University of Bristol - Department of Computer Science

COMS40110/1 - Individual Project

OSDigger

Open Source Mailing List Archive

June 1, 2000

Matthew HAMILTON

mh6225@bris.ac.uk

Project Thesis submitted in support of the degree of Master of Engineering in Computer Science

Contents

I Specification and Design **1**

1 User Requirements **2**
1.1 The Customer . . . 2
1.2 Motivation . . . 2
1.3 Requirements . . . 3

2 Initial Specification **5**
2.1 System Overview . . . 5
2.2 Adding new messages . . . 6
2.2.1 Mail Server . . . 6
2.2.2 Parser . . . 6
2.2.3 Database . . . 6
2.3 Indexing Messages . . . 7
2.3.1 Indexer . . . 7
2.4 Serving User Requests . . . 7
2.4.1 Searching . . . 8
2.4.2 Browsing . . . 8
2.4.3 Customisability . . . 8
2.4.4 Query Server . . . 9
2.4.5 Web Server . . . 9
2.5 Misc . . . 9
2.5.1 Privacy . . . 9
2.5.2 Platform . . . 9

3 Risk Analysis **10**
3.1 Project . . . 10
3.2 Technical . . . 11
3.3 Business . . . 13

4 Prototype **14**
4.1 Indexer . . . 14
4.2 Front End . . . 14
4.3 Conclusion . . . 16

II Planning **17**

5 Project Plan **18**
5.1 Timeline 18
5.1.1 Task list 18
5.1.2 Resources and task assignment 21
5.2 Resources 23
5.2.1 Server 23

III Business Plan **25**

6 Executive Summary **26**

7 The Business and Its Management **27**
7.1 Introduction to the Business 27
7.1.1 What are Mailing Lists? 27
7.1.2 What is Open Source Software? 28
7.1.3 Why Open Source Software? 28
7.1.4 Motivation 29
7.2 Mission Statement 29
7.3 History and Position to Date 29
7.4 Objectives, Near Term 30
7.5 Objectives, Long Term 30
7.6 Key Staff 30
7.6.1 Matt Hamilton 30
7.6.2 Tim Saigol 31
7.6.3 Chris Green 31
7.6.4 Chris Parsons 31

8 Products **32**
8.1 Product Description 32
8.1.1 Searching 32
8.1.2 Browsing 33
8.2 Readiness for Market 33
8.3 Applications 33
8.4 Comparison of Competition 33
8.5 Legal Protection 33
8.6 Product Differentiation 34

9 Market and Competitors **35**
9.1 Service Scope 35
9.2 Description of Customers 35
9.2.1 Developers 35
9.2.2 Novice Users 36

9.3 Customer Needs and Benefits 36
9.4 Market Size and Growth 39
9.4.1 Estimating the Market Size 39
9.4.2 Market Growth 40
9.5 Competitors 40
9.5.1 GeoCrawler 40
9.5.2 MARC 41
9.5.3 eGroups 41
9.5.4 DejaNews 41
9.5.5 SourceForge 41
9.5.6 Strengths and Weaknesses 42
9.5.7 Critical Success Factors 42

10 Competetive Business Strategy **44**
10.1 Pricing 44
10.2 Advertising Revenue 44
10.2.1 Advertising Rates 45
10.2.2 In-House Advertising 45
10.2.3 Other Revenue 46

11 Forcasts and Financial Data **47**
11.1 Sales Forcasts 47

IV Implementation **49**

12 Related Work **50**
12.1 Digital Libraries 50
12.2 Information Storage 51
12.3 Indexing Methods 51
12.3.1 Inverted Indexes 51
12.3.2 Signatures 52
12.3.3 Suffix Tries 53
12.3.4 Sequential Searching 54
12.4 Compression 54
12.4.1 Text Compression 54
12.4.2 Index Compression 55
12.5 Storage 56
12.5.1 Index Storage 56
12.5.2 Updating Indexes 56
12.5.3 Distributed Searches 56
12.6 Algorithms 57
12.6.1 Compression Schemes 57
12.6.2 Document Ranking 60

13 Technical Basis **64**
13.1 OSDigger Implementation 64
13.1.1 Storage 64
13.1.2 Word Stemming 65
13.1.3 Index Compression 65
13.1.4 Index Construction 66
13.1.5 Searches 67
13.2 User Interface 68

14 Specification and Design **72**
14.1 Overall Design 72
14.2 Database 72
14.3 Indexer 74
14.3.1 Indexer 74
14.3.2 Inverter 75
14.4 Query Server 75
14.4.1 Onestep Queries 75
14.4.2 Twostep Queries 75
14.5 Web Site 76
14.6 Parser 76

