Outline - Introduction What problem? Why is the website important/word storms important Why might people use it (not Contribution

Background Word clouds Visualization Quims Ruby on Rails

Requirements Analysis Use Case

Design Implementation

Evaluation Screenshots

Compare to initial requiements Test suite

Conclusion and future work

Chapter 1

Abstract

Chapter 2

Introduction

In the age of the Internet, with an ever-increasing amount of data in our possession, it is important to have tools to deal with this data. To be able to analyze large corpora of documents easily and to achieve that any place with simplicity. My project, to design a scalable web 2.0 extension of the word storm project, aims to assist in the visualization of large data, and provide the capacity to share this visualization technique with a number of concurrent users, in a manner that encourages reuse and conforms to user expectations.

It will achieve this using the word storm. I will go into further depth about what a word storm is, and how one is created, but for now, it can be said that a word storm is a visualization tool for analyzing corpora[0]. Graphical visualization is an important method of conveying information - humans process visuals faster and better than any other medium. Taking advantage of this, if the important information in a collection of documents can be extracted and presented in a visual manner on the screen, the similarities and differences between them can be quickly analyzed and important features can be extracted.

Word storms achieve this using a group of related word clouds. These clouds are colorful representations of the important words within a document, each portrayed by an angle, size and color. The word storm synchronizes the important attributes across these clouds - identical word position, color and angle can be made identical during the creation process.

With this synchronization, word storms allow large amounts of data to be viewed and understood quickly within their context to each other. All the applications of the singular word cloud are applicable to word storms - each cloud itself gives insight into the important features of that document. When taken as a whole, however, important parallels can be drawn between the documents, and new insights can be made from this visualization. The capability to customize a word storm's options can create different layouts and impressions, increasing the potential for use across multiple domains.

By making these word storms accessible online, a number of benefits and auxilary goals are met. Firstly, with the growing improtance of the web, a web site is needed to be relevant. By designing a website, word storms gain

credibility and presence within the online community. As soon as it is placed online, the potential reach of the word storms grows to millions of people. While this growth can be achieved by uploading the project, the inclusion of a user interface and database changes how the word storm can be used, and increases its reach to non-technology inclined individuals.

With a user interface, individuals lacking a coding background can use the word storms. By removing the requirement of knowing Java and providing the tools to generate a storm, users who are not technologically saavy can still create their own products. The increase in potential users improves the chances of discerning meaningful discoveries from the data. Encouraging the sharing of storms by providing email generation or gallery views improves the usability and reach of the storms. By this reasoning, it is important to maintain a level of usability. Each feature and design needs to be

Finally, the Internet's prolificacy means that word storms can be accessed from nearly anywhere. A user has the ability to create, modify, and download word storms on different computers without needing to download code, or possess the files previously used for generation. Both the ease and the potential to share and use word storms increases when the code is deployed to the web.

Details paragraph

Evaluation

Chapter 3

Background

In this section, I present the information required to understand word storms. I describe the core component of the word storm - the word cloud, what it has been used for in the past, and how it is created. Next, summarize what a word storm is, how it is implemented, and . Finally, I describe the project with context

3.1 Visualization- TODO: This section is heavily quoted - reduce quotes (rewrite in your own words if time)

Visualization is the effective graphical presentation of knowledge[6]. There are two primary areas of visualization - scientific and information. Information visualization, that is "visualising abstract data with no spatial structure"[6], is the area of research that word storms can be categorized as. The aim of visualization is to improve "our understanding of data by leverging the human visual system's highly tuned ability to see patterns, spot trends and identify outliers"[4]. Visualization transforms data (in this case, textual data) into a graphical representation and allows us to gain an impression of the data.

These impressions should convey the nature of the data, and give insight into the data. Visualizations can replace cognitive efforts with immediate perceptions, increasing comprehension, memory and decision making. Thus, the immediate benefit of visualizing data is speed, and its long-term application is a more memorable method of conveying information with clarity. Furthermore, because of the stimulation of the visual cortex, visualizing encourages a level of interactivity and engagement that does not emerge from the data alone.

3.2 Word clouds

To understand the nature of the word storm, we introduce its subcomponent - the word cloud.

The idea of using a layout of words first came from the web navigation tool 'Tag Clouds' - an alphabetical collection of words that represented locations/links within the website. The more frequently used words in the tag cloud were sized larger, to give users a quick impression of the data and allow them to quickly follow a popular link. While tag clouds as a concept have existed since 1992, (first published in the book 'Tausend Plateaus. Kapitalismus und Schizophrenie' by Gilles Deleuze and Felix Guattari), their popularity has expanded with the advent of Web 2.0. Flickr, a popular photo sharing website, employs these tag clouds to facilitate the search of similar images. Each image is assigned multiple tags by its owner; these tags are gathered to create a tag cloud.

The word cloud emerged as an evolution of the tag cloud. Rather than being used to aid navigation, the word cloud is used for document comprehension and analysis. It displays the most frequently occuring words in a variety of colors and angles, so that the result is a compressed cloud that summarizes the important words of a document. The end result allows the user to gain a fast understanding of the important information contained within. **Give example here**.

Several popular word cloud generators exist online, but each follows a similar algorithm to create its visualization. The following algorithm is used to create a word cloud:

```
Count the words
Remove trivial words
Sort by the count, descending.
Keep the top N words for some N.
Assign each word a font size proportional to its count.
Generate a Java2D Shape for each word, using the Java2D API.
While (words need to be placed)

    Place the word where it wants to be
    While it intersects any of the previously placed words
    Move it one step along an ever-increasing spiral
```

This pseudo-code was given by Jonathan Feinberg, the creator of the popular word cloud generator, Wordle. **CITE** While the algorithm itself is simple, more advanced elements are required to maintain speed - Feinberg references last-hit caching, hierarchical bounding boxes, and quadtree spatial indexes as techniques used to speed up the intersection test (the generator's bottleneck).

In the algorithm above, the words selection first needs a number of preprocessing steps. The text is tokenized (split into individual words without punctuation) and each word is counted. Next, stop words - common functional words that add no meaning to the context (i.e. the, and, a, etc.) - are removed to prevent them from dominating the cloud. In practice, each word in the list is

reduced to its stem to eliminate duplicates like plurality and verb tenses, and a single word is chosen to represent this word. From this list, they are ranked and sized according to their term frequency (amount they occur in the document) and the first word is placed.

Use figure from Quims Code

From here, the algorithm moves in a clockwise spiral and attempts to place the next word. At this stage, the algorithm checks if there will be any intersections between a previously placed word and the current one. This check is performed by hierarchical bounding boxes **Another figure** since checking the intersection of rectangles is faster than comparing glyphs. The bounding boxes are stored in a tree for each word. During the placement section, each descending node of this tree is checked for a potential intersection. If none exists, the word is placed and the spiral increments. Once all words have been placed, the algorithm terminates and a drawing function is called to display the word cloud.

3.2.1 WordCram

WordCram is an open source word cloud generator developed by Daniel Bernier. Its implementation resembles the above procedure for word cloud creation, with the spiral algorithm and hierarchical bounding boxes already present in its code. A few notes about WordCram's implementation are relevant for the rest of the paper: WordCram uses Processing, a multiplatform imaging tool, as its means of generating the image containing the word cloud.

The word storm code, by Joaquim Castella, was integrated with WordCram - certain functionality was modified to accomodate for the word storm requirements:

1. WordCram skips words when unable to fit them within the frame. Rather than lose information, Castella increases the size of the frame to allow this word to be placed.

2. The font size is not set by the word's term frequency. Instead, Castella calculates a target area for the word to fill using **Formula awi = amin + swi (amax - amin) **. If two words have equal weight, they are represented on the screen with the same area. By this, Castella reduces the human tendency to view longer words as more important - a word with less letters is drawn with larger font to offset its length.

3. The layout, algorithm and drawing sections of the code have been separated to ease any modifications.

4. The configuration options have been reduced to their simplest form.

5. The word ranking is ordered by a normalized term frequency. Normalizing the term frequency reduces the impact of longer documents; crucial in the word storms project.

During testing, both the word make and making appeared in a cloud. This may indicate that the stemming process did not occur, or that this could be a particular outlying case that needs to be addressed. **Insert sample graph/Point to sample graph**

WordCram also features additional algorithms for placing words, but since Castella's code only applies the spiral algorithm, it is the only case we consider. Finally, a newer version of WordCram has been released, and so the methods described in this code may differ from the current implementation.

