

CALJAX: An In-Browser Digital Repository System

Marc Bowes

Supervised by Hussein Suleman

	Category	Min	Max	Chosen
1	Software Engineering/System Analysis	0	15	0
2	Theoretical Analysis	0	25	0
3	Experiment Design and Execution	0	20	10
4	System Development and Implementation	0	15	15
5	Results, Findings and Conclusion	10	20	15
6	Aim Formulation and Background Work	10		10
7	Quality of Report Writing and Presentation	10		10
8	Adherence to Project Proposal and Quality of Deliverables	10		10
9	Overall General Project Evaluation	0	10	10
Total marks		80		80



Department of Computer Science
University of Cape Town
2009

Honours Project Report

CALJAX: An In-Browser Digital Repository System

Marc Bowes

Supervised by Hussein Suleman



Department of Computer Science
University of Cape Town
2009

Abstract

Searching is a key function to extract previously stored information from Digital Repository systems. A lot of research has been done on information retrieval in the past, with the grand vision only now coming to fruition. While these fruits now form the basis of billion dollar companies, there has always been an emphasis on on-line retrieval. This report will investigate a completely off-line Digital Repository system built on AJAX technology – specifically how to search information given the limitations of AJAX. Emphasis will be placed on the integration of existing technology and how it can be used to implement information retrieval using AJAX.

This system, named CALJAX is implemented in two parts – a preprocessor and a Web site. The preprocessor, implemented in Java, generates indices which are then used by a JavaScript search engine to generate results in response to a query. This combination allows for a collection to be exported for off-line use (specifically searching) on a removable medium such as DVD-ROM.

Experimentation shows that the system delivers good results with acceptable performance and is compatible with modern Web browsers. While the implementation has met the original specifications, there is scope for both improved and new features.

Acknowledgements

I would like to thank Hussein Suleman for being a great supervisor. His involvement and guidance was very much appreciated. To the Masters students who played audience to our proposals, prototypes and presentations – thank you for your time.

Table of Contents

Abstract	i
Acknowledgements	ii
List of Figures	vii
Introduction	1
Project description	1
Problem Statement.....	1
Research question	1
Motivation	1
Searching	1
Background.....	2
Introduction	2
AJAX	2
Background.....	2
Approaches	3
Information Retrieval	4
Background.....	4
Full-text searching	4
Preprocessing	5
Using the index	5
Search in unusual systems	6
Similar systems	6
Greenstone	6
EPrints and DSpace.....	7
Synthesis	7
Design and Implementation.....	8
Global decisions	8
Layout	8
Data	9
Site	10

Preprocessor	11
Java.....	11
Algorithm	12
Web Site.....	16
HTML	16
JavaScript	17
Design process.....	20
Prototype	20
Version 2	20
Final	20
Synthesis	22
Evaluation.....	23
Performance and compatibility	23
Aim.....	23
Methodology.....	23
Results	24
Analysis	26
Conclusion.....	26
User evaluation.....	26
Aim.....	26
Methodology.....	27
Results	27
Analysis	27
Conclusion.....	28
Information retrieval.....	28
Aim.....	28
Methodology.....	28
Results	29
Analysis	29
Conclusion.....	29

Future work.....	31
Improvements to current features.....	31
Performance.....	31
Online integration	31
New features.....	32
Advanced search.....	32
Suggestions.....	32
Conclusion	33
Bibliography	34
Appendix A: Results of “Performance and compatibility” evaluation	36
Computer 1: Ubuntu 9.10 with Firefox 3.5.....	36
Average result for step 1	36
Average result for step 2	36
Computer 1: Windows 7 with Firefox 3.5	36
Average result for step 1	36
Average result for step 2	36
Computer 1: Windows 7 with Internet Explorer 8.....	36
Average result for step 1	36
Average result for step 2	37
Computer 1: Windows 7 with Chrome 3	37
Average result for step 1	37
Average result for step 2	37
Computer 2: Ubuntu 9.10 with Firefox 3.5.....	37
Average result for step 1	37
Average result for step 2	37
Computer 2: Windows XP with Firefox 3.5	38
Average result for step 1	38
Average result for step 2	38
Computer 3: OSX with Firefox 3.5	38
Average result for step 1	38

Average result for step 2	38
Computer 3: OSX with Safari 4	38
Average result for step 1	38
Average result for step 2	39
Appendix B: User evaluation handout	40
Overview	40
Searching	40
Tasks	40
Feedback	41
Comments	41

List of Figures

Figure 1: A Java Applet allows for writing to disk.	4
Figure 2: Data is inverted so that words map to document pointers. Pointers are looped up in the document index which is used to generate human readable results.	6
Figure 3: CALJAX is layed out so that the site/ folder can be exported. The Java code in preprocessor/ need only generate files in the indices/ and lists/ folders in site/.	9
Figure 4: Each object must have metadata which describes it.	10
Figure 5: These five transformations turn a query string into an array of search terms.	13
Figure 6: Searching without CSS or JS produces a very bland page. CSS provides the look and JS provides the brains.	17
Figure 7: Search logic. JS applies behaviour to the page and calculates results when the user clicks search. These results are calculated off the inverted indices.	19
Figure 8: Paginating results.	20
Figure 9: Searching the Bleek and Lloyd collection. The table of results lists the page images which match the search query. The collection name and story summary are given as detail on the right.	21
Figure 10: Results for Computer 1.	24
Figure 11: Results for Computer 2.	25
Figure 12: Results for Computer 3.	25

Introduction

Project description

CALJAX is a system to browse, search and manage a digital collection. This system is lightweight, extensible and distributable, with a minimal software footprint without sacrificing functionality. The system features basic browsing, searching and updating. The update functionality includes the ability to update a central repository.

The key feature of CALAX is that it can be run in an off-line environment using only a modern Web browser. Numerous technical difficulties, due to limitations (sometimes intentional) in the capabilities of Web browsers, make such a system challenging to implement. This report discusses these difficulties and possible solutions thereof.

Problem Statement

Research question

Digital Repository systems are typically heavy-weight, requiring a Web server in both online and offline forms. This project attempts to overcome this through the use of Asynchronous JavaScript and XML (AJAX) in a Web browser. The research question asks whether this approach is feasible and able to provide at least the basic functionality expected of a Digital Repository system.

Motivation

An offline Digital Repository system is desirable for distributing content in environments which have limited to no Internet connectivity. Greenstone, a popular distributable Digital Repository system, distributes 50 000 copies of its software each year (Witten I. B., 2001). However, it is not always possible to install software in the target environment to run a system such as Greenstone. This project aims to plug this gap by minimising software requirements, using only the typical suite of installed software. The Bleek and Lloyd collection provides a way of distributing a digital collection, allowing users to browse and search the collection using only their Web browser (Suleman, 2007(a)). However, this system is specific for their content and cannot be generalised to an arbitrary collection of data. This project aims at a more general solution - that is, it should be applicable to a wide variety of content types and collections.

Searching

This report deals with searching the resulting collection. Searching the Digital Repository requires that preprocessing be done in order to quickly perform queries. The preprocessor needs to build inverted files: mapping words to the documents in which they appear. These indices are then used by AJAX code in the Web browser to perform a search.

Background

Introduction

Digital Libraries describe the conceptual space whereby a computer system makes it possible to store and retrieve a collection of digital objects (Schatz, 1997) (Lagoze, 2005) (Digital Libraries, 1995) (Witten I. M., 2000).

While a lot of research has gone into digital libraries, most digital library solutions follow a model where information is retrieved across a network (Schatz, 1997) (Lagoze, 2005) (Unknown, 1995) (Witten I. M., 2000). Historically, this has been necessary in order to overcome limitations in hardware, specifically storage (Schatz, 1997). Modern technology has not only overcome these limitations, but has done so in a way which is affordable.

These advances allow a digital library system to run on a single machine. This chapter investigates the applications of existing information retrieval techniques to an AJAX-based Digital Library solution.

The Digital Library system will be distributed on some sort of removable medium (for example, a DVD) and should “just work”, without needing to install any software on the target machine. A DVD reader and a Web browser with JavaScript support should be good enough. This approach is feasible, as demonstrated by the Bleek and Lloyd Collection prototype (Suleman, 2007(b)) (Suleman, 2007(a)).

AJAX

Background

AJAX, or Asynchronous JavaScript and XML, is a technology which allows one to add programming logic to an HTML page (Wusteman, 2006) (O'Reilly, 2005). AJAX describes the set of technologies and programming approaches to making the Web more dynamic. Traditionally, AJAX is used to increase responsiveness and usability on a Web page, by only updating portions of a page at a time (Wusteman, 2006). However, AJAX can be used to build an application without any need for external connectivity, that is, a desktop-style application in a Web browser.

AJAX allows for the dynamism and richness typically reserved for desktop applications (Wusteman, 2006) (Coombs, 2007) (Miller, 2005) (O'Reilly, 2005). This is due to the programmable layer provided by JavaScript. This layer allows for manipulation of the otherwise static HTML. JavaScript uses XMLHttpRequest (XHR), a de facto standard developed by Microsoft for transferring data (Wusteman, 2006) (Apple, 2005).

In regular HTTP communication, a client sends requests to a Web server. This server then returns HTML, which is rendered in the client's browser. This model is inefficient when the

returned page is largely similar to the previous page (Wusteman, 2006). AJAX uses asynchronous techniques to improve the user's experience – reloading only aspects of the page which change. While the asynchronous requests are almost always over a network, this is not a limitation.

The AJAX-based Digital Repository system will make use of AJAX techniques to provide desktop software reliant only on a browser with JavaScript support.

Approaches

JavaScript is limited in the connections it can make, including access to the computer it is being run on. These limitations are for security purposes (Anupam, 1998) (Garrett, 2005) (Mikkonen, 2007).