15 Software Guide **77**
15.1 Software Availability 77
15.2 Choice of Programming Languages 77
15.3 Software Structure 78
15.3.1 Indexer 78
15.3.2 Parser 80
15.3.3 Java Servlets 81
15.4 Code Samples 81
15.4.1 Parser 81
15.4.2 Query Server Ranking 83
15.5 Third Party Software Components 85
15.5.1 Compression routines 85
15.5.2 MIME-Tools 86
15.5.3 Xerces XML Parser 86
15.5.4 Xalan XSLT Stylesheet Processor 86
15.5.5 BerkeleyDB 3.x 86
15.5.6 MySQL 86
15.5.7 ZLib 87
15.5.8 MM MySQL JDBC Drivers 87

16 Maintenance Guide **88**
16.1 Source Control 88
16.2 Makefiles 89
16.2.1 Documentation 89
16.2.2 Indexer 89
16.3 Portability 90

V Operations **91**

17 Security **92**

18 Testing **93**

19 Integration **94**

VI Project Review **95**

20 Project Review **96**

21 Future Work **97**
21.1 Tilebars 97
21.2 LSI 97

References **97**

Abstract

This document describes OSDigger, a full text mailing list archive and search system.

Part I

Specification and Design

Chapter 1

User Requirements

1.1 The Customer

Etherworks, a startup, Bristol based company has been formed by the author and several University of Bristol graduates. The aim of the company is to setup a web based archive of Open Source Software [39] electronic mailing lists. The founders of the company have collectively put up the money for capital. So far this capital has been used to purchase a server to develop and host the archive on. Revenue is expected to be generated via advertising on the site once it is established.

The work completed for this project will comprise mainly of the backend systems to operate the archive. The other members of Etherworks will be working on the web site design, graphics and administrative tools. These are not to be considered as part of this project.

1.2 Motivation

Much of the development of Open Source software is discussed on mailing lists. There are hundreds of these lists covering various topics to do with the use and development of the software. The main essence of Open Source software is communication between the community developing the software.

Most of these lists are not archived in any way, and the archives that do exist only offer very basic searching and browsing functions. Due to the computationally expensive nature of searching through large archives, most of the archives only allow the user to search through a single subsection at a time.

Many of the questions asked on the lists have often been previously asked, and more importantly answered. What is needed is a single central archive of the various lists, which users can search through to find what they are looking for.

This site has the potential to be a very valuable resource used by both developers and users of all levels.

Nothing like this at this scale currently exists on the internet. There are quite a few smaller sites that archive multiple lists, but none with a site-wide search facility.

Many web search engines, such as AltaVista and Infoseek, also sell their indexing and search software as a separate standalone commercial product. However this software is geared towards indexing web sites and not mailing lists.

1.3 Requirements

In order to set this archive apart from other similar sites on the net it will have to offer much larger capacity and more advanced features:

Capacity

The archive will have to be able to handle the storage, retrieval, and searching of millions of messages. It must be capable of handling several simultaneous users a second visiting the site without significant slowdown.

Scalability

As the service become more popular and the load increases, it should be possible to split the components of the archive up to run on seperate servers. This should increase the capacity that can be handled by the archive.

Near Real-Time

The archive must make messages available for browsing as soon as they arrive. The search indexes must be updated regularly, at least once a day, so that searches return recent messages.

Browsing

Users must be able to easily and efficiently browse the archive and see the most recently sent messages of a particular list. They must be able view the individual messages and move to next and previous messages in the message's thread.

Searching

The site must allow users to search through the archives of messages for keywords. They must be able to search across all lists held or just a select set of lists. The results must be ranked by relevance and must be returned within a couple of seconds.

Customisation

The site must be customisable by the users. They must be able personalise various aspects of the site, such as default searches, time and date formats, number of messages per screen to show.

Cost

Etherworks are supplying a server to develop the site and software on. The server should be sufficient to run the full site once opened to the public for the first couple of months. The only major cost anticipated will be the cost of co-locating the server at an ISP once the site goes commercial.

Advertising Space

The site is expected to generate revenue through advertisements placed on the site. Therefore the site must be designed in such a way to accommodate various standard sized banner ads.

Chapter 2

Initial Specification

This specification details the initial components of the system. The final details of the system are left until the *Technical Specification* which appears in Chapter 14.

2.1 System Overview

The archive has three main functions: *adding new messages*, *indexing messages* and *serving user requests*. In order to fulfil these functions the system is comprised of various components as shown in Figure 2.1.

