Geo Mosaic

Ben Russell
Benjamin.R.Russell@Colorado.EDU

Rob Elsner
Robert.Elsner@Colorado.EDU

Chris Grosshans
Chris_Grosshans@Comcast.net

ABSTRACT
A system is presented which will index images from the World Wide Web that have Global Positioning System coordinates for the image location. It will then generate a new image by combining source images, where the process of selecting source images takes in to account geographic locality and then average color match.

Categories and Subject Descriptors
J.5 **[Computer Applications]**: ARTS AND HUMANITIES

General Terms
Design, Experimentation

Keywords
Image processing, Image Mosaics, Distributed Systems.

1. INTRODUCTION

For our project we created a distributed web application that generates photo mosaics. To generate a mosaic the user simply uploads an image that has been encoded with geographic location information. Geo mosaic then selects images taken near the uploaded image. These images are combined to create a photo mosaic. When viewed from afar the mosaic resembles the uploaded image. Closer inspection reveals that the mosaic is a composite of images from nearby locations. The first image shows an example uploaded image. The second shows a mosaic of the uploaded image.

Figure 1 Source Image

Creative Commons copyright image courtesy gnuckx and can be viewed at: http://www.flickr.com/photos/gnuckx/4276820451/

Figure 2 Mosaic Image

Figure 3: Source Image Location

The inset yellow box expands some of the source images. The image highlighted by the arrow is located in Turkey (see figure 4);

Figure 4: Member Image Location

The remainder of this paper reviews the architecture of the Geo-Mosaic. Section 2 provides a high level architectural overview of the application. The following three sections review the sub systems of the Geo-Mosaic application in detail.

2. Architectural Overview

The Geo-Mosaic application is comprised of three distinct sub-systems.

1. Image Locator Sub-system
2. Image Storage Sub-system
3. Mosaic Creation Sub-system

The Image Locator Sub-system is responsible finding photos that have been encoded with geographic location information. These photos are referred to as geo-tagged images. The Image Locator Sub-system consists of several web crawlers that scan the Internet to find geo-tagged images. When a crawler finds a geo-tagged image it notifies the Image Storage Sub-system. The Image Storage Sub-system stores longitude/latitude and color information for each geo-tagged image.

The Mosaic Creation Sub-system provides a web interface that allows users to upload images. When an image is uploaded the Mosaic Creation Sub-system queries the Image Storage Sub-system for images taken near the location of the original image and creates a mosaic from these images.

Each sub sub-system is a self contained application. The sub-systems communicate using web services. The following figure provides a visual representation of the three systems.

The system architecture is provided in figure 5.

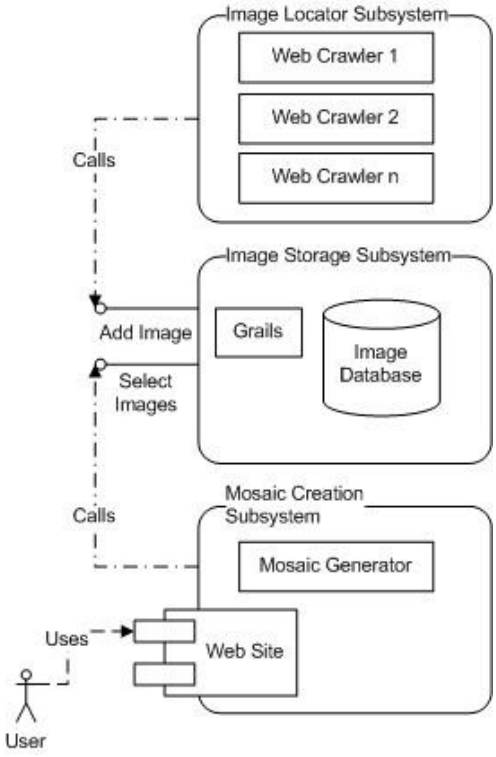

Figure 5 System Architecture

3. Image Locator Sub-system

3.1 Goal and Overview

In order for the application to function, we require a large database of geo-tagged images. After exploring several possible sources, we settled on flickr as an excellent source for photographs with geographic information attached. In addition to offering a large number of source photos, it provides a convenient Multilanguage API [1]. We chose to implement this tool in perl, largely for convenience of implementation and accessible flickr and SOAP modules. Up to four hosts were used at a time, running on consumer hardware under OSX and EC2 instances with Ubuntu Linux installed.