3.3 Word Storms

Word storms are an extension of the word cloud that aim to improve the analysis of a large amount of documents by using a series of word clouds. If each document is created independantly, the word clouds are difficult to compare against each other. By placing the clouds beside each other, and coordinating the appearance across them, the word clouds can convey more important information faster. This is the basis for word storms.

Word storms were created by Joaquim Castella in 2012. In the paper describing word storms **Cite**, Castella indicates there are two attributes for gauging a storm's importance. Firstly, each cloud must be a good representation of its parent document and secondly, the storm is constructed so that the color, position and orientation of the same words are shared between clouds. If these metrics are followed, then the storm will represent each document, and the visual similarities between the clouds will be easy to see. A user will be able to evaluate the entire corpora by looking at the overall storm, compare individual documents through visual similarities/differences between their respective clouds, and, as with an singular cloud, gain an understanding of the document.

3.3.1 Overall Storm Creation

The simplest manner of creating a word storm is to run the word cloud algorithm on each document in the corpora. This, however, produces none of the desired outcomes from the storm - the documents are difficult to compare against each other, and the information contains nothing representing the corpora as a whole. Instead, Castella suggests making the storm 'homogenous' - syncronize the orientation and color of any word which appears in multiple clouds.

To assist in identifying the important properties of the storm, Castella manipulates the visual appearance of the words by using tf-idf (term frequency - inverse document frequency). Term Frequency refers to the number of times a word appears in the corpora and the Inverse Document Frequency calculates how rare the word is within the corpora. By setting the font brightness of words according to their tf or idf value, the storm can showcase different factors. Using tf highlights the similarity between clouds, while idf values brighten the documents that are unique between clouds. According to Castella's paper, the storm

then sets the hue of the word randomly, while the saturation is constant to prevent dull storms. However, in the code, the values corresponded to the rgba color model, with the alpha channels relating to the word's idf value **ADD REFERENCE HERE**.

Certain aspects of the word storm are just extended concepts of the word cloud. For example, the number of words in a single cloud will be the same for each cloud of the storm. The font typography is shared across each of the storms to help maintain consistency and aid visual identification.

Four different algorithms exist for creating the storms. The first is the naive approach - generate each cloud independently. The other three - iterative, gradient and combined - create storms with the desired similarities and appearances and are considered in depth below. While the details of storm creation varies depending on the algorithm chosen, certain factors are common to all in their implementation. Each document in the corpora is preprocessed - in addition to the cloud preprocessing seen in a single cloud, the term frequency and inverse document frequency are calculated for the entire collection. Furthermore, the coordinated attributes are stored in a mapping between the word and an object containing all the relevant properties, which vary depending on the algorithm chosen.

3.3.2 Algorthims

To generate a cloud, Castella employed three different algorithms to coordinate words across multiple clouds. The descriptions of these algorithms is taken from **Cite paper**.

3.3.2.1 Iterative

The iterative algorithm aims to coordinate the position of words across documents. As the name suggests, the algorithm iteratively shifts the words in the cloud until each word is positioned within a distance of the others. To do so, each cloud is first generated independantly with the standard spiral algorithm. Next, the average position of each word across the storm is calculated, and the spiral algorithm is run again to attempt to place these words near this average.

The algorithm's speed is improved by relaxing two conditions on its convergence: the thresold distance for completion is increased if the storm could not be placeed, and a maximum amount of iterations is allowed before termination. Furthermore, the algorithm won't move words if they have already converged, and checks to see if, in an earlier cloud, the word has been placed. If it has, it places the new word as near to that location as possible without overlap.

These factors mean that the storm can be generated quickly **Quim's stats**, but each cloud tends to be sparse. The spiral algorithm shifts the words away - given enough space, each word can have its own converged location. But as the distance between words increases, the cloud's visual appeal and information content decreases. Early termination limits this effect, but these concerns needed to be addressed with gradient algorithm described below.

3.3.2.2 Gradient

The gradient algorithm reduces the sparseness of the iterative algorithm through a tailored MDS function (titled Discrepency Between Similarities [9] in the paper by Castella and Sutton). A stress value measures the relationship between documents and the relationship between clouds - similar documents/clouds will score low. A correspondance sum measures how well each cloud represents the document: Finally, a penalty keeps the words from overlapping and minimizes the word's distance from the center. The stress value, the correspondance sum and the penalty are summed together to find the DBS value. Gradient descent is then used to find the local minum to optimize the weighted parameters in the equation **Final equation here?**.

3.3.2.3 Combined

The combined algorithm aims to combine the strengths of the iterative and gradient algorithms. First, the iterative algorithm runs to completion (either termination or convergence). The gradient algorithm then performs the MDS optimization, having the effect of pulling the words, spread out by the iterative algorithm, back together. This gradient section performs better after the iterative algorithm has run - the words are optimally positioned for the gradient descent's performance. [9]

3.3.3 Implemenation

The word storm code is layered overtop the WordCram code. The code implements a preprocessing stage, where the words are loaded in, tokenized and stored. The values of the term frequency and inverse document frequency are calculated, and the index is constructed. Once the initial setup is complete, the chosen algorithm runs. The index maintains the coordinated properties shared amongst the cloud. It contains the color, angle, and idf values for the word, the average location of the word across its clouds, the word's actual locations as a list and a boolean indicating whether the word has converged.

In addition to the choice of algorithm, a number of customization options are provided. These allow the user to manipulate some storm attributes and are discussed in further detail in section 3.54.1.4.

It is important to note that this is a black box system - the user provides a folder and a series of images are returned. Apart from command line modifications (i.e. specifying the algorithm used to generate the clouds), the way the image files are generated cannot be modified. This setup proved troublesome when I added the capability to modify the storms after their creation.

3.4 Alternatives to Word Storms

Some work has been done on analyzing and visualizing corporas of textual data. Key differences exist between these and word storms, and these differences

should be mentioned.

3.4.1 Collocate Clouds

SAMPLE HERE? Cite creator* Collocate clouds can be used to analyse a corpora of documents, so the distinction must be made between these and word storms.Collocate clouds share more in common with tag clouds than word clouds - they are layed out in alphabetical order with important words larger/brighter. However, Though both tools are used to analyze a corpora of documents, collocate clouds focus on visualizing the context of certain words within a corpora with regard to other words - which words occur in the presence of others. Thus, a collocate cloud will only showcase the word with context to other words, and does not distinguish between the different documents. Word storms focus on creating similar clouds between the different documents, while each cloud will preserve the identity of the document and can be used independantly.

3.4.2 Context Perserving Dynamic Word Cloud Visualization

Cui et al propose an idea similar to the word storm [11] - a series of word clouds that perserve spatial and semantic relations. They seek to perserve the context of the word within the cloud

3.4.3 Word Storm Website

It should be noted that a 'WordStorm' project is online at www.lonji.net. This project focusses on dictionary entries and the relationships between words, and care should be taken, in the case of future deployment, to ensure that a distinction is made between the sites.

3.5 Web 2.0

Within the project scope is the requirement that the website be 'Web 2.0'. Given the number of varying definitions for what Web 2.0 is, I should define Web 2.0, and the list of features required by the website to meet this definition. An article by Tim O'Reilly, President and CEO of O'Relly Media (a company closely associated with popularizing Web 2.0) loosely defines Web 2.0 as

1. Having user driven content - A user's content determines the behavior of the website. The website should focus on this user data, and provide context and functionality around it.
2. Focussed on the Long Tail - Targe
3. Providing a Rich User Experience
4. Using Data as the Intel Inside

5. Designed to

3.6 Ruby on Rails

3.6.1 Rails' Strengths

The project was implemented using Ruby on Rails . By using a framework, the basic website functionality comes ready-to-use. This decreases development time, critical in a time-sensitive project. The overall security of the web site increases - known security exploits are safeguarded against. With websites like Basecamp, Twitter, and Github running Rails, the Rails' community is one of the largest web frameworks. Such a prolific nature enables an easier effort with debugging, and provides greater support.