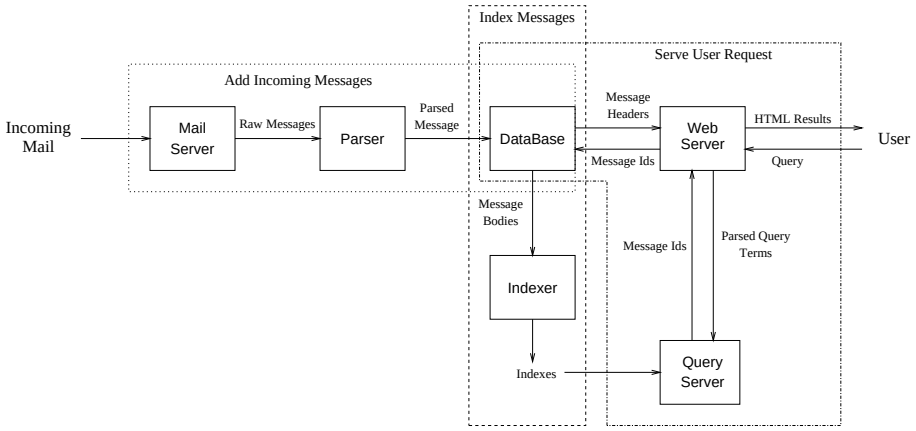


Figure 2.1: System Overview

2.2 Adding new messages

New messages will arrive at the mail server via standard SMTP transports from the mailing lists. These messages will then be passed through a parser to convert them into a suitable form to add to the database. The parser will also take care of adding the actual messages to the database.

2.2.1 Mail Server

The mail server will accept the messages from the mailing lists. It must be highly configurable, allowing messages to be piped to external programs, and allow the creation of arbitrary addresses that are not users according to the underlying operating system.

2.2.2 Parser

The parser must be able to take one or more messages in on standard input from the mail server. This means that the parser will be able to be used like any other unix mail filter and be spawned by the mail server each time a message appears.

During the parsing process the message will be split into its constituent parts – headers and body. The relevant headers (eg. To:, From:, Date:, Subject) will be inserted into the database along with the message body.

The message body must first be converted from any encoding that may be used (eg. Quoted-Printable, or Base64) into plain 8bit ASCII. If the message is a multipart MIME message then only the parts of mime type `text/plain` will be added. This prevents the system from archiving any binary attachments such as images or applications.

2.2.3 Database

The parsed messages will be stored in a database for later retrieval and indexing. The database must allow large `TEXT` or `BLOB` objects to be inserted and be able to handle millions of rows efficiently.

Although, not strictly necessary it would be desirable if the database allowed user defined functions to be coded to do particular tasks at the server side – eg. compression, parsing.

The database must be accessible from the parser, indexer and the web server, hence must have an API in whatever language is going to be used for those components.

2.3 Indexing Messages

At periodic intervals the messages must be (re)indexed. This process involves fetching the messages from the database and creating some sort of index structure that can be used to quickly locate messages that match a certain search criteria.

2.3.1 Indexer

The indexer will fetch the messages from the database and create an inverted index of the messages. This index will then be stored to disk for the *Query Server* to use to answer queries.

The indexer must be extremely fast and efficient on system resources. It is expected that the indexes will be rebuilt at least once a day, and that there will be millions of messages in the database. Assuming the the average message is 2 Kilobytes, this means that the indexer must be able to index several Gigabytes of text an hour.

As system resources are limited to the server that has been supplied, the indexer must be able to create a full index in memory in one go, or index chunks of the text at a time and merge the results.

The mailing lists are constantly used, thousands of messages are sent to these mailing lists a day. The indexer must be able to cope with the continually growing and changing archive.

2.4 Serving User Requests

There are two main activities supported by the archive – browsing and searching. This means that the archive must be able to show a chronological list of the latest messages for a user to read through; and be able to search through the millions of messages and return the results of a query.

2.4.1 Searching

The search interface presented to the user must be intuitive and easy to use. The query entered by the user will be sent to the query server by a script on the web server.

The query server will return a list of document ids and scores to the script which will then fetch extra meta-data (eg. Subject, Author, List) from the database. The final results will be parsed by the script into HTML and presented to the user.

2.4.2 Browsing

The user will be presented with a hierarchy of subjects and lists similar in style to Yahoo [61]. The catagories should be cross-linked such that a user can can move from one related catagory to another.

Once a list has been chosen the user should be shown a view with the most recent messages in that list. The view should show the date the message arrived, the author and the subject.

Clicking on a message should show the user that particular message with the relevent headers and meta-data. The user should be able to move easily to previous and next messages in the list and in the subject thread of the message.

2.4.3 Customisability

A large proportion of the target users of the archive are very technically competent and much prefer speed and efficiency over flashy graphics and bells-and-whistles. On the other hand, there may be many novices who would prefer the extra features. The ideal situation is a system that is customisable, so that users can make their own minds up about what they want.

