Scalability Project 2

Thor Olesen and Stig Killendahl

November 2017

1 Introduction

In Assignment 1, a webservice was designed and implemented to return links to granules in the Sentinel2 dataset, that match to user-provided latitude and longitude values. Thus, this has now been adjusted in the webservice to return links/ids of images in the Sentinel2 dataset.

1.1 Color Feature

We been asked to add color as another dimension besides location, but have deemed it very inconvenient to implement in practice. Namely, you have to breach from using pure Go code to handle jpeg2 sentinel-2 images that are not supported natively in Go (only jpeg). However, a hypothetical model for the color functionality may be devised that describes how the service could potentially enable querying by color, under the assumption that we have a library to process jpeg2 images in Go.

Querying by longitude and latitude is already handled in the current implementation, so the only real work is in ranking the images based on color. The assignment introduces three different ranking factors, which are all based on processing the images and outputting RGB values. If we had to rank by one color band, we assume that we would have to look at each pixel in the picture, get the RBG values and in that way, calculate the average R, G and B value for the entire picture. Once we have the average RGB values, the images may be sorted based on either of the three factors, as described in the assignment.

Arguably, sequential processing of the 1830x1830 pixel pictures would be slow. Thus, parallel processing could speed up the processing. This would be a complexity vs speed trade-off. Another trade-off would be precision vs speed. If speed is more important, than precision, one could process only every 5th, 10th or 15th pixel, and in that way speed up the process significantly. A third trade-off is space vs speed. If the same images are queried a lot, one could pre-process those images and store the result. This could make sense if the service was used a lot for the same geographical areas. These are all decisions that should be taken into account based on what is important when designing the solution.

1.2 Geographical Location Feature (Scaling spatially)

The requirement is to support querying images located in a geographical area of interest, specified by two latitude and longitude bands. This service is provided in the "service.go" file that has a new handler function. Namely, the "area" function that, provided two pairs of latitude and longitude coordinates, will fetch all image ids within the area they cover (i.e. area of interest). This is done by fetching a link to all the relevant granules and image folders via the BigQuery API in "query.go". This is used in combination with the Google Cloud Bucket API in "query.go" to fetch all the images that reside in the bucket image folders. Finally, all the image links within the area of interest returned as a JSON array to the client in the area handler function in "service.go".

The images can be downloaded with the gsutil tool, using the copy "cp" command and prepending "gs://" to the image path: gsutil cp "gs://**imagePath**. Image form: "gcp-public-data-sentinel-2/**base_URL**/GRANULE/**granule_id/img.jp2**.

@ https://tvao-178408.appspot.com/area?lat1=55lng1=12lat2=55lng2=12

1.3 Profiling (Benchmarks)

The webservice solution has been benchmarked in terms of response time, memory usage and scalability, using the built-in benchmarking tools in Go and the HTTP benchmarking tool "go-wrk". The benchmark queries cover different request use-cases and are located in "benchmarks_test.go", and the response time has been benchmarked with the http web server performance tool "go-wrk", yielding the results shown in section 1.3.1, where an area of interest with 1563 image links is queried on 1, 10 and 32 client connections respectively.

Mind you, the timeout flag has been set to 100 seconds to allow the roundtrip of request responses to complete. Further, when serving multiple HTTP client connections (as indicated by the -c concurrent goroutines flag), a point above 32 goroutines is reached, where both the tool and web service fails with messages like: "2017/11/09 12:23:08 http: panic serving 127.0.0.1:50852: runtime error: invalid memory address or nil pointer dereference" and "AttributeError: 'module' object has no attribute 'apiproxy'" in handle_request. This could be due to a safety measure in OAuth that is used during each call for authentication with the BigQuery and Storage service API .

However, it can be concluded from the response test results that around 1500 image links take 1 minute to query from the web service. These response time results have been complemented with Go's profiling tools to identify and correct any potential bottlenecks that may improve the performance further.

1.3.1 Response Time Test Results

Response time when fetching 1500 images links on one goroutine:

```
$ go-wrk -c 1 -T 100000 'service/area?lat1=55&lng1=12&lat2=55&lng2=12'
Running 10s test @ service/area?lat1=55&lng1=12&lat2=55&lng2=12
  1 goroutine(s) running concurrently
1 requests in 1m20.554213498s, 383.85KB read
Requests/sec:           0.01
Transfer/sec:           4.77KB
Avg Req Time:           1m20.554213498s
Fastest Request:        1m0s
Slowest Request:        1m20.554213498s
Number of Errors:       0
```

