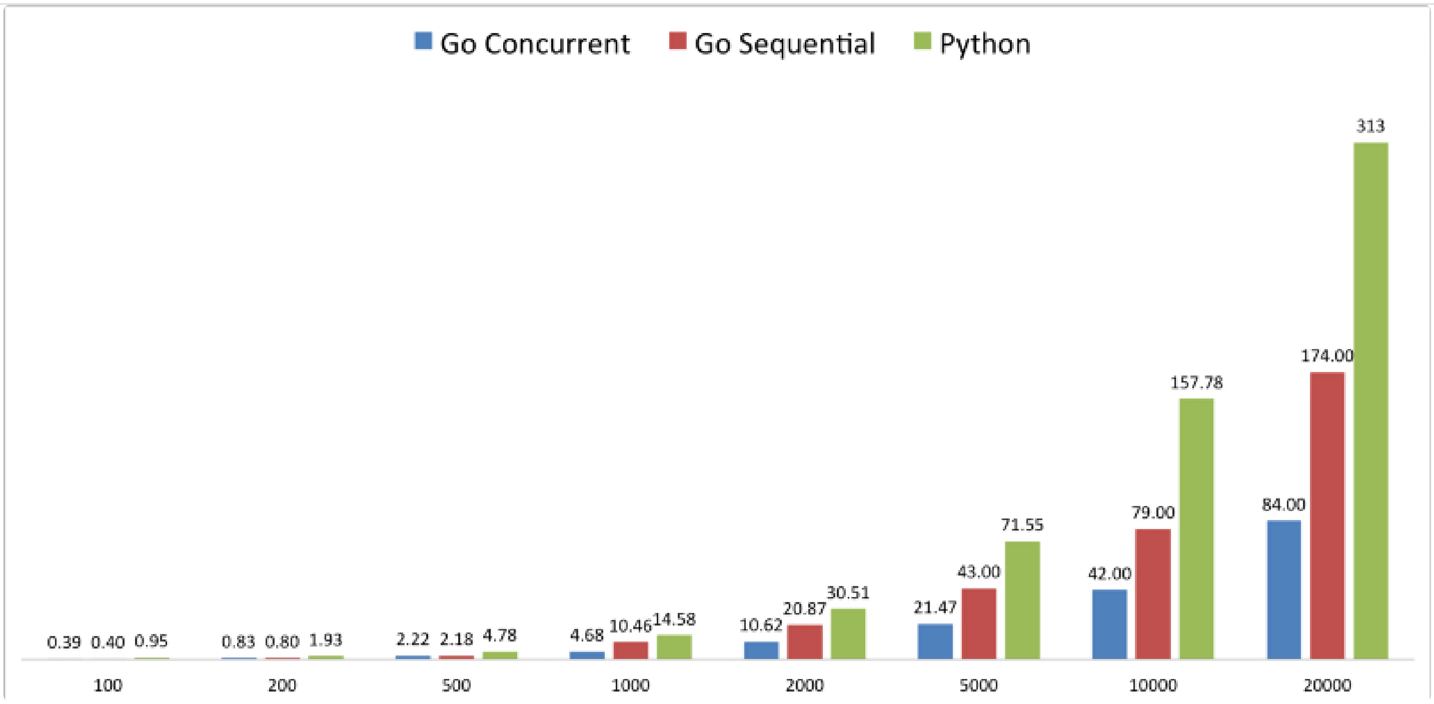
```
def parse(filename):
htmlfile = open(filename).read()

titleregex = re.compile('''<div class="[^<]*?prd-brand-detail">(.*?)<div class="mid-ro
w mb5 full-width"''', re.S|re.I)
fabricregex = re.compile('''\{"simple_price":(.*?),"simple_special_price":(.*?)\}''',
re.S|re.I)
mrpregex = re.compile('''<meta property="og:image" content="(.*?)"''', re.S|re.I)
thumbnailregex = re.compile('''Fabric</td>\s*<td[^>]*?>(.*?)</td>''', re.S|re.I)


result = re.search(titleregex, htmlfile)
title = result.group(1)

result = re.search(fabricregex, htmlfile)
fabric = result.group(1)

result = re.search(mrpregex, htmlfile)
mrp = result.group(1)

result = re.search(thumbnailregex, htmlfile)
thumbnail = result.group(1)
print filename, title, fabric, mrp, thumbnail
```