Aspects of the site that should be customisable should include:

- The date format (mm/dd/yy, dd/mm/yy, etc.) and the timezone.
- The number of messages to display per screen.
- The default lists to show when browsing.
- Default search parameters.

The archive should keep a record of the users preferences such that when they return to the site is is automatically displayed how they want.

2.4.4 Query Server

The query server accepts the queries from the scripts running on the web server and returns the raw results. The query will consist of the search terms including any modifiers – such as negation, the number of results to return, and the lists to search. The results will be a list of message ids and their relative score according to some ranking algorithm.

The query server must return the search results within a couple of seconds otherwise users will loose interest.

2.4.5 Web Server

The web server will be the main interface for the users to the archive. Therefore it must be reliable. It must support scripting languages or have an API to allow the connection to the Query Server.

It is possible that performing the searches on the archive may take some time and hence the web server must be able to continue processing requests while waiting.

2.5 Misc

2.5.1 Privacy

Not everybody wants their messages archived and there needs to be a way to opt out of have their messages archived. The most common method used for archives it to set a `X-No-Archive` header in the message. All messages with this header will not be stored in the database.

2.5.2 Platform

The archive and indexing system must be able to run on moderate hardware. A server has been supplied to host the archive and indexes. The specifications of the server are detailed in Section 5.2.1.

Chapter 3

Risk Analysis

The risks have been divided up into three sections *Project*, *Technical* and *Business*. Each risk has been given an estimated probability of occuring, and an estimation of the impact to the project.

3.1 Project

Department Y2K shutdown
Probability: 90%, Impact: low
Detail: The University will be shutting down over the millennium for safety reasons. This means that the computing facilities will be out of use. There may be residue effects of the shutdown such as lost working time.
Reduction: Most of the development of the software from the project can also be done from home. Backups of the main CVS repository are taken each night so there will always be access to recent revisions of the software to work on should the department close.

Illness
Probability: 5%, Impact: High
Detail: As I am the only person working on the project, should I become ill, the project will probably not be completed on time.
Reduction: As this is an individual project, there is not much really I can do about it.

Run out of time
Probability: 20%, Impact: Medium
Detail: I have other assignments to do within the university which may take up more of my time than planned. It is also possible that I might

under-estimate the time needed to complete the project.
Reduction: Proper planning. I need to make a project plan and timeline, detailing how much time each part of the project will take, and stick to it.

3.2 Technical

Parser fails to parse all forms of messages

Probability: 10%, Impact: Medium
Detail: The parser has to parse all sorts of messages coming in, conforming to several standards [10, 17, 18, 33, 46]. If it fails to parse the messages correctly then messages may be missed by the archive, or worse still bounced messages may cause the mailing list to unsubscribe us.
Reduction: The only way to test that the parser handles all the standards of messages (and their idiosyncrasies) is to parse as much mail as possible and watch the outcome. There are various libraries for parsing Internet mail and hopefully we can use one of them. If a message cannot be parsed an error should be logged so we can refine the parser.

Indexer cannot index text fast enough

Probability: 5%, Impact: Medium
Detail: The indexer must be able to index the messages fast enough such that the index can be kept as up-to-date as possible. We hope to rebuild the index at least once a day.
Reduction: By carefully studying existing algorithms for creating indexes and profiling existing indexers the common bottlenecks have been found and removed or optimised. In the worst case a faster server can be purchased (at the time of writing, the fastest production processors are 2-3 times faster than the ones used in the supplied server).

Query Server cannot perform queries fast enough

Probability: 10%, Impact: Medium
Detail: As the size of the index grows and the popularity of the archive increases the queries may take longer to run. If the results are not returned within a couple of seconds then the user will lose interest.
Reduction: By benchmarking and profiling the the query server we should be able to find out any slow spots in its execution. Where possible it should use a minimum number of disk accesses

Server security is breached

Probability: 1%, Impact: Severe
Detail: If someone gained unauthorised access to the server, they could delete valuable archives of messages, or take the server offline.

Reduction: The server will be running FreeBSD, a fairly secure variety of Unix. A firewall will be running and only specific ports will be let through. Only specifically needed services will be running. Further information about the security of the server can be found in Section 17.

3.3 Business

Competition

Probability: 20%, Impact: Medium

Detail: There are currently no archives of this scale on the web, which puts us in the strong position as being the first, however large web portals with existing presence could still release a competing site.

Reduction: We have found that there is currently no out-of-the box software to do what we want, this means that any competitors would have to develop their own software as well. They could still launch before us, but having done much of the research already we would still have the lead. By keeping in close contact with the Open Source community we hope that our site will be more popular than any competitors.

