

Technical Report CS14-11

**Approximate Algorithmic Image  
Matching to Reduce Online Storage  
Overhead of User Submitted Images**

Braden D. Licastro

Submitted to the Faculty of  
The Department of Computer Science

Project Director: Dr. Robert Roos  
Second Reader: Dr. Gregory Kapfhammer

Allegheny College  
2014

*I hereby recognize and pledge to fulfill my  
responsibilities as defined in the Honor Code, and  
to maintain the integrity of both myself and the  
college community as a whole.*

---

Braden D. Licastro

Copyright © 2014  
Braden D. Licastro  
All rights reserved

**BRADEN D. LICASTRO.** Approximate Algorithmic Image Matching to  
Reduce Online Storage Overhead of User Submitted Images.  
(Under the direction of Dr. Robert Roos.)

## ABSTRACT

Reducing the number of duplicate images uploaded to public servers is an ever more relevant problem as the number of images shared increases dramatically every day. Methods of data reduction such as file expiration dates only lessen this load by a small amount while common methods of image matching are in many cases resource exhaustive, time consuming, or highly inaccurate. This research utilizes technologies capable of identifying near-duplicate images through file hashing, pixel difference, and histogram comparisons. In order to evaluate the feasibility of implementing such a system, a basic photo sharing website has been developed and run on a fixed collection of images.

# Dedication

To Professor Cupper. He was more than just an advisor and professor; he was a member of the Alden family and a father away from home.

# Acknowledgements

I would like to thank my thesis advisor, Professor Roos for the time, expertise, and guidance he provided me as I worked through this project. I would also like to thank my family and girlfriend for their support and encouragement that kept me going until the end.

# Contents

<b>Acknowledgements</b>	<b>v</b>
<b>List of Figures</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Image Hosting Services . . . . .	1
1.2 Data Management Technologies . . . . .	3
1.3 Motivation . . . . .	4
1.4 Goals of the Project . . . . .	5
1.5 Thesis Outline . . . . .	6
<b>2 Related Work</b>	<b>7</b>
2.1 Primary Sources . . . . .	7
<b>3 Method of Approach</b>	<b>12</b>
3.1 Server Configuration . . . . .	12
3.2 Website Design . . . . .	16
3.3 Color Profile and Histogram Comparison . . . . .	23
3.4 Threats to Validity . . . . .	26
<b>4 Results and Evaluation</b>	<b>28</b>
4.1 Functionality . . . . .	29
4.2 Performance . . . . .	31

4.3	Conclusion . . . . .	36
<b>5</b>	<b>Discussion and Future Work</b>	<b>37</b>
5.1	Summary of Results . . . . .	37
5.2	Future Work . . . . .	37
5.3	Conclusion . . . . .	37
<b>A</b>	<b>Duplicate Image Detection Code</b>	<b>38</b>
A.1	Upload Procedures . . . . .	38
A.2	Exact Duplicate Detection Code . . . . .	53
A.3	Approximate Duplicate Detection Code . . . . .	55
<b>B</b>	<b>Website Framework Code</b>	<b>62</b>
<b>C</b>	<b>Website Framework Code</b>	<b>63</b>
<b>D</b>	<b>Miscellaneous Code and Configurations</b>	<b>64</b>
	<b>Bibliography</b>	<b>65</b>

# List of Figures

1.1	Various Image Hosting Service Structures . . . . .	2
2.1	Possible image manipulations . . . . .	8
3.1	Proposed schema for file uploads . . . . .	18
3.2	Successful image upload response when no duplicate is located. . . . .	21
3.3	Successful image upload response when a duplicate is located. . . . .	22
3.4	Streamlined upload process . . . . .	23
3.5	Visual representation of an image's histogram . . . . .	25
4.1	Initial state of the upload form upon page load. . . . .	29
4.2	Error displayed when a user attempts to upload a non-jpeg file. . . . .	29
4.3	Error displayed when attempting to upload a jpeg image that is too large. . . . .	30
4.4	Image processing after upload completes. . . . .	31
4.5	Processing time on data set without duplicates. . . . .	32
4.6	Processing time on data set with 20% duplicate images. . . . .	32
4.7	Average of 5 tests using different groups of sub 1MB images. . . . .	34
4.8	Average of 5 tests using different groups of greater than 1MB images.	35

# Chapter 1

## Introduction

Sharing media with the public is becoming a more integral part of social interaction every day. Static images are just one of these many forms of media, and the number of daily uploads to image hosting websites is absolutely staggering. According to a recent survey by All Things Digital, as of May 2013, more than 500 million images are uploaded to image sharing websites each day, and this number is expected to double by the end of 2014 [5]. With figures this large, it immediately becomes apparent that multiple issues come with this trend of increased image sharing, namely, how much space does this number of images require, and is there a technology available to reduce this requirement? The simple answer is yes, there are tried and tested technologies that will reduce storage costs, but before looking into these technologies, it is important to understand what image hosting websites are and how they function.

### 1.1 Image Hosting Services

Image hosting services, or image sharing websites, are sites that allow users to upload images to the Internet and share them publicly with the link they are provided. These image sharing websites mostly operate in the same fashion, but recently a new breed has emerged. As seen in Figure 1.1, both submission processes are similar, but both

have inherent advantages and disadvantages.

Looking at the first submission process variant at the top of Figure 1.1, a user would like to share an image with the public. The user can upload this image through the Internet from any Internet connected device, and the image will be stored on a publicly accessible server. From here, any number of people can access this shared image through an Internet-connected device indefinitely. Nearly all image hosting services operate on this model. Some of the more popular services include Flickr, Imgur, Photobucket, Shutterfly, and Instagram.

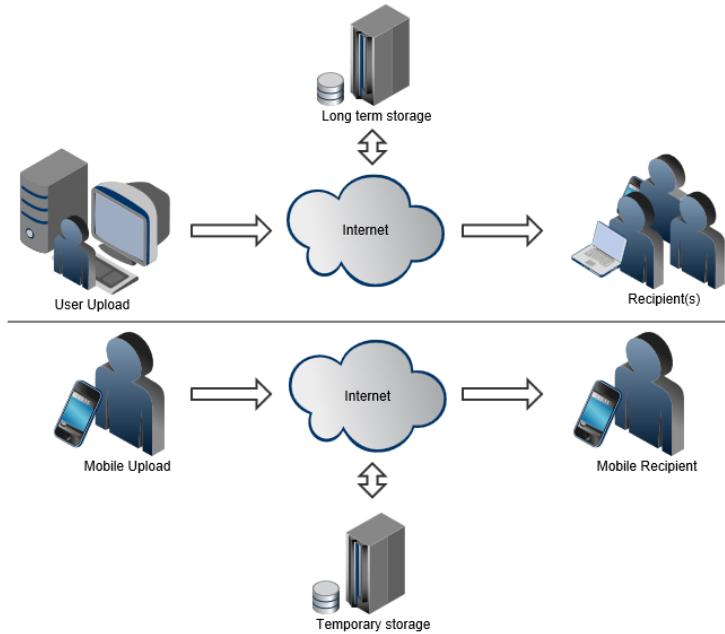


Figure 1.1: Various Image Hosting Service Structures

The second image submission variant shown at the bottom half of Figure 1.1 is currently, as of December 2013, only used by one host called Snapchat. With this system, multiple differences are immediately apparent. First, this service model will accept images only from mobile users. It can also be seen in the diagram that when a user uploads an image through the Internet, it is no longer accessible long term from a server, but it is in fact only temporarily available for a specified recipient to

receive. This service actually allows the uploader to set an expiration time anywhere from 1 second to 10 seconds [10]. After the recipient opens the image and this time period has expired, the image is deleted permanently from the server and no longer accessible to either party.

While both systems have their own inherent benefits, they also have drawbacks. Although they reach from the infrastructure needed to support the service all the way out to the end user, this research focuses only on the infrastructure function. In order to realize the application of the proposed research, it is important to understand why such a focused application is required. For the first image hosting variant in Figure 1.1, the long term storage of image files raises concern. By storing files for an extended period of time, it is a guarantee that duplicate data will make its way to the storage servers given enough time. Determining duplicates and preventing their addition can save not only a considerable amount of space as the amount of redundant data increases, but it can also save a large amount of money when discussing the cost per gigabyte of data storage which will be discussed in Section 1.3. This research will not target the application of the temporary system seen at the bottom of the figure as the amount of stored data is comparatively minimal due to the rapid turnover of data being housed.

## 1.2 Data Management Technologies

The image hosting services outlined in Section 1.1 utilize numerous technologies to help lessen the load of the files they must store. Although articles discussing the technologies these companies implement are nonexistent, after examining the services several different technologies became immediately apparent. The most prominent method of space reduction is the reduction in size of uploaded images. By reducing the quality of the image by a small percent, these websites are able to significantly reduce

the amount of space needed to store the uploaded images. The next most common method of space reduction used is the expiration of uploads. After locating the earliest available images across the websites it was determined that the file expiration can possibly work in one of two ways. The implementation either functions as a countdown from the date of upload or in the other case the countdown begins after the last date the file was accessed. Each time the file is accessed, the timer will reset. Finally, some of these sites also limit the size of the uploaded files and the number of uploads per user. In order to remove these restrictions, many sites also offer paid services which help offset the cost of upkeep.

### 1.3 Motivation

Summarizing the information presented thus far, image hosting websites clearly require complex infrastructures, not only to allow the sharing of files but to manage the large numbers of submissions. Although the numerous technologies used to lessen the space requirements of the submissions work well, it should be possible to further improve on the system by reducing the number of duplicate submissions. Remembering that more than 500 million images are uploaded to long-term image sharing websites each day, we have a baseline to calculate potential savings from. In addition, a study published by NTP Software found that nearly 20% of stored data is duplicate [15]. These numbers are staggering, especially when there is a possibility of reducing an additional 100 million images of required storage space.

To bring the possible savings into light a rough calculation based on the 2013 average of \$.05 per gigabyte and an assumed image size of 1MB will show that by removing the duplicates a significant amount of money could be saved. More specifically, approximately \$1.8 million can be saved annually at the current sharing levels. This number was calculated using the equation:

$$\frac{((Images_{Total} * \frac{\%Duplicate}{100}) * 365.25) * ImageSize_{Average}}{1024} * CostPerGB = TotalSavings$$

Multiplying the total number of images uploaded each year by the percent duplicate we can find an approximation of how many duplicate images are stored annually. By multiplying this by the average image size we will know approximately how many megabytes of space this duplicate data will require to be stored. Finally, if we convert the cost per gigabyte to cost per megabyte to store the data and multiply that by the number of megabytes of data, the final cost per year to host this duplicate image data can be obtained. By allowing unregulated user submitted data, it quickly becomes apparent that data redundancy reduction can save a significant amount of storage space and money.