3.2 Workflow

The main loop of our image crawler involves downloading all of the geo-tagged images from a given pair of longitude/latitude coordinates, processing them, and then moving on to another set of coordinates. The initial communication is performed as a flickr.photos.search command which returns an XML tree of every photo within 32 km of the supplied longitude and latitude. The tool then parses this xml file to obtain the internal photo ID and the external image URL. The photo ID is then passed to a

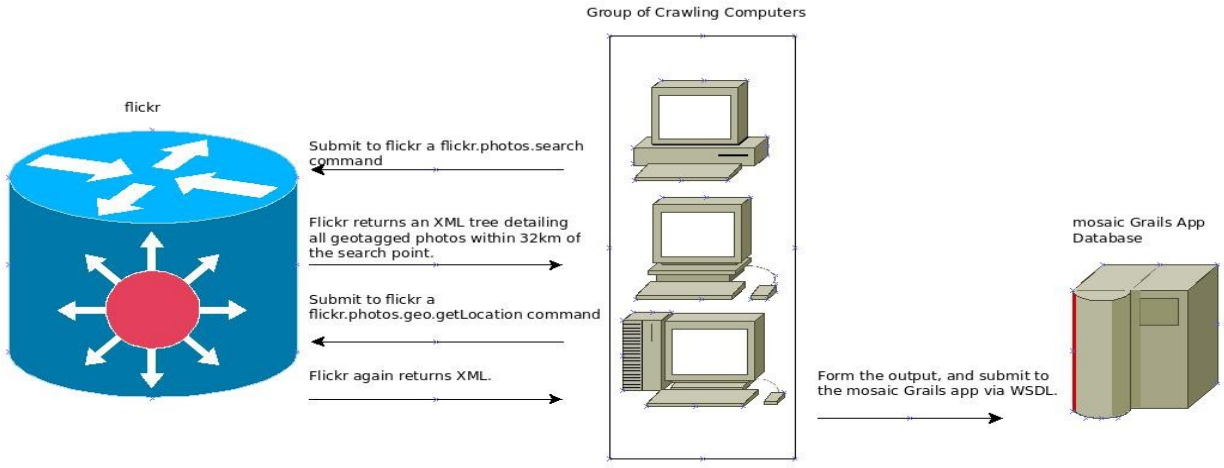

Figure 6 Web Crawler Workflow

flickr.photos.geo.getLocation command which returns via XML the longitude and latitude attributes of the image.

To obtain the needed average red, blue and green color data the tool downloads the image using LWP and generates a color histogram via ImageMagick. The downloader then communicates with the database via WSDL saveImage call. Each URL visited is saved in an internal data structure to ensure that communication with the database is restricted to new images.

After examining every image at a given set of coordinates the crawler increments the longitude value by .25 degrees with a special condition to reset the targeted value to -180 degrees should it exceed 180. After the crawler has traversed the planet in this fashion it then increments the latitude by .25 degrees in an outer loop. This program would terminate after examining every point on the earth, but in practice this never happened. After several weeks the program was terminated on each host manually as a sufficient body of images had been amassed.

3.3 Performance

The design of the downloader was heavily influenced by the amount of time allotted for the project. As we had numerous weeks to obtain the needed data, the crawling rate was decreased in order to avoid flooding the flickr servers, and possibly being black listed. Consequently the speeds obtained are far below the true possible throughput.

Each instance of the downloader managed to analyze on average 27 images per minute. As the number of photos available on the flickr service are extremely numerous and spanned the globe individual crawlers could run for an extended period without duplicating downloaded files. Consequently for these regions the number of images examined per minute scaled linearly.

As the image throughput was limited, CPU consumption was negligible throughout. While memory consumption did have the potential to grow without bound in the form of the previously visited datastructure, in practice this did not prove to be a problem, both the OSX and Ubuntu hosts allowed the crawler to run for several weeks without issue.

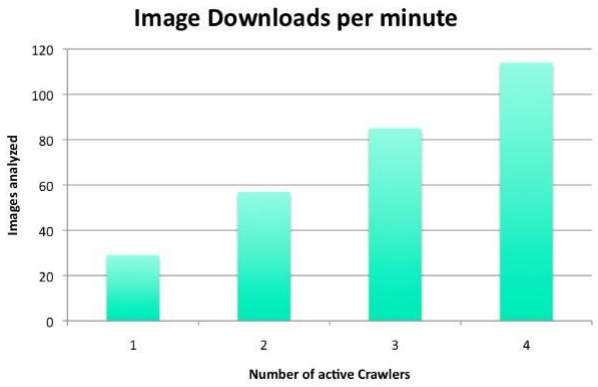

Figure 7 Download Rate By Crawler Count

4. Image Storage Sub-system

4.1 Image Storage Sub-system overview

The image storage sub system is a standalone application that consists of two parts. The first part is an Amazon RDS database. The Second part is a web application that allows external systems to access the database using web services. The remainder of this section explores those two sub components in additional detail. The following image provides an overview of the Image Storage Sub-system