One of Rails' key strengths is its ability to rapidly generate functional code with minimal effort. This scaffolding code forms the basis for each Rails application. These quotes, from the Rails website, reflect the consensus of Rails' speed as one of its strengths:

> *"[W]eb designers and software engineers can develop a website much faster and more simply"* - Bruce Perens, Open Source advocate
>
> *"Powerful web applications that formerly might have taken weeks or months to develop can be produced in a matter of days."* - Tim O'Reilly, founder of O'Reilly Media[1]

The ability to create prototypes quickly and experiment with different approaches is critical in a time-sensitive project such as this. Choosing Rails maximizes the time spent on creating the main functionality of the word storms website, and less time on the less important aspects.

3.6.2 Issues with Scalability

A well-known issue with Rails is its purported lack of scaling[2]. With Twitter's high-profile departure from using Rails as a message queue back-end in 2008, questions arose whether Rails could handle scaling. In the creation of a scalable website, such limitations must be addressed before choosing such a framework. With our website, it does not matter for two reasons:

1. Rails Scalability in General - While the common consensus online is that Rails cannot scale, there are counterpoints to address. Twitter's front-end still uses Rails and can handle the traffic with minimal outage. Other high-traffic websites, including the U.S. based video site Hulu and the YellowPages site, use Rails without difficulty despite several million users. The volume of users does not affect the scalability of Rails.

[1] http://rubyonrails.org/quotes
[2] The website http://canrailsscale.com/ displays a single 'NO' on it.

2. Twitter's Difficulty Stems from Tweets - Tweets are designed to be constant, rapidly flowing information that require constant updates to the database. Twitter's global reach also implies that such data must be stored in databases spread across dozens of countries and locations. To improve performance, websites can typically employ caching to decrease server load. In Twitter's case, caching would be rendered ineffective in a few minutes by the influx of data. Contrast the nature of tweets with the static nature of a word storm. A single call to the database is required to fetch the word storm, which can then be cached for future viewing. The issues faced by Twitter are inherently different than the ones faced by the word storm generator, so Rails as a framework poses little risk.

Due to the academic nature of this website, it is unlikely that it will achieve the levels of traffic experienced by Twitter or other high-profile sites. Such considerations, however, reflect the foresight required for software development and should be considered**Word used twice** in the case that such a website receives considerable traffic.

Chapter 4

Goals and Requirements Analysis

4.1 Requirements Capture

In the Software Engineering discipline, an important step in the creation of a software project is the requirements analysis. Such an analysis allows a focus on obtaining the necessary functionalities of the program. In commercial applications, the application requirements are gathered from the user - in this academic exercise, without a set of clients to elicit these requirements, I treated myself as the typical user. From the requirements, I identified the following goals to reflect these needs:

4.1.0.1 Usage of Word Storms

Some of the potential uses of the word storm website are identified as follows:

1. Temporal evolution - View the history of a series of documents and discover how they have changed or evolved over time. For linguists, one could analyze the similarity of words over time, while a historian may discover trends in historical documents. In the context of the web, users can view how their Facebook/Twitter content has evolved over time.

2. Literature analysis - Determine important thematic elements that appear in an author's novel (analysing chapters) or entire work (analysing an entire bibliography). In the context of the web, blog posts can be scruntized in this manner.

3. Analysis of Ted Talks – By taking the transcript provided at the bottom of certain Ted Talks, an individual can quickly compare content of individual videos to identify videos of interest.

4. Analysis of Critical Works - Summarize a series of critiques provided by reviewers and compare the similarities to identify the common consensus about a work (i.e. film, game, or art).

5. Summarize Lectures - Present the core concepts of a year's lecture material into core concepts and identify where common topics appear. This can be applied to both university level education, or early schooling years.

4.1.1 Adapt Word Storm Code for online usage

Code for creating word storms was written by Joaquim Castella during the implementation phase of his project. This code remains much as it was, though adjustments to the code had to be made. The code was created with a local implementation in mind, relying on local file systems and a number of Java jar files. My task was to adapt the code in such a manner that the functionality of the program remained untouched, but allowed the necessary customization. Steps had to be taken to ensure that the code would remain as modular as possible.

4.1.2 Allow Users to Upload Files

To generate a word storm, the user needs to provide a series of files for the algorithm to process. In a client-server website, users should upload their files to the server. The server then performs the computations and returns a word storm for the user to save or share.

With a web 2.0 website, the convenience must remain with the user. Functionality should be guided by the user's expectations and intentions. Certain behaviours are expected when dealing with files. Users, when uploading individual files, will expect a standard file navigation system. Furthermore, it should be convenient for the user to upload multiple files. If the purpose of the word storm is to generate a series of word clouds based on a corpora of documents, the entire corpora should be easily uploaded.

4.1.3 Generate Word Storm

From the uploaded files, the user should be able to create a word storm with as little hassle as possible. If the user has uploaded a series of files, the word storm should be created from those files and displayed as soon it is ready. If the user chooses an alternate means (i.e. Twitter username), the server should perform the actions to create the word storm without user input. The process should involve as much automation as possible, to avoid confusion with the user. **HCI Principle**

4.1.4 Customize Word Storm **Give as past test**

Word storms were designed with a number of customization options. Each of these should be easily accessible to the user**HCI Principle** and describe the

effects the option has on the storm. Also, given that the run time can increase dramatically if certain values are changed (i.e. iteration time), the user should be warned about the potential duration that it takes to create. Additionally, these options should not interfere with a casual user's use - these options should be optional . The possible customizations are as follows:

4.1.4.1 Algorithms

Castella's code had four algorithms to choose from, which result in slightly different clouds. The user can select:

- Independant algorithm: The naive algorithm that applies the WordCram code to each storm independently.
- Iterative algorithm: This algorithm iterates over possible positions and slowly shifts the words to a point of convergence across all clouds.
- Force algorithm: This algorithm forces a layout based on the DBS gradient approach.
- Combination algorithm: This hybridized algorithm combines both the Iterative algorithm and the Force algorithm. This is the current default of the program.

4.1.4.2 Number of Words

The amount of words in each cloud can be set. The default value in the implementation is 25. The user can choose a number between a maximum value and a minimum value, depending on performance speeds.

4.1.4.3 Word Angle Synchronization

The words that occur in multiple clouds will have the same angle by default. The user can have the option to remove this constraint.

4.1.4.4 Word Color Synchronization

The color of the words can vary depending on the color scheme chosen. The current options allow the colors between clouds to be synchronized or to remain independent.

4.1.4.5 Word Scale Synchronization **Add**

4.1.4.6 Tf-Idf Options **Check**

The user can select which statistic to use - term frequency or the combination of the two.

4.1.4.7 Font

The user should have the capability to change the font of the word storm. However, each individual cloud will not be customizable. The font selection should take place from a short list of predefined fonts, with the option to extend it to other fonts in the future.

4.1.4.8 Iterations

The user should be able to control the number of iterations the word storm algorithm goes through, as this number will produce different clouds. The more iterations, the more accurate the positioning of the words will be, at a cost of time. This trade-off should be customizable.

4.1.4.9 Width/Height of Image

The final width and height of the individual clouds can be set (to a maximum value). The user should be able to customize this if they seek to limit the final size of the Storm.

4.1.4.10 Cloud Positioning

Otherwise, the users will want to be able to arrange the Word Clouds in such a manner of their choosing. Thus, the images should be positionable about the screen and savable in these positions.

4.1.5 Scaling Website

If the amount of users or the amount of CPU stress exceeds the server's capabilities, the word storm website should demonstrate the capability to scale. This means that, under duress, additional servers should be added. The measured performance of the website should not decrease due to an increase in users. Such scalability also prevents the worse-case scenario of a complete site failure.

4.1.6 Saving and Sharing

One of the key elements of the Web 2.0 website is the capability for users to share information between other users. In the case of a word storms project, the ability to share the created word storms is a fundamental feature. This can either be achieved by having a share feature button (which could send out the word storm as an email), or including a gallery view where users can showcase a list of previously created word storms. Once the user is provided with the link, each of the created word storms can be visible. Privacy options should also be considered when dealing with user interaction - some users will prefer their information/creations to remain anonymous. Sharing should therefore remain opt-in.

Users should also be able to download and save the word storms for later viewing. This static image, containing the word storm images in their user-selected locations, should be a PNG to prevent loss of quality (critical in the case of a word storm composed of dozens of clouds).

4.1.7 User Login

To maintain the user preferences described in the previous section, and to ease sharing between users, the website should feature a user login system. The user should have a home page dedicated to their previously created word storms, and have their customization settings related to their username.