Response time for 1500 images on 10 goroutines (15.000 total image links):

```
$ go-wrk -c 10 -T 200000 'service/area?lat1=55&lng1=12&lat2=55&lng2=12'
Running 10s test @ service/area?lat1=55&lng1=12&lat2=55&lng2=12
  10 goroutine(s) running concurrently
10 requests in 1m7.209131623s, 3.75MB read
Requests/sec:           0.15
Transfer/sec:           57.11KB
Avg Req Time:           1m7.209131623s
Fastest Request:        1m0s
Slowest Request:        1m9.661708598s
Number of Errors:       0
```

Response time for 1500 images on 32 goroutines (48.000 total image links):

```
$ go-wrk -c 32 -T 100000 'service/area?lat1=55&lng1=12&lat2=55&lng2=12'
Running 10s test @ service/area?lat1=55&lng1=12&lat2=55&lng2=12
  32 goroutine(s) running concurrently
32 requests in 1m19.397931339s, 12.00MB read
Requests/sec:           0.40
Transfer/sec:           154.70KB
Avg Req Time:           1m19.397931339s
Fastest Request:        1m0s
Slowest Request:        1m19.79735394s
Number of Errors:       0
```

We do not quite know why is does not scale above 32 concurrent client connections but maybe it has to do with a certain max limit set, when using the Google Cloud service APIs. Ultimately, this might be investigated by trying it out on the final deployment version as opposed to the local test deployment app instance - or finding out how to configure the API services to allow more connections. It would have been interesting to test it on more connections and more data for benchmark comparisons.

1.3.2 Memory Test Results

Memory allocations when fetching all granules for a specific lat and lng:

```
$ go test -bench=Images
BenchmarkImages-8 1       3501121132 ns/op 599168 B/op 2822 allocs/op
PASS
ok        github.com/tvao/SatelliteWebApi/src       81.241s
```

As shown above, 3,5 seconds are spent per request with 2822 memory allocations and 599168 bytes spent per request operation.

Memory allocations used when fetching 1563 image links in area of interest from lat1 '55.660797', lng1 '12.5896', lat2 '55.663369' and lng2 '12.584670'

```
$ go test -bench=Area
BenchmarkArea-8 1 58636079673 ns/op       26090912 B/op 100953 allocs/op
PASS
ok        github.com/tvao/SatelliteWebApi/src       128.122s
```

This benchmark shows how well the new geography feature (i.e. area handler) performs, when fetching 1500 image links in an area between two pairs of coordinates. To test this further, you could increase the area of interest with a benchmark on two geographical pairs of coordinates that are further apart.

Now, to get an idea of the dominant memory allocations, one may add the -memprofile flag and analyze the output file with the built-in "pprof". The goal is to gain a better understanding on the potential performance bottlenecks in the program. For this purpose, we have first produced a cpu and memory profile, when fetching 1500 images in a geographical area:

```
$ go test -bench=Area -benchmem -cpuprofile=cpu.out -memprofile=mem.out
BenchmarkArea-8 1 51629207238 ns/op       25672448 B/op 100619 allocs/op
PASS
ok        github.com/tvao/SatelliteWebApi/src       113.194s
```

Based on this, we use the "pprof" tool to analyze the data:

```
$ go tool pprof mem.prof
(pprof) top5
Showing nodes accounting for 2659.24kB, 100% of 2659.24kB total
Showing top 5 nodes out of 54
      flat  flat%   sum%        cum   cum%
  512.12kB 46.39%   100%   512.12kB 46.39%  main.getImagesFromBucket query.go
         0     0%   100%   528.17kB 19.86%  bigquery.insertJob bigquery.go
```

Seemingly, a lot of memory allocations are spent on the queries in BigQuery and Storage API.

To investigate this, one may run the same query and ran the following command once in a while to see the memory consumption in the heap throughout the request lifecycle: "go tool pprof goprofex http://localhost:8080/debug/pprof/heap. Interestingly, if you then use the list command, one may see exactly what allocates so much memory in the query service:

```
$(pprof) list main.getImagesFromBucket query.go
Total: 1.08MB
ROUTINE ======================== main.getImagesFromBucket in query.go
  512.12kB   512.12kB (flat, cum) 46.39% of Total
         .          .    124: if err != nil {
         .          .    125:   log.Fatalln(err)
         .          .    126:   return nil, err
         .          .    127: }
  512.12kB   512.12kB    128: fullImageURL := bucketName + "/" + attrs.Name
```

In this case, a new string is allocated for each link retrieved from the cloud bucket, which does not scale, when increasing the size of the input (i.e. geographical area). Thus, by using the "pprof" profiling tool in Go, one is able to pinpoint weak areas like this in code. A solution might be to use a byte array and flush it on each iteration to avoid spawning many short-lived string objects.