## 1.4 Goals of the Project

This project develops and tests a system capable of efficiently identifying and reducing the number of duplicate image submissions on an image sharing website. In order to fulfill this goal, an image sharing website was developed which is capable of accepting images as uploads and providing a link to the user for public access of the image. To develop a functional website a set of requirements were met allowing it to perform several functions. The website was designed to accept an image file through an online submission form. At this point, it processes the image and matches it against a collection of images currently stored on the server. This process is completed using a series of algorithms and checks outlined in Section 3. The website was developed using PHP, the GD image library, and HTML5 to ensure minimal conflicts between scripts and languages. Using this site, the duplicate image detection tool was implemented using both original and existing code from other resources and tested thoroughly to assess the effectiveness of using this method of duplicate reduction.

## 1.5 Thesis Outline

The remainder of this thesis will discuss the work in further detail in addition to existing information relating to the topic. Chapter 2 covers related works and existing research that has been completed on the topic of image matching and comparison. Chapter 3 covers the project details pertaining to the website and supporting infrastructure it will operate on. This includes a brief discussion of the hardware and configuration of the web server in addition to the website design and implementation of the duplicate image detection tool that will be integrated and tested. Chapter 4 discusses the collected results and the accompanying evaluation of the collected data. This evaluation includes the performance costs of implementing this system over a passive one, which only acts to prevent a specific file from ever being submitted to the server. The evaluation will also include the results of the image de-duplication and a determination of whether such a system is a feasible method of reducing storage requirements. An additional section outlining threats to validity will also be included in this chapter. Finally, Chapter 5 summarizes the research and conclusions completed throughout this senior comprehensive project and include possible areas of future work.

# **Chapter 2**

## **Related Work**

As expressed in the previous chapter, storage space on Internet accessible servers is at a premium when users have the ability to freely add content. In order to reduce the cost of operating image sharing websites it is necessary to develop and utilize technologies to reduce the space requirements. It is unsurprising that a large amount of research exists which targets the problem of locating similar images. Often, these comparison techniques were developed to be run on a local computer and not over the Internet, although the methods could be adapted to operate on a web based platform in many cases. By utilizing a combination of the research found in this chapter, it is possible to build a functional system capable of eliminating a large percentage of duplicate images on the server. When choosing a comparison method several characteristics were evaluated, namely time efficiency, accuracy, and computing resource requirements.

### **2.1 Primary Sources**

Although comparing existing images located in a database is not directly relevant to this research, the paper by Lee, Ke, and Isard [11] can be useful in method of approach. During their research they utilized a partition min-hash function for discovering near-

duplicate images. Partition by hash functions are used to break down a large data set into a number of equal sized segments that can be identified by a generated hash. Instead of looking at the image as a whole as many other algorithms do, it looks at the image as several smaller pieces through the use of the partition hashing function. They claim that the algorithm is actually faster and more effective than standard functions which look at the entire image. Since the research is highly efficiency sensitive due to its web based structure, implementing a variant of this comparison method could prove beneficial.

Another article that is vital to creating an effective and efficient image matching algorithm is the 2010 paper by Srinivasan. This research was targeted at web-based image matching, which is directly related to this research. This research claims that traditional near-duplicate detection systems are not applicable for the de-duplication of large-scale web image collections [16], the research performed targeted an image matching system which was scalable, highly efficient, and effective.

In order to perform the effective image matching the authors required, they decided to implement a thumbnail matching based system. This algorithm would generate a 130-bit thumbnail representation of the image and was capable of finding near-duplicate images while operating in  $O(1)$  time [16].

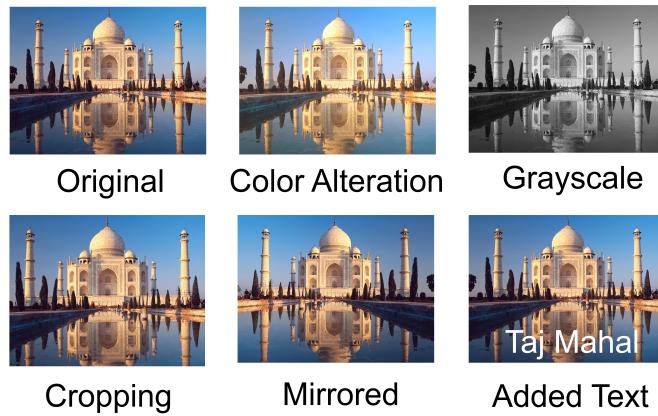


Figure 2.1: Possible image manipulations

When using this thumbnail method, the authors were able to adjust automatically for differences in resolution, arbitrary amounts of cropping, caption, logo, and other manipulations, in addition to color and rotation variations, examples which can be seen in Figure 2.1. Implementing a variant of this method was useful as each of these are highly probable manipulations in a real world scenario, and have been included in the final tests performed on the implemented system.

Due to the web based nature of the proposed research, another article, written by Foo, Zobel, Sinha, and Tahaghoghi in 2007 is highly related to the proposed research. This paper discusses the topic of web search based image matching [7], mainly the finding and determination of types of copyright infringement as most near-duplicate images are derived from one source. The goal of their research was not to derive an algorithm capable of matching images, but to locate duplicate and near-duplicate images on the web using search engines, and determine the most common methods of image alteration leading to redundancy and possibly copyright infringement. Their research was invaluable to the proposed research as it allows targeting of the most common methods of image alteration when creating a matching algorithm. To locate their image set to work with, they used the most popular search queries of 2005 and collected the number of images, and determined the number of unique images based on the results returned. As a note, images with non-unique URLs were not included to prevent false duplicates from the same sites. In conclusion, the authors found that the most common alterations, ordered from one to 10, one being the most common, were as follows [7]:

1. **Combination:** Images with more than one alteration.
2. **Scale:** Images that differ in size.
3. **Crop/Zoom:** Images that are cropped from the original.

4. **Picture in Picture:** Images that contain another image.
5. **Contrast:** Images that have adjusted contrast.
6. **Border/Frame:** Images that have added borders or frames.
7. **Grayscale:** Images that have been converted to grayscale.
8. **Recoloring:** Images with colors that have been modified.
9. **Mirror:** Images that have been mirrored to prevent copyright infringement detection.
10. **Rotate:** Images that have been rotated.

The research does not focus on every one of these common parameters that have been found, but will focus on a select group of these common alterations outlined in Section 4.

Finally, the most relevant information found provides a starting codebase for the algorithm and research. An online coding tutorial provider, CatPa, outlined a method of using PHP libraries to generate real-time thumbnails and hashes of images and compare them to ones currently located on a server before uploading [3]. If the image is a duplicate it would be denied the privilege of being uploaded and the function would notify the user. This system is extremely limited as it provides the user with no way of adding an image to the server if it is not a duplicate and is only a false positive, and if it is unique, does not provide them with a method of accessing the already-present file.

The image comparison code developed by CatPa has been used as a base to generate a fully functional image sharing site with duplicate-reduction systems and allow for the testing of the effectiveness of such a system over traditional methods of sharing with duplicates allowed. The system created by the author [3] generates a

$16 \times 16$  thumbnail of each image on the server, and of the image being uploaded. From here, the algorithm generates and compares black and white and color histograms, the thumbnails, and determine if it is a duplicate based on the allowable difference threshold setting which was determined after running the initial tests outlined in Section 4 and analyzing the results [3]. The research utilizes this exact method, but also provides code improvements to utilize less memory by implementing the PHP Improved libraries and also checking images for immediate similarities in resolution and exact image matching with a full MD5 image hash. These methods require minimal calculation, just a simple comparison of strings compared to the generation of histograms and re-sizing of images. After the initial string comparison approach, the code was used to calculate the average deviation between the two images and will allow easy expansion of the system to provide a faster, more effective image matching system.

# **Chapter 3**

## **Method of Approach**

In order to create an effective image matching algorithm, it was necessary to pull research, knowledge, and pre-existing code from a number of resources in addition to completing original code. Although the research does not target the hardware infrastructure of the network that image sharing sites use, a simple public web server was built and configured. This topic will be briefly discussed in Section 3.1 To test the proposed image matching method, it was also necessary to build a skeleton that would act as a simple image sharing website. This skeleton is capable of accepting an image file and returning a link to the user which will allow the submission to be viewed. The site also allows for the development of the proposed matching function and will provide the necessary functions to accept and handle the output of the research.

### **3.1 Server Configuration**

In order to implement the website, it is necessary to have an environment capable of running the required processes. To do this, several options were considered. The first option was to run the site on a locally installed Apache web server. After a small amount of testing, this was determined not to be the best option. Due to the lack of a dedicated machine the web server had wildly varying performance due to

interference from other processes that could not be closed. This was even a problem when operating with a fixed amount of memory. In addition to this, the XAMPP environment was also tested for performance. This environment gets its name from the functionality it provides pre-configured and out of the box. In other words, X (Cross-Platform) Apache HTTP Server with MySQL, PHP, and Perl. This package functions across any operating system platform as a fully functional web server. In many cases the configuration is acceptable as is, but in the case of this research, even after changing resource limits and other settings the software still frequently became non-responsive with resource hungry processes such as analysis of large files.

The next alternative was to purchase web space on a shared server. These servers are readily available at minimal cost from dozens of providers such as A Small Orange, BlueHost, and Byethost. After careful consideration several concerns prevented the use of this method. The primary factor was that these servers are shared with numerous users. Due to the nature of this setup, it is unknown how many websites are being hosted by a particular server, and how many concurrent users are accessing these sites at any given time. In addition, it is not possible to set an allotted amount of memory for a particular environment or control what programs are running that may possibly impact the performance of the research.

Another option that looked very promising at first was to rent server time from a provider such as Amazon Web Services. With this method not only it is possible to control what programs are operational, but it also allows for more freedom of configuration. This would seem like the most promising solution to the problem, but there are still factors that cannot be eliminated, such as the speed of the Internet connection at any given moment. By hosting a local web server, it is possible to control this factor, but the issue remains of how much it could affect the results. To test this concern, a server was rented using the free tier of services. From this point,

a simple timer was configured and one 15 megabyte file was transferred using SFTP. This process was repeated 10 times and the results were analyzed. After reviewing the results, there was no concerning transfer speed variance, making this a good match. In the end, it was discovered that there are restrictions on the service that do not allow an individual to alter server settings relating to resource allocation, which was a key concern from the start.

The final option was to implement a local dedicated web server and operate the website and scripts from that. The machine allows not only very tight control over variables such as resource allocation, installed programs, and custom network hardware configuration, but it also allows usage on a local network or over an internet connection. By running initial tests on a local area network, it is possible to eliminate Internet speed fluctuations and control the number of devices utilizing the network bandwidth. This also opens another path where a real world simulation can be run by submitting images to the system over an internet connection and comparing the behavior with only one variable at a time differing.

