SoWS, Assignment 2 Report

By:

Arleth, Johan

Ulrik, Kenneth Ry

Date:

November 10, 2017

1 Introduction

Assignment 1 in the course Scalabilty of Web Systems asked the students to build a web-server that can access the Sentinel 2 datasets available online, by allowing a web-browser user to supply search parameters in the shape of longitude and latitude values, and the web-server should use these parameters to find the closest Sentinel 2 image of the ground, unfiltered in terms of time and return a direct (download) link to that image.

The purpose of "Assignment 2" in the course Scalability of Web Systems is to extend on "Assignment 1", by (a) allowing the user to supply a range of latitude as well as longitude parameters, and retrieving all images within this rectangular space on earth, and (b) allow for various ways to rank by the average color in an image, or by the distance to a desired color, or by the distance to the euclidean line between two colors in the 3-dimensional color-space.

All of the color-related sorting-options have been excluded due to technical properties of the bridge between Go and C (which has a library required to perform reading of the ".jp2" files on the Sentinel 2 dataset) and deployment to the Google Cloud AppEngine.

2 Design

2.1 Latitude and Longitude search

The update to this part in Assignment two, is that we should now be able to take 2 latitude and 2 longitude parameters, which forms a rectangular space on earth. The idea is then that more than a single image-download-path is returned, since there will be several images within such a range. Our approach is to query up against the BigQuery Go API library and we can then simply supply an SQL-statement that filters so called "Granules" by their related longitude and latitude values, meaning we can query directly onto the BigQuery storage, without requiring any conversion from the longitude and latitude parameters.

2.2 Ordering by color

In particular, the feature discussed here regards **ordering** by color, rather than *filtering* by color. This means that the latitude and longitude parameters continue to be the only way to filter the retrieved images, whereas the *set* of returned image-links should be ordered in terms of the average color within that image.

2.2.1 In general

First, to achieve general ordering by color of a set of images, we would need a way to calculate the average color for each of these images. We would do this by getting all the RGB-values for every pixel in the image, sum all the R, G and B channels and use integer-division by the amount of pixels (size of the image in pixels), thus getting the average RGB value. Integer-division would be used to ensure that we get a proper RGB-value. Ideally, for optimization, we would take a new query parameter "accuracy" which contains an integer value. This parameter would define the sampling frequency of our calculation of the average colors in the found images, meaning with a value of 5, we would only read the color of every 5 pixels in the image, and sum these (of course dividing only by the amount of pixels that are being summed). This part has not been implemented in code.

2.2.2 With regards to a color-band

For sorting by a certain color-band (which we, like the other sorting-challenges, didn't prepare HTTP parameters for handling), we would grab the relevant colorband-image out of the 3 representing the entire Sentinel 2 bucket's full image, sum all the pixels in that image representing e.g. the Red colorband, and then we could just order the different buckets by their red images' summed amount of pixel-values.

2.2.3 With regards to given RGB

We compute the Euclidean distance between a supplied (as a query parameter) RGB color and the average RGB colors, and the sorting is based on this calculated distance from a lower distance to a larger distance (ascending once again). This allows a user to ask for quite green images (perhaps looking for forest-images in the spanned area) by for instance supplying an RGB parameter of (0,255,0), to sort by images distance to the purely green color.

2.2.4 With regards to two given RGB's

Given two RGB parameter values to use for ordering, what we do is that we compare average RGB values of images by their distance to the 3D line between the two given RGB parameters in the 3D RGB space, and once again we would sort in an ascending manner, such that images that are closest to the line between the two parameter RGB's are listed first, and those farther from this line are listed last.

3 Implementation

3.1 Latitude and Longitude search/filtering

We had a method which uses the BigQuery client library to speak through to the BigQuery API, which took singular latitude and longitude arguments, added a +/- 0.5 accuracy ratio around the two values, and handled a single one of the base-URLs in the area given by the ranges from -0.5 lat and long to +0.5 lat and long. This was done to make sure that a user's parameters would always find some result (avoid having to do direct float precision comparison. This was the solution to assignment 1.

For assignment 2, we have extended this behaviour, by optionally allowing the "lat" and "long" query parameters to take two values each, comma-separated, e.g. "?lat=20,22long=37,41". A user is not allowed to provide a range for one, but not the other. Also, the first number should be smaller than the second. Given this, we then perform the same BigQuery SQL query, but in this case without changing the given parameters and instead applying these number-values directly in the area-filtering query onto the BigQuery table with Sentinel 2 base-URLs.

3.2 Color-sorting

For this, since this part of Assignment 2 was redacted, we have not finished an implementation regarding this part of the task. However, we do have some untested code that implements the color-ordering, assuming that these functions can receive a list of R, G, or B values (as an integer matrix) for an image. This corresponds to the storage of the Sentinel 2 data, where each bucket has at least 3 images stored, one for each color-channel. They are stored as grayscale images. If these were to be mapped onto each other with the correct mapping of which is which, one would be able to construct the complete image. Knowing this structure, the function `getAverage_R_G_or_B_Value` in "colors.go" takes one such channel-file and extracts the average pixel value throughout that entire image. As a detail, it arbitrarily reads the Red channel of the ARGB structs that are being read using the "image" library in Go. We could also have read the Blue or Green channel - since it is a grayscale image (by assumption/knowledge), all channels have the same values for all pixels (locations in the image).

For the actual color-ordering, we also prepared logic, which is also unused, that could perform all the 3 required ordering-methods. These are defined respectively (as per the assignment description) as: `orderByColorBand`, `orderByColor` and `orderBy2Colors`.