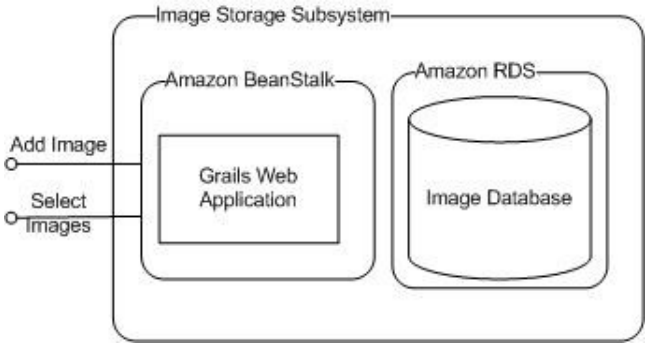

Figure 8 System Diagram of Storage Subsystem

4.2 Database Selection

The goal of our project was to create a distributed system. We chose to build our project on top of Amazon cloud technologies. Amazon provides a number of database alternatives: Amazon SimpleDB, Amazon EC2 Relational Database AMIs and Amazon RDS. This section highlights the differences between these technologies and explains why we chose Amazon RDS.

As its name implies Amazon SimpleDB [2] keeps things simple. Most of the database administration, including scaling and geographical distribution, is configured via an easy to use web page. SimpleDB is a not a relational database. There are no schemas so there is no automatic data validation. You must store all of your information in one table. As our application database needs are minimal, SimpleDB looked like the most promising choice. However upon closer inspection SimpleDB was too simplistic. SimpleDB can only store UTF-8 string values. Our application requires several numeric fields. More specifically we needed to store and select images by longitude and latitude. Storing and selecting against numeric values is possible in SimpleDB. However you need to convert your numeric values into strings using negative number offsets and zero padding to transform numbers into selectable strings. So SimpleDB's UTF-8 string constraint was awkward to program against. A second short coming of SimpleDB is that it limits select results to 2500 values. This is a major problem as it might take more than 2500 images to create a mosaic.

Amazon's second database option, EC2 Relational Database AMIs, are simply virtual machines with databases preloaded. This is the most flexible option offered by Amazon as it allows you to create and configure the database in any manner you want. However our group wanted to explore Amazon specific technologies we did not pursue this option.

We decided to utilize Amazon RDS [3] for our project. Amazon RDS is a MySQL database. The benefit of Amazon RDS is that many administrative tasks are managed by Amazon. Amazon backs up your data, installs software updates. For an additional fee Amazon will replicate data across multiple availability zones and provide automatic fail over.

Underneath the hood Amazon RDS is just a MySQL database. It is a relational database with all of the powerful features that that entails (multiple tables, constraints, indexes, …). As Amazon RDS is a MySQL database you can use any MySQL compatible tool to manage Amazon RDS. This allows you to utilize powerful well established tools like MySQL Admin to create and fine tune your database. Amazon RDS is essentially a mixture of SimpleDB and Relational AMIs. Like SimpleDB, AmazonRDS has numerous tools which simplify the administration of your distributed database. Like AMIs, AmazonRDS is a full fledged database

4.3 Database Design

The Geo-Mosaic application has very simple database requirements. The application requires an image URL and information about where the image was taken (longitude and latitude). The application also needs average red, blue and green color information. The color information is used by the Mosaic Creation sub-system to select the best images for creating a Mosaic. As our data requirements are so simple our database only consists of one table. The primary key of the table is the image URL. The table contains the following columns: longitude, latitude, red, blue and green.

4.4 Web Service Technology Stack

The second part of the image storage sub-system is the web services that allow the other sub-systems to populate and select from the database.

The application contains two web services. The first web service allows the Image Locator Sub-system to add new images or update old images in the database. The second web service is used by the Mosaic Creation Sub-system to select images from the database. The select web service requires the caller to specify a longitude, latitude and number of images to select. The service then selects images near the specified longitude and latitude. If the number of images selected does not exceed the number of images requested by the caller the web service executes up to two more selects with broader ranges. If at the end of three selects the web service will select images from around the world. For example, if the web service caller requested images near the pink dot, the web service would first select images from the green region. If the first select did not retrieve enough images the web service would select from yellow and then red regions.