To build the server a specification had to be determined that would allow optimal performance. To allow the greatest flexibility, the Ubuntu Server operating system was chosen. From this, the server's hardware specifications were chosen based on the minimum system requirements given by the operating system, Apache suite, and MySQL Database. This information was used to pick a quantity of memory, hard disk space, and processor speed. The server was built with 4 gigabytes of random access memory (RAM), 3 gigabytes allocated to the programs and operating system, a 2.43 GHz Intel Core i3 processor, and dual 7200 RPM 500 gigabyte hard drives. The integrated network card was faster than the available network equipment, so it was not of direct concern. The hard disks were chosen to provide ample space for any reasonable number of tests but not provide so much space as to be considered excessive

for their purpose. A 7200 RPM variant was also chosen to allow the maximum data throughput and not become a bottleneck when working with great numbers of large image files. Finally, the disks were configured as a redundant mirror to emulate a simple backup system that duplicates the data as a form of backup. This allows the collection of data and comparison of storage requirement improvements in a non-redundant system, and one with a worst case scenario backup implementation that simply creates copies of the files.

The software selection is more straightforward. After selecting to use a Linux operating system, Ubuntu Server was chosen as the distribution. I had the most experience with using, configuring, and troubleshooting issues with this environment and decided it was best for this reason. The accompanying software was fairly simple to select as a quick Google search for “Ubuntu web server” will return thousands of results outlining the setup and configuration of a basic Linux-Apache-MySQL-PHP, or LAMP server. The setup process was completed step by step using the ApacheMySQLPHP LAMP Server Setup Guide provided through the Ubuntu Documentation [6]. Next, the code which will be discussed shortly requires that the PHP GD Image Library be installed. To prepare the PHP installation to use this library, the server was configured using the direction of the tutorial hosted by nixCraft [14].

Upon the completion of the prior configurations, the server was updated and running the latest version of all installed software. To prevent updates from altering the outcomes of the future, a hold was placed on all packages to prevent all non system security related updates from being installed. In addition to this, the firewall was configured to allow HTTP communication through port 80, which allows interaction through an internet browser with the website hosted on the server. This was followed by a reconfiguration of the connected network router giving access to the internet through port 80. MySQL did not require any setup past the installation of the

program and was left alone. At this point, the server configuration was complete and the default Apache “Success” page was displayed when accessing the server showing that everything was working properly.

## 3.2 Website Design

The core of this research hinges on the successful implementation of an image comparison algorithm. In order to do this, a website needed to be developed that acted in a manner similar to a simple image sharing site. In order to do this, a specific demographic of users had been targeted. Due to the code limitations of the PHP GD Duplicate Image Finder written by CatPa [3], only jpeg submissions are accepted for the purpose of this research. In addition to this, a 15 MB file size limit is enforced. This is to prevent excessive wait times when transferring the image file to the server during tests run on a large number of files. The website was designed to be lightweight; more specifically, it has no extraneous scripts or applets running on the upload page. The purpose of this is to give processing times relating only to the actual upload process and not nonessential scripts. Finally, the last restriction placed on the website development is cross browser compatibility. Instead of placing focus on making a website that functions across all common web browsers, it was decided it would be highly beneficial to work with the Gecko browser platform, such as Firefox, due to the vast array of development tools available. This limitation will also allow for a focused effort in file management on a specific platform and will allow room for further research after the tool has been optimized.

To begin, a database was implemented in such a way that it holds a vast array of information. The image sharing functionality of the website requires several different pieces of information to be stored in a database. The first column of the database houses the identifier for each entry. This is known as the primary key, which is a

column in a database where all entries must be unique and the key is used to identify the information in the row. This identifier is used by the website to track the order of the submissions to the server since the date uploaded is not important to the research and will not be tracked. The next column, as seen in Figure 3.1, is the `ILookup` column. This is what the website uses to look up each image location on the server when supplied with a URL containing this identification code. This column must always have a value so the `NOT NULL` flag was set. The third column is `IName`, the column that houses the actual file name of the image being stored on the server. This column must also have a value so the `NOT NULL` flag was set. This is concatenated to the end of the `directory` entry which cannot be null, and provides the website with the exact location of the image file on the server with respect to the Linux root directory.

Each of the remaining database columns are specific to the image de-duplication scripts. First, the `uMethod` column is nothing other than a single integer that marks what upload method was used to place the image on the server. If this number is set to “0”, the image was uploaded using the non-duplicate reducing functions. On the other hand, if this number is set to “1” it is known that the image was uploaded and checked for duplicates before committing to long term storage. Both the `hash` and `fingerprint` fields contain information that allows the scripts to rapidly search the database for duplicate files when a new file is uploaded. Both of these columns are allowed to have a null value. This is allowable due to the fact that non duplicate reduced images will not have any image matching data associated with them. These non duplicate reduced images will be discussed in Section 4 when discussing the base case that the results will be compared to. Finally, the last column tracks the total time in milliseconds that it took to upload the image to the server. This is not allowed to be null as both the base case and the duplicate reduced case will

require this information. In Section 4, this data will be used to compare the resulting data gathered by uploading images in a traditional manner and using this duplicate reduced function being researched.

Database: thesisDB   Table: shareTracker			
Name:	Type:	Description:	Extra
<b>ID</b>	int(11)	<i>Gives every image a unique ID</i>	Primary Key Auto-Increment
<b>ILookup</b>	varchar(6)	<i>Unique URL ID Lookup</i>	Not Null
<b>IName</b>	varchar(21)	<i>Images file name on server</i>	Not Null
<b>directory</b>	varchar(15)	<i>File location from server root</i>	Not Null
<b>uMethod</b>	int(1)	<i>Upload method used</i>	Not Null
<b>hash</b>	varchar(40)	<i>Hash of the image for exact dup matching</i>	
<b>fingerprint</b>	varchar(32)	<i>The MD5 fingerprint of the histogram array.</i>	
<b>processTime</b>	int(11)	<i>Upload time, from start to completion</i>	Not Null

Figure 3.1: Proposed schema for file uploads

In order to test the effectiveness of the research being discussed, a base case must be created against which to compare the results of running the algorithms. To create this data, the website performs one function before moving on to the duplicate reduction test. A separate directory was created on the server where every single image submission will be placed regardless of whether it is duplicate or not. This will mimic the upload process of a non-duplicate reducing image sharing website. Each entry into this folder will be placed in the database exactly the same as the duplicate reduced entries are, but will contain no image identification information. This first process is referenced henceforth as a traditional upload method. This upload method is unknown to the user as it will not return a URL allowing access to the image at a later time. During this process, each file is assigned a new, unique name to prevent collisions upon upload. A SQL "INSERT" will then be run on the database for the image and will record the file's location on the server, the file name, the fact it was uploaded using the traditional method, the unique URL to access it from (which is not

given to the user), and an ID for each insert into the database. After the completion of the process an "UPDATE" will be run on the database entry created by the "INSERT" above. The time taken to complete the task will then be included in that entry and can be used to analyze the efficiency of both systems upon the completion of the tests outlined in Section 4.

After the initial upload completes, a second script is called. The second script has been designed to operate in two steps. This function performs mostly the same tasks as the traditional upload, but checks the image being uploaded for duplicates and handles each case appropriately. First, the image will be matched in the most basic form by hashing the image file using an MD5 file hashing function, after which the resulting hash will be compared to all images in the database using the following SQL Statement: "SELECT \* FROM 'share\_tracker' WHERE 'uMethod' = '1' AND 'hash' = 'fileHash' ". If this statement returns any results relating to a matching hash on a duplicate reduced uploaded image, it has found an exact matching image.

In the event where a match is not found, the script will continue to the second stage of duplicate finding function. This function will operate in several different stages. The first stage toward finding an approximate image match is creating a fingerprint of the image file. In order to do this, the script will create a GD image object by using the PHP GD Image Library that was configured on the server in Section 3.1. After the object is created, the GD Library has built in functions that allow iteration through every pixel in the image and view each pixel's red, green, blue value, or RGB value. These values will then be used to return a fingerprint in the form of an MD5 hash of the color frequency of the image. This color frequency is also known as a color profile. The next step to identifying a duplicate image would be to query the database for any images that have a perfectly matching color profile. This profile will be the same for an image no matter the variation as long as the

image resolution, color and contrast are left unmodified. At this point, if there is no matching color profile, it can be said that there is most likely not a matching image on the server and the function will return no matches found.

If the color profile does in fact match that of one or more images on the server, the script will then make a  $16 \times 16$  pixel full color copy of the image being uploaded in addition to  $16 \times 16$  copies of each of the images with a matching color profile. This process will also utilize functions provided by the GD Library that will allow pixel by pixel comparison of each image. When re-sizing an image, a number of techniques can be used when changing the total number of pixels. One popular method of re-sizing an image is called interpolation. With this process, when reducing the number of pixels, the algorithm will select one pixel and any immediate neighbors to it, at most there will be four. The algorithm will take the values of all five pixels and average them together eliminating the need for the neighbors as the original now represents the values of all five. This process is continued pixel by pixel until the image has reached its desired smaller dimension [1]. This fact will allow the comparison of both images even if they were different dimensions, as the resulting  $16 \times 16$  files will have similar averages after re-sizing if they are in fact a match. Because of a possibility of slight variations in color, this function will not look for exact match, but will allow a set amount of deviation between two thumbnails. To compare the pixels of both images, the RGB values of the corresponding pixels on each image will be analyzed. First, the red value of a pixel will be subtracted from the red value of the corresponding pixel's red value in the other thumbnail. The absolute value of the difference will be recorded. The same process will be completed for the green and blue values of that pixel. Once the difference from that pixel is calculated, the same comparison will be performed on each of the remaining pixels. In order to allow for very slight variations in color with each pixel, the resulting total calculated deviation will be divided by

the number of pixels to ‘normalize’ the result. After that calculation is complete, if the final deviation is less than or equal to the limit set by the website administrator, the function will return that a matching image was found.

If after all of the above processes are completed and no duplicate image is located on the server, the image upload will be accepted by the function and stored on the server’s disk. At this time a SQL "INSERT" will then be run on the database for the image and will record the file’s long term location on the server, the file name, the fact that it was uploaded using the duplicate-reduced method, the unique URL to access it from, the image’s MD5 hashed histogram that is described below, the thumbnail created for histogram comparison, and an ID for each insert into the database. After the completion of the process an "UPDATE" will be run on the database entry created by the "INSERT" above. A view of the functioning duplicate-reduced upload process can be seen in Figure 3.2 when no duplicate image is located.

