Assignment 2

Adrian Westh, Emil Lynegaard and Simon Munck

November 8, 2017

Link to prototype: `https://ecly-178408.appspot.com /images?address=` to query an address, `images?lat=&lng=` to query a coordinate, and finally `images/area/north_lat=&south_lat=&east_lng=&west_lng=` to query an area: The area bounded by the two coordinates.

Examples:
`https://ecly-178408.appspot.com/images?lat=37.4224764&lng=-122.0842499`
`https://ecly-178408.appspot.com/images?address=RuedLanggaardsVej,7, 2300,K\OT1\obenhavnS`
`https://ecly-178408.appspot.com/images/area?north_lat=-2.89&south_ lat=-6.55&east_lng=29.63&west_lng=25.93`

1 Adding Color

Due to the updated assignment, and the format used for the image data, we have chosen to only describe how we could have implemented color querying.

If we were to implement the color querying, we would expand our existing implementation as follows: Prior to returning the retrieved URLs color ranked, we would download the associated images, calculating their color properties locally. This would be done using Go's image package[1] assuming we could convert the images to a recognized format. Once an image's color index was calculated, we would cache their color properties either in memory or in a database, to avoid downloading and evaluating any image twice. With this cache we would then prior to any image retrieval check whether it exists in our cache and only download it and evaluate it if this were not the case. With this overall design, we would be able to allow color indexing, although it would be incredibly slow for uncached images, as the average filesize of the `.jp2` images is around 100MB.

2 Scaling Spatially

Our code is designed to scale well. This is certainly needed for the area query, since the resulting list of images is potentially enormous. When fetching the ap-

[1] https://golang.org/pkg/image/

	Attempted Requests	Elapsed Time	Requests/sec:	Transfer/sec:	Avg Req Time:	Fastest Request:	Slowest Request:
Area	17	37.673s	0.45	873.87KB	22.161s	7.149s	40.185s
Address	64	32.650s	1.96	694.41KB	5.102s	1.851s	9.795s
Coords	90	31.801s	2.83	578.11KB	3.533s	1.737s	11.279s

Table 1: Benchmark Specifications

propriate URLs, `goroutines` are utilized. When executing these `goroutines`, the work of fetching an URL does not slow the program down considerably: It is done in a separate lightweight thread. To aggregate the result of all the `goroutines`, we use `channels` to return lists of URLs, all of which are appended to the single list we return.
After making some tests on the Google Cloud Platform with the program deployed, we found that executing a very large amount of `goroutines` would exceed the memory usage allowed using the free plan (or at least the subscription we have). This lead to the introduction of a semaphore disallowing more than 100 `goroutines` to execute simultaneously.
Concluding, in regards to scalability, due to the lack of shared state outside of the `goroutine` limiting semaphore, everything else works independently allowing requests to naturally happen concurrently.

3 Benchmarking

The benchmark results are shown in Table 1. We have chosen to cover three use-cases: Finding the images within the tiles between two coordinates (*area*), the images in a tile from an address and the images in a tile from a coordinate. The three experiments were all limited to run 10 `goroutines` concurrently.

The benchmarks were implemented using the `go-wrk`[2] library to execute commands recording measurements for the benchmarks.

[2]https://github.com/tsliwowicz/go-wrk