This system should also remain secure and provide the functionality expected of a website.

4.1.8 Testing

The website should be tested with regard to two overall goals.

1. The software should be tested to avoid any software bugs. The standard practice to ensure this is to build a test suite to test the functionalities of your code. This involves constructing a series of unit tests to test individual code segments. There are also automated tests that ensure the functionality behaves as expected. The web tool Selenium will provide an additional layer of testing to ensure that each button on the web page is functional and responsive.

2. The software should undergo system testing to verify the overall functionality of the program. These tests will involve stress testing (to test the capabilities of the database) and scalability testing (to test the scaling functionality). Both of these require an automated load generator to quickly create varied data usable for such tests. Therefore, a sub-requirement of this section is to create a load generator.

4.1.9 Design

The initial design began as a series of sketched layouts of proposed user interfaces. To maintain simplicity, the pages choosen for the project were as follows:

- Home Page - A page to house all of the navigational links within the site, as well as provide a brief introduction to the concept of the word storm. A gallery view of globally shared word storms was originally intended to appear here.

- User Page - A page dedicated to an individual user. A gallery composed of the user's word storms is the primary focus of the page. The preferences of the user would also be indicated here (currently only the capacity for sharing is an option, but further options could be placed here as well).

- Upload Page - Structured as a series of different tabs, each with a unique method of creating word storms. One tab would be dedicated to uploading documents, another would allow the inclusion of a Twitter enter, another would allow for PDFs, etc. Options for customizing the word storm are presented as a dropdown menu on each of these tabs, and a button for the creation of the Storm appears at the bottom of the page.

- Word Storm Edit Page - A page that displays the individual Word Clouds. Each of these clouds is adjustable and there is an option to recreate the cloud based on a new series of options.

- About Page - Page that describes the history of the word storm content, and a brief description of how the word storm algorithm works. Also provided are links to relevent sites (Word Storm github code, Joaquim's Paper, Word Cram)

The actual design of each of these pages should follow the design of these pages as closely as possible. Smaller details, like the positioning of the buttons, did not reflect these initial designs. Furthermore, to aid in navigation, a navigational bar was placed at the top of the page. The constant presence limits the screen size, forcing certain designs to be constrained. Additional pages (such as a Login Page/Signup Page) were also added as needed for basic online functionality.

Chapter 5

Implementation

In this section, I will outline the actions taken in the creation of the web site, and justify their necessity. I begin by discussing the modifications and additions that I have made to the word storm code. I outline the website structure and design decisions. I finish with a summary of the work done

5.1 Overall Design/Pages

The final design

5.2 Word Storm Code

5.2.1 Modifications to Original Word Storm Code

The word storm continues to be developed and enhanced. Any additions made in my implementation could not change the core development of the program. However, certain modifications were made to adapt the code for remote access. Each of the customization options 4.1.4are passed to the launcher file - if the prerequiste amount of arguments is not there, an error code is returned. This helps the server validate if the program ran successfully. A number of other minor functions were added for ease of access and functionality (i.e. changing the font was not originally present).

In the middle of the project, a new version of the code was released by Joaquim Castella. My design needed only a few minor modifications to adapt to this new code (mainly to include a call to intialize the new algorithm).

5.2.2 Integrating the Previous Code

I have packaged the code into a executable jar file. This file is located within the file system of the Rails code. This code is executed after the user has pressed the 'Create Word Storms' button. This passes the current customization options, as

well as the necessary file locations, to the jar file. **Strengths and weaknesses of jar files**

5.2.3 Word Storm State

Because the word storm code was a closed system - given a file input, it returned a series of images - any changes to the storm required generating it from scratch, and then applying the changes. Given the time to create a storm (**Statistic**), and the changes to the code needed to implement this, I chose to add the concept of state to the storm. The idea is that, once the code has created a storm, it saves the positions of the words, their colours, their relationships and the configuration used to generate the storm. From this saved file, the user can reload it at a later instance and perform modifications.

To achieve this, I used Java serialization to save the storm object as a byte file in the target folder. Serialization

There were three downsides in using serialization.

1. Any changes to the storm object or any subclasses renders the object unserializable - it cannot be loaded if changed. There are two possible solutions:

   (a) Rerun the original loader if the file is changed - whenever the storm is unserialized, it checks if it can still unserialize. If an error is thrown, the code generates a new storm from the original documentation, and the storm is saved again. To implement this method, each of the file uploads needs to be stored long-term (removing one , increasing the size of the storage needed, and

   (b) Introduce versions into the code. If the . I opted to implement the second one - the storm is saved in the database with the version of the code it was created with. Whenever the word storm code has changed, the newest version should be placed into a new folder labelled as the version number and the version label changed in the configuration file. This means that, despite modifications to the storm code, the serialization process will not break for this particular instance. This increases the burden of maintenence, and requires users to regenerate the storm if they wish to access features of later versions. However, this way prevents every storm from needing to be regenerated, and lessens the amount of strain on the server, as it does not have to store the files.

2. The precise state is not saved. So any information pertaining to relations between objects is not reloaded and must be reconfigured. This means that anything saved within the context of the processing library does not maintain its state and must be reloaded.

3. The code itself was not immediately serializable. Each component of an object needs to be serializable before the object can be. Because since the

WordCram code uses the image library, Processing, the imaging aspects could not be serialized. To fix this, I refactored the code so that each accessible object (the WordCram object, the Word object, the EngineWord object, etc) was serializable. Next, I added subclass to replicate the code in the processing library - each subclass extends a particular aspect of the graphics library by calling the super method (PVector became MyPVector, PGraphics became MyPGraphics, etc).

(**Describe what factors you do get out of serialization - configuration file, coordinated property attributes (apart from location) **)

From serialization, there are a number of benefits. All of the relevant words in the storm are saved to file, so the original documentation could be removed.

From these changes, the word storm object could be serialized.

5.2.3.1 Saving and Loading the Storm **Quicker**

However, serializing the object at the end of the program, and unserializing it immediately did not replicate the storm. The positions of the storm are not saved within the coordinated property, due to the loss of the relationship with the processing library. All MyPVector instances within the CoordProp class needed to be written to file and reloaded.

If the values are saved after drawing, the storm will be reloaded as a zoomed in version of itself. When drawn, the storm and all word positions are scaled and translated - the fraw() function crops the images to the size of the largest cloud in the storm, and scales the image to the ratio of 640:480. If the values are saved after the storm has been drawn, each of the values has been scaled by a relative factor and the image appears zoomed in.

To fix this, I saved the state of the storm and the coordinated positions of the words before drawing. Reloading the storm instead resulted in a storm where the positions of the words are correct, but the colors, set during the drawing process, were wrong. Therefore, the entire saving process was split into four sections: saving the word storm as a serialized object, saving the positions of the cloud to file, drawing the images to file, and then saving the chosen colors.

The resulting files (three for each storm) are stored on the local server in a file system. These files take up a small amount of memory - for a storm of six documents, the three files are comparable to a single image (60 kb). **When testing a large amount ~100, make sure you record the size**

Reloading the storm requires a number of steps. First, the storm is read back from file. Then, the rendered place and the target place of each word is reloaded back into the index that relates the word to a coordinated property. These values represent the pre-drawn values of the storm. Next, each word in the index has its color reloaded from the saved file. The configuration of the storm is taken from the serialized object, and a new MyPApplet (the serialization-correction subclass used to draw the storm) is created. The storm is reloaded into the applet, and the defaults are reset. From this state, the word storm performs as before it was saved - if drawn, the image will be the same.

5.2.4 Additions to Word Storm Code **Higher Level**

5.2.4.1 Moving a word

I implemented an extension of Castella's code to move a word. After reloading the saved state of storm, I pass four parameters to the file. The first value is the pixel coordinates of the word you want to move. The user can only interact with the image, so these coordinates are given relative to the scaled 640:480 ratio. The second value is the pixel coordinates of where the user wants to move the word to. The third specifies the cloud to manipulate and the fourth specifies the output folder.

The algorithm converts the pixels into local coordinates (before they are stretched) by dividing the pixel coordinates by the scale used when it drew the storm. To find the target word, I iterate over the list of words in the chosen cloud and check if the pixel value is within that word's bounding box (the box used to draw the word). The index then updates this word's values to the target pixels.