JavaScript limitations prohibit reading or writing to disk. This is a formidable obstacle, and not easily overcome. There are three primary approaches to solving this problem.

ActiveX

Internet Explorer provides ActiveXObject, which allows file access. However, while this is a viable workaround, it would limit the Digital Library system to Internet Explorer only. This is highly undesirable, as Internet Explorer runs on Microsoft Windows only, which requires licensing. This will impose limitations on who can use the Digital Library.

The upside of this route is that the code is very simple – sample code is widely available on the Internet (Tim Stall, 2005), including Microsoft's own documentation (Microsoft, 2007). If one had control over the target market, using ActiveX could be the best solution.

Mozilla extensions

On the opposite side of the fence, Mozilla provides extensions to their Web browser, Firefox. These extensions make it possible to compile additional functionality into the browser's JavaScript (C-Point, Unknown). While these extensions make it possible to read or write to files, this approach suffers from similar drawbacks to the ActiveX approach by limiting the code to Firefox. While it is possible to write in support for both ActiveX and Mozilla extensions, this is not a good solution. There are more browsers to cater for, such as Apple's Safari browser, or the popular Opera browser.

Applet

This approach involves embedding a Java Applet in the Web page. The JavaScript then connects to the Applet as if the Applet was on any other Web server. The Applet has access to the client's computer.

Provided the Applet is on the same disk as the data, the Applet (usually) requires little to no authorisation. In the case of the Digital Library system, this means that the Applet would have to reside on the same media as the HTML pages and data repository (Kourbatov, Unknown).

JavaScript is able to access the Applet, and its public methods, as if it were any other object in the DOM (see Figure 1). This approach means that the JavaScript code remains clean, while keeping cross-browser support. The restrictions on domain could prove problematic should the repository ever be split across media. In this case, code in the Applet would need to be electronically signed.



Figure 1: A Java Applet allows for writing to disk.

More information on this topic is available in the SUN developer guide (SUN Microsystems, Unknown).

Information Retrieval

Background

Information retrieval is not a new topic in Computer Science. Since Vannevar Bush's publication in 1945, significant research has been done in the field (Schatz, 1997). Research has typically focused on how to search ever increasing volumes of information, as well as how to improve the quality of the yielded results (Baeza-Yates, 1999).

Initially, information retrieval techniques were limited by the available hardware in the 1960s, where storage space limited searching to title, author, journal and keywords (Schatz, 1997). As hardware improved, systems were able to support full-text searches, where the full text of the article is indexed (Schatz, 1997). Beyond full-text searching lies semantic searching, where emphasis is placed on concepts conveyed by the query, rather than the occurrence of the exact words.

The existing research is sufficient for the needs of the AJAX-based Digital Library system. This chapter will investigate the application of existing techniques.

Full-text searching

Searching the full text of a document is an expensive operation. It is an exhaustive search, as the full text (title, author, keywords, abstract) in all documents must be scanned in order to report results.

To lessen the cost of full-text searching, a technique known as indexing is used (Schatz, 1997) (Witten I. M., 1999) (Suleman, 2007(b)) (Suleman, 2007(a)). A so-called “inverted index” is created, where words are mapped to articles (Schatz, 1997). This inverted index makes lookup operations cheap: one only needs to apply Boolean operators (such as AND, OR) to the list of returned article pointers.

Inverted indices can be extended with proximity operators, which allow queries to narrow the scope of the returned results by enforcing that certain terms appear near each other in the text. This is achieved by storing the offset of the word along with the article pointer (Schatz, 1997).

Preprocessing

In order for the above-mentioned technique to work, the inverted index needs to be generated. Index generation has been extensively covered by research into information retrieval (Schatz, 1997) (Witten I. M., 1999) (Suleman, 2007(b)) (Suleman, 2007(a)).

The preprocessor needs to traverse the digital collection: inverting each document so that words are mapped to their occurrences in articles (Schatz, 1997) (Witten I. M., 1999).

Certain words, such as 'and' or 'of', are considered too noisy to be indexed, and are omitted from the procedure (Schatz, 1997) (Witten I. M., 1999).

Words are shrunk to a canonical form, so that matches can be performed against the base word, rather than the specific version – for example, 'retriev' rather than 'retrieve' or 'retrieving' (Schatz, 1997).

A document index need also be created. This is a mapping of document identifiers to the actual resources (Witten I. M., 1999).

Using the index

The search page will read from the index, perform the necessary operations, and generate a list of results (see Figure 2). The generated results will provide summary information (such as title and author) as well as links to the actual digital objects. This page will need to use a workaround to read from the indices, as mentioned in section 2: AJAX. The search is entirely off-line: it is only displaying a subset of pregenerated content.



Figure 2: Data is inverted so that words map to document pointers. Pointers are looped up in the document index which is used to generate human readable results.

Search in unusual systems

In traditional information retrieval, a search query is built and then results are retrieved. The user is then able to browse the results or search again (Wusteman, 2006). With the advent of Web 2.0 has come the notion of dynamic searching, a hybrid of searching and browsing (Wusteman, 2006). The principal is that the system provides suggestions while the query is being built.

Facebook (<http://facebook.com>), a social networking site, allows one to search one's friends. While entering a name, a drop down list appears with partial matches to the query. For example, typing in “John” might yield a list containing “John Smith” or “Andrew Johnson”. This behaviour is achieved using AJAX to request results based on the current input. Facebook uses the user's history to rank the results, so that friends who are visited more often are ranked higher. This technique allows quick matching of queries to results, usually without leaving the page.

It is also possible to embed all search results in the search page using JavaScript (Mikkonen, 2007). This allows for rapid negotiation of results as no network traffic is required, making it suitable in cases where latency is an issue.

However, this technique is not suitable for handling lots of data due to memory restrictions. A natural overlap of the two approaches is to only embed frequently referenced friends in JavaScript variables, making the search appear very responsive.

This is useful in scenarios where latency may be high and memory cannot hold all the results. An example of this scenario is mobile devices such as cellphones. MXit (<http://mxit.com>), a mobile chat client, use an adaptation of this technique by only listing frequently referenced online contacts (the rest of the contact list is sent later) (MXit, 2009). This is useful in cases where a user may have a thousand contacts, which could dramatically slow down the login process.

Similar systems

Greenstone

Greenstone is a Digital Library system which makes it possible to export a digital collection for off-line viewing (Witten I. M., 2000). The resulting system is static, but needs a Web server in

order to run. Greenstone collections can be updated, but the rebuilding process can take a day or two for large collections (Witten I. M., 2000).

Greenstone supports full-text searching and metadata browsing. The system is extensible, meaning it can support a wide variety of metadata types. Additionally, plugins can be written to handle new types (by performing a conversion to Dublin Core) (Witten I. M., 2000).

EPrints and DSpace

EPrints is an open-source package for building open access repositories. It is written in Perl and runs under most operating systems, although the Windows version is still under development (Wikipedia, 2009).

DSpace is an open-source package for managing a wide variety of digital objects. It is written in Java and JSP, using Oracle or PostgreSQL for storage (Wikipedia, 2009).

Both systems provide similar functionality (browsing and searching digital collections) and there is much contention as to which system is better (Kim, 2005). However, neither system provides functionality to export the collection to static media: they are exclusively on-line solutions.

Synthesis

This chapter investigated the applications of existing technology towards a completely off-line Digital Library solution. Software requirements are minimal, and modern operating systems come standard with the required tools.

The combination of a Web browser and JavaScript provides an almost perfect environment to build a desktop application. However, security concerns limit what JavaScript has access to. As a workaround, it was suggested that an in-page Applet could be built. This Applet will act as a proxy: it will do the disk access on behalf of the page (AJAX) code. However, this is not necessary in newer browsers (such as Mozilla Firefox 3.5) which allow certain cross-site scripts to execute.

Information retrieval techniques are already well established and suit the needs of the proposed off-line AJAX Digital Library. A preprocessor is inevitable, as an index needs to be generated in order to search the digital collection. The preprocessor will generate an inverted index: mapping words to the documents in which they appear. Logical operators such as AND/OR are the applied to the query to narrow the scope of the lookup.

Design and Implementation

The design of CALJAX was split into three parts. This chapter will discuss the design and implementation of the searching feature of the repository. Some of this design overlaps with the other (browsing and management) sections, as consistency would be important when merging the three together.

Global decisions

Layout

Project data layout (directory structure) was an important decision to make. CALJAX's duty is to export digital collections into static HTML (HyperText Markup Language) for offline viewing. This means that there are two key aspects to the system: exporting and viewing. The preprocessor's duty is to do the exporting. HTML, CSS (Cascading Style Sheet) and JavaScript would present a Web site which would allow for searching the exported collection.

The root directory structure was therefore split into two folders named **preprocessor/** and **site/**. **preprocessor/** houses source code, while **site/** contains many subdirectories. Of these subdirectories, the most important one is **data/** which contains the raw data and metadata on which the preprocessor needs to operate. The decision was made to store **data/** in **site/** because the resulting Web site would need access to the original data. With **data/** in **site/**, the only step to distribute the Web Site would be to copy the **site/** folder to some other place (for example, burn it to a DVD-ROM). This layout is demonstrated in Figure 3.

