Scalability Project ~~2~~

Thor Olesen and Stig Killendahl

November 2017

Contents

1 Introduction

For this assignment, the web service has been extended with a **geometry** package, located in **geometry.go**. It supports the functionality used to both fetch and parse Geofabrik polygon data and construct S2 region covers that yield an approximate polygon representation of a given country. This may be used to query and count all satellite images that match a given country by reusing the functionality from assignment 2. Namely, by querying the sentinel-2 data bucket via the **worker pool** functions in **service.go** that spawn goroutines for all granules, that match the specified country, and finally fetch images for all these granules concurrently, where each query is handled in the **query** package.

2 Geofabrik Parsing Feature

Firstly, the web service nos supports fetching and parsing PSLG data of a country the user inputs from Geofabrik. The country is given as a query parameter in a HTTP request of the form "/geo?country=denmark", which is handled in the **service geo** handler (see **??**. The service then uses the **geometry** package to parse and construct a polygon representing the given country:

```
func parse(r *http.Request, country string) ([]float64, error) {
        client := urlfetch.Client(r.Context())
        request := fmt.Sprintf("http://download.geofabrik.de/europe/%s.poly", country)
        resp, err := client.Get(request)
        if err != nil { // Retry if error
                err := retry(DefaultRetry().MaxRetries,
                             DefaultRetry().Duration*time.Second,
                            func() (err error) {
                                    resp, err = client.Get(request)
                                    return
                            })
                if err != nil {
                        return nil, err
                }
        }
        defer resp.Body.Close()
        regex := regexp.MustCompile(floatExponentPattern)
        bytes, err := ioutil.ReadAll(resp.Body)
    ...
        data := regex.FindAllString(string(bytes), -1)
        countryCoords, err := normalizeCoords(data, strconv.ParseFloat)
    ...
        return countryCoords, nil
}
```

As shown in the snippet, the country specified by the client is used to fetch the polygon data from Geofabrik and parse it to a set of latitudes and longitude coordinates used to construct a polygon of the given country. This is done using a **urlfetch.Client**. Notice that the context is now passed on with the http request, instead of spawning off a new one for each use like in assignment 1 and 2 using: **appengine.NewContext(request)**. This has been done to keep the context scoped to each request. By analogy, the Context flows through the program like a river or running water. Consequently, it is never stored or kept around more than strictly needed and does not leak any ressources. Instead, it is passed along with each request and runs during the lifetime of a request:

```
func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        ctx := appengine.NewContext(r)
        if err := fn(w, r.WithContext(ctx)); err != nil {
                http.Error(w, err.Message, err.Code)
        }
        defer cancel() // Cancel ctx as soon as request returns
}
```

3 Region Cover Feature

The country polygon is constructed by using the **S2** geometry library nad its **RegionCover** type. The S2's RegionCover datatype is used to create an approximation of the resulting polygon (consisting of rectangles). This allows arbitrary geographical regions to be approximated as unions of cells (**CellUnion** datatype). This is useful for approximating operations on a country and is used to fetch and count then umber of satellite images associated to a country (at a given point in time). Namely, the parsed coordinates are converted to S2 points that are used to build an "approximated" region cover of the country:

```
func regionCover(coords []float64, maxLevel, maxCells int) s2.CellUnion {
        points := []s2.Point{} // Construct points
        for len(coords) > 0 {
                lat, lng := coords[0], coords[1]
                p := s2.PointFromLatLng(s2.LatLngFromDegrees(lat, lng))
                points = append(points, p)
                coords = coords[2:] // Rest coords
        }
        // Construct loop (i.e. spherical polygon) and polygon
        l1 := s2.LoopFromPoints(points)
        loops := []*s2.Loop{l1}
        poly := s2.PolygonFromLoops(loops)
        // Construct region cover
        rc := &s2.RegionCoverer{MaxLevel: maxLevel, MaxCells: maxCells}
        cover := rc.Covering(poly)
        return cover
}
```