Castella's algorithm then can be applied to these new word locations. The iterative algorithm corrects any overlapping values caused by the move, and the gradient algorithm improves the overall appearance, exactly as before. As a result, our cloud will appear visually similar to the previous clouds, but the target word will be in the correct location.

5.2.4.2 Changing the color

To change a word's color, four parameters are passed. A pixel value, as a coordinate on the screen, represents the word that should be colored (found via the bounding boxes). As with moving the word, the color's input coordinates must be converted to local coordinates. The second parameter is the color to change the word to, represented in RGB space. Because the alpha value represents relationships within the storm (the idf value of the word), I have not allowed the user to change it. Note that this differs from Castella's original paper, and does not use HSB.

Changing the color of a word is simple. Once the storm is reloaded, the index mapping words to the storm's coordinated properties is accessed. The color of that particular word is set to the corresponding rgba value (where alpha is set automatically). The storm is then drawn with the new color for the chosen word. However, this implementation assumes that all words are synchronized by color. If the color's are not synchronized,

Both of these are created and executed as .jar files, placed in the same path as the original storm code. These jar files should be subject to the versioning required for deserialization. Another constraint is that the files need to be ordered alphabetically - my code saves the storm state alphabetically. The cloud id must be set according to its alphabetical index, or the wrong cloud will be chosen. s

It is possible to call these functions any number of times in sequence - a user can move one word, color another, and move a third. The state will remain

constant and accessible. However, only a single word can be moved/colored at once - once the user can only make one change before the storm is recalculated.

Example

5.3 Website Structure and Implementation

5.3.1 Website Architecture

The code was structured using the Model-View-Controller (MVC) architecture, something inherent in the design of all Ruby on Rails projects. MVC architecture divides the code logic into seperate sections - the model contains the data, the view contains the user interface information, and the controller manages the flow of data between the two. The user/client only sees the view, and interacts with the controller. The benefits of using MVC include structuring the code in a logic manner, allowing modular views that do not depend on the data involved, isolating business logic from the view[1], and allowing multiple views to be active at the same time[2]. There are only a few drawbacks for using MVC - it can overcomplicate simple projects, updating the controller almost always means updating the view, and it sometimes can be difficult to determine where in the MVC the logic needs to be.

In relation to my code, I structured it as demonstrated in **Graph**. The view holds my HTML code, the controller contains the functionality, and the model represents my database (5.3.2). Because the word storm code is contained within a jar file, and is not included within the main system, I have indicated it's presence and interaction outside. In addition, the call to 'Amazon S3' is only in the case of online functionality. In the previous iterations, the calls to the cloud were represented by a local file system. Toward the end, the storage service was switched to S3 and is represented in the diagram as such. **Fix chart**

5.3.2 Database Setup and Relationships

The database utilizes Rails' built-in Active Record objects. Each entry from the database table has a corresponding object associated with it. Calling the object's methods simulates SQL calls to the database (Object-Relational mapping), rather than using explicit SQL language. Beneath this, Rails employs a database management system. Multiple systems can be used - I initially used the default, SQLite. However, SQLite does not provide adequate scaling - each read to the database locks the entire database file [8]. To prevent issues with multiple users accessing the database at the same time, I chose to change the database system to PostgreSQL.

Swapping the system was trivial, due to Rails' abstraction of the database. PostgreSQL allowed the Ruby Gem Rubber to autodeploy the website to my EC2 instance (5.9.1). PostgreSQL also provides backups, general support for scalability, security (a username and password are now needed in the database.yml

file to run) and other advanced features. These features should ease maintenence and future development.

I created three different Active Record objects (User, WordStorm, Image), and used two others (Uploads and Settings), adapted from online code. Each active record object has, by default, three values - an auto-incrementing ID for each entry in the database, and two datetimes (Created_At and Updated_At) for when the entry was created and when it was last updated.

5.3.2.1 User

The User object represents the individual using the website. A database entry is created when the user signs up (a mandatory action to use the websiteby design), with a username and an email. The password is stored as a salted encryption (to prevent both rainbow table attacks and reading the unencrypted value in the case of an security attacker) and verifies the user upon logging in. The storm_num value represents the amount of storms that the user has created, and is used in the creation of a unique filepath for each storm. For security, usernames and emails must not have been entered before, the password is stored as a salted hash in the database, and a password confirmation is required before sign-up is allowed.

5.3.2.2 Uploads

The Upload object represents the files uploaded to the server. This was sample code taken from **Cite**. The upload contains a name, content_type (in our case, always restricted to text), and size of the file. The entry is only created once the file has been successfully uploaded - if there is an error with the server, or the user cancels the upload before . These entries can be deleted once finished, and are automatically deleted once the word storm has been created. Further information can be found in 5.4.2.

5.3.2.3 Settings

The Settings object is created via the ledermann-rails-settings gem **cite**. This integrates with a Active Record object and allows the settings to be set without creating an entry in the database. Further settings can be added without needing to perform a migration (adding new columns to an existing database table), allowing for user settings to be added by default.

Here, each of the customization options used in the storm generation are saved as part of the user's settings. When a new user is created, these values are set to the defaults (the original values found in Castella's code).

5.3.2.4 WordStorm

The WordStorm object represents the individual word storm. A database entry in this table is created whenever a user successfully generates a word storm. The file location represents where the data files and image (if not using Amazon

S3 (5.9.2)) associated with the word storm are located. The name is a user chosen name to describe the storm (defaults to 'Untitled'). The size represents the width of each cloud when displayed in the gallery. The height is calculated via the width to maintain the 4:3 ratio (i.e. full size is 640:480).

The version value indicates the version of the word storm code used for creation. This is my method of solving the backwards compatiability. If the storm's state wants to be accessed, the file system will look in the folder with the version number and call the code from there.

The algorithm represents the the manner in which the storm was generated - certain storms have different properties than others. The algorithm value is used to check if a particular storm can be colored or manipulated.

5.3.2.5 Image

The Image object represents the individual word clouds. A database entry is created for each unique image/document in the word storm. The file location represents the individual path for the image - this can either be the value of a . Pos_X and Pos_Y values represent the draggable positions of the image within the storm.

The user has several associations to maintain the relationships between the tables, and to ensure the correct information is updated during creation or deletion. A user has many uploads and many word storms (has_many relationship specified in the model). Each word storm and upload can only belong to one user (belongs_to relationship), and a word storm has many images (has_many relationship). An image belongs to a single word storm (belongs_to relationship).

The complete graph of the dependencies can be seen in the following entity-relationship diagram.

INSERT GRAPH HERE

5.3.2.6 Storing Files

Neither the files created to save the storm's state (5.2.3) nor the images are saved in the database. These files are either stored in the local file system or on the cloud. This was done under the initially assumption that databases would suffer from performance issues when loading images [16]. However, research from Sears et al. [17] suggests that BLOB (binary large objects - images in this example) under 256KB perform better in the database. Because of the large amounts of white space in each of our images, the file sizes do not typically exceed that amount (this is dependant on the algorithm used, font choice, letter case used, and number of words in a cloud). The images could be stored in the database from this paper's findings.

However, the paper also suggests that fragmentation resulting from storing a file long-term is easier to handle by the file system. Additionally, the database management system would need to be maintained - the relationships between the images in the database and the original storm needs to be preserved in the

database itself. By storing the images online reduces the strain on the server's bandwidth.