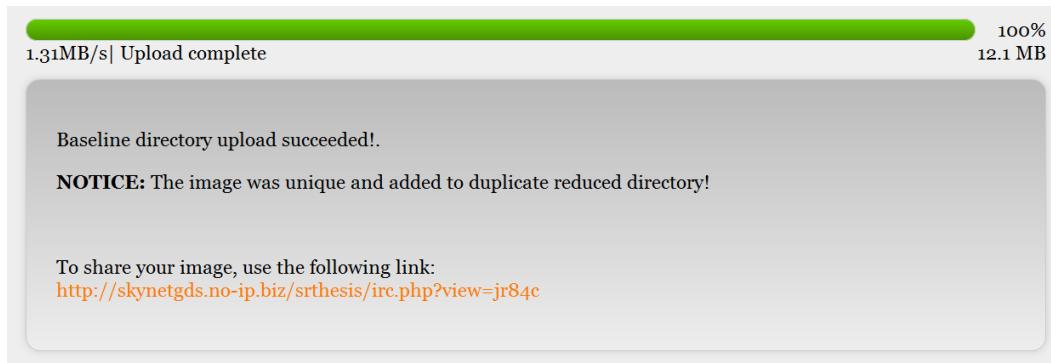


Figure 3.2: Successful image upload response when no duplicate is located.

If the image is determined to be a match after the completion of the full image hash function or the completion of the color profile and  $16 \times 16$  pixel by pixel comparison, the submitted image is assumed to be a duplicate and a prompt will be displayed to the user showing the image they provided and the possible match that already exists. If the image is verified a match, the user will be given a unique link to the

image that is already on the server, and the image being uploaded will be discarded as a duplicate. At this time the system will run an "INSERT" on the database linking the new identifier to the image already on the server, and it will be added to the database. If the user decides the image is not a duplicate, the system will run an "INSERT" statement and it will be added to the database. Following the completion of this process an "UPDATE" will be run on the last "INSERT" and the time taken to complete the task will be included in that entry. A view of the functioning duplicate-reduced upload process can be seen in Figure 3.3 when a duplicate image is located.

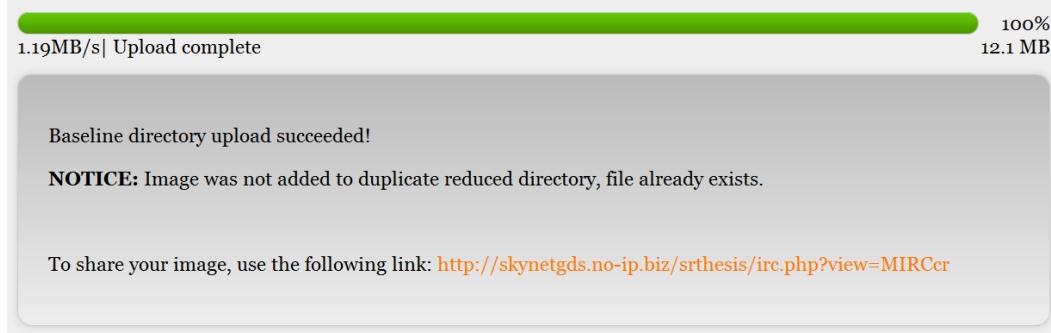


Figure 3.3: Successful image upload response when a duplicate is located.

In the case of requiring duplicate files, where the user decides to upload the image regardless of uniqueness, the script will be able to differentiate between the two images with the unique image ID that is generated at the time of insertion into the database. The closest matching occurrence of an image on the server compared to a new upload will be used in the prompt and displayed to the user. This will prevent frustration with multiple prompts every time more than one duplicate is found. This technique will also allow multiple links that point to the same image while leaving the user unaware of the system operating in the background to provide a consistent experience. An outline of this full process can be seen in Figure 3.4.

When a user accesses the file from the provided link, the system will run a query that looks up the image identifier provided in the URL. If it matches an image on

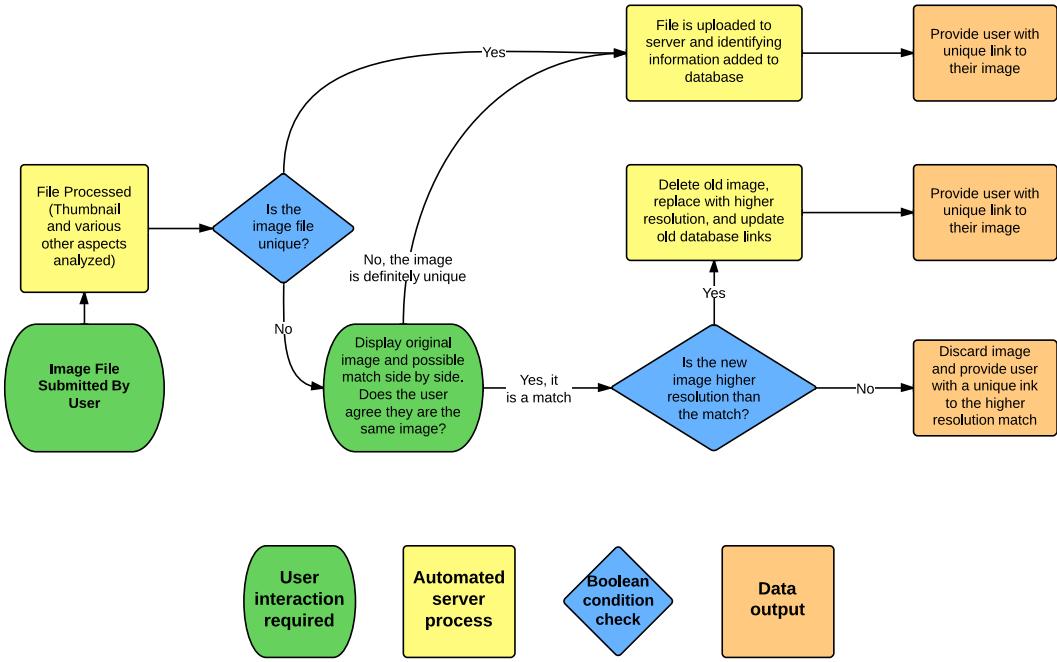


Figure 3.4: Streamlined upload process

the server, the image location will be used to provide the image to the user for viewing. The user never notices a difference, but on the server side we have ensured file redundancy has been eliminated and possibly improved user experience by providing the user with higher quality content than what they were expecting. If the requested image is not found, a 404 “Image cannot be found.” error will be displayed to let the user know something went wrong.

### 3.3 Color Profile and Histogram Comparison

As mentioned in Section 3.2, an image is composed of RGB values associated with every pixel in an image. In order to use these values in an effective manner, a thumbprint of the color profile had to be created. This color profile is represented as a histogram and allows the visualization of each color’s frequency. This section will

break down the process of comparing histograms and their accompanying data and explain the reasoning behind the comparison method chosen for this research.

As mentioned before, each pixel in an image is composed of an RGB value that allows the storage of the correct mixture of each color in that pixel. This value is separated into three separate ranges of 0-255 values where the higher the number the brighter the color. As seen in Figure 3.5, a histogram has been generated using a popular photo editor by the name of Paint.NET. On the right hand side the four different overlaid histograms can be seen. The ‘x’-axis represents the relative number of occurrences of each 0-255 value while the ‘y’-axis represents the values themselves. The red layer indicates the frequency of each red value, the lowest of the 0-255 values being at the bottom of the image, and the highest at the top. The green and blue indicate the same for their respective colors. Also, note that blacks, whites, and grays can be represented by the RGB values. For example 255, 255, 255 represents a solid white pixel while 000, 000, 000 represents a pure black pixel, thus these colors do not require their own array.

This collection of values that make up the histogram is where the comparison process begins. For any comparison of data to occur, each pixel’s RGB values must be recorded into three arrays of size 256. The index of the red, green, and blue arrays will correlate to the 256 possible colors within the respective color. As a color is encountered, a counter in the code will increment the value by one at that array index. This will allow the tracking of the frequency of each of the RGB values. It does not matter what pixel these are related to as the overall number and frequency of each color is the only concern.

At this point, every RGB value in the image should be stored within the color’s respective array. This is the array of values that can be graphed using a histogram, and the visual of the color profile can be printed out for visual inspection. Since we

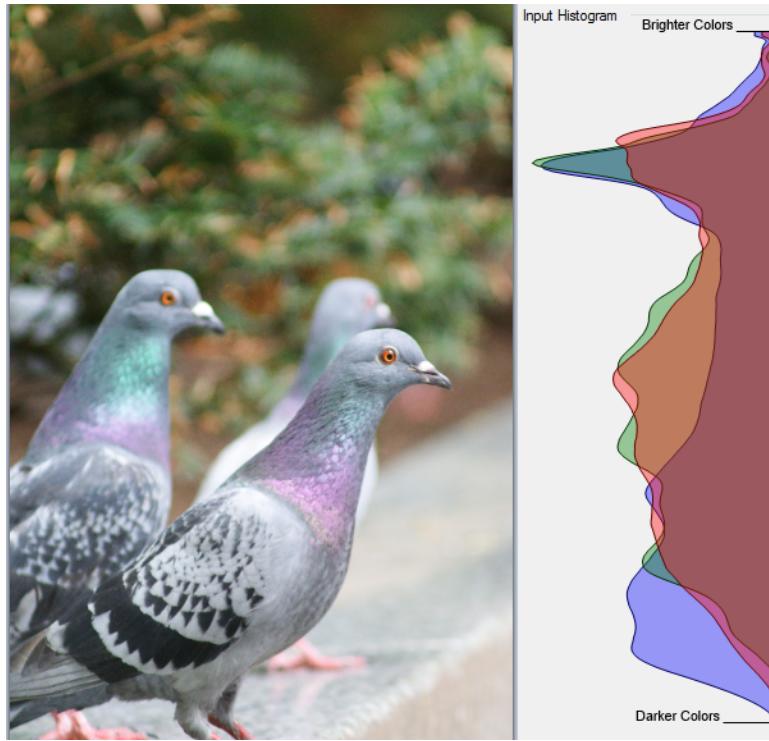


Figure 3.5: Visual representation of an image's histogram

will be using a function to process the values, the actual generation of said histogram would be unnecessary. These values can be handled and compared in a few ways, but the best method for the purpose of this research much be chosen. This array could be processed using the PHP `serialize()` function and stored in the database for later examination. By doing this, each of the arrays would require separate serialization, and all three would require their own columns. With this method, the array could be pulled from the database and run through the `deserialize()` function to return the array to its original state. This would be beneficial for applications that would require the comparison of each individual array index to another array. Due to the large amount of processing time required to iterate over multiple arrays of size 256, this is not optimal. In addition to this, we are not concerned with calculating the difference in color profiles, but are only concerned with an exact match. The use of exact-only matching is valuable with this research topic. If two images with different

color profiles are being compared, it will be assumed that they are unique. An identical color profile could potentially signify a duplicate image as explained below.

The color profile requirement better lends itself to a different method of histogram value comparison. Due to the fact that we need to find only exact matches and that the system accounts for a possible false positive, hashing the three arrays will suffice. For this process, the three arrays will be concatenated in the order `Red.Green.Blue`. This will generate one array with 1020 indexes representing the total relative count of each individual RGB color value. This value will then be taken and processed with the PHP `md5()` function which will return a 40 character alphanumeric string representing the array. When an uploads fingerprint is created, the database will be searched for an exact match hash. If a matching hash is found, it is known that a photo with the same color profile is already on the server. In the event of a hash collision where two different histograms have the same resulting hash, the script will be performing a pixel by pixel comparison anyway and will be able to decide that the images are not unique, so this is not of concern.