Overwhelming success

Probability: 30%, Impact: Medium

Detail: Should the site become an overwhelming success, resulting in more visitors to the site than we can handle, we could easily fail.

Reduction: Before we launch we will do load simulations on the server, to assess the effects of lots of simultaneous requests. From these tests we will be able to work out what the highest hit rate the server can safely handle. With this data we will be able to decide whether we have enough capacity to handle the initial interest. More details of these simulations will be detailed in Section 18

Alienating the Open Source community

Probability: 10%, Impact: High

Detail: The Open Source community is very wary. They want the best for their software and their community, and hold ideals of privacy and open standards very high. Should we do something that grossly flaunted these ideals we could risk losing credibility and popularity.

Reduction: We must make sure that we consult various people within the community to make sure we are providing a service that they will use. Once the site has been launched we will be releasing the source code to the system under one of the Open Source licenses. This way we will be releasing something back into the community and giving the opportunity for others to make enhancements and changes to our software should they feel fit.

Chapter 4

Prototype

4.1 Indexer

A prototype of the indexer was developed. This was primarily to test out various indexing methods and have a chance to do some research into the area. This research is shown in more detail later in Chapter 13.

A lot of research has been done previously into the area of Information Retrieval, and several books [29, 2] and papers have been quite a useful base.

The prototype indexer was written in C and created compressed inverted indexes based on the *Local Bernoulli Model* introduced by Witten et Al [29]. These schemes use the fact that a list of document numbers are an increasing stream of integers with small intervals. As such they can be represented in a much more compact form by storing the list of document numbers in which a particular word occurs as just a list of intervals.

This resulted in an average of just over 4 bits per index entry, compared to 32 or 64 bits if just storing the entry as an `int` or `long`.

The speed of the indexer was very impressive as well, resulting in around 3GB of text indexed per hour – with 2-3 term ranked queries being answered in under a second.

4.2 Front End

A simple web page was created using Perl and HTML to interact with the database. A simple parser was written in Perl that would take a message in

Figure 4.1: Prototype Screenshot

on standard input and insert it into the database. The web page shows messages in a particular list sorted in reverse chronological order. A screenshot of this interface can be seen in Figure 4.1.

4.3 Conclusion

From coding and using the prototype a number of points were discovered:

- The indexing algorithms discussed by Witten et Al will be fast enough for our needs.
- Parsing messages is actually quite difficult, as there are so many standards out there and many mailers do not follow them all correctly.
- Due to its great support for text manipulation, Perl seems like a good language to write both the parser and web scripts in.
- The server supplied by Etherworks should have enough memory and CPU power to efficiently construct the indexes.

Part II

Planning

Chapter 5

Project Plan

In order for the project to be a success it must be ensured that it finishes on time and that we have allocated enough time and resources for each task.

5.1 Timeline

The project has been broken up into individual tasks that need to be completed. A Gantt chart representing these tasks is shown in Figure 5.1. The other members of Etherworks have been included in this plan, however their actual work should not be considered part of the project, and hence is not shown to any detail.

5.1.1 Task list

- Customer Requirements
  1999.Oct.04 - 1999.Oct.15; assigned to Matt Hamilton

  The other members of Etherworks and the Open Source community as a whole were asked for what they thought was required of the service.

- Initial Specification
  1999.Oct.18 - 1999.Oct.22; assigned to Matt Hamilton

  An overall structure for the archive was devised and the principle components marked out. This will later be expanded into the Technical Specification

- Prototype
  1999.Nov.01 - 1999.Nov.26; assigned to Matt Hamilton

A prototype was built to test if the idea is feasible. The prototype was written in C and specifically tested the speed and scalability of the compression algorithms found in [29].

- Risk Analysis
  1999.Dec.10 - 1999.Dec.16; assigned to Matt Hamilton

  The main risks that could cause the project to fail or not be completed on time were predicted. Their probability of occurrence and consequences as well as what could be done to reduce this risk were estimated.

- Technical Specification
  2000.Jan.11 - 2000.Jan.21; assigned to Matt Hamilton

  The Technical Specification will contain the exact design of the backend components for the archive. The coding of the end product will be based upon this document and will be useful for anybody else wanting to modify the code.

- Design DB Structure
  2000.Jan.25 - 2000.Jan.27; assigned to Matt Hamilton

  The messages will be stored in a database, the structure of which needs to be designed. The database will also hold information such as user preferences and administravia.

- Coding Parser
  2000.Jan.28 - 2000.Feb.16; assigned to Matt Hamilton

  The parser that decodes messages and inserts the into the database will be coded

- Test Parser
  2000.Feb.17 - 2000.Feb.23; assigned to Matt Hamilton

  Testing of the parser will involve subscribing to many mailing lists and testing to see if the parser can parse all of the messages.