5.3.3 Tools/Packages Used

5.3.3.1 Gems

Ruby on Rails makes extensive use of Gems - packages that contain useful functionality. The website makes use of following gems:

rails: The gem allowing Rails to run

ledermann-rails-settings: A settings package which creates a database table for an ActiveRecord instance.

jquery-rails: A gem that allows easier integration with JQuery

jquery-ui-rails: A gem that allows easier integration with the JQuery UI

jquery-fileupload-rails: Upload manager that uses JQuery

sqlite3/pg: Database used within the application. Database System changed in middle of project 5.3.2

paperclip: File attachment library for an ActiveRecord instance, used for document uploads

twitter-bootstrap-rails: Appearance

bcrypt: Encryption tool used to safeguard information within the database

MORE NEEDED

5.3.3.2 Javascript Useage

JQuery and, by extension, Javascript are used extensively within the word storms website. Rather than allow users to access certain elements of the website and risk errors, each page checks for the presence of Javascript, and redirects the user to a page . that redirects users to the main page and ask them to reenable javascript before proceeding. Doing this ensures a consistent experience, and limits the potential errors. The decision was made to disable the website if there was no Javascript on the basis that certain aspects of the website would not work. Rather than provide the user with some functionality, and provide a potentially faulty service, in accordance with the HCI principle **1 needs to be cited here**, the website can only provide full service. The users expectations are never failed and the user is never confused about a feature that would work on one computer and not another. Additional work could be done to find workarounds to guarantee all functionality, but I did not have the time. **see if you can find some statistics about stuff**

5.4 Users

Users are required to sign-up to generate a word storm. This design helps maintain the file structure for both uploads and created storms. To sign up, users provide a username, password and email which is then entered into a database. Users have their own homepage associated with their username. There are two

showcase a gallery of all the word storms that the user has created. The option to share the images is tied into the user preferences as an opt-in choice. Also related to the preferences are the previous configurations for a word storm (font choice, tf-idf configuration, etc). These choices satisify user desire for privacy, while providing a greater user experience.

5.4.1 Profile Page

5.4.2 Uploading Files

Single file upload is a trivial issue - many different versions exist online. However, word storms require a corpora of documents to function effectively. In the case of a hundred files to analyse, users will not want to individually upload each file, so alternatives must be considered. Several methods were considered for users to upload their files:

1. Zipping the Files - A singular file upload can be used if a zip is passed to the server. However, the contents of the zip need to be verified, the user is inconvienced by needing to zip the files beforehand **HCI principle**, and the server needs to spend time unzipping the file. For these reasons, this was discarded.

2. File Transfer Protocol - Either standard FTP (File Transfer Protocol) or SFTP (SSH File Transfer Protocol) could upload the files. Using FTP can potentially open a number of potential vulnerabilities to the server (including bounce and spoof attacks) while SFTP requires additional setups for the authentication and a public key system for each user. If security was a priority, I would have chosen this method.

3. JQuery Uploader - A Ruby gem that allows for simultaneous file upload. As a gem, its functionality has been secured, verified and tested. It provides a simple user interface (showcases the progress for the upload), and does not require flash. The files are displayed on screen using an AJAX call, so users receive visual feedback for their interaction with the system. For these reasons, this was the uploader chosen. **HCI**

The uploaded files are attached to the user who uploads them and are stored temporarily in a database with a dependency on the users. Users are currently only allowed to upload files if they are logged in, as this simplifies the database for testing purposes. Future implementations could implement cookies as a means of allowing temporary users to create word storms.

Uploading files are limited to text files, due to the constaint in the original codebase. **REPEATED** At present, the contents of these documents are not checked for malicious code. Limiting the file types to documents only mitigates the problem - a clever hacker could still attempt an injection attempt. Such a santization effort needs to take place at the level of the word storm creation, at the cost of speed. Hence, all measurements recorded during the academic testing were done so without sanitation in place - metrics would increase corresponding

by the thoroughness of the sanitatization. In practice, the files would be ignored if their contents contained unsafe information, as processing such code could take down the word storm website.

5.5 Creating the Storm

After the files have been uploaded, the required options are passed to the jar file (input and output locations, and customization options) and the word storm code is executed. If the results are okay (the files are found, and the image can be created), the word storm is added to the database. Each of the created images is then added to the database

The user is then redirected to a page where the results of the word storm are displayed. This is a collection of thumbnail images arranged in a series of draggable boxes. When the images are clicked, their contents are displayed in full view on the page using the Lightbox JQuery plugin. Users are able to view the individual Word Clouds this way (and save them to their local file system if desired).

On this page, there are three options:

The user can name the storm.

The user can save the storm.

The user can adjust the size of the storm's clouds. **Describe**

This will create a unique saved object that contains the unique positioning of the word storm (see 4.7 for further details). This saved word storm forms the basis for the entry into the database and is what the user will share with other users. Finally, the third option allows users to recreate the cloud, based on the series of customization options. This requires regenerating the cloud from scratch (and looping back to the same page) and potentially taking more time, depending on the options selected. Therefore, for this reason, clicking the 'Recreate Storm' button showcases a warning indicating the possibilities for recreating a word storm.

Once the storm has been created, the files uploaded to the server are removed. This is done for two reasons. Firstly, to perserve a large amount of data over time in a database is costly, will likely cause runtime delays, and foster questions about the legality of storing data for long periods of time. Secondly, eliminating the files removes the user's question as to whether the files they upload in their next session belong to a new word storm or an old one. By assuming that each new upload starts a new word storm, it removes the possibility for conflicting behavior.

However, the tradeoff with this approach is that word storms will not be editable after their initial creation (in terms of the individual clouds or selecting the options). Users will have to reload the files if they want a different word storm. However, the word storm can still be adjusted with regard to the positions of the individual clouds. (Note: This may change if I can call the iterative algorithm to change individual words - then future customization might be worth saving the data).

5.6 Customizing the Storm **Clean up**

To customize the word storm, the user is presented with a tab of the options listed in Section 3.5. This option is located in the same place as the file upload, to further the idea this is the creation page. The tabbed menu allows for casual users to generate a word storm without worrying about the details. The settings are set to reflect the user's preferences, and are, by default, the standards defined in the original word storm code.

These options are presented as a series of sliders, menus and buttons. By eliminating the use of number forms and text boxes, the user does not have understand what are a range of suitable numbers or selections. This allows them to use the word storm without having to understand the precise implications of each of the options. Furthermore, to clarify the understanding, hovering over the individual options provides an indepth analysis of the functionality.

After the user has created the storm, he is directed to a display page with each individual cloud displayed as a thumbnail. These thumbnails not only present the user with a method of viewing their clouds (and scrutinzing whether the customized options are to their liking), but as a means of further adjusting the word storm. Each of these thumbnails can be dragged about the page, and certain clusters can be positioned nearby. Once the object is saved, the values of each of the images is saved as part of the word storm. That way, in the future if the user wants to view the Storm, the clouds are positioned in the manner that he has customized and still fully functional (with regard to viewing individual Word Clouds).

As stated above, once the object has been saved, these customizations are permanent. The user needs to create an entirely new word storm from scratch if wanting to customize.

Changing the font was also not present in the original code as an option. As part of my modifications, I added this to the storm configuration file.

5.7 Viewing the Storm

5.7.0.1 Gallery Layout Page

There are two different layout pages for viewing storms. Both galleries have checks that ensure that at least one storm is available for viewing before displaying. If a storm is not available, the user is redirected to the upload page, with a prompt stating the error . Each image in the gallery views can be clicked to increase its size and be displayed within a modal dialog. **Show here** I am using a modified version of the JQuery plugin LightBox2 by Lokesh Dhakar. These modifications were introduced by myself as a means of manipulating the image - clicking left and right will perform actions necessary for editing the storm (before they were not allowed). **Reference** **Search function to ease use**

User Created Storms The first is a gallery of the user created storms. This page allows the user to view their created storms, and access . The second contains a list of all the storms that have been created. This gallery shows the storm's name, and the user who created the storm if the privacy settings allow it. A cookie is used here to maintain the position, so that, if the user exits and returns, the storm they were previously viewing is reloaded.

All Storms **Web 2.0 justification**

5.7.0.2 Sharing the Storm

When the share function on either the user's gallery page or universal gallery page is pressed, the user receives a prompt to enter an email address. Upon submitting, the email address is verified via both a regular expression and the Sanitziation gem, and an email is sent to the target address. The server needs to have smtp configured for this to work (i.e. postfix on Ubuntu). Using email to share storms increases the reach of the storms and provides a level of familiarity to the users. **CITE INFOGRAPH HERE**3

5.7.0.3 Editting the Storm

A link from the user's gallery page allows the user to edit the storm. The edit page allows the user (and only the user who the storm belongs to) to manipulate the storm. The edit page

5.8 Twitter Scraper

To ensure extensibility, I have designed the website to allow alternative means of uploading documents for storm analysis. As a proof of concept, I constructed a Twitter Scraper to create a storm from a Twitter username. I access the Twitter API and fetch a list of the usertweets (up to a limit of 200 as per Twitter's design) as an XML file. I then parse the file for the Tweet's content. Because each Tweet is limited to 140 characters, the resulting storm is a large number of clouds, each containing little information.