Due to the fact that the only images of concern are exact color profile matches, it is not required to store the values of the color profile in a recoverable format. This leads away from storing either a plain text array of values in a database, or storing a serialized version of the array in the database. Because of this, it is acceptable for a 40 character alphanumeric hash to be stored in the database and directly compared to the hash of a newly submitted image in order to detect a possible duplicate image.

## 3.4 Threats to Validity

With any area of research comes inherent shortcomings no matter the care taken to eliminate free variables. By creating a locally hosted server and excluding any extraneous scripts from the website, these variables were controlled to the greatest extent

possible. Due to the nature of a publicly hosted web server, there is always a chance of malicious attack which could skew the results one way or another. As discussed in Chapter 4, four sets of experiments were performed. One test was performed offline using only a standard test image library with a very specific image alteration to each image. This environment gives the best possible chance of gathering unbiased results. The second test performed utilized the same image library, but was run over an internet connection instead of a LAN connection. This allowed for performance testing over a connection of fluctuating load and speed. An identical set of two experiments were tested using a set of 10 photographs taken around the Allegheny College campus during the winter months. This gives a collection of unique images, some of which would inherently have similar color profiles due to the reduced color intensity of the winter months.

Finally, the last variable that was not able to be determined was a possibility of image corruption. Due to the nature of images, it is possible that minor file corruption can occur on physical media. Since this is nothing more than the loss or incorrect representation of data, an image will still process using this system, but may yield incorrect results. In addition, if corruption occurs on a very small number of pixels, it is possible that the image will visually look identical to the original but still differ enough that it will be seen as an original image instead of a duplicate. This corruption can happen due to a momentarily lost connection, a web browser mishandling an HTTP request, or a server side fault. There are currently numerous open bugs in both the PHP language and the Apache server applications, none of which were researched to ensure the proper function of scripts. It was assumed that a functioning script returning expected results during trial runs is a fully operational script when running final tests and analyzing the results.

# Chapter 4

## Results and Evaluation

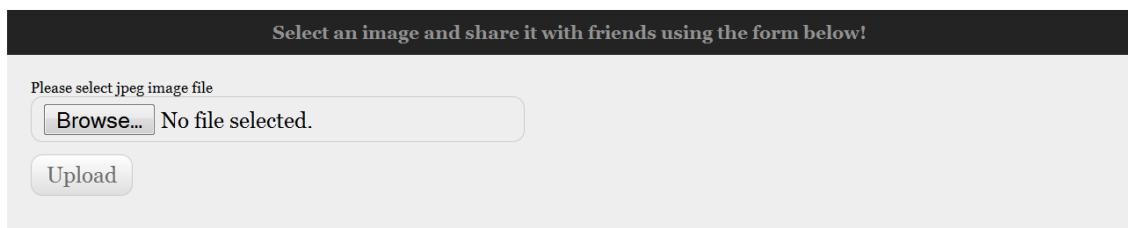
Following the implementation specifications set forth in the previous chapters, a functional image sharing website was developed. This website was not only able to successfully accept images, store them for later access, and provide these features in an intuitive manner, but it did so while reducing its storage space requirements. In order to gauge the success of the research project, several metrics must be evaluated. The first to be considered is the functionality of the image sharing system. This includes usability and the requirement of keeping user interaction at a minimum while the space optimization system operates behind the scenes. Along with functionality, the performance of the system must be evaluated. If the addition of the developed system takes too much time to operate, a user might be dissuaded from using the system in the first place. This would in fact lessen storage requirements of the image sharing website, but defeats the purpose of the research. Performance has been measured in several ways, namely the processing time to upload an image to the server, accuracy of duplicate detection, and the amount of storage space required after duplicate detection has run. As this research hinges on the performance of the duplicate detection algorithm from a time and effectiveness standpoint, the results are of extreme importance. This chapter presents the results of the system's evaluation against the characteristics outlined above.

## 4.1 Functionality

As mentioned previously, the functionality of the image sharing website developed involves both the usability and autonomous function of the server side duplicate detection system. In order to gain insight into the usability of the image sharing system, one must first understand how it is operated. Upon loading of the web page the user is presented with a screen as shown in Figure 4.1.

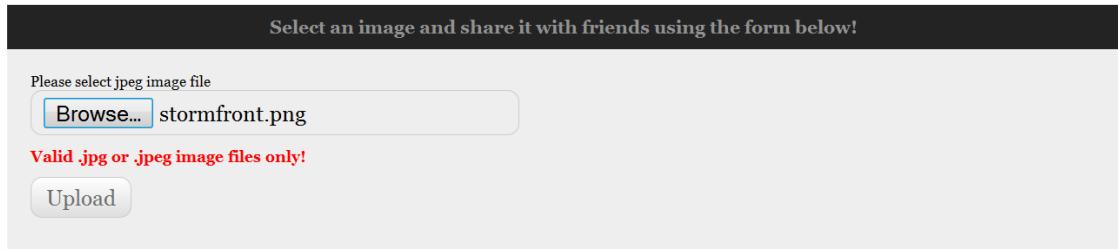
At this time, the user is asked to select an image for sharing and is told that only jpeg images are allowed. This can be accomplished by using the given option to browse for an image file stored on their local computer. Upon choosing an allowed image, a preview will be displayed along with the details of the image to be uploaded.

If a user would choose to upload a file that is not of the jpeg format or if the image size is larger than what the server allows, a warning will be displayed and the upload button will remain disabled as seen in Figures 4.2 and 4.3.



The screenshot shows a simple web form for image upload. At the top, a dark header bar contains the text "Select an image and share it with friends using the form below!". Below this, the main content area has a light gray background. It features a text input field with the placeholder "Please select jpeg image file". To the right of the input field is a "Browse..." button and the text "No file selected.". At the bottom of the form is a single "Upload" button.

Figure 4.1: Initial state of the upload form upon page load.



The screenshot shows the same upload form as Figure 4.1, but with an error message. The "Browse..." button is now highlighted in blue, and the text next to it shows "stormfront.png". Below the input field, a red error message reads "Valid jpg or jpeg image files only!". The "Upload" button remains at the bottom of the form.

Figure 4.2: Error displayed when a user attempts to upload a non-jpeg file.

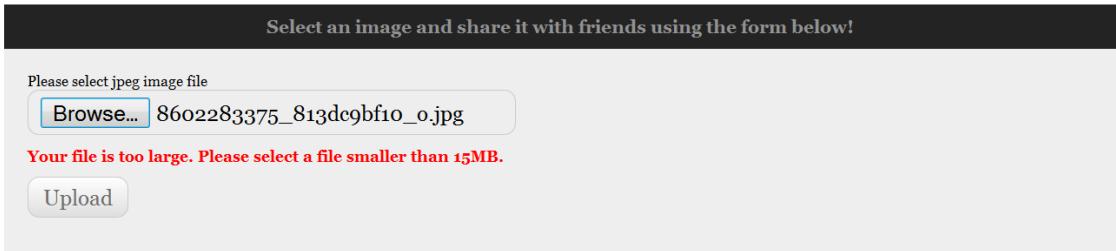


Figure 4.3: Error displayed when attempting to upload a jpeg image that is too large.

By designing the system in this manner, it ensures that the image upload can be completed in an intuitive way. After any errors have been resolved by selecting an appropriate sized jpeg file, the upload button will enable and the user can click it to start the sharing process. Once the upload begins a new section of the user interface will be built and displayed. In this section, a progress bar is visible that visualizes the current progress of the image transfer to the server, the speed in which the transfer is occurring, and the estimated time remaining. Upon completion of the upload, the user is presented with a message saying "PLEASE WAIT...PROCESSING", which can be seen in figure 4.4. During this time, the algorithms operate in the background determining the images uniqueness as implemented in Chapter 3. To this point, the only interaction required by the user is supplying an acceptable image file and clicking the upload button. After processing completes, there are several options that can be displayed.

If it is determined that the image being uploaded is a potential duplicate, a pop-up is displayed to the user with a preview of the image they are trying to upload shown side by side with a preview of the potential duplicate on the server. The image with a higher resolution has text displayed under it specifying. The user can then click the higher resolution image and the server will provide the link to share it with, or they can choose to upload their image anyway. At this point, the server assumes the image

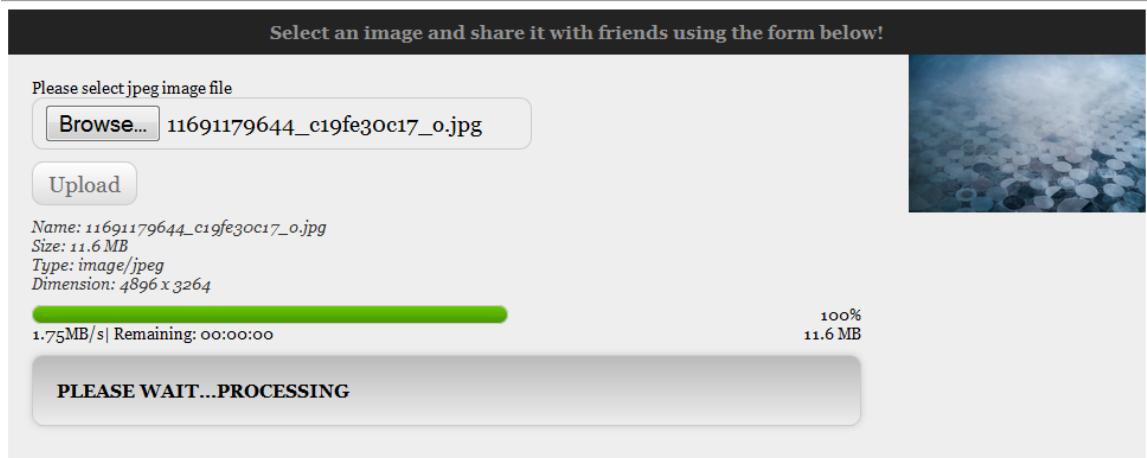


Figure 4.4: Image processing after upload completes.

is unique and will generate and display a link to the user. This feature was disabled for the purpose of testing the effectiveness as it introduces a human error which could bias collected results. If the image is in fact deemed to be unique, user interaction is not required. The server will generate a link to the new image and return it to the user for sharing.

From this point on, the full upload process is complete, and users can access an image using one of the generated links, never knowing if that image was unique or is displaying the copy already on the server. Although this system is very basic on the surface, it is very powerful. Not only is the server reducing image duplication in real time, but it is simultaneously generating links, thumbnails and possibly performing other uploads and serving images to users all with the click of two buttons or one link.

## 4.2 Performance

Processing time in the scope of this research refers to the amount of real world time that it takes to complete the image upload. This begins at the point where the server

receives an image to the time where a link is provided to the user. As seen in Figures 4.5 and 4.6, several factors play a major role in the variance of processing time.