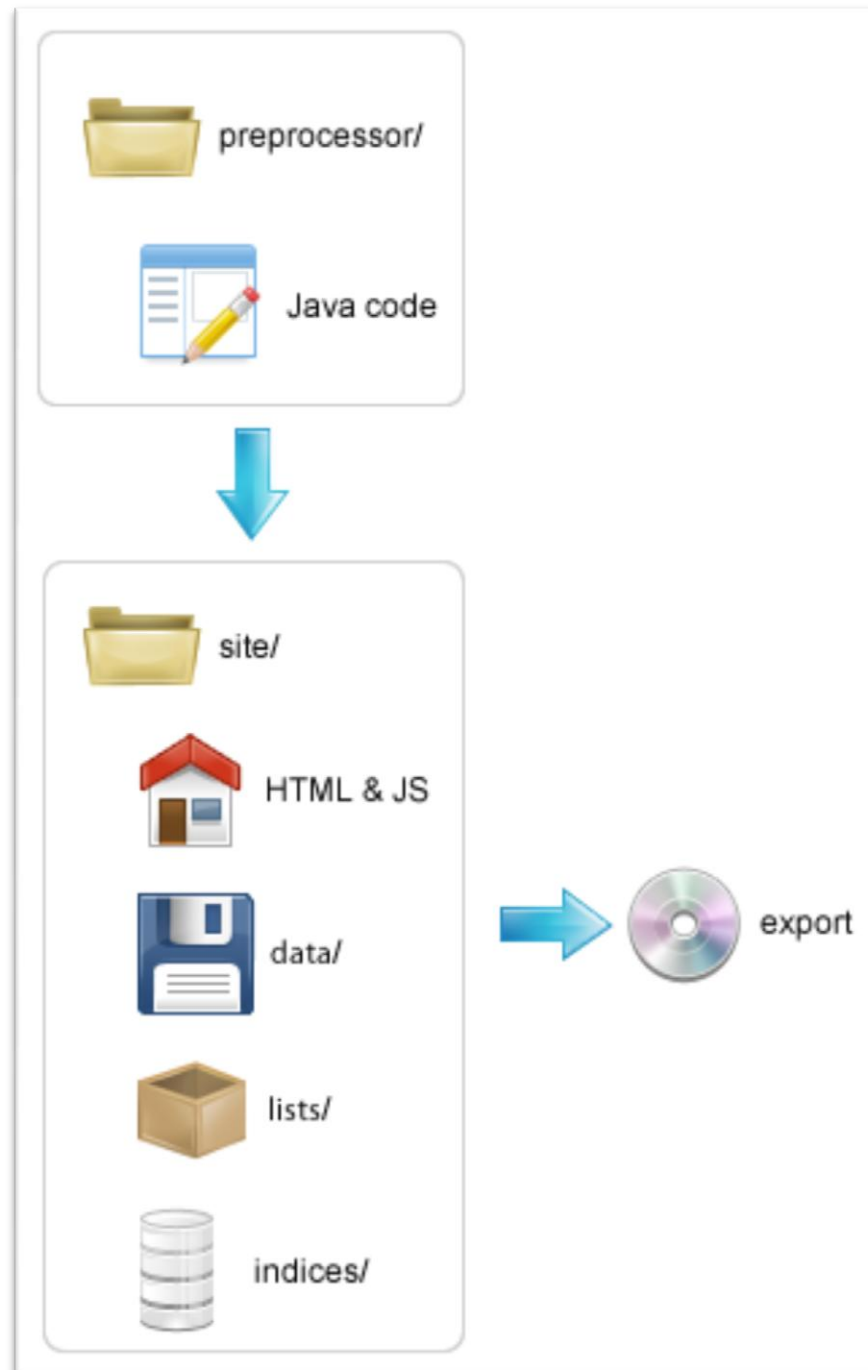


Figure 3: CALJAX is layed out so that the `site/` folder can be exported. The Java code in `preprocessor/` need only generate files in the `indices/` and `lists/` folders in `site/`.

Data

It was clear early on in development that for CALJAX to be a general solution, the emphasis had to be on the metadata over the data. This is because data could be in any format (e.g. text or video) while metadata is in a standardised textual (usually XML) format. Metadata is usually

created by humans (e.g. adding keywords to an image) and is a formatted formal representation of the meaning, category and so forth of its underlying data.

CALJAX only processes data files which have metadata. For each filename discovered in the data folder, the system would require that it be paired with another. The decision was made to group data/metadata pairs by filename (see Figure 4). The convention is to append “.metadata” to the name of the data file and the resulting string is the name of the metadata file. CALJAX enforces this pairing. If a “foo.jpg” were to be discovered, it would be ignored unless a “foo.jpg.metadata” is also present (and vice versa).



Figure 4: Each object must have metadata which describes it.

Since the system design is to build a generic solution for any type of data, CALJAX needs to be able to deal with a potentially large number of objects. The **data/** folder could contain many thousands of objects and their metadata. This is a potential issue for some Operating Systems which do work well when folders have many files.

To combat this, the **data/** folder is allowed to have arbitrary depth and structure. For example, objects beginning with the letter “A” could be placed in **data/A/**. Alternatively, objects in a certain category could be grouped together. This allows for the Operating System to process each folder at a time in more manageable chunks. It also allows for additional features, which will be discussed in the report on browsing the Repository.

Site

The site is comprised of only two HTML pages – one for browsing, one for searching. As a consolidated project, there were certain decisions to make when building these pages. Most importantly, the look and feel had to be consistent between the two. A CSS file was used to apply this look and feel.

CSS allows one to use styles (such as `<h1>Heading level 1</h1>`) which are then transformed by the stylesheet according to certain rules. For example, if one wanted the heading to be blue, the stylesheet would dictate that the `h1` tag has a `text-color` of `#0000FF`. This is preferred over in-line HTML which would specify in the tag how to render the text (e.g. `blue text`).

The stylesheet was developed late into the project and only made an appearance in the final version. This allowed both members to focus on writing code and placing it in appropriate tags without worrying about how it looked. When combining the projects, a stylesheet was created and applied to the satisfaction of both without needing to alter code to change appearance.

The JavaScript files for browsing and searching did not need to bear resemblance to each other. Each member was responsible for compatibility and performance of their own script. However, both scripts made use of the jQuery library. jQuery is a “fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development” (jQuery, 2009). Besides speed and simplicity, jQuery is also cross-browser compliant, which would help solve compatibility issues.

Preprocessor

The end goal is to produce a list of objects which match a query. It is not feasible to traverse the metadata, matching each file against the search query. Therefore, the goal of the preprocessor is to generate some sort of index which JavaScript can search.

As mentioned in the background chapter, a common technique in searching is to create inverted files. For each word encountered, an inverted index is created for it. This inverted index stores which objects (technically, the metadata for the object) “contain” that word. Thus, the inverted index becomes a listing of where to find a certain term in the collection.

The preprocessor traverses the **data/** folder loading metadata objects and producing inverted indices. This has several challenges. Traversing a directory is not inherently cross-platform as each Operating System has unique selection techniques. For example, Windows separates folders by backslash, while Unix-like systems use forward-slashes.

Java

In general, CALJAX has to maintain cross-platform (and cross-browser) compatibility and performance. The decision was made to create the preprocessor in Java. This made sense for a number of reasons. The Web site heavily reliant on JavaScript and it was apparent that a Java Applet may be needed to integrate updates into the system. After the decision to use Java was made, Firefox 3.5 was released. This version allows for (limited, but secure) cross-site scripting. This would negate the need for an Applet. However, the decision to use Java for the preprocessor still made sense.

Java is widely supported, has an excellent collection of libraries and can deliver good performance. Java is innately cross-platform. Java code is partially compiled and partially interpreted. The compilation causes the “.java” files to become “.class” files. These files are then interpreted by the Java Virtual Machine (JVM), which needs to be installed on the target machine. The interpretation stage means that code is cross-platform, dealing with issues such as file listing. Java is widely available and in some cases comes preinstalled with the operating

system, so little to no setup would be required for a user of CALJAX's preprocessor. Java also provides great library support – most importantly, an XML library. Metadata would be XML-formatted, and thus reading XML is a critical part of the preprocessor.

Java thus meets the three core needs for the preprocessor: compatibility, an XML library and good enough performance. While it is true that these three could be covered by another language, none do so as easily. A C++ preprocessor could achieve all three, but would have to be compiled for all target architectures. A scripting solution would require the runtime to be installed on each target machine.

Algorithm

The preprocessor traverses the data, collecting valid metadata objects. Each metadata object needs to be accumulated so that inverted indices can be created. Thus, for each metadata object several operations need to be done.

Loading the metadata

There are multiple XML libraries for Java. Two of these come with Java by default so it made sense to pick one of them to avoid external dependencies.

The Document Object Model (DOM) parser uses a tree structure to load the XML (To The River, 2005). This makes it easy to manipulate the document and traverse back and forth. However, this structure can consume a lot of memory and is thus only suitable for small XML files.

The Simple API for XML (SAX) parser uses an event driven interface to parse the document, invoking callbacks as it progresses (To The River, 2005). This is not suitable for document modification, but is able to deal with large XML files.

The nature of metadata files dictates that they are not large. The Dublin Core format has only fifteen metadata elements. The simplicity provided by the DOM parser, combined with the expected small sizes of the metadata files made it an obvious choice.

Finding words

Once the metadata has been loaded into the DOM, the program needs to extract words and build inverted indices. The DOM is used to traverse elements, extracting the inner text. This text then has all punctuation replaced with whitespace (“an-example” becomes “an example”). The new text is then split by whitespace, resulting in a set of words.

Each word is then transformed to lowercase. While this transformation is simple, it is important to realize that any transformation can only work if it is mirrored when the search is performed. That is, if the preprocessor applies a transformation, then so too must the searching algorithm. In

this case, the JavaScript must be aware that all terms need to be lowercased. This is the second transformation (the first being punctuation stripping).

Words are then shrunk to their canonical form. This process is known as stemming and is language-specific. This project uses an implementation of Porter's Stemming Algorithm which is specific to English (Porter, 2006). The result of this step is that “retrieve” or “retrieving” would be shortened to “retriev”. This is the third transformation.