As a result, a region cover is built from the country coordinates. The region cover is represented internally as a union of cells (i.e. CellUnion). Further, the size of the cells covering the country region may be specified, based on a granularity level. Specifically, the **maxLeveL** parameter determines the granularity of the cells covering a region, where a levle of 30 is extremely small (¡ cm^2)*andalevelof*1*isverycoarse.Further*,*the***maxCells***parameterdetermineshowmanycellsareusedatmosttocoveragiv*

```
func imagesByRegion(cover s2.CellUnion, r *http.Request) (Links, error) {
    // Spawn concurrent jobs
        for i := 0; i < len(cover); i++ {
                c := s2.CellFromCellID(cover[i])
                go getImageBaseLinksSafe(client, r, results, errChan,c)
        }
        // Await concurrent results on channel
        for range cover {
                select {
                case err := <-errChan:
                        return nil, err
                case links := <-results:
                        mutex.Lock() // Protect non-threadsafe slice against races
                        rectangleImages = append(rectangleImages, links...)
                        mutex.Unlock()
}}}
```

Do you really need to synchronize on the non-threadsafe slice or does the channel automatically block ?

4 Design and Implementation

Besides the above implementation details, some scalability issues arose, when trying to fetch and count satellite images for a whole country region. Specifically, the web service became dramatically slower and less resilient. For example, it may take several minutes to fetch images for a given country and images may be lost. To cope with this, a retry mechanism has been implemented that serves to make the web service more resilient by retrying failed calls to the Google Cloud services, BigQuery and Storage respectively. Further, goroutines are still used to concurrently fetch all the images that reside in each granule subfolder of images.

```
func retry(attempts int, sleep time.Duration, callback func() error) (err error) {
        for i := 0; ; i++ {
                err = callback()
                if err == nil {
                        return
                }
                if i >= (attempts - 1) {
                        break
                }
                // Add randomness to prevent Thundering Herd (process retry overlaps)
                jitter := time.Duration(rand.Int63n(int64(sleep)))
                sleep = sleep + jitter/2
                time.Sleep(sleep)
        }
        return fmt.Errorf("after %d attempts, last error: %s", attempts, err)
}
```

The retry function is then used in the **service** package to retry failed attempts to query a given granule subfolder in the bucket:

```
func worker(client *storage.Client, r *http.Request, jobs <-chan string, results chan<- Result) {
    for imgLink := range jobs {
                result, err := getImagesFromBucket(client, bucketName, imageObject, r)
                if err != nil { // Retry for better resilience
                        err := retry(DefaultRetry().MaxRetries,
                                     DefaultRetry().Duration*time.Second,
                                     func() (err error) {
                                            result, err = getImagesFromBucket(...)
                        return
                        }) ... }}}
```

As a result, the web service now behaves consistent and correct at a cost of performance. Arguably, the web service in its current form favors accuracy over throughput. The retry is set to happen 5 times by default, meaning each request may potentially run 5 times per granule. By comparison, an example may be explained based on the default granularity settings set to 15-30 max level and 100 max cells in the region cover. Namely, when querying all images in Denmark, the region cover of Denmark is used to find 4070 granules. Currently, the images that reside in each granule are fetched concurrently, meaning 4070 goroutines are spawned that each fetch 10-15 images. In the worst case, these requests may happen up to 5 times, meaning it is fail safe but it may take up to 20.000 concurrent requests.

5 Scalability Evaluation

Based on the previous example of finding satellite images of e.g. Denmark, one may consider how to evaluate the web service. That is to say, evaluating scalability as the capability of the web service to handle a growing amount of work, which in this case is based on the size of a country. In this project, it has been important to strike a balance between the trade offs of accuracy, latency and throughput. On one hand, one may want to consistently find the correct amount of images that match the country, which favors accuracy over throughput and latency. On the other hand, one may compromise this in favor of throughput and low latency, by approximating the result. Thus, the configuration of concurrency, geographic granularity (i.e. approximations) and request retries will ultimately affect the web service performance - and how well it scales with the size of the country.