To gain more meaningful clouds, I grouped each tweet by month. However, this posed a problem for individuals who tweet quite often - the BBC, tweeting around thirty times, has one or two clouds created before Twitter's access limit is reached. Instead, I limited the amount per cloud to thirty tweets, an arbitary number to divide the clouds. Further refinement would require preprocessing (i.e. calculate the average number of tweets per month and divide the clouds accordingly), but this serves as a proof of concept.

Integrating further scrapers/uploaders into the code is trivial. Possible extensions, or a Ted Talk scraper) would either need a new upload page (i.e. a PDF uploader) or a new scraper (i.e. enter a web site URL and scrape the data). The files gathered only need to be preprocessed into text files before they can be analyzed as a word storm.

5.9 Scalability

To achieve scalability in this project, two methods were looked at. EC2, Amazon's virtual servers, are created on demand as customizable instances. The goal of using EC2 was to allocate the necessary servers needed for this project, and increase them as needed. EC2 allows customization of the server size, number of servers that work in conjecture, and the operating system specifications for each server. Amazon provides load balancing to ensure that each instance receieves an equal amount of traffic.

The second, S3, is Amazon's online storage facility. The storage itself is scalable - the amount of storage needed will increase upon demand. It also provides a level of security. S3 represents its overall structure in a bucket. Information can then be pushed to and fetched from this bucket using S3 requests. These requests decrease the bandwidth cost on your server by shifting the load to Amazon.

By using Amazon's services, the information is provided online and accessible from multiple locations. The cost depends on your use, and a servers can adjust to cope with an increase in useage. The size of Amazon, and the number of high-profile customers (**list**) ensure that the software provided is maintained, available, and the latest security features are implemented. Both the EC2 instances and the S3 bucket were created from Amazon's Ireland location, and must be accessed from these locations.

5.9.1 EC2

EC2 allows websites to add instances of a particular server when needed. Each server would perform a specific operation - one would act as the database, one as the web portal, and the other as the application. If the load increased on the database, the database could replicate and horizontally scale by adding another database. This database, a slave to the original, would decrease the load undertaken by the master. A load generator performs this switching functionality.

To deploy a Rails application to EC2, a gem called 'Rubber' can be used to deploy to the instance using your Amazon credentials. Using Rubber required the use of a Linux OS, so the project switched operating systems (6.0.3.2). Each of my servers - application, database, and web - ran an Ubuntu instance provided by alestic.com.

5.9.1.1 EC2 Issues**HIGHER LEVEL**

After deployment to EC2, the instance's permissions prevented the client from uploading files to it (the user needed write permission). Each time code is deployed to EC2, a mounted directory is created with new file permissions. I would have modified the deployment mechanism to automatically set permissions, but had a larger issue not taken priority.

When attempting to generate the storm from the uploaded files, the Processing library threw an error with regard to the display. The Processing library

was not '..designed for headless run'[10]. It needed a display to generate the images, something that the Ubuntu EC2 instance did not initially provide. X11 can be bypassed by ssh-ing with the command '-X' to enable X11 forwarding, but this route is not optimal for a web application.

Instead, I looked at setting up an instance of X11 on the server. By setting up xvfb (a frame buffer to perform all graphical operations in memory), and using a hack from this blog post[10], I could run a bash script to set up an xvfb on the instance and execute the code. However, this doubled the runtime for a small sample (three documents), and introduced a number of potential security flaws (with regard to necessary permissions to execute the bash script). Furthermore, the multiple files required to configure xvfb to run without errors pushed the size of the server configuration to 3GB of the 7 provided. Each new instance would have to undergo this configuration when added, undesirable if autoscaling were implemented.

As a result, the code was taken down from Amazon EC2 and the instances torn down. In my opinion, the issues surrounding this hack were not worth the added effort and concerns solving it in this manner would solve. I could not find other alternatives for running processing on a headless server. Instead, I would contend that, if the website wants to use Amazon's EC2 instances, then the foundation for the word storm - WordCram - needs to be refactored to not require graphical memory to use. The time commitment involved in the large-scale refactoring of the codebase was deemed too great. The EC2 code, autogenerated by Rubber, remains in the code in case of future use.

5.9.2 S3

Instead, I used S3 to host the images online. Once the images are created from the storm, each of the files is pushed into an S3 'bucket' via the S3 gem. The bucket is structured in the same style that the original file system used - each user has a folder within a bucket; each storm has a folder inside that user's folder. Upon successful upload, the word storm is created as normal (the path still points to the local file system).

Each of the image database entries are modified - their file location, instead of pointing to a local file, points to an amazon URL (i.e. https://s3-eu-west-1.amazonaws.com/wordstorm.bucket/1/8/...). The permissions of this file need to be set to public read-write for the user to update this file. By using S3, the bandwidth cost of the overall

S3 will autoscale the size of the storage as needed.

5.10 Testing

5.11 Legality

5.11.0.1 Cookies Useage

A Rails session id is used to maintain a user login. An additional cookie is used for the gallery page to maintain viewing position. In June 2012, Article 5.3 of Directive 2002/58/EC **3** was drafted by the EU Directive on Privacy and Electronic Communications to safeguard user data. In compliance with these new regulations, a warning detailing the use of these cookies is given at the forefront of the website. The implicit consent of the user, and their agreement to the use of cookies, as outlined by the Information Commissioners Office**4**, complies with the legal requirements of this directive. The session id performs in accordance to this law and expires 1 hour after user login.

5.11.0.2 Data Protection Act

The Data Protection Act of 1998 [12] regulates personal data if any indentifing information is contained within. Data, by the Act's definition [13], is any information which is processed. By reading and tokenizing the uploaded files, the storm website processes the information and thus any uploaded file is considered data.

Personal data, by the Act's definition, is any information that can identify a user. The website cannot know if the user information is personally identifiable - the information may or may not contain sensative information and could identify the user (names, locations, or other information). Data from a Twitter stream, or the uploaded textual documents, in conjecture with the email address used to sign up, could identify an individual under certain circumstances. Due to the user requiring an account to upload storms, the information is linked with their username, thus holding them accountable.

Due to the situation where an individual can be identified by this data, the Data Protection Act should be followed. In this case, we can assume consent to process the files has been given by the user's upload. In accordance with the Data Protection Principles, the data must be adaquately safeguarded, and the information contained within cannot be extended to outside the European Economic Area (unless the area in question is secure)[14].

5.12 Security

The Open Web Application Security Project (OWASP) has published a list of their top ten security faults on the internet in March 2013. Although only an interim report, the contents provide insight into popular attack vectors and methods to safeguard a web application. From this, I have addressed a number of security attack vectors and discuss how they have been protected in the word storm website.

- 1) Injection - An attacker exploits the code intrepreter to insert his code into a request. Rails, in certain regards, provides security for this. However, for any value that the user can potentially modify, I use the Sanitize gem to help secure the input by removing unsafe elements (HTML code, SQL code, etc).

- 2) Broken Authentication and Session Management - An attacker exploits poorly managed sessions or authentication. The session therefore is expires after one hour, and the password information is stored in a secure manner (as a salted hash). Thin is used as a web server instead of webrick, because WEBrick is primarily for development rather than production level applications.

- 3) Cross-Site Scripting - The attacker exploits the browser's intrepreter to execute code. The Sanitize gem is used on textual input, and the parameters passed via post are checked for validity.

- 4) Insecure Direct Object References - The attacker gains access to unsecure data by posing as another user. To prevent this, the user is required to log into the word storm website. All of the data provided (personal storm access, uploads, settings) is only accessible via this account.

- 8) Cross-Site Request Forgery - The attacker forges an HTTP request and . A crsf_meta_tag inserts a crsf_param and a crsf_token on each page, acting as a identifying signature for the page. Whenever a form is submitted, these values are also submitted. Each time a form is received, the values are checked to confirm where the request came from. If the request fails, the form data is discarded.

- 9) Using Components with Known Vulnerabilities - A component used in the website has been exploited. In January 2013, Rails discovered several security flaws**Cite**. I have upgraded to the latest version of Rails to prevent these attack vectors. The Rails application needs to be constantly upgraded while in use.

- 10) Unvalidated Redirects and Forwards - The attacker exploits the automatic redirect to another site. I check each parameter during a redirect, and ensure the user is still logged into the application before accepting a redirect.