Certain “stop” words are then discarded to prevent noise in the inverted indices. In the final implementation, the system discarded all words shorter than two characters. However, this should be customisable. Further discussion on is held in the chapter on evaluation. Regardless of the settings, this is (the fourth) a transformation and must be reapplied in exactly the same manner in the Web site.

The final transformation is to split by whitespace. Figure 5 demonstrates the complete chain.

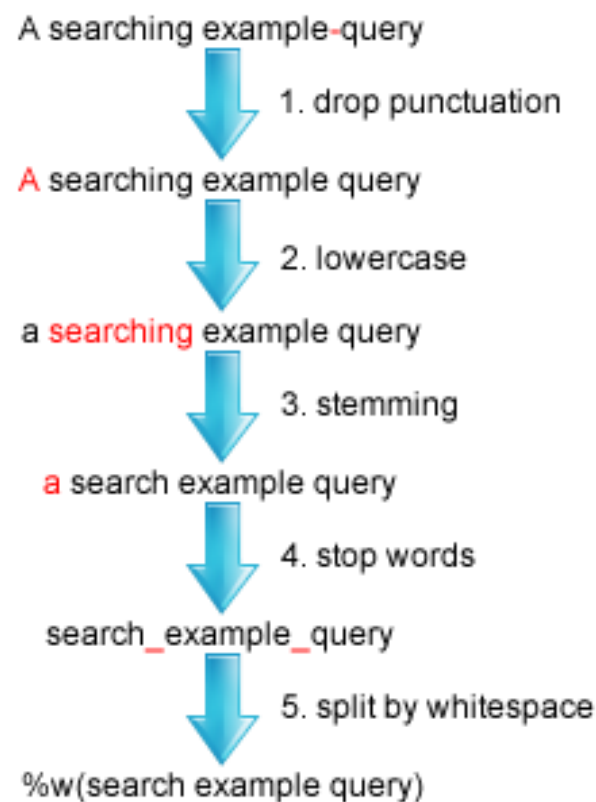


Figure 5: These five transformations turn a query string into an array of search terms.

Building the inverted indices

As each metadata file is processed, each word contributes to its specific index. For example, if the word “foo” is found in the metadata for object “bar”, then the foo index needs be updated

with a reference to “bar”. Inverted indices are accumulated over the run time of the preprocessor – they are built up as each piece of metadata is processed.

In the interest of saving disk space, each object is assigned a number upon discovery. Instead of storing the object name in the index, the CALJAX preprocessor will store the identifier. An identifiers file is then built so that the JavaScript will be able to determine which object a certain number corresponds to.

A piece of metadata may have multiple references to the same word. The index therefore stores the number of occurrences of the word in each piece of metadata.

Each inverted index therefore stores two pieces of information per entry: the identifier of the document and the number of occurrences. This inverted index is then named the same as the name of the word and has “.idx” appended to it. These indices are stored in the “site/indices” folder.

Field scoping

In addition to building an inverted index for each word, the preprocessor must also build additional indices per metadata field. If a user wishes to search for objects with a specific title, then there must be a folder containing inverted indices for the title field only.

Thus, for each word found in the metadata, it must be indexed twice: once in the “global scope” and once in the “scope for that field”. If the word “foo” appears in a <title> tag, then two entries must be created: one for `indices/foo.idx` and one for `indices/title/foo.idx`.

The specific implementation for CALJAX only allows for single field depth. A user would therefore be able to search “title:foo” but not “published:year:2009” (assuming the XML was structured like <published><year>2009</year></published>). However, the user would be able to search “published:2009” or “year:2009”. This decision was made for simplicity, but further investigation would need to be conducted to validate it.

Implementation

There are several implementation issues to consider. There are three main steps in the indexing: parsing metadata, accumulating indices and saving indices. Parsing the metadata is achieved through the DOM parser and saving the indices is achieved through some sort of file writer. The critical step is the accumulation of indices.

These indices have several properties to consider. One cannot simply open a file and write to it as words are discovered. This would be too expensive as the file needs to be continuously modified and not just appended to. Thus, the indices need to be stored in memory. Since the

number of indices is proportional to the number of unique words in all metadata, there is potential for a large number of indices.

The most expensive operation in index accumulation is finding the index to modify. The only other step is the modification – and that is as simple as making a new entry for the object, or incrementing the occurrence value if it is already in the index. Thus, having fast lookup for the index is crucial.

As previously discussed, indices are scoped to either the global or metadata field scope. It is therefore important to be able to look up an index in a specific scope. Once that index is located, it becomes important to determine the current (if any) occurrence value for the specific object. The sequence of operations is:

1. Specify a scope
2. Specify a word
3. Specify a document identifier
4. Increment the occurrence value

A hash is a storage structure which allows for fast lookups. Data is inserted as a key/value pair. The hash of the key is what determines where the value is stored. To retrieve the value, specify a key and the hash can be recomputed – which in turn divulges where the value is stored. This gives (nearly) constant lookup time. It is not exactly constant, because there is a potential for collisions. However, this is rare and resolving them is cheap.

Java has built in support for the hash structure, made available through `java.util.HashMap`. This is a templated type which means one can specify exactly what to store in the `HashMap`. Here is the above sequence rewritten with data types:

1. Specify a scope (String)
2. Specify a word (String)
3. Specify a document identifier (Integer)
4. Increment the occurrence value (Integer)

The document identifier itself must store an occurrence value. The `HashMap` definition is therefore:

```
HashMap<String, HashMap<String, HashMap<Integer, Integer>>>
```

For example, using associative array notation, one could increment the occurrences of “foo” in the third object 3 in the global scope by using:

```
hash[""]["foo"][3]++
```

The empty string means “no specific metadata field” or the “global scope”

In the Java implementation, some extra work has to be done to ensure that the key/value pair exists. This is necessary for first time insertion of any of the sub-hashes – such as the first time a word is seen in a specific object.

Once this HashMap is populated, saving the indices is trivial. Looping over key sets provides an easy means of writing the index to disk one word and one scope at a time. The HashMap solution provides an excellent space-time tradeoff: memory usage is slightly worse, but index lookup time is nearly instant. Looking up the index information is the most critical step in the algorithm, so this tradeoff is a key success factor for speed.

It is unlikely that the hash will consume more memory than the system has available. The preprocessor is expected to be run by people with more knowledge than an end-user, and it is reasonable to expect that such a person has a modern machine. However, even if the hash uses more memory than is available, it is not a problem. The Operating System will use memory management techniques (specifically paging) to deal with this scenario. Paging will naturally select infrequently used indices to be paged out, so there is no need to deal with this programmatically.

Once all metadata has been processed, the document lengths are calculated and stored in the `identifiers` file. This file is then stored in the `lists/` folder inside `site/`.

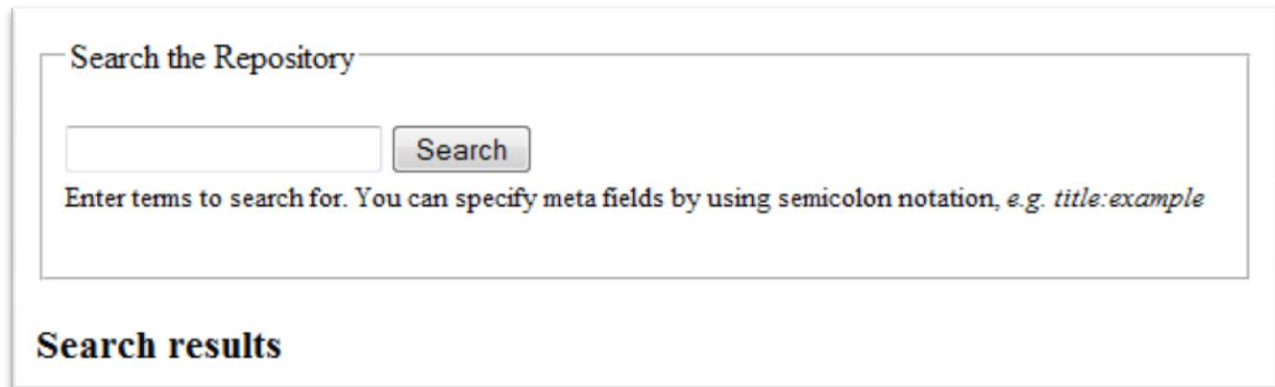
Web Site

Searching the Repository consists of a single HTML page and several stylesheets and JavaScripts. The HTML page was defined using the v4.0.1 Document Definition Type (DTD) and uses the modern best practices of unobtrusive JavaScript and styling (Willison, 2008).

HTML

In essence, the HTML page is simply a scaffold for a search engine, without the stylesheets and JavaScripts. The visual effects (color, size and rounded edges) are applied via CSS and the programmatic layer is provided through the JavaScripts. The HTML scaffold provides hooks for the JavaScripts to interact with the user. In short, the HTML page acts as both `stdin` and `stdout`, while the JavaScripts are where the meat resides (see Figure 6).

The HTML provides a title (formatted with `<h1>`), a search box (a `<input type="text">`) with a button and a `<div>`. The `<div>` has an identifier which will be used by the JavaScripts so that it can be located and used to contain the search results.



Search the Repository

Enter terms to search for. You can specify meta fields by using semicolon notation, e.g. title:example

Search results

Figure 6: Searching without CSS or JS produces a very bland page. CSS provides the look and JS provides the brains.

JavaScript

The JavaScripts for searching come in three files: jQuery v.1.3.2, an implementation of Porter’s Stemming algorithm and the actual search logic. Each file can be found in the `javascripts/` directory of the `site/` folder.

jQuery provides fast selection and traversal of the DOM as well as some other AJAX convenience functions. jQuery was not strictly necessary, but provides the cross-browser AJAX calls for accessing and altering the DOM as well as fetching and parsing data.