Figure 4.5: Processing time on data set without duplicates.



Figure 4.6: Processing time on data set with 20% duplicate images.

Of any variable tested, file size was determined to have the largest impact on processing time. Although at first glance, the large image upload using this system may appear too inefficient for a real world scenario, it is important to remember the scale being used. Traditional upload methods appeared almost instantaneous to the

user while duplicate reduced took less than two seconds to process, even on the largest file. This amount of time was not considered unreasonable as fluctuations in Internet connection speed can cause greater variance in upload times than was ever observed with processing times from the server side. To interpret the graphs in an intuitive manner, it can be said that processing time is directly proportional to the size of the files being uploaded. In addition, the average processing time can be greatly affected by the frequency of duplicates being uploaded. When exact duplicates are uploaded the processing time is near that of a traditional upload as the in-depth image scan is not needed. This fact alone can account for another large source of processing time variance. In Figure 4.5, every file was chosen to be unique, this would cause data to be collected that reflected worst case processing time where every file was scanned in depth. In Figure 4.6, a data set was created that contained 20% exact duplicate images. This number was chosen as it correlated to the estimated amount of duplicate data on servers which is discussed previously in Section 1.3. In this case a smaller number of images required a detailed analysis, while the remaining images were able to be detected with only a first pass scan. In this scenario, the processing time was reduced by approximately 30% of the worst case time.

Although these results point toward a feasible system with respect to wait time, server configuration and load can affect these greatly. A webmaster can decide to accept larger or smaller images than the 15MB limit that was configured for this experiment. Seeing that processing times increase rapidly with respect to file size, this system leans toward a usage on high volume servers with a smaller image size versus a low volume situation with large files.

Next, another important metric is the total space consumption of the images uploaded. Using four test sets of images the results were compiled and quite promising. The first test set included images smaller than 1MB with no duplicates included. As

expected, this set required the same amount of space after duplicate reduction as a traditional upload system. The same could be said for the other group of large images with no duplicates. In the group containing small images with one of five chosen modifications including 85% quality reduction, 25% resolution reduction, horizontal flip, vertical flip, and the addition of text, the results become somewhat surprising. In the instances where the image was flipped or text was overlaid, image uploads succeeded on every single test. Where image resolution and quality had been reduced, the results showed no discernible pattern with detection with duplicate reduction ranging anywhere from approximately 5%-20% in each test. The average of five tests using this group of small images can be seen in Figure 4.7 and represents nearly a 12% reduction in storage space. In the group where large files were used, nearly every one of the duplicates were detected using the same modifications and test structure as discussed with the small image group before. Both the small and large test groups contained 50% duplicate images in order to cover the widest variety of potential image modifications. Results for the large file size set of images can be seen in Figure 4.8.

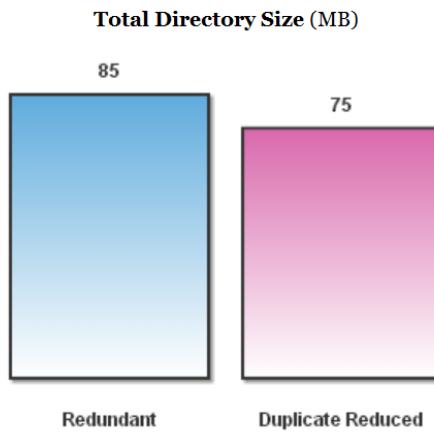


Figure 4.7: Average of 5 tests using different groups of sub 1MB images.

Last, and perhaps most importantly, accuracy for duplicate detection is how well the system worked when finding duplicates in a set of known files. After analyzing the

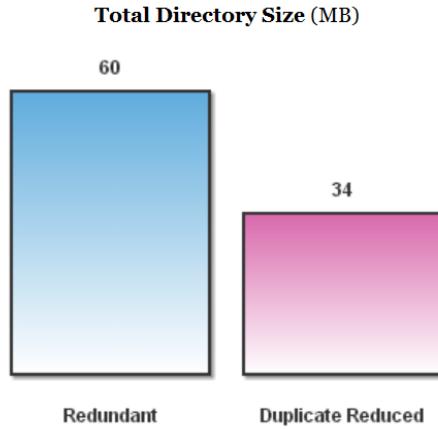


Figure 4.8: Average of 5 tests using different groups of greater than 1MB images.

results, the data was able to be broken down into three groups. These groups include small size photographs, large size photographs, and computer generated graphics. With both photographic groups, the accuracy was within the expected and acceptable range for locating duplicates. The system determined the expected outcome for each of 100 small photographic image files 82 times. It also determined the expected outcome for each of 100 large photographic image files 91 times. Many variables could have lead to the difference in correct detection between the two sets even though they contained the same images in different resolutions. After analyzing the results, the most probable cause of incorrect classification came from the loss of data involved with re-sizing the images to  $16 \times 16$  pixels for deep analysis. This will be discussed in further detail when analyzing the computer generated graphics. Although the thumbnail size can be increased to improve accuracy, the processing time increases significantly due to the higher number of individual pixels needing analyzed.

Finally, the computer generated graphics provided completely unexpected results. On a set of 100 known images, only 33 were correctly identified. As mentioned previously, the process of generating the thumbnail for analysis causes a significant amount of data loss within the image. The computer graphics used all consisted of

repetitive patterns of alternating black and white. As it turns out, the color profile is identical with each as each was developed to be half black and half white. If the image contained too much detail, when the thumbnail was generated the pattern is eliminated in many cases. This can be seen in Figure

### 4.3 Conclusion

Coming soon.

# **Chapter 5**

## **Discussion and Future Work**

Conclusion to be written in the Spring after research and testing concludes.

### **5.1 Summary of Results**

Coming soon.

### **5.2 Future Work**

Coming soon.

### **5.3 Conclusion**

Coming soon.

# Appendix A

## Duplicate Image Detection Code

This appendix contains code relating directly to the uploading and duplicate image detection process. More precisely, the code relating to the exact image matching function is contained within Section A.2 while code relating to the more in-depth approximate duplicate image detection process is in Section A.3.

### A.1 Upload Procedures

```
<?php  
*****  
* *  
* Debugging *  
* *  
*****  
  
// Uncomment to turn on debugging, ignore notices.  
//error_reporting(E_ALL & ~E_NOTICE);  
//ini_set('display_errors', 'On');  
  
// Print the PHP max stats for this directory... more debug info.  
//echo 'upload_max_filesize' . ini_get('upload_max_filesize');  
//echo 'post_max_size' . ini_get('post_max_size');
```

```

*****  

*          *  

*      File/Script Imports      *  

*          *  

*****  

// Allow us to connect to the database, IMPORTANT!  

include "scripts/connect.php";  

// Provide us with the missing functions.  

include "scripts/dupFindSimple.php";  

// Provide access to thumbnail creation methods.  

include "scripts/thumbnailer.php";  

*****  

*          *  

*      Environment Variables      *  

*          *  

*****  

// Upload directory information.  

$truTarget = "/var/www/srthesis/uploads/";  

$druTarget = "/var/www/srthesis/uploads_reduced/";  

$publicThumbTarget = "/var/www/srthesis-thumbnails_reduced/";  

// Details about the file being used in the system.  

$sFileName = $_FILES['image_file']['name'];  

$sFileType = $_FILES['image_file']['type'];  

$sFileSize = bytesToSize1024($_FILES['image_file']['size'], 1);  

*****  

/* Thumbnail function variables. */  

*****

```

```

// Width for thumbnail images we use for fingerprinting; Recommended
150.

$thumbWidth = 150;

// Width for public visible thumbnail (Whatever the <img /> width is
).

$publicThumbSize = 215;

// Sets how sensitive the fingerprinting will be.

// Higher numbers are less sensitive (more likely to match). Floats
are allowed.

$sensitivity = 2;

/*
 *
 *      Global Variables
 *
 *****/
// Global notification control for response to user.

$error = '';
$trNotice = '';
$drNotice = '';

// Global filename storage.

$uFileName = '';

/*
 *
 *      Misc Functionality
 *
 *****/
/*
 *      Function bytesToSize1024(params)
 *
 */

```

```

*   Description: Converts from bytes to a more *
*               human readable format *
*
*   bytes - The number to be converted *
*   precision - Number of decimal places to round to *
*
*   Returns: Number in B, KB, MB, or GB *
*   Calls:   N/A *
*****
function bytesToSize1024($bytes, $precision = 2) {
    $unit = array('B', 'KB', 'MB', 'GB');
    return @round($bytes / pow(1024, ($i = floor(log($bytes, 1024))) ),
        $precision).'.'.$unit[$i];
}

*****
*   Function createImageHandle(param) *
*
*   Description: Creates a case sensitive alphanumeric *
*               image handle for the URL *
*
*   bytes - The number to be converted *
*
*   Returns: $length character alphanumeric string *
*   Calls:   N/A *
*****
function createImageHandle($length){
    $handle = "";
    $codeAlphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    $codeAlphabet .= "abcdefghijklmnopqrstuvwxyz";
    $codeAlphabet .= "0123456789";
}

```

```

for($i=0;$i<$length;$i++){
    $handle .= $codeAlphabet[mt_rand(0,strlen($codeAlphabet))];
}

return $handle;
}

/***** Adapted with permission from: *****
*          CatPA PHP GD Image Finder code
*          Function createFingerprint(param)
*
* Description: Analyses the filename passed to it and
*               returns an md5 checksum of the
*               file's histogram
*
* filePathAndName - File and location on server the
*                   fingerprint will be created for
*
* Returns: MD5 hashed histogram of the supplied file
* Calls:   N/A
*****/

function createFingerprint($filePathAndName) {
    // Load the image. Escape out if it's not a valid jpeg (can be
    // extended later).
    if (!$image = @imagecreatefromjpeg($filePathAndName)) {
        return null;
    }

    // Create thumbnail sized copy for fingerprinting.
    $width = imagesx($image);
    $height = imagesy($image);
}

```

```

$ratio = $GLOBALS["thumbWidth"] / $width;
$newwidth = $GLOBALS["thumbWidth"];
$newheight = round($height * $ratio);
$smallimage = imagecreatetruecolor($newwidth, $newheight);
imagecopyresampled($smallimage, $image, 0, 0, 0, 0, $newwidth,
    $newheight, $width, $height);
$palette = imagecreatetruecolor(1, 1);
$gsimage = imagecreatetruecolor($newwidth, $newheight);

// Calculate the number of pixels in the resized image.
$numpixels = $newwidth * $newheight;
$histogram = array();
// Iterate over x-axis pixels.
for ($i = 0; $i < $newwidth; $i++) {
    // Iterate over the y-axis pixels.
    for ($j = 0; $j < $newheight; $j++) {
        // Pull the index of the current pixel.
        $pos = imagecolorat($smallimage, $i, $j);
        // Pull RGB value of the current pixel.
        $cols = imagecolorsforindex($smallimage, $pos);
        // Split apart the RGB values of the pixel, throwing out
            alpha.
        $r = $cols['red'];
        $g = $cols['green'];
        $b = $cols['blue'];
        // Convert the colour to greyscale using 30% Red, 59%
            Blue and 11% Green.
        // Gets a single integer value representing all three
            values.
        $greyscale = round((($r * 0.3) + ($g * 0.59) + ($b *
            0.11)));
        $greyscale++;
    }
}