- Code Indexer
  2000.Feb.24 - 2000.Mar.14; assigned to Matt Hamilton

  The indexer that fetches messages from the database and produces the index files used by the Query Server will be written.

- Test Indexer
  2000.Mar.15 - 2000.Mar.21; assigned to Matt Hamilton

  The indexer will be tested and profiled to make sure it is fast enough. It is expected that there will be quite a few refinements of the code to try and squeeze as much speed as possible out of the indexer. This

task will also involve coding a decoder to make sure that the indexes produced are valid.

- Code Query Server
  2000.Jan.28 - 2000.Apr.10; assigned to Matt Hamilton

  The daemon that accepts queries from the web server will be coded.

- Test Query Server
  2000.Apr.11 - 2000.Apr.17; assigned to Matt Hamilton

  Testing the Query Server will involve creating sample scripts to submit queries to the Query Server and time response times.

- Code Web Scripts
  2000.Jan.28 - 2000.May.05; assigned to Matt Hamilton

  These scripts will form a library or module with functions to submit queries, and retrieve list indexes and messages. They will be called by the web pages being designed by Chris Parsons.

- Test Web Scripts
  2000.May.08 - 2000.May.12; assigned to Matt Hamilton

  The scripts will be tested with some simple HTML pages to make sure they work correctly. At this point we should have all of the functionality of the archive implemented.

- Backend Integration Testing
  2000.May.15 - 2000.May.26; assigned to Matt Hamilton

  All of the backend components will be tested together to make sure that they work correctly with each other.

- Thesis Writeup
  2000.Jun.06 - 2000.Jul.03; assigned to Matt Hamilton

  The remaining parts of the this thesis will be written up. Also a complete business plan will be written to project where we go from the end of the project.

- Design Admin Site Tools
  2000.Jan.11 - 2000.Apr.28; assigned to Tim Saigol

  Tim Saigol will be designing administrative tools that will be needed for the day-to-day running of the archive once we launch.

- Design Web Site Graphics
  2000.Jan.11 - 2000.Apr.28; assigned to Chris Green

  Chris Green will be designing the graphics for the web site and deciding the overall look and feel of the site.

- Design Web Site Layout
  2000.Jan.11 - 2000.Apr.28; assigned to Chris Parsons

  Chris Parsons will be working on designing the final web site layout. He will be coding the needed HTML and calling the backend scripts that will be written as part of this project.

- Frontend Integration Testing
  2000.May.01 - 2000.May.12; assigned to Chris Parsons

  The graphics, HTML and scripts will be tested together to ensure a consistent look and feel across the site and that the site works under different web browsers and at different connection speeds.

- Beta Test
  2000.May.29 - 2000.Jun.02; assigned to Resource group

  Selected groups of people will be asked to test the site and let us know what they think. Any changes and improvements will be made at this time. We will also do synthetic load benchmarks to see how the server will handle heavy loads and realistically how many concurrent users we can support.

- Archive Site Launch
  2000.Jun.05 - 2000.Jun.05; assigned to Resource group

  Launch the site! We will advertise the site on various other web sites and discussion forums and try and get as much interest as possible.

5.1.2 Resources and task assignment

Matt Hamilton

Start	Finish	Days	Done	Task
1999.Oct.04	1999.Oct.15	10	10	Customer Requirements
1999.Oct.18	1999.Oct.22	5	5	Initial Specification
1999.Nov.01	1999.Nov.26	20	20	Prototype
1999.Dec.10	1999.Dec.16	5	5	Risk Analysis
1999.Dec.21	2000.Jan.05	12	0	Vacation
2000.Jan.11	2000.Jan.21	9	0	Technical Specification
2000.Jan.25	2000.Jan.27	3	0	Design DB Structure
2000.Jan.28	2000.Feb.16	14	0	Coding Parser
2000.Feb.17	2000.Feb.23	5	0	Test Parser
2000.Feb.24	2000.Mar.14	14	0	Code Indexer
2000.Mar.15	2000.Mar.21	5	0	Test Indexer

2000.Jan.28	2000.Apr.10	14	0	Code Query Server
2000.Apr.11	2000.Apr.17	5	0	Test Query Server
2000.Jan.28	2000.May.05	14	0	Code Web Scripts
2000.May.08	2000.May.12	5	0	Test Web Scripts
2000.May.15	2000.May.26	10	0	Backend Integration Testing
2000.Jun.06	2000.Jul.03	20	0	Thesis Writeup

Total: 158 days scheduled, 40 days completed.

Tim Saigol

Start	Finish	Days	Done	Task
1999.Dec.21	2000.Jan.05	12	0	Vacation
2000.Jan.11	2000.Apr.28	79	0	Design Admin Site Tools

Total: 79 days scheduled, 0 days completed.