“stemmer.js” provides a JavaScript implementation of Porter’s Stemming Algorithm. It performs the exact transformations that the preprocessor version does. However, the JavaScript implementation applies rules via Regular Expressions. Implementation aside, the results are identical, which is of critical importance.

“search.js” is the script which provides the search functionality to an otherwise static page. It applies this behaviour unobtrusively using the selectors and manipulators provided by jQuery. This is achieved by hooking into the HTML and applying behaviour to the input box and button. Most importantly, the submit button is redirected to invoke `find(querystring)` instead of submitting the form. For completeness, the input box will invoke `click` on the button if `return` is pressed.

When the page loads, it first loads `identifiers` (which was generated by the preprocessor). This allows the scripts to lookup the real object information when generating results (the inverted indices only contain a pointer).

The `find` function is where the search logic begins. All transformations made in the preprocessor are reimplemented in JavaScript: punctuation is stripped, terms are stemmed and converted to lowercase. Stop words are not discarded – the index loader will simply cause the engine to skip the term because the index for the term will not exist.

For each term, an asynchronous read is issued and a callback function is supplied. This callback function will load the index into a series of hashes, similar to the preprocessor (by scope and word) and trigger another function which checks if all indices are loaded. Should this check pass, `find_real` is called, which actually performs the search. The hash acts as an index cache, so each index will only be loaded once during the page lifetime. This is desirable for situations where the user is only slightly modifying their query (adding or subtracting a term) but can be undesirable over time as memory becomes clogged. Figure 7 illustrates the sequence of events.

`find_real` computes rankings of objects by measuring the inner product of the occurrences against the search query. This value is normalized against the document length, which is stored in `identifiers`. As each rank is calculated, the result is inserted into a priority queue so that the output is correctly ordered. This priority queue is then used to generate tables of results - each `<table>` has a specific page number. These tables are then inserted into the `<div>`.



Figure 7: Search logic. JS applies behaviour to the page and calculates results when the user clicks search. These results are calculated off the inverted indices.

Pagination is achieved by hiding all the tables by setting `style="display: none;"` on all but the first page. Links are then generated so that the current `<table>` will be shown, while the rest will be hidden (see Figure 8). This can be achieved with a single line of jQuery and gives great performance (jQuery is designed for fast selection and iteration), even with thousands of pages of results.



Figure 8: Paginating results.

Design process

The implementation of the project happened in three phases two weeks apart.

Prototype

The idea of this phase was to ensure that the project was programmatically feasible as well as to test against misconceptions or design flaws. This proof of concept stage would implement basic features for all three aspects of the system. For searching, indices were built and searched on but no transformations were applied. The three aspects of the project were kept completely separate and no attempt was made to consolidate anything.

The prototype project was demonstrated to several Masters Students as well as the second reader for this project. Much valuable feedback was gleaned from both the development and feedback. Minimal testing was done in this phase as the lack and instability of features did not warrant it.

Version 2

This phase was to add features to the prototype. The search prototype applied no transformations to the query string, and thus the main focus of version 2 was to implement these transformations. All of them were completed and some of the prototype work was refactored into a more manageable state.

This phase was also demonstrated to a similar group of students as was the prototype. Again, feedback was collected and used to flesh out a design for the final version of the code. No formal evaluation was done on this iteration. However, simple sanity checks were performed such as “can item X be found?” and “is the code compatible?”

The code was not compatible with other Web browsers. Additionally, results were not ranked properly meaning that queries returning a lot of results were useless. Pagination was not yet implemented, which further devalued the results. This information, together with feedback from the students was used to generate a design for the final version.

Final

The final version of the project implanted ranking and pagination and fixed compatibility issues. At this point, effort was made to consolidate the project as a whole. A stylesheet was developed and applied to both offline pages (browsing and searching) and an attempt was made at online

integration. Figure 9 demonstrates searching on a collection containing images from a collection of notebooks.

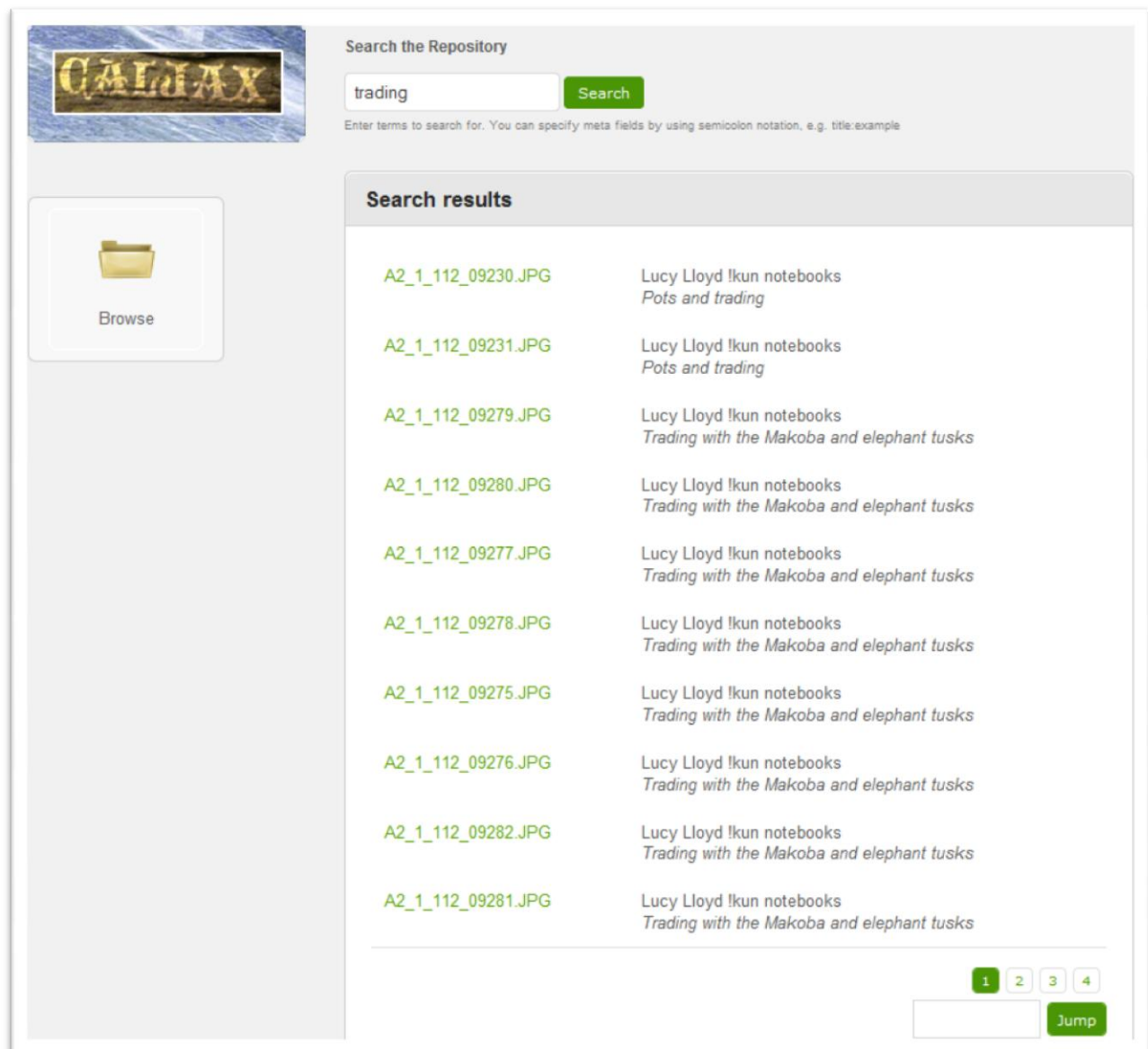


Figure 9: Searching the Bleek and Lloyd collection. The table of results lists the page images which match the search query. The collection name and story summary are given as detail on the right.

The online integration fetches a list of new metadata from a given server and performs a simple search over the retrieved data and appends the results to a separate results window. However, given limited time, this integration was not finished. More will be discussed in future work.

A formal evaluation was completed for this version of the project and will be discussed in the next chapter.

Synthesis

CALJAX was developed over three phases, with each phase building upon information learnt from the previous stage. The first stage was built using knowledge learnt during the literature synthesis, which has been presented in the Background chapter.

Searching CALJAX required two pieces of software to be developed: a preprocessor and a Web page. The preprocessor was written in Java and produced output into a folder which could then be burnt to DVD-ROM (or some other media) as a self-contained entity. The Web page was written in HTML, using AJAX to read the data, process and display the data generated by the preprocessor.

The final version of the project was consolidated so that a consistent look and feel was present in both the browsing and searching aspects. Incomplete features and evaluation of this final version will be discussed in upcoming chapters.

Evaluation

The research question underpinning CALJAX asks whether it is feasible to create such a system. Performance and compatibility play a huge role with respect to feasibility, as deficiencies in either category could justify failure for the project as a whole.

An information retrieval evaluation is needed to investigate the quality of the results. This ties in with feasibility so that the real question is whether the system is able to produce good results in good time. A user evaluation was carried out to determine this.

Performance and compatibility

Aim

This evaluation will investigate code compatibility and performance over several computers, operating systems and Web browsers.

Methodology

The system was seeded with data from the Bleek and Lloyd collection. The resulting `site/` folder was then written to a DVD. However, due to the size of the collection, not all data items were copied across. While all the metadata was present, roughly half of the real objects (images) were stubbed out.

This DVD was then used for compatibility and performance tests. The system was tested on the following three machines:

1. **Intel(R) Core(TM)2 Quad CPU @ 2.40GHz, 4GB memory**

This is a desktop running Ubuntu 9.10 and Windows 7. Firefox 3.5, Internet Explorer 8 and Chrome 3 were tested.