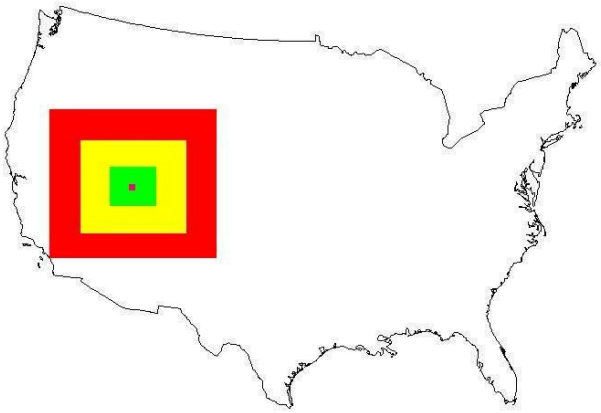

Figure 9 Search Expansion Pattern for Image Searches

4.5 Web Service Tests

The web services were created using Groovy on Grails. The Groovy application was deployed on Amazon's beanstalk [4]. In addition to standard unit and integration tests that directly test the groovy applications code our group created load tests. These load tests were designed to test the performance of the application when it was deployed to Amazon's beanstalk. The web services load tests were created and executed using SoapUI [5].

The load tests were relatively straight forward. 10 threads called our web services for 5 minutes. To emulate realistic loads 90 percent of the calls were to the save image web service. 10 percent of the calls were to the select images web service. Each thread paused between 50 and 150 milliseconds between calls. All of the web service parameters were randomly generated for each call.

Amazon's Beanstalk service managed our application as advertised. When the user defined threshold for average bytes per minute was exceeded Amazon created an additional instance of our application to handle the increased computation load. After the load test finished Amazon removed the additional instance after the specified cool down period.

We ran our load tests twice. For the first test our application was deployed to a micro EC2 instance. During the second test our application was deployed to a medium EC2 instance. The point of the two tests was to determine if our web service calls were executed faster when we had a more powerful EC2 instance. The results of the web service select calls are shown in the table below (all times are in milliseconds).

Table 1 Query Performance by EC2 instance type

Instance Type	Min select time	Max select time	Average Select Time
micro	162	30383	2109
medium	142	3671	937

As expected calls to the more powerful medium instance were quicker than they were to micro instances.

5. Mosaic Creation Sub-System

There are a number of mosaic creation libraries available as open source software. The initial implementation was based on [5], however it has been rewritten in PHP to make use of native C image processing libraries. The rewrite was largely a simplified version of [5] since we chose to allow duplicate tiles outside a specific bounding box. The built-in PHP SOAP processing library was found to be defective at receiving chunked HTTP streams, so the nusoap library was integrated. This brought additional challenges in terms of memory use so the actual SOAP HTTP request is performed by issuing a cURL function call and the XML response is parsed by the nusoap library. [6] [7]

The creation sub-system is outlined in figure 10.

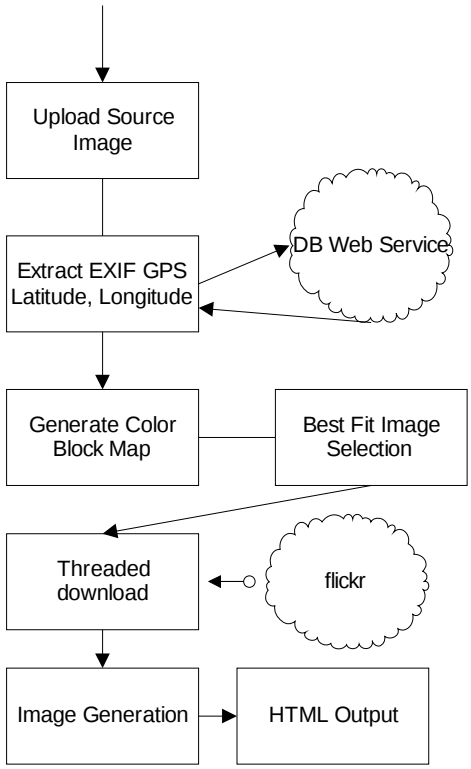

Figure 10 Image Mosaic Front-end Flow

When a user request for a mosaic is received (via file upload), the following steps happen:

1) The file is parsed to extract the EXIF GPS data
2) The web service is then queried for a list of images around the extracted GPS coordinates
3) The file is segmented in to blocks, and for each block the average color is computed
4) Based on the data returned from the web service (which has previously computed average image color values), a best fit is selected where
    a. The absolute value difference of each component is summed to provide an error term
    b. If this is lower than the current error term, the image is checked to verify it will not already be used in the surrounding N pictures
5) Once the images have been selected, a unique list of images to be downloaded is then built
6) A multi-threaded download of the images caches them on local disk
7) Once all images have been downloaded, the final image is stitched together based on the block to image mapping

5.1 Problems with Mosaic creation

Downloading hundreds of images takes time, even with big data center bandwidth it takes about 5-8 seconds to download 1,000 images. Issuing hundreds of HTTP requests to the same host can look like a denial of service attack so requests must be throttled to avoid triggering any countermeasures. In our testing flickr allowed, with no failure, tens of requests per host before denying future requests. To avoid problems we settled on issuing up to 35 simultaneous requests for images.