By checking the heap repeatedly, numerous optimizations may be found, especially regarding string concatenation, when constructing image links from base URLs and granules:

```
$(pprof) list main.getImageBaseURL query.go
Total: 1.58MB
ROUTINE ======================== main.getImageBaseURL in query.go
  512.09kB   512.09kB (flat, cum) 31.69% of Total
         .          .     91:   if err != nil {
         .          .     92:       return nil, err
         .          .     93:   }
         .          .     94:   imageBaseURL = row[0].(string)
  512.09kB   512.09kB     95:   granuleID = row[1].(string)
         .          .     97:   links = append(links, fullImageURL)
```

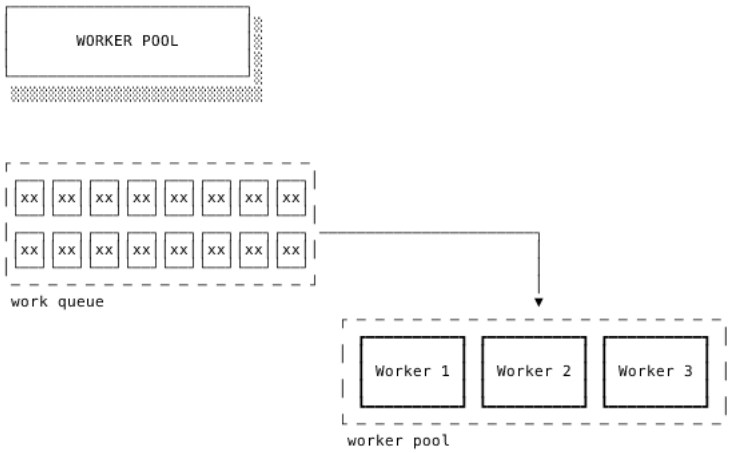
Again, this may be fixed by introducing a single byte array or string object, reused throughout the whole iteration of image links. By doing this and running the response time benchmark again, a reduction from 70 to 50 seconds was achieved. Besides this, we did not manage to find out how to keep track of all memory allocations with specific code references like the above, throughout the whole request lifecycle. However, one may perhaps look at the execution tracer in Go for an even deeper understanding of the memory consumption and lifecycle of the web service. The tracer provides information about goroutines, system calls, the heap and processes. Commands for tracing are: "go tool pprof http://localhost:8080/debug/pprof/profile" or from an output file: "$ go tool trace trace.out".

1.4 Scalability Improvements

As a final remark, it may be noted that the previous response time benchmark results certainly indicate that the web service is somewhat slow. This was assumed to be due to the fact that all images residing in different folders are fetched sequentially.

To fix this and make the web service truly scalable, a simple worker pool has been implemented, using goroutines and channels. The worker pool is a model in which a fixed number of m workers (implemented in Go as goroutines) work their way through n tasks (image links) in a work queue (implemented in Go with a channel). The job/worker design pattern has been used with Go channels to create a channel system, on which image query jobs can be queued and operated on my multiple workers (i.e. goroutines) concurrently. The example below illustrates the idea of a worker pool:

Figure 1: Worker Pool visualization: workers work concurrently on work items



A simple code implementation has been included on the next page, for the sake of illustrating how such a massive performance gain may be achieved with a very simple concurrency pattern in Go. Obviously, this may be improved in the future by making a separate worker pool abstraction that decoupled from its specific use in the "area" request handler function. Instead, it may then be used in other request handlers and be made more robust with error handling.

In terms of benchmarking, the web service is now capable of fetching the 1563 images in less than 3 seconds - as opposed to the previous response time of 1 minute. Note that the other benchmarks have not been rerun, as this new solution was found on the day of submission. Thus, only the response time was reevaluated to get an indication of the performance gain.

1.5 Worker Pool Code

The following code is from the web service located in "service.go". Notice some code have been omitted to illustrate only the parts that actually help provide the significant performance (i.e. response time) improvement, using the work pool pattern for concurrency.

```
// Worker receives work on jobs channel and send back images links on results channel
func worker(id int, jobs <-chan string, results chan<- Links) {
        for imgFolder := range jobs {
                result, err := getImagesFromBucket(bucketName, imageObject, r)
        }
}
func area(w http.ResponseWriter, r *http.Request) *appError {

        // Create a set of worker jobs for each link
        numberOfJobs := len(links)
        jobs := make(chan string, numberOfJobs)
        results := make(chan Links, numberOfJobs)

        // Setup worker pool
        for i := 0; i <= numberOfJobs; i++ {
                go worker(r, jobs, results)
        }

        // Send jobs
        for _, imgLink := range links {
                jobs <- imgLink
        }
        close(jobs) // Close do indicate this is all work to be done

        // Collect worker results and write them back to client in JSON result
        imageResult := Links{}
        for i := 0; i <= numberOfJobs; i++ {
                imageResult = append(imageResult, <-results...)
        }
        close(results)
        encode := json.NewEncoder(w).Encode(imageResult)
}
```