Nothing fancy, just 4 simple regexes. My testbed is a Mid-2012 MacBook Pro with 2.5 GHz i5 processor and 4 gigs of RAM. Now, I had two options. First, run the script with 100 HTML files sequentially and monitor the running time. Or, I could create 100 goroutines and give each of them a single HTML file and let them handle its own parsing using Go's built in concurrency feature. I decided to do both. Also, I made a Python script which does a similar thing, to compare its performance against the Go script. Each experiment was performed 3–4 times and the best running time was taken into account. Following is the result I received:

```
# of Files    Go Concurrent   Go Sequential   Python
 100           0.39            0.40            0.95
 200           0.83            0.80            1.93
 500           2.22            2.18            4.78
 1000          4.68            10.46           14.58
 2000          10.62           20.87           30.51
 5000          21.47           43              71.55
 10000         42              79              157.78
 20000         84              174             313
```

It is obvious that using Go beats Python in all the experiments and more than 2x speed in every cases. What is interesting, using concurrency results in much faster performance than processing the HTML files sequentially. Possibly, deploying 500–1000 goroutines together actually speeds up the execution process as each goroutine can work without coming in each other's way. However, I was able to deploy maximum 5000 goroutines at once. My machine could not handle more than that. I believe using powerful a CPU will let you process the files faster than what I have done. This begs for more experiments and benchmarking, something that I absolutely love to do! The following is a graphical view of the same experiment:

If the differences are not visible properly, perhaps this one will be helpful:



If you are not satisfied yet, keep reading!

I was able to get Murthy intrigued into this. He started playing with the Python code and kept optimizing it. He felt that compiling regexes every time the loop is executed adds overheads and takes unnecessary time. Globally compiled and initialized regexes will be a better option. He tried out two experiments, one with plain CPython and the other with PyPy which has JIT compiler for Python. Following is the result he observed. He tested this on a CentOS server with Python 2.6 running, 8 cores and 16 gigs of RAM.

```
1000 HTML files                                        Time
 CPython (without fn calls, global regexes)             16.091
 CPython (with fn calls, global regexes)                20.483
 CPython (with fn calls, global regexes as fn params)   16.898
 CPython (with fn calls, local regexes)                 17.582
 PyPy (with fn calls, global regexes as fn params)      4.510
 PyPy (without fn calls)                                3.567
 CPython + re2 (with fn calls,global regexes)           1.020
 CPython + re2 (without fn calls)                       0.946
```

Not bad at all! PyPy does it under 4 seconds. Although the CPython result is same as mine. Go provides a runtime parameter GOMAXPROCS which can be passed as a parameter while running the go code. The variable sets and limits the number of OS threads that can execute user-level Go code simultaneously. I believe that by default the limit is set as 1. But to parallelize the process using multiple threads I set it as 4. This is the result I got.

```
# of HTML Files    Go Concurrent 1 Thread    Go Concurrent 4 Threads
 100                0.397                              0.196
```

```
200                 0.827                          0.396
500                 2.22                           0.988
1000                4.68                           1.98
2000                8.27                           4.03
```

So, 1000 HTML files under 2 seconds and 2000 files in around 4 seconds. As you can observe from the results, using multiple threads gives you 2x more gain than the previous result, more than 4x gain over sequential processing result and more than 7x gain from CPython result. However, I was not able to run 5000 goroutines concurrently as my OS did not permit. Still, if you can run your job in batches, the advantages will be good enough. *Google if you are reading this, can you invite me over to your Mountain View office so that I can run this on your 16000 core machine?*

- DataWeave Marketing

4th Aug, 2015

DATA ENGINEERING