Memory consumption will be high (both in terms of disk drive storage and RAM) per client. Each image is typically between 40KB and 200 KB. For a typical mosaic creation the service will cache between 500 and 1300 images resulting in 4MB to 20 megabytes of data per request being temporarily stored. While generating the final image the HTTP process will consume around 3% of available RAM while downloading images and peak at 13% while creating images. In our case, that means each request will peak at about 130MB.

The final and largest problem we encountered is CPU utilization. Each client process fully consumed one core of our server for an extended period of time. This occurred during the final image generation, and accounts for the majority of time that a request takes to process. That means that any given c1.medium node from Amazon would be able to serve 2 clients, dedicating one CPU or core to each client.

Currently the front end mosaic generation is run on an Amazon EC2 c1.medium, which is the compute heavy variety Amazon provides for. This process takes, on average, 40 seconds for a single request or 70 seconds for multiple concurrent requests primarily due to CPU resource contention.

Since the web crawler and database levels performed sufficiently for their respective tasks a good strategy would involve offloading more of the final image generation tasks to these nodes. The web crawler could be tweaked to build a distributed hash table, where the key is the image URL (as currently) and the value is the image bytes. The mosaic program would be located in the same data-center to make efficient use of local bandwidth to retrieve the selected images. Searching for color matches in the database layer by weighting color match higher and distance from source lower would provide a larger set of images to select from, while reducing the computations required on the image creation nodes.

GPU acceleration would make image manipulation trivial, but require more expensive virtual machines to host the service.

The image manipulation for this task is inherently parallel, no process would need to access any shared data except for the output image. Using compute nodes to scale the image could increase throughput at the cost of higher internal bandwidth and larger complexity. There are some uses of Hadoop in such a manner [8] [9] [10].

We also discovered a few cases where the GPS coordinates returned from flickr placed an image in the wrong location. We found a few images where the user had tagged the location as Antarctica when they claim to have tagged it in Pennsylvania. We have also seem images which should probably be filtered from typical results. Both cases remind us that dealing with random user contributed images can be risky in terms of accuracy and general population suitability.

5.2 Related Work

[11] Provides an improved matching system which will give visually more accurate results by utilizing feature recognition, such a technique could be adapted to provide better block mapping to retrieve images which have feature similar to the original block features.

[12] Is a very similar system,

6. Conclusions

The final stage of the Geo Mosaic project turned out to be significantly more challenging than expected. Image manipulation on the Java(TM) platform lacks performance when using portable libraries, and is tricky when using native libraries. Even with the problems encountered we believe that scaling the image generation is possible, though the added costs of more nodes or GPU-enabled nodes might outweigh any potential business case. One could imagine an interactive map, where clicking on a sub-tile would generate a new mosaic, each time getting further away from the original coordinates but allowing the user to visually explore a region through a series of image mosaics.

7. Acknowledgements

We would like to acknowledge Amazon for providing a generous grant that allowed us to host our project on their cloud. We would also like to thank the PHP community for its ample examples utilizing cURL and the GD image library

8. References

[1] Flickr API Documentation
http://www.flickr.com/services/api/

[2] Amazon SimpleDB Developer guide.
http://docs.amazonwebservices.com/AmazonSimpleDB/latest/DeveloperGuide/

[3] Amazon Relational Database Service Developer guide
http://docs.amazonwebservices.com/AmazonRDS/latest/DeveloperGuide/

[4] Amazon Elastic Beanstalk
http://aws.amazon.com/elasticbeanstalk/

[5] JImageMosaic http://jimage-mosaic.sourceforge.net/

[6] Soap UI homepage. http://www.soapui.org/

[7] http://www.scottnichol.com/nusoapintro.htm

[8] http://escience.washington.edu/get-help-now/astronomical-image-processing-hadoop

[9] https://docs.google.com/present/view?id=dc4kjnw4_65dswgs9t7

[10] http://www.slideshare.net/ydn/8-image-stackinghadoopsummit2010

[11] Chungnam Lee; Hsu, C.C.; Yai, S.B.; Chuang, W.C.; Lin, W.C. 1998. Distributed robust image mosaics. *Pattern Recognition, 1998. Proceedings, Fourteenth International Conference on.*