Chris Green

Start	Finish	Days	Done	Task
1999.Dec.21	2000.Jan.05	12	0	Vacation
2000.Jan.11	2000.Apr.28	79	0	Design Web Site Graphics

Total: 79 days scheduled, 0 days completed.

Chris Parsons

Start	Finish	Days	Done	Task
1999.Dec.21	2000.Jan.05	12	0	Vacation
2000.Jan.11	2000.Apr.28	79	0	Design Web Site Layout
2000.May.01	2000.May.12	10	0	Frontend Integration Testing

Total: 89 days scheduled, 0 days completed.

Chris Green, Chris Parsons, Tim Saigol, Matt Hamilton

Start	Finish	Days	Done	Task
2000.May.29	2000.Jun.02	5	0	Beta Test
2000.Jun.05	2000.Jun.05	1	0	Archive Site Launch

Total: 6 days scheduled, 0 days completed.

5.2 Resources

5.2.1 Server

A server has been provided by Etherworks for the development and hosting of the archive. The Department of Computer Science has kindly allowed the server to be connected to their network, where is can stay during the development stage of the project, however it must be moved for the launch of the site.

Memory	300MB
Processors	2 x 300Mhz Intel Pentium II
Storage	4 x 4GB Wide SCSI Hard Disks DPT RAID 5 Controller with 8MB cache
OS	FreeBSD 3.4