Number 5 (Security Misconfiguration), Number 6 (Sensitive Data) and Number 7 (Missing Function Level Access Control), while important, were not directly addressed directly in my application and are not mentioned. While certain security flaws will still exist within the application, a basic level of security has been implemented.

A potential drawback is the lack of screening that occurs with the file uploads.

With S3, the files of the storm need to be set to public_read_write. This

Chapter 6

Evaluation

Did not warn user about useage, nor provide asynchronous loading to indicate how far the user is along

6.0.1 Screenshots

6.0.2 Comparison Against Requirements Analysis

Website
Here is stuff
Web 2.0
There's more stuff here
Sharing is caring

6.0.2.1 Scalability

The scalability metric was not be achieved. While the website could be deployed to EC2, the server configurations could not be customized to allow the word storm code to run without a number of hacks that limited its effectiveness. If the project wants to use EC2 (or another Ubuntu server), either the word storm code needs to be changed or another workaround needs to be found.

The scalability In addition, the scalability was not tested to a large scale - each file upload and call to S3 costs a small amount of money. Proper scalability testing requires funding not available in this project.

Additionally, the switch of the database from SQLite to PostgreSQL, while it does not immediately guarentee

6.0.3 Performance Metrics

6.0.3.1 Results on Different Websites

The word storms project has been tested on Firefox, and Chrome, without any difference in functionality. Any functionality either not allowed (accessing

another user's storms) or does not exist (accessing a non-existant storm) results in either a 404 or a redirect with a flash message indicating the error response.

6.0.3.2 Hosting Enviroments

During the implementation of the website, the hosting enviroment changed with the needs of the project. For the majority of the project, the enviroment was Windows 7. Due to technical complications, the enviroment shifted to Windows Vista. Later, when the project was deployed to EC2, limitations of the Rubber Gem changed the system to a virtual-machine-based Ubuntu.

The differences between these enviroments ensures the implementation is independent of the operating system specifics. During application migration, several bugs regarding specific operating system implementations were discovered. This completeness verifies the application's functionality and improves its longterm extensability - users can both develop/improve the application on multiple platforms.

6.0.4 Test Suite

The test suite

The

6.1 Future Work

There are a number of possibilities for future work with the word storms website:

6.1.1 New Algorithms

Joaquim Castella and Charles Sutton's new algorithm for generating word storms reflects the potential for new algorithms to be developed. These algorithms could be introduced as alternative options for generating the storm. Another useful feature would include the ability to introduce a new document into the corpora, and have it update without being forced to start the algorithm from scratch would be useful for any long term prospect of the word storm.

6.1.2 User Testing

Long-term user heuristics could illuminate both the manner in which word storms are used, and, with regard to usability, how the website performs. From these results, certain features could be added, redesigned or removed from the website.

6.1.3 Different Display Mechanisms:

Currently, the word storms are displayed on the screen without any relation between the clouds' positions. While the user should still be able to customize

their position, it might be revealing to have the initial location reflect metrics between the clouds themselves (i.e. similarity measure between two clouds represented by distance between the words). By having this as an option for positioning/default, and have different metrics for comparing clouds, the position can add another dimensionality to the word storms.

Users may want to be able to combine different clouds together, or create subsets of storms. These would need independent display mechanisms, and different methods for achieving this.

6.1.4 Improve/Extend Information Gathering

As mentioned in section 5.8, the Twitter scraper suffers from the inability to be customized relative to the number of Tweets a user makes. Additionally, the web has a large amount of potential information sources that can be visualized within a word storm (i.e. different sections on Wikipedia, TED talks, university lectures), but the website lacks the necessary capability to parse the data. Finally, word storms can only process .txt files, and so cannot deal with the majority of the information on the web. Changing the input process, by including both the ability to deal with .pdf or .doc files and adding scraper to popular websites increases the number of potential usages.

6.1.5 Refactoring the Codebase

Two issues encountered during the implementation of the project - serialization and EC2 errors - were the result of limitiations in the Processing Library. While there are methods around these limitations, these hacks hamper the project's extensibility. The project is highly-coupled - WordCram relies on the Processing Library, Castella's word storm code relies on WordCram and my code now relies on the word storm implementation. For the project to continue to evolve, and adapt (run on a headless server), the codebase should be refactored to remove the coupling and provide alternative means of generating the storm.

Chapter 7

Conclusion

Bibliography

http://www.oreilly.de/artikel/web20.html

http://www.flickr.com/photos/tags/

http://stackoverflow.com/questions/342687/algorithm-to-implement-a-word-cloud-like-wordle

http://www.amazon.co.uk/Tausend-Plateaus-Kapitalismus-Schizophrenie-Deleuze/dp/3883960942

http://www.joelamantia.com/tag-clouds/text-clouds-a-new-form-of-tag-cloud

http://eprints.gla.ac.uk/56039/1/56039.pdf

http://www.careerride.com/MVC-disadvantages.aspx

http://www.buildingwebapps.com/articles/6419-can-rails-scale-absolutely

http://blogs.discovermagazine.com/cosmicvariance/2009/01/20/a-recommendation-letter-word-cloud/

http://groups.inf.ed.ac.uk/cup/wordstorm/wordstorm.html

https://github.com/blueimp/jQuery-File-Upload

[0] http://groups.inf.ed.ac.uk/cup/wordstorm/wordstorm.html

[1]http://guides.rubyonrails.org/getting_started.html

[2]http://www.phpexpertsforum.com/what-is-are-the-benefits-and-drawbacks-of-mvc-t862.html

[3]http://www.cs.utexas.edu/users/nn/web-pubs/sirosh/pvc.html

[4] http://delivery.acm.org/10.1145/1750000/1743567/p59-heer.pdf?ip=129.215.58.109&acc=OPEN&CFII

[5]http://spyrestudios.com/the-anatomy-of-an-infographic-5-steps-to-create-a-powerful-visual/

[6] http://homepages.inf.ed.ac.uk/tkomura/cav/presentation11_2013.pdf

[7] http://oreilly.com/web2/archive/what-is-web-20.html?page=3

[8] http://www.sqlite.org/whentouse.html

[9] arivx quim/sutton

[10] http://www.processing.org/discourse/beta/num_1272308820.html

[11] http://research.microsoft.com/en-us/um/people/weiweicu/images/cloud.pdf

[12] http://ico.org.uk/for_organisations/data_protection

[13] http://www.ico.org.uk/for_organisations/data_protection/the_guide/key_definitions

[14] http://www.ico.org.uk/for_organisations/data_protection/the_guide/the_principles

[15] http://www.lonij.net/wordstorm/wordstorm.php

[16] http://stackoverflow.com/questions/3748/storing-images-in-db-yea-or-nay

[17] http://arxiv.org/pdf/cs/0701168.pdf

*3 - http://blog.getresponse.com/social-sharing-boosts-email-ctr-up-to-115.html

1** http://thegeekhead.blogspot.co.uk/2009/06/hci-and-rai-applications.html

2** - http://guides.rubyonrails.org/security.html

3** - http://ec.europa.eu/justice/data-protection/article-29/documentation/opinion-recommendation/files/2012/wp194_en.pdf

4** - http://www.ico.org.uk/for_organisations/privacy_and_electronic_communications/the_guide/cook

http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013%20-%20RC1.pdf

Part I

TODO

1. Visualization - Clean up quotes, ensure that information provided doesn't get redundant elsewhere
2. Database writeup
3. HCI writeup
4. Introduction to each section
5. Define Web 2.0
6. Relate web 2.0 to stuff
7. Web 2.0 evaluation
8. Twitter Screenshots
9. Abstract
10. Conclusion
11. Evaluation ?
12. Wordstorm background - Alternative websites
13. Improve Introduction/Conclusion
14. Compare Against Quim - Evaluation
15. Mention each page at least once
16. File size => small
17. Clean/Edit - Past Tense, Fix stars, Spell Check
18. Change to other latex
19. Diagram insertion - ER, MVC, Web path
20. Fix Citations
21. Image size statistics - decreased
22. Define scalability
23. Charles Edits

Code

1. Test on different websites
2. Verify that twitter code removes when done
3. Remove uploads on completion

4. Clear database - folders/data
5. Test Suite
6. Web Metrics
7. Screenshots - gallery/main page/uploads/
8. Scale - what does it do?
9. Remove all gems
10. Rehost Java Code
11. Tutorial
12. Selenium