Currently, the web service is set to retry any failed request attempts to the sentinel-2 bucket 5 times by default, with a maximum duration of 10 seconds each. Further, the region cover is set to be constructed with a cell level between 15-30, where 1 is never used due to it being very fine-grained, resulting in too many granules being created to query in the storage bucket. Initially, the web service was run on a few countries, including the following:

Results with Storage Bucket Client

- Denmark yields 50.000 images in 1 min
- Germany yields 133.000 images in 3-4 min
- Russia: N/A

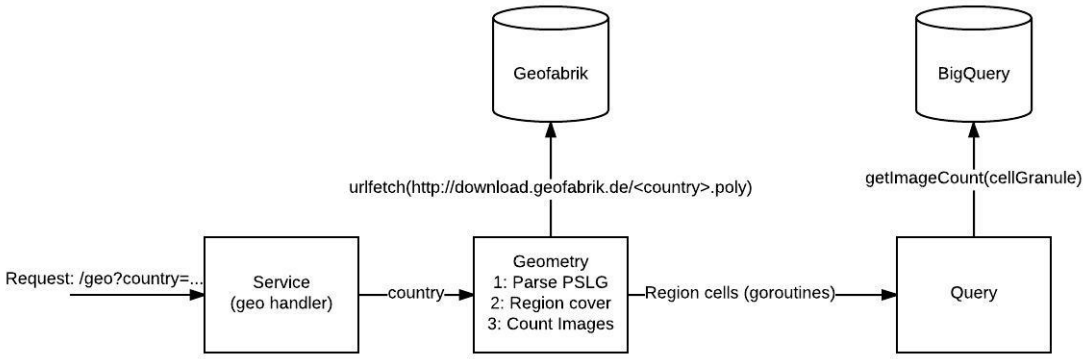
Seemingly, the web service did not scale well with the size of the country, since a bigger country is comprised of more granules, and each granule spawns a new goroutine to request images in the sentinel-2 storage client. However, it should not cause any performance issues to spawn this many thousands of goroutines, as they represent lightweight threads of execution that are "cheap" to create[1]. Rather, the bottleneck seems to be caused by the many granules that are used in concurrent storage bucket requests. Arguably, certain request limits are enforced by the Cloud Storage[2] that prevent all of the storage requests to execute concurrenctly as intended. On top of that, the "retry" mechanism introduces yet another performance overhead, by increasing the total number of storage requests, based on the amount of storage request failures. Seemingly, the problem lies in striking a balance between either counting the images accurately with a consistent result or approximating the count in favor of reduced latency.

[1] http://blog.nindalf.com/how-goroutines-work/ seen 22/11/17
[2] https://cloud.google.com/storage/quotas seen 22/11/16

6 Improvements

In his regard, another approach has been used to improve performance of the web service dramatically and reduce the overall latency. Namely, the granules are counted directly via BigQuery instead of querying each granule in the storage bucket. In addition, the number of images per granule has been inspected to be 13 in the sentinel-2 bucket. Thus, under the assumption that this granule size remains unchanged for all granules, one may completely avoid querying the Storage API. Although this is a strong assumption, one could also determine the granule size dynamically with a single image count request to a given granule in the storage bucket. In this way, one does not need to iterate all images for all granules, which does not scale well in practice. As a result, one no longer needs to create goroutines for each granule to fetch the total image count. Although this solution is less "dynamic", it dramatically improves the performance. For example, it previously took 120 seconds to query the image count of Denmark, using the storage client. By comparison, it now only takes less than 2 seconds, when querying the granules only in BigQuery, knowing each granule contains 13 images. Most importantly, the same final image count result is yielded with a huge performance gain. As shown below, the storage client is not used in the web service request:

Figure 1: Country Request Flow



New Results with BigQuery Client only

- Denmark with 5000 granules yields 50.000 images in 1 second
- Germany yields 133042 images in 8 seconds
- Russia (biggest country) yields an image count of 36.615.475 in 10 seconds

All together, the web service has been optimized by counting the images directly via BigQuery, using a fixed size for granule images and setting the max region cover level to 15. In the future, one might look into how to leverage many thousand concurrent requests to the storage API, which did not work out in this project. Arguably, it might be due to a cap limit enforced by Google Cloud on the service or associated billing account issued by ITU. Nevertheless, it leaves room for further investigation, along with further improvements on setting the appropriate level of concurrency, geographical approximations and request retries on failed Google Cloud requests.