4 Benchmarking

4.1 Method

We have used an ASUS X550JX[1] connected via LAN on the ITU internet for benchmarking our solution.

The solution-space, as previously touched upon, doesn't involve any real color-sorting computations. Indeed, our code also does not download any of the hosted .jp2 files. Thus, our test is more focused on the overhead of sequentially accessing the client library for the Sentinel 2 datasets, navigating through the directories of the datasets, to discover the full paths for the various stored jp2 files.

Our tests do not go over 1000 base URLs, since we have put 1000 results as the limit on our BigQuery queries. This is to avoid getting unreasonably large amounts of results, which can eat a lot of the data limit on Google Cloud.

Given this setup, the most predictable of outcomes, and what we indeed would expect, is a directly linear increase in time spent to treat these base-URLs. This is based on the simple thought-experiment that each base-URL requires a certain amount of calls to the Sentinel 2 dataset's API, in order to discover the full image-paths. By having to treat a larger amount of base-URLs, the amount of API-calls is simply multiplied by the amount of base-URLs to deal with.

Looking at figure 1, where our benchmarking results are depicted, we mostly see the expected, as base-URL handling increases linearly for the most part, *however*; a slight indication of exponential growth does appear near the final test-runs. The test is probably too limited to really conclude anything based on this however. One qualified guess could be that some list-operations (lookups, searches, scans, etc.) in the source-code of the treatment of individual base-URLs is slowly eating more and more of the time-cake. This guess should however not suggest exponentiality, but should only affect the measurements in the sense of a steeper slope on the linear result function.

The slight rise in time spent on Bigquery is probably down to two things: The query is touching upon more data, which means longer data processing time, and since we are getting more results, it's a bigger data transfer from BigQuery to our server.

[1] `https://www.komplett.dk/product/853766#`

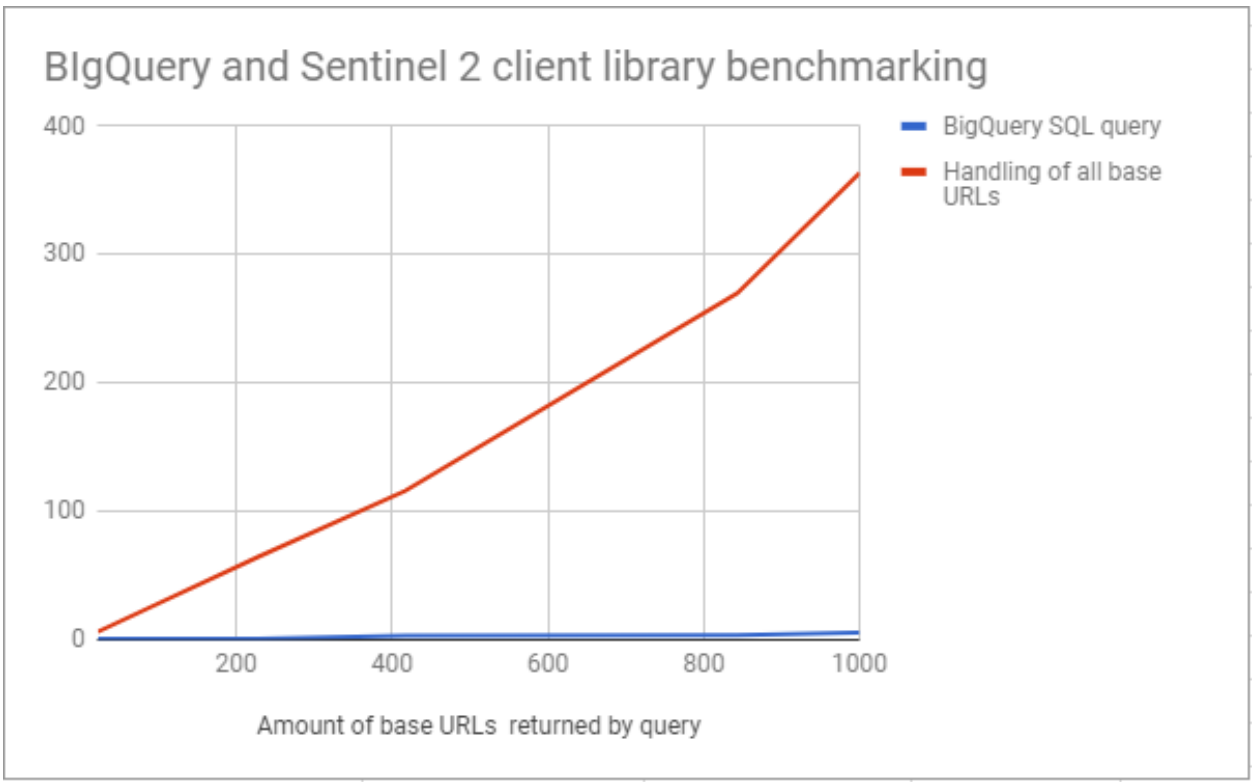

Figure 1: Our benchmarking results visualized. The X-axis is the amount of base-URL's returned by the BigQuery SQL query, and the Y-axis is time measured in seconds.

4.2 Appengine limitations

For reasons unknown, we are getting 500 (Internal Server Error) when our query touches upon a sufficiently large area. This is however not an issue when hosting our server on localhost. An example is latitude-range: 20-24 and longitude-range: 40-44. On localhost it is not an issue to process this, but on Google Cloud it crashes with error 500. For this reason we had to do our benchmarking with our webserver deployed on localhost.