2. **Intel(R) Core(TM)2 Duo CPU @ 2.00GHz, 2GB memory**

This is a laptop running Ubuntu 9.10 and Windows XP. Firefox 3.5 and Chrome 3 were tested.

3. **MacBook Pro with Intel(R) Core(TM)2 Duo @ 2.66Ghz, 4GB memory**

This is a laptop running OSX. Firefox 3.5 and Safari 4 were tested.

For each of these systems, the following five searches were performed. Each search query is designed to test the system at handling a specific type of query. The size of the index can vary from a few bytes to a megabyte. Therefore, indices ranging from smallest to largest were used in the tests.

1. **acquir appli**

two very small indices

2. **lion**

random query of moderate size

3. **notebook.jpg xam**
three biggest indices
4. **notebook name:jpg**
biggest index and biggest index in biggest scope
5. **strike string stripe strive strong struck stupid substanc success suffer suitor summer
sung sun sunlight sunrise sunset suppli surpris surround**
long query – 20 terms

Each query was performed six times as follows:

1. Load the site and perform the query, record the time.
2. Without refreshing, click search again and record the time.
3. Eject the DVD and close the Web browser. Repeat steps 1 and 2 another two times.

Across the six queries, an average is taken over step 1 and over step 2. For example, if step 1 and step 2 consistently yield 100 and 10 milliseconds respectively, then the two averages are 100ms and 10ms. It is necessary to eject the disk before repeating step 1 so that the cache is cleared.

Results

For raw data, please see Appendix A.