```

```

    $value = (round($greyscale / 16) * 16) -1;

    // Use the integer value as an index and increment that
    // index.

    $histogram[$value]++;
}

}

// Normalize the histogram by dividing the total of each colour by
// the total number of pixels.

$normhist = array();

foreach ($histogram as $value => $count) {

    $normhist[$value] = $count / $numpixels;
}

// Find maximum value (most frequent colour).

$max = 0;

for ($i=0; $i<255; $i++) {

    if ($normhist[$i] > $max) {

        $max = $normhist[$i];
    }
}

// Create a string from the histogram (with all possible values)

.

$histstring = "";

for ($i = -1; $i <= 255; $i = $i + 16) {

    $h = ($normhist[$i] / $max) * $GLOBALS["sensitivity"];

    if ($i < 0) {

        $index = 0;
    } else {

        $index = $i;
    }
}

```

```

    $height = round($h);

    $histstring .= $height;

}

// Destroy all the temporary images that we've created.

imagedestroy($image);

imagedestroy($smallimage);

imagedestroy($palette);

imagedestroy($gsimage);

// Generate an md5sum of the histogram values and return it.

return md5($histstring);

}

```

```

*****
*
*      File Operations
*
*****
*
*      Function findext(param)
*
*      Description: Separates the extension from the
*                    rest of the filename and returns it
*
*      filename - Filename to be split apart
*
*      Returns: File extension of the supplied file
*
*      Calls:   N/A
*
*****

```

```
function findext($filename)
```

```

{

    // Only allow lowercase , split at the period and keep the trailing
    // characters.

    $filename = strtolower($filename);

    $exts = explode(".", $filename);

    $n = count($exts)-1;

    $exts = $exts[$n];



    // Returns the extension.

    return $exts;

}

*****



*   Function renameFile(param)                               *

*                                                       *

*   Description: File renamed using the below pattern      *

*               TTTTTTTTTT-IIIII.EEE(E)                      *

*               - where -                                     *

*               Timestamp-ImageHandle.Extension             *

*                                                       *

*   filename - Filename to be renamed                     *

*   imageHandle - Newly generated image handle           *

*                                                       *

*   Returns: New filename with same extension as the old *

*   Calls:     N/A                                         *

*****



function renameFile($filename, $imageHandle)
{
    // Finds the file extension of the supplied file.

    $ext = finext($filename);




    // Creates a timestamp to use in the filename.
}

```

```

$time = time();

// Combine the the unix timestamp file name, and the extension.

$reName = $time . "-" . $imageHandle . "." . $ext;

// Returns the new file name.

return $reName;
}

/******************
*
*      Upload handlers
*
*      *
*****/


/******************
*
*      Function trUpload()
*
*      *
*      Description: Traditional upload function
*
*      *
*      Returns: N/A
*
*      Calls:    N/A
*
*****/


function trUpload()
{
    // Allow access to the global (scope) variables.

    global $strNotice, $error, $truTarget, $uFileName;

    // Create an image handle for the baseline upload.

    $trImageHandle = createImageHandle(6);

    // Globally rename the file using this original filename.

    $uFileName = renameFile($GLOBALS["sFileName"], $trImageHandle);
}

```

```

// Make sure we aren't accidentally overwriting anything this time

.

if (file_exists($truTarget . $uFileName))
{
    // Upload will cause more than one file with the same name in
    // the directory? Error.

    $error = '<p><strong>ERROR:</strong> Filename uniqueness not
preserved.<br />Please try again or contact the webmaster
if this problem persists.</p>';
} else {
    // Actually upload the file!

    if(move_uploaded_file($_FILES['image_file']['tmp_name'],
        $truTarget . $uFileName))
    {

        try {
            // Add the image to the database!

            $stmt = $GLOBALS["conn"]->prepare('INSERT INTO share_tracker
(ILookup, IName, directory, uMethod) VALUES (:imageHandle,:imageName,:directory,:uMethod)');
            $stmt->execute(array(':imageHandle'=>$trImageHandle,
                ':imageName'=>$uFileName,
                ':directory'=>'uploads',
                ':uMethod'=>'0'));

            // Report back a success!

            $trNotice = "<p>Baseline directory upload succeeded! You may
view this image <a href=\"http://skynetgds.no-ip.biz/
srthesis/irc.php?view={$trImageHandle}\\">HERE</a>. </p>";
        } catch(PDOException $e) {
            // Or an error...
            $error = '<p><strong>ERROR:</strong> ' . $e->getMessage() .
            '</p>';
        }
    }
}

```

```

        }

    } else {
        $error = '<p><strong>ERROR:</strong> File could not be saved
                  to the server.<br />Please retry or contact the webmaster
                  if this problem persists.</p>';
    }
}

*******/

*   Function drUpload()                               *
*
*   Description: Duplicate reduced upload function   *
*
*   Returns: N/A                                     *
*
*   Calls:   N/A                                     *

function drUpload()
{
    // Allow access to the global (scope) variables.
    global $drNotice, $error, $truTarget, $druTarget, $uFileName;

    // Generate the 40-bit SHA1 file hash for simple dup lookup.
    $shaHash = sha1_file($truTarget . $uFileName);

    // Retrieve the MD5 fingerprint for storage in the database.
    $md5Fingerprint = createFingerprint($truTarget . $uFileName);

    // Make sure we aren't accidentally overwriting anything this time
    .

    if (file_exists($druTarget . $uFileName))

```

```

{

    // Upload will cause more than one file with the same name in
    // the directory? Error.

    $error = '<p><strong>ERROR:</strong> Filename uniqueness not
    preserved.<br />Please try again or contact the webmaster
    if this problem persists.</p>'; } else {

    // Uniqueness check...

    $dupResponse = simpleDupCheck($truTarget . $uFileName, $shaHash,
        $druTarget, $md5Fingerprint);

    if($dupResponse !== null)

    {

        // Every image gets a new handle, even if it's in the reduced
        // folder. (For consistency)

        $newiHandle = createImageHandle(6);

        try {

            // Exact dup found, add to DB and respond appropriately.

            $stmt = $GLOBALS["conn"]->prepare('INSERT INTO share_tracker
                (ILookup, IName, directory, uMethod) VALUES (:imageHandle,:imageName,:directory,:uMethod)');
            $stmt->execute(array(':imageHandle'=>$newiHandle,
                ':imageName'=>$dupResponse,
                ':directory'=>'uploads_reduced',
                ':uMethod'=>'1'));

            // Report back a success!

            $drNotice = "<p><strong>NOTICE:</strong> Image was not added
            to duplicate reduced directory, file already exists.</p>
            "
            "<br /> .
            "<p>To share your image, use the following link: " .


```

```

        "<a href=\"http://skynetgds.no-ip.biz/srthesis/irc.php
            ?view={$newiHandle}\">http://skynetgds.no-ip.biz/
            srthesis/irc.php?view={$newiHandle}</a></p>";

    } catch(PDOException $e) {
        // Or a failure...
        $error = '<p><strong>ERROR:</strong> ' . $e->getMessage() .
        '</p>';
    }

    // Unique so actually upload the file!
} else {
    // Create an image handle for the duplicate reduced link.
    $drImageHandle= createImageHandle(6);
    // Globally rename the file using this original filename.
    $relinkFileName = renameFile($uFileName, $drImageHandle);

    // Copy image to dup reduced and see if success.
    if(copy($truTarget . $uFileName, $druTarget . $relinkFileName)
    )
    {
        // Create a thumbnail for lightweight public viewing.
        makeThumb($druTarget, $GLOBALS["publicThumbTarget"],
        $relinkFileName, $GLOBALS["publicThumbSize"]);
    }

    try {
        // Add the image to the database!
        $stmt = $GLOBALS["conn"]->prepare('INSERT INTO
            share_tracker (ILookup, IName, directory, uMethod, hash
            , fingerprint) VALUES (:imageHandle,:imageName,:
            directory,:uMethod, :shaHash, :fingerprint)');
        $stmt->execute(array(':imageHandle'=>$drImageHandle,

```

```

        ':imageName'=>$relinkFileName,
        ':directory'=>'uploads_reduced',
        ':uMethod'=>'1',
        ':shaHash'=>$shaHash,
        ':fingerprint'=>$md5Fingerprint));
    }

    // Report a success!
    $drNotice = "<p><strong>NOTICE:</strong> The image was
unique and added to duplicate reduced directory!</p>" .
"<br />" .
"<p>To share your image, use the following link: " .
"<br />" .
"<a href=\"http://skynetgds.no-ip.biz/srthesis/irc.php
?view={$drImageHandle}\">>http://skynetgds.no-ip.biz
/srthesis/irc.php?view={$drImageHandle}</a></p>";

} catch(PDOException $e) {
    // Or a failure...
    $error = '<p><strong>ERROR:</strong> ' . $e->getMessage()
. '</p>';
}

} else {
    $error = '<p><strong>ERROR!</strong> File could not be
saved to the server.<br />Please retry or contact the
webmaster if this problem persists.</p>';
}

}

}

// Call the upload functions and actually do some work!