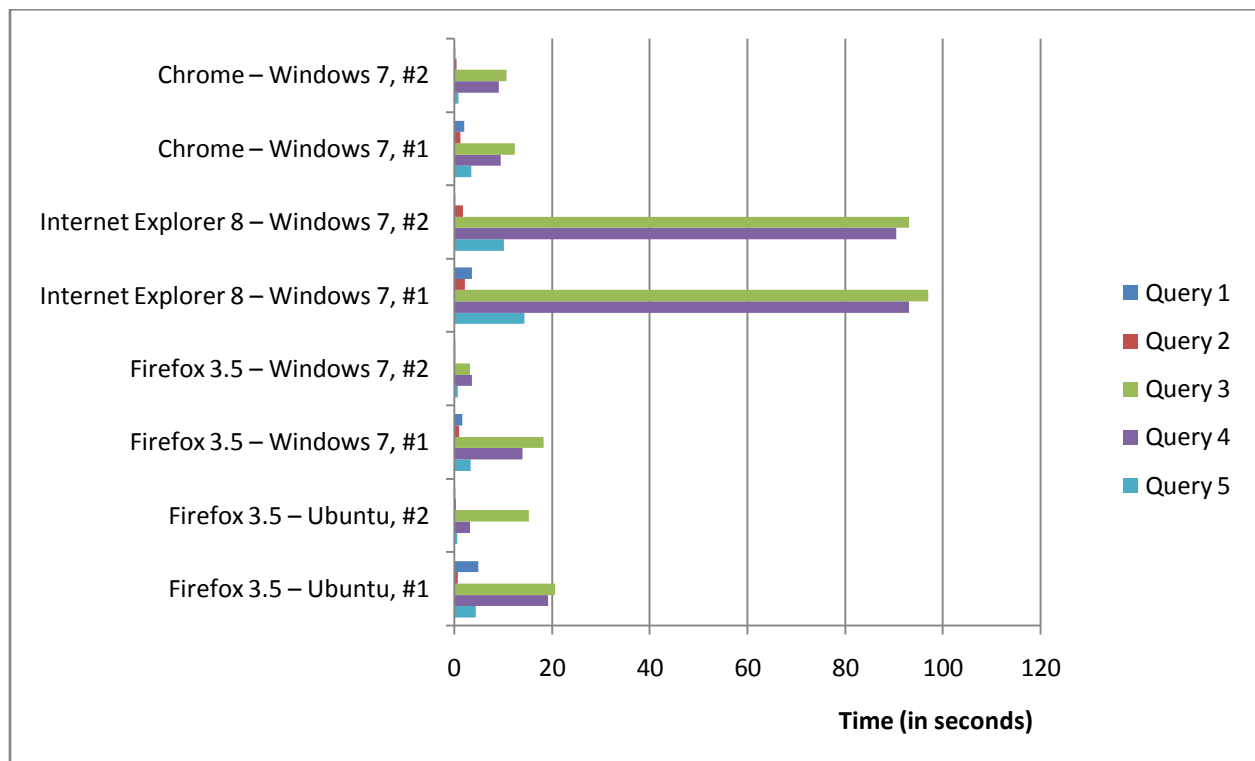


Figure 10: Results for Computer 1

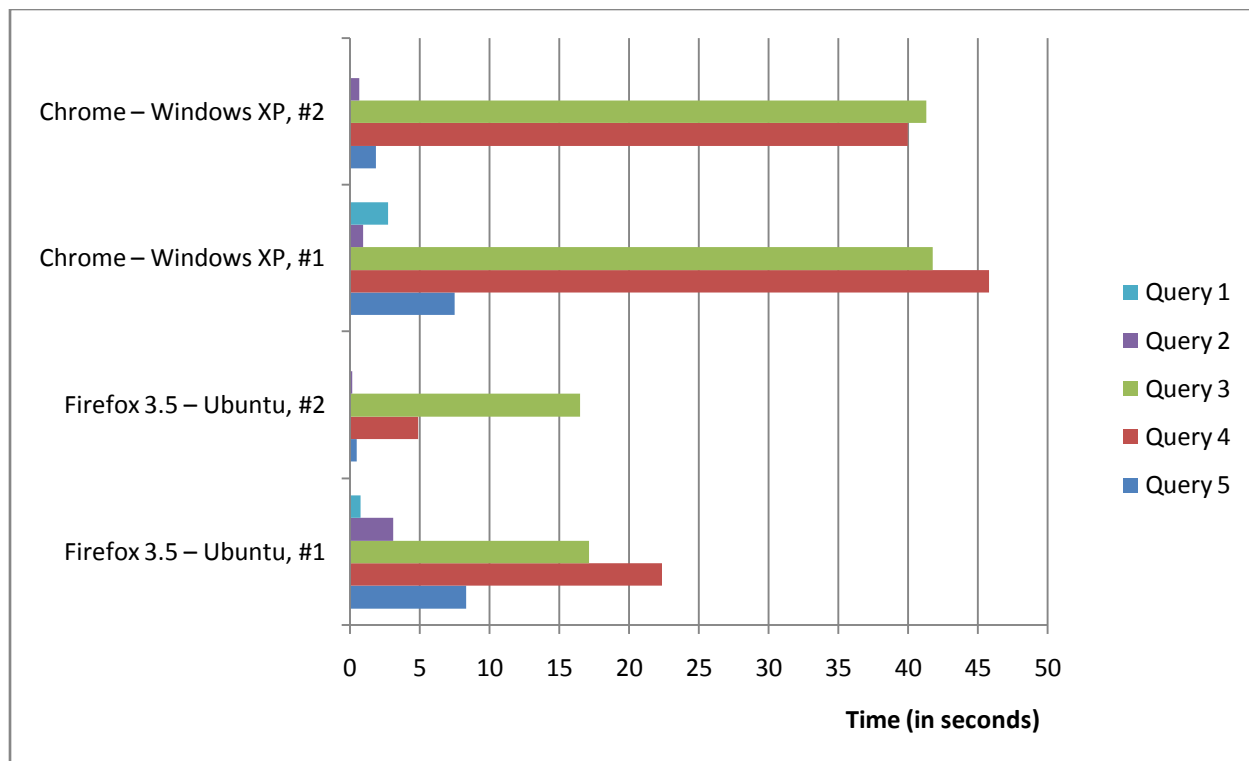


Figure 11: Results for Computer 2

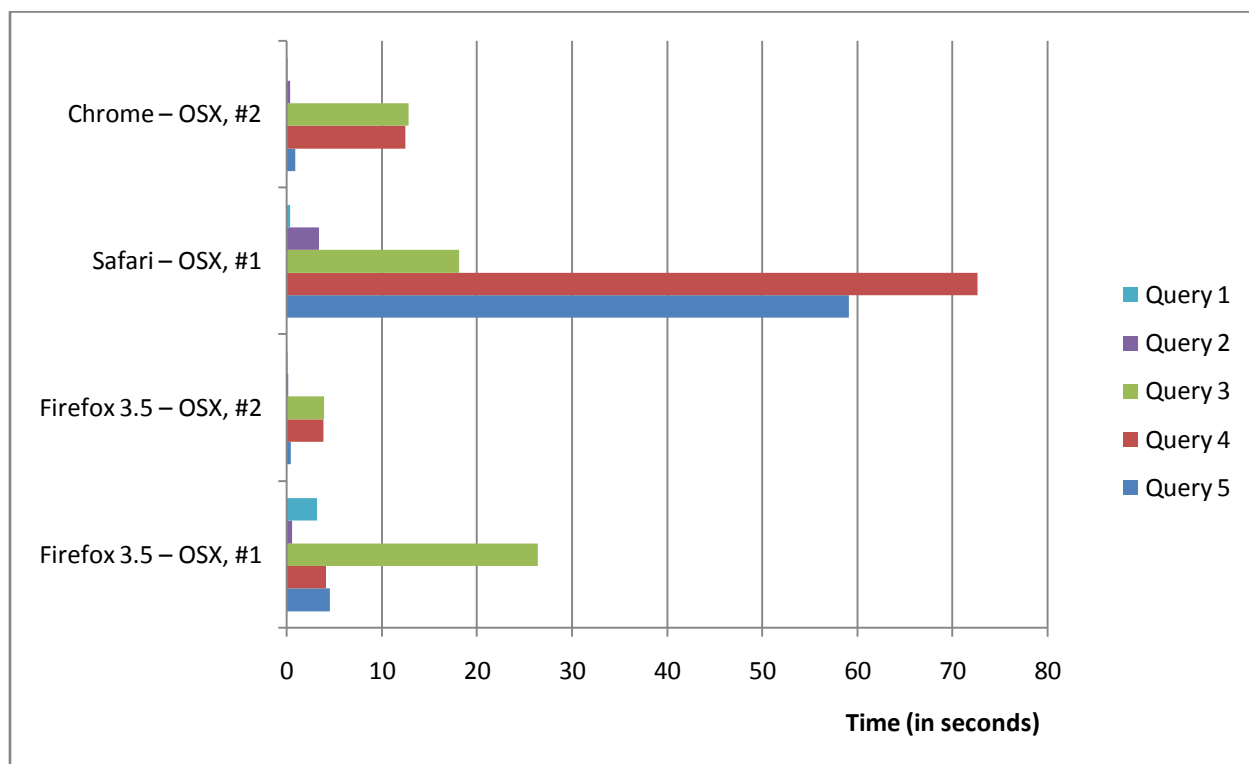


Figure 12: Results for Computer 3

Analysis

CALJAX was compliant with Web browsers on all three operating systems, with the exception of Internet Explorer. Some variable renaming was needed in order to avoid conflicts with the internal workings of Internet Explorer. That was the only compatibility glitch discovered.

Performance was heavily reliant upon browser and caching. Other than Internet Explorer, the other Browsers exhibited similar performance, although Firefox had the best overall performance. Internet Explorer had very bad performance, which only worsened as the size of the indices increased. There was nearly a minute and a half difference between Firefox and Internet Explorer. Additionally, the run-time of the script caused Internet Explorer to continuously throw up messages asking if the user wished to terminate the script.

On average, performance is hindered more by DVD-ROM latency than by JavaScript. In query 4, this latency added up to sixteen seconds additional time. Operating System and Computer specifications also play a big impact here as it impacts how caching and prefetching works. In all cases, the cached query was significantly faster because the disk cache bypasses the drive latency.

All browsers struggled with queries involving large indices over one megabyte.

Conclusion

Compatibility was not an issue on any system or Browser. Queries involving large indices produce slow results on the current implementation. However, these large indices were a function of the seeded data. For example, the “notebook” index contained virtually every page in the collection. This is clearly an edge case.

Internet Explorer is definitely not a desirable browser to be using. The stylesheet does not work properly as the browser provides no advanced features (such as rounded edges). The Mozilla and WebKit engines both provide these extensions, although not in a standard manner. The search times are consistently slower and the browser uses a lot more resources, which causes cached items to be dumped. However, the system is still compatible with Internet Explorer.

The code could be improved to deal with larger indices by breaking them up into more manageable chunks. However, DVD-ROM spin-up plays a very significant role in result delay and means to mitigate this should be looked at (such as preloading popular indices).

User evaluation

Aim

The aim of this experiment is to determine the usability of the system. A search engine must facilitate the retrieval of items that match a query. The query should be easy to build and the

results should be returned in a reasonable manner. In general, the system should be intuitive to use and avoid surprising the user.

Methodology

Eleven users were selected for this test. These users have moderate to extensive experience with search systems, but few had digital library experience. Each user was asked to complete tasks and a questionnaire. This handout (see Appendix A) explained the purpose of the system to the user and gave them certain tasks to complete. Once all were completed, the user was then asked to agree or disagree with certain observations. This feedback was used to determine the success of the system with regards to usability.

Results

Question	Strongly Agree	Agree	Neutral	Disagree	Strong Disagree
1	11				
2	10	1			
3	11				
4	6	5			
5	11				
6	8	3			
7	7	2	2		
8	11				
9	7	4			1
10	11				

Analysis

Each observation was phrased in such a way that agreements correspond to positive feedback and vice versa. From the results, it is clear that tasks 4, 6, 7 and 9 need investigating as the others received positive responses.

Task 4, 6 and 7 all refer to results and interpreting the results. While the feedback is mostly positive, the positivity wasn't as pronounced as the other questions (which deal with speed and intuitiveness). It is clear that the results aren't that easy to interpret, making it harder to find what you want in the results.

This is largely because the Bleek and Lloyd collection is primarily composed of stories, which have useful information such as a name, author and keywords. However, CALJAX currently only supports single-entities. As a result, the system was seeded with pages from the stories. Pages then had their story metadata attached to them. The result is that the name of an object is the name of the page, which is actually just the image file name of the page. These names are not useful to the users, which in turn hampers the usefulness of the results.

Additionally, a search will return a result for each page in the story that matches. If a story titled “About Cattle” has 3 pages, there will be 3 results – 1 for each page in the story. This means that results are bloated, as the story metadata is duplicated per page.

However, despite this, users still gave reasonably positive feedback for the result interpretation. This means that they were able to find what they were looking for among the results without too much effort.

The only negative feedback was for task 9. In this task, the user was asked if additional terms helped improve the quality of the results. The search algorithm uses OR-like logic to build results – so adding terms will never reduce the number of results. One particular user found that the increased number of results was undesirable, because a more specific query should yield more specific results.

On the other hand, the ranking algorithm pushes more relevant results to the top, making the initial results more specific to the new query. This is probably why the other users agreed that the additional term helped them – the results on page 1 were better, even if the total pages had increased.

Conclusion

CALJAX searching was designed to return good results in a reasonable time. The users felt that the searching was fast enough and were able to find the example stories mentioned in the tasks. They felt the system was intuitive and were not bewildered by any aspects. The OR logic can build very large result sets, but this can be argued to be a good thing. The collection size is limited by the size of the media (in this case, a DVD-ROM) and so perfect recall might be desired over condensed results. The result set was bloated by the duplication of story metadata, which further bloats the results.

Speed was never an issue when searching. Although the tasks did not involve terms with massive indices, none of the massive indices could be used in examples. This is because the big indices are a by-product of the data – such as “JPG” and “notebook” - and are not related to the actual content in the stories.

Information retrieval

Aim

This test is designed to measure the quality of the returned results, thereby investigating the usefulness of the information retrieval algorithm implementation.

Methodology

A set of query strings were built by examining content in the Repository. For each query string, the recall and precision was measured. Recall indicates if all possible relevant results were

returned, while precision measures how many results are actually relevant. These results were then recorded and analyzed.

Typically, recall and precision is determined up front by experts on the collection. However, a test this comprehensive was not feasible in the allotted time frame. Therefore, the original metadata (in XML format) file was manually analyzed to determine expected recall and precision. Recall is perfect if all the possibly relevant objects are returned. Precision is perfect if all results are relevant to the query.

The following query strings were used:

1. lion
2. author:willem
3. lion collection:|xam
4. rain weather clouds seasons

Results

Query	Recall	Precision
1	100% (2903)	100%
2	100% (16)	100%
3	100% (10683)	2% (246)
4	100% (2189)	0%

The numbers in parenthesis refer to the number of articles which make up the percentage. For query three, 246 results is roughly 2% of 10683 results.

Analysis

The system has 100% recall because the results are built using OR logic. For small or single-word queries, results are very good as the system gives 100% precision. However, the longer the query becomes, the worse the precision becomes. This is because each additional query term can only add results, instead of refining them.

On the contrary, the ranking works very well. If one searches “lion collection:|xam”, more results are returned than with either “lion” or “collection:|xam”, but the ranking ensures that the results are ordered in such a way that “lion AND collection:|xam” would yield the same top results.

Conclusion

The system has 100% recall, which means it is good at dealing with simple queries which scope results to certain words or metadata fields. However, adding additional terms widens the scope, instead of narrowing it.

The OR operator makes it impossible to refine results. However, given the limitations of the collection (such as DVD size), there is an upper bound on how many results there will be. As

such, high precision may not be as necessary given the limited number of objects in the collection.

While precision worsens with additional terms, the ranking algorithm does a good job at ensuring precision across the initial results.

Future work

Improvements to current features

Performance

Incremental results

The JavaScript could be altered to improve performance. Currently, the system is not intelligent about what it is loading. Files are loaded in their entirety, which is not strictly necessary as processing time is wasted calculating information for results which will rank poorly. Instead, the system could read off the top of each index and calculate results. Once those results are displayed, the system could continue to refine the results. In this manner, rough results are quickly generated (and will most likely be accurate for the first few pages), while more results can be generated over time.

This would give yield faster response times without decreasing quality (by the time the user is finished scanning the first page of results, the second page will be finalised and so forth). Google uses a similar technique – more complex results are only generated if the user browses beyond the first page of results.

To achieve this, indices will have to be split into multiple parts as JavaScript can only read a file in its entirety. This would also require modifications to the preprocessor and the list files so that JavaScript will have enough information to know which files to load.

Cache

The engine currently caches indices for previous searches. This was done to improve the expected use case of the system: a user enters a query, inspects the results and refines the query to improve the results. However, large indices can consume large amounts of memory causing performance degradation over time. This could be ameliorated by only partially caching indices. This is dependent on the previously mentioned improvement which allows for incremental result generation.

Online integration

Online integration was not properly implemented. Currently, the system requests new metadata from the online server and the collection version or server location is not customisable. This should be configurable in the interface and settings should be saved in a cookie. The engine does a simple search of the new metadata, but does not display the results to the user. This should be displayed in a separate section to the normal results so that the user is aware that the results are of items not in the local repository (and thus might not be viewable).