```

```

trupload(); // Creates a baseline to compare results to.

drupload(); // Upload method targeted by research.

/*
 *      Response handlers
 *
 *****/
// Give the user some much needed output.

if(! $error) {

//Tell the user what has been going on behind the scenes.

echo <<<EOF

$trNotice

$drNotice

EOF;

} else {

// Something interrupted the process and shouldn't have...

echo $error;

}

?>

```

## A.2 Exact Duplicate Detection Code

```

<?php

/*
 *      Debugging
 *
 *****/
// Uncomment to turn on debugging
//error_reporting(E_ALL & ~E_NOTICE);
//ini_set('display_errors', 'On');

```

```

*****  

*          *  

*      File/Script Importing      *  

*          *  

*****/  

// Allow us to connect to the database, IMPORTANT!  

include "connect.php";  

// Provide us with the missing functions.  

include "dupFindComplex.php";  

*****  

*          *  

*      Duplication handler      *  

*          *  

*****/  

*****  

*      Function simpleDupCheck(params)          *  

*          *  

*      Description: Uses an SHA1 File hash to initially          *  

*                  check for duplicates.          *  

*          *  

*      filePath - File path and name of the image          *  

*      fileHash - SHA1 hash of the full file being uploaded *  

*      uploadDir - Directory path where uploads are stored *  

*      fingerprint - MD5 fingerprint of color profile          *  

*          *  

*      Returns: Name of matching image file, if found.          *  

*      Calls: If no exact match, calls deepDupCheck(*)          *  

*****/

```

```

function simpleDupCheck($filePath, $fileHash, $uploadDir,
    $fingerprint) {

    // Check for exact file matches with the file hash stored in the
    // DB.

    try {
        $stmt = $GLOBALS["conn"]->prepare('SELECT `IName` FROM `'
            . 'share_tracker` WHERE `uMethod` = \'1\' AND `hash` = :fileHash
        ');
        $stmt->execute(array(':fileHash' => $fileHash));
    } catch(PDOException $e) {
        $error = 'ERROR: ' . $e->getMessage();
    }

    // Was there a match?
    if ($stmt->rowCount() > 0) {
        // Match found, return the file name of the file on the server.
        $nMatch = $stmt->fetch();
        echo "Debug: Simple Check was enough to find a match."; /*
            Temporary for visual effect */
        return $nMatch[0];
    } else {
        // No exact match, look for 'approximate' candidates.
        echo "Deep Check Run";
        return deepDupCheck($filePath, $uploadDir, $fingerprint);
    }
}
?>

```

### A.3 Approximate Duplicate Detection Code

<?php

```

*****  

*          *  

*      Debugging      *  

*          *  

*****/  

// Uncomment to turn on debugging  

//error_reporting(E_ALL & ~E_NOTICE);  

ini_set('display_errors', 'On');  

*****  

*          *  

*      File/Script Importing      *  

*          *  

*****/  

// Allow us to connect to the database, IMPORTANT!  

include "connect.php";  

// PHP GD script to check for duplicates.  

include "phpDupeImage.php";  

*****  

*          *  

*      Duplication handler      *  

*          *  

*****/  

*****  

*      Function deepDupCheck(params)          *  

*          *  

*      Description: Uses modified PHP GD Image Finder          *  

*                  to determine uniqueness.          *  

*          *  

*      filePath - File path and name of the image          *

```

```

*      uploadDir - Directory path where uploads are stored   *
*      fingerprint - MD5 fingerprint of color profile        *
*
*      Returns: Result from running isUnique(*) function      *
*      Calls:    N/A                                         *
***** */

function deepDupCheck($filePath, $uploadDir, $fingerprint) {
    // Create a new instance of phpDupeImage and set the upload
    // directory.
    $dc = new phpDupeImage();
    $dc->completed_files_path = $uploadDir;

    // Check and see if the file is already no the server the GD
    // functions.
    $matchResult = $dc->is_unique($filePath, $fingerprint);

    // Return the result after thorough examination.
    return $matchResult;
}

?>

<?php
// Code based off of works by pawz, 2010.
// www.catpa.ws/php-duplicate-image-finder/

***** */

*                                *
*      Debugging                 *
*                                *
***** */

// Uncomment to turn on debugging.
//error_reporting(E_ALL & ~E_NOTICE);

```

```

//ini_set('display_errors', 'On');

class phpDupeImage {

    // Sets how much deviation is tolerated between two images when
    // doing an thorough comparison.
    public $deviation = 10;

    // Sets the path to where the files are stored on the server.
    // Needed to perform deep deep analysis.
    public $completed_files_path = '';

    // Sets the width and height of the thumbnail sized image we use
    // for deep comparison.
    public $small_size = 16;

    /***** Function isUnique(params) *****
    *
    * Description: Checks fingerprint against DB and
    *               determines uniqueness.
    *
    * filePathAndName - File path and name of the image
    *                   to check for uniqueness
    * fingerprint - MD5 fingerprint of color profile
    *
    * Returns: Null for no match found
    *          ImageName of the match if one is found
    * Calls:   If no exact match, calls deepDupCheck(*)
    *****/
    function is_unique($filePathAndName, $fingerprint) {

        // Select any images in database with a matching fingerprint.
        try {

```

```

$stmt= $GLOBALS["conn"]->prepare('SELECT * FROM `share_tracker
    WHERE `fingerprint` = :fingerprint');
$stmt->execute(array(':fingerprint'=>$fingerprint));
$matches = $stmt->fetchAll();
} catch(PDOException $e) {
    echo 'ERROR: ' . $e->getMessage();
}

if (!count($matches)) {
    // No matching fingerprints found so return null.
    return null;
} else {
    // Matching fingerprint located, run deep check on each
    // possible match.
    $match_found = null;
    foreach($matches as $match) {
        if ($this->are_duplicates($filePathAndName, $this->
            completed_files_path.$match['IName'])) {
            // We found a match, die and return the file
            // name of the duplicate found.
            return $match['IName'];
        }
    }
}

// Fingerprint was a match but deep check determined it
// wasn't the same image.
return null;
}

}

```

```
*****
```

```

*   Function are_duplicates(params) *
*
* Description: Compares two images by analyzing the *
*               colors of each pixel difference of *
*               each corresponding pixel in the images *
*
*   file1 - The first of the files to compare *
*   file2 - File to compare to the first file (file1) *
*
* Returns: 1 (True) for a match found. *
*          0 (False) for no match found. *
*
* Calls:  N/A *
***** */
function are_duplicates($file1, $file2) {

    // Load in both images and resize them to small_size x
    // small_size.

    $image1_src = @imagecreatefromjpeg($file1);
    $image2_src = @imagecreatefromjpeg($file2);
    list($image1_width, $image1_height) = getimagesize($file1);
    list($image2_width, $image2_height) = getimagesize($file2);
    $image1_small = imagecreatetruecolor($this->small_size,
                                         $this->small_size);
    $image2_small = imagecreatetruecolor($this->small_size,
                                         $this->small_size);
    imagecopyresampled($image1_small, $image1_src, 0, 0, 0, 0,
                       $this->small_size, $this->small_size, $image1_width,
                       $image1_height);
    imagecopyresampled($image2_small, $image2_src, 0, 0, 0, 0,
                       $this->small_size, $this->small_size, $image2_width,
                       $image2_height);
}

```

```

// Compare the pixels of each image and figure out the
// colour difference between them.

for ($x = 0; $x < $this->small_size; $x++) {
    for ($y = 0; $y < $this->small_size; $y++) {
        $image1_color = imagecolorsforindex($image1_small,
            imagecolorat($image1_small, $x, $y));
        $image2_color = imagecolorsforindex($image2_small,
            imagecolorat($image2_small, $x, $y));
        $difference += abs($image1_color['red'] -
            $image2_color['red']) +
            abs($image1_color['green'] -
            $image2_color['green']) +
            abs($image1_color['blue'] -
            $image2_color['blue']);
    }
}

// Normalize the image by diving it by the number of pixeles
// in the resized image.

$difference = $difference / pow($this->small_size, 2);

if ($difference <= $this->deviation) {
    // The images match!
    return 1;
} else {
    // The images don't match.
    return 0;
}
}

?>

```

## **Appendix B**

### **Website Framework Code**

The code contained within this appendix describes only the research oriented pages and supporting files that were

## **Appendix C**

### **Website Framework Code**

All program code should be fully commented. Authorship of all parts of the code should be clearly specified.

## **Appendix D**

# **Miscellaneous Code and Configurations**

# Bibliography

- [1] Tinku Acharya and Ping-Sing Tsai. Computational Foundations of Image Interpolation Algorithms. *Ubiquity*, 2007(October):4:1–4:17, October 2007.
- [2] Angela Bradley. Renaming PHP Uploads. [http://php.about.com/od/advancedphp/ss/rename\\_upload.htm](http://php.about.com/od/advancedphp/ss/rename_upload.htm), 2011. [Online; accessed 10-October-2013].
- [3] CatPa.ws. PHP GD Duplicate Image Finder. <http://www.catpa.ws/php-duplicate-image-finder/>, 2010. [Online; accessed 13-September-2013].
- [4] Dictionary.com. Definition of Photo Sharing. [www.dictionary.com](http://www.dictionary.com), 2013. [Online; accessed 20-November-2013].
- [5] All Things Digital. Meeker: 500 Million Photos Shared Per Day and That's on Track to Double in 12 Months. <http://allthingsd.com/20130529/meeker-500-million-photos-shared-per-day-and-thats-on-track-to-double-in-12-months/>, 2013. [Online; accessed 20-November-2013].
- [6] Ubuntu Documentation. ApacheMySQLPHP - LAMP Server Setup Guide. <https://help.ubuntu.com/community/ApacheMySQLPHP>, 2013. [Online; accessed 21-August-2013].
- [7] Jun Jie Foo, Justin Zobel, Ranjan Sinha, and S. M. M. Tahaghoghi. Detection of Near-duplicate Images for Web Search. In *Proceedings of the 6th ACM International Conference on Image and Video Retrieval*, CIVR '07, pages 557–564, New York, NY, USA, 2007. ACM.
- [8] The PHP Group. GD and Image Functions. <http://php.net/manual/en/ref.image.php>, 2013. [Online; accessed 13-October-2013].
- [9] The PHP Group. POST Method Uploads. <http://de.php.net/manual/en/features.file-upload.post-method.php>, 2013. [Online; accessed 09-September-2013].
- [10] Snapchat Inc. Snapchat Support. <http://support.snapchat.com/>, 2013. [Online; accessed 9-December-2013].
- [11] David C. Lee, Qifa Ke, and Michael Isard. Partition Min-hash for Partial Duplicate Image Discovery. In *Proceedings of the 11th European Conference on*

*Computer Vision: Part I*, ECCV'10, pages 648–662, Berlin, Heidelberg, 2010. Springer-Verlag.

- [12] Dejan Marjanovic. PDO vs. MySQLi: Which Should You Use? <http://net.tutsplus.com/tutorials/php/pdo-vs-mysqli-which-should-you-use/>, 2012. [Online; accessed 02-October-2013].
- [13] TecNick LTD Nicola Asuni. TESTIMAGES. [www.tecnick.com/public/code/cp\\_dpage.php?aiocp\\_dp=testimages](http://www.tecnick.com/public/code/cp_dpage.php?aiocp_dp=testimages), 2013. [Online; accessed 3-November-2013].
- [14] nixCraft. Ubuntu Linux: Install or Add PHP-GD Support to Apache Web Server. <http://www.cyberciti.biz/faq/ubuntu-linux-install-or-add-php-gd-support-to-apache/>, 2012. [Online; accessed 30-September-2013].
- [15] NTP Software. Survey Says Nearly Two-Thirds of Files on Primary Storage Are Stale. [www.ntpsoftware.com/pressroom/survey-says-nearly-two-thirds-files-primary-storage-are-stale](http://www.ntpsoftware.com/pressroom/survey-says-nearly-two-thirds-files-primary-storage-are-stale), 2013. [Online; accessed 31-October-2013].
- [16] S H. Srinivasan and Neela Sawant. Finding Near-duplicate Images on the Web Using Fingerprints. In *Proceedings of the 16th ACM International Conference on Multimedia*, MM '08, pages 881–884, New York, NY, USA, 2008. ACM.