New features

Advanced search

CALJAX does not interpret queries – it only matches terms. Modern search engines support advanced queries where the user is able to impart more information to the engine by specifying how joins happen (AND/OR/NOT etc.).

Such a feature would require alterations to the JavaScript engine only. Certain terms would be interpreted as function calls rather than as a term to search for. This feature would allow for finer grained retrieval of results and would improve the search experience for advanced users.

Suggestions

The current system provides little assistance to first time users. This could be improved by implementing search suggestions. These suggestions could be generated by an altered preprocessor. Alternatively, search queries could be sent (via an HTTP POST) to the central repository. These queries would then be stored to generate common search queries which would then be suggested to the user.

Conclusion

CALJAX is a system designed to export digital collections for offline viewing. The system makes it possible to browse, search and update the resulting collection. This report has focused on searching the repository.

A preprocessor is used to generate inverted indices which are then used by an HTML page powered by AJAX technology. These indices are used to quickly generate results and display them to the user. The collection can be written to a DVD (or any distributable media) and requires no software to be installed in order to work.

The system could have a profound impact as a means to distribute information to areas without readily available Internet or software access as it requires only a modern Web browser in order to function.

The evaluation of the system has provided positive feedback. The system is successful in that most foreseen obstacles have been overcome. Some work is outstanding which would improve the quality and speed of the results.

Bibliography

Anupam, V. M. (1998). Security of Web Browser Scripting Languages: Vulnerabilities, Attacks, and Remedies. *7th USENIX Security Symposium*.

Apple. (2005, 06 24). *Dynamic HTML and XML: The XMLHttpRequest Object*. Retrieved May 2009, from Apple Developer Connection: <http://developer.apple.com/internet/webcontent/xmlhttpreq.html>

Baeza-Yates, R. R.-N. (1999). Modern Information Retrieval. *SMLI TR-2007-168*.

Coombs, K. (2007, January). Building a Library Web Site on the Pillars of Web 2.0. *Computers in Libraries*, 27(1).

C-Point. (Unknown). *How to read and write files in JavaScript*. Retrieved May 2009, from C Point: http://www.c-point.com/JavaScript/articles/file_access_with_JavaScript.htm

Garrett, J. (2005, February 18). *Ajax: A New Approach to Web Applications*. Retrieved May 2009, from Adaptive Path: <http://adaptivepath.com/publications/essays/archives/000385.php>

jQuery. (2009). *jQuery Javascript Library*. Retrieved 2009, from jQuery: http://docs.jquery.com/Main_Page

Kim, J. (2005). Finding Documents in a Digital Institutional Repository: DSpace and Eprints. *68th Annual Meeting of the American Society for Information Science and Technology*.

Kourbatov, A. (Unknown). *The Javascript FAQ*. Retrieved May 2009, from Linux Topia: http://www.linuxtopia.org/online_books/javascript_guides/javascript_faq/reading2.htm

Lagoze, C. K. (2005, November). What is a Digital Library Anymore, Anyway? *D-Lib Magazine*, 11(5).

Microsoft. (2007, March). *ActiveX Data Objects (ADO) Frequently Asked Questions*. Retrieved May 2009, from Microsoft KB: <http://support.microsoft.com/kb/183606>

Mikkonen, T. T. (2007). Using JavaScript as a Real Programming Language. *SMLI TR-2007-168*.

Miller, P. (2005, October). Web 2.0: Building the New Library. *Ariadne*, p. 45.

MXit. (2009, March). *MXit Open Protocol 5.8.2*. Retrieved May 2009, from MXit Devzone: <http://devzone.mxit.com/download/2>

O'Reilly. (2005, September 30). *What Is Web 2.0*. Retrieved May 2009, from O'Reilly Net: <http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>

- Porter, M. (2006). *The Porter Stemming Algorithm*. Retrieved from Tartarus: <http://tartarus.org/~martin/PorterStemmer/>
- Schatz, B. (1997, January). Information Retrieval in Digital Libraries. *Science* , pp. 237-334.
- Suleman, H. (2007(a)). Digital Libraries without Databases. *ECDL* .
- Suleman, H. (2007(b)). in-Browser Digital Library Services. *ECDL* .
- SUN Microsystems. (Unknown). *JavaScript to Java Communication (Scripting)*. Retrieved 2009, from J2SE Docs: http://java.sun.com/j2se/1.4.2/docs/guide/plugin/developer_guide/js_java.html
- Tim Stall. (2005, April 24). *Using JavaScript to read a client-side file*. Retrieved May 2009, from Dot Net Developer's Journal: http://timstall.dotnetdevelopersjournal.com/using_javascript_to_read_a_clientside_file.htm
- To The River. (2005). *Introduction to XML and XML With Java*. Retrieved May 2009, from To The River: <http://www.totheriver.com/learn/xml/xmltutorial.html>
- Unknown. (1995). Digital Libraries. *Communications of the ACM* 38(4) .
- Wikipedia. (2009). *DSpace*. Retrieved May 2009, from Wikipedia: <http://en.wikipedia.org/wiki/DSpace>
- Wikipedia. (2009). *EPrints*. Retrieved May 2009, from Wikipedia: <http://en.wikipedia.org/wiki/Eprints>
- Willison, S. (2008). *Unobtrusive JavaScript with jQuery*. Retrieved 2009, from Simon Willison: <http://simonwillison.net/static/2008/xtech/>
- Witten, I. B. (2001). Power to the People: End-user Building of Digital Library Collections. *1st ACM/IEEE-CS joint conference on Digital Libraries*, (pp. 94-103). Virginia, USA.
- Witten, I. M. (2000). *Greenstone: A Comprehensive Open-Source Digital Library Software System*. UNiversity of Waikato.
- Witten, I. M. (1999). *Managing Gigabytes*. San Francisco: Morgan Kaufmann Publishers, Inc.
- Wusteman, J. O. (2006, June). Using Ajax to Empower Dynamic Searching. *Information Technology and Libraries*, 25(2) , pp. 57-64.

Appendix A: Results of “Performance and compatibility” evaluation

Computer 1: Ubuntu 9.10 with Firefox 3.5

Average result for step 1

1. 4856
2. 709
3. 20526
4. 19099
5. 4284

Average result for step 2

1. 23
2. 228
3. 15240
4. 3191
5. 474

Computer 1: Windows 7 with Firefox 3.5

Average result for step 1

1. 1547
2. 905
3. 18230
4. 13960
5. 3247

Average result for step 2

1. 42
2. 163
3. 3110
4. 3586
5. 674

Computer 1: Windows 7 with Internet Explorer 8

Average result for step 1

1. 3542
2. 2034
3. 97029
4. 93148
5. 14233

Average result for step 2

1. 69
2. 1640
3. 93168
4. 90557
5. 10123

Computer 1: Windows 7 with Chrome 3

Average result for step 1

1. 1920
2. 1175
3. 12263
4. 9463
5. 3405

Average result for step 2

1. 16
2. 349
3. 10648
4. 9023
5. 778

Computer 2: Ubuntu 9.10 with Firefox 3.5

Average result for step 1

1. 801
2. 3135
3. 17167
4. 22376
5. 8336

Average result for step 2

1. 30
2. 164
3. 16500
4. 4915
5. 515

Computer 2: Windows XP with Firefox 3.5

Average result for step 1

1. 2744
2. 954
3. 41776
4. 45827
5. 7516

Average result for step 2

1. 41
2. 693
3. 41329
4. 40001
5. 1858

Computer 3: OSX with Firefox 3.5

Average result for step 1

1. 3147
2. 519
3. 26402
4. 4138
5. 4520

Average result for step 2

1. 30
2. 132
3. 3908
4. 3808
5. 411

Computer 3: OSX with Safari 4

Average result for step 1

1. 310
2. 3392
3. 18119
4. 72656
5. 59158

Average result for step 2

1. 17
2. 361
3. 12771
4. 12435
5. 874

Appendix B: User evaluation handout

Overview

CALJAX is a generic Digital Repository system. This means that it is designed to store digital objects (such as articles or media) and allow for users to browse, search and manage the resulting collection.

For testing purposes, CALJAX has been seeded with data from the Bleek and Lloyd collection which is a series of stories from the Bushmen people of Southern Africa. CALJAX generates a static website which can be distributed on any media (such as DVD-ROM).

Searching

CALJAX allows users to search for specific objects in a collection. In this case, the searchable objects are pages from the Bleek and Lloyd collection. Below are a number of tasks which a user of the system might wish to undertake. For each task, please place a mark against the rating which most accurately answers the relevant question.

Tasks

As a researcher, you wish to use the repository to learn more about the bushmen people. As you do not have access to an Internet connection, you are unable to use the online website. Grabbing a copy of the CALJAX version, you decide to use it to conduct your research.

After completing all the tasks, please complete the table on the following page. You may take as much time as you wish. Feel free to experiment with the system beyond the scope of these tasks.

Open the site's search page. This was easy to do.

Upon seeing the interface, it is immediately clear what you need to do to begin searching. You wish to find a specific story which you had previously done research on. The story name is "About Cattle".

The site made it easy to find this story.

The search results were meaningful and easy to interpret.

The search results were generated quickly.

You wish to find all stories by the author "Willem".

The site made it easy to find this story.

The search results were meaningful and easy to interpret.

The results were generated quickly.

You recall reading an interesting story about a lion, but are unable to recall more details. Typing "lion" in the search box yields nearly 300 pages. You decided to narrow it down by specifying the collection to come from the "[xam]" notebooks.

You feel like the narrower scope has helped you.

The search results were generated quickly.

Feedback

Please place a mark in the column which best describes your response to the statement given in each task. Please place only one mark per task.

Tasks	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1					
2					
3a					
3b					
3c					
4a					
4b					
4c					
5a					
5b					

Comments