Table 5.1: Server Specification

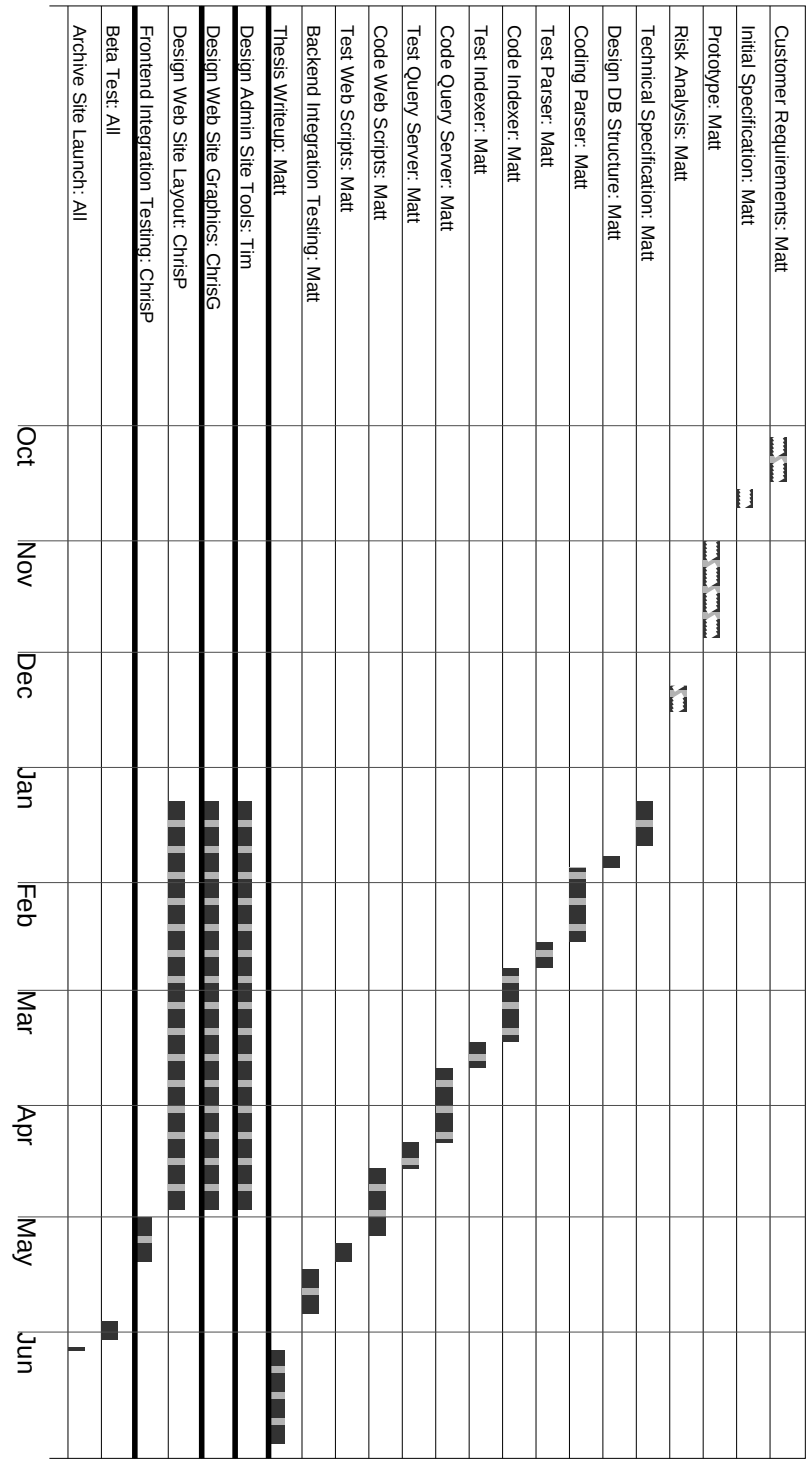


Figure 5.1: Project Gantt Chart

Part III

Business Plan

Chapter 6

Executive Summary

Chapter 7

The Business and Its Management

7.1 Introduction to the Business

Etherworks, a startup IT venture was formed by four students from the University of Bristol, UK. The main aim of the company was to develop and launch OSDigger, an archive/search service for mailing lists on the Internet.

The primary target is to archive mailing lists for Open Source Software development projects. These projects are developed by loose-knit groups of individuals and organizations spread across the globe, whose primary means of co-ordination for these project is via mailing lists.

7.1.1 What are Mailing Lists?

Electronic mailing lists are similar to their conventional counterparts in that they serve to distribute information to a subscribed audience. With electronic mailing lists a particular email address is setup, eg. `freebsd-questions@freebsd.org`. Any email messages sent to that address are automatically re-sent to all subscribers of the list. There may be thousands of subscribers on a list. Anybody can subscribe to a mailing list by sending a simple email to an automated management address with the word `subscribe` in the body of the message.

7.1.2 What is Open Source Software?

The basic idea behind open source is very simple. When programmers on the Internet can read, redistribute, and modify the source for a piece of software, it evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing.

Some facts about Open Source software:

- Linux-based OSes have the leading market share of Web servers powering the Internet's public Web sites, with 31 percent of all sites, according to a Netcraft study [36]. The next most popular OS has only 20 percent.
- Open-source e-mail routing tool Sendmail handles 80 percent of all the e-mail traffic on the net, according to a study by International Data Corp. (IDC).
- More than 61 percent of all public Web sites are powered by the open-source Apache Web Server, according to a Netcraft study [36].
- DNS and Bind, the Domain Name Server tools that route all your URL requests across the Net, may represent as much as 100 percent market share, according to industry analysts.

In effect, the Internet itself would not function without the open-source tools that power it.

7.1.3 Why Open Source Software?

As explained by Bob Young, founder of Red Hat Linux:

> The best analogy that illustrates this benefit is with the way we buy cars. Just ask the question, "Would you buy a car with the hood welded shut?" and we all answer an emphatic "No." So ask the follow-up question, "What do you know about modern internal-combustion engines?" and the answer for most of us is, "Not much."
>
> We demand the ability to open the hood of our cars because it gives us, the consumer, control over the product we've bought and takes it away from the vendor. We can take the car back to the dealer; if he does a good job, doesn't overcharge us and

> adds the features we need, we may keep taking it back to that dealer. But if he overcharges us, won't fix the problem we are having or refuses to install that musical horn we always wanted – well, there are 10,000 other car-repair companies that would be happy to have our business.
>
> In the proprietary software business, the customer has no control over the technology he is building his business around. If his vendor overcharges him, refuses to fix the bug that causes his system to crash or chooses not to introduce the feature that the customer needs, the customer has no choice. This lack of control results in high cost, low reliability and lots of frustration.
>
> With Open Source, companies like Red Hat are able to treat our customers as partners in the use of the technology they're building their businesses around.
>
> Having control over the technology they are using is the benefit that is enabling users of open-source tools to build more-reliable, more-customized and lower-cost systems than ever before.

7.1.4 Motivation

The idea for the service arrived out of our own experiences in trying to search through the vast back catalogue of mailing list messages to find answers to questions we had that we knew had probably already been addressed previously.

7.2 Mission Statement

Arising from our own experiences, we intend to provide a service to allow the preservation and extraction of knowledge from mailing lists. We are aiming directly at the Open Source community to add value to the freely available mailing lists. We also aim to help novice users to more easily enter the Open Source arena by allowing easy access to this knowledge.

7.3 History and Position to Date

So far a prototype site has been setup to demonstrate the workings of the site. The search engine, which is the main component of the site has been mostly designed and written. Users can currently enter a search query and

have the results returned to them. More testing is needed before the site is ready to be publically launched.

A bank account has been setup for the company.

The main developer, Matt Hamilton, is about to graduate from University and will be working full-time on the final development of the site.

7.4 Objectives, Near Term

- To start an advertising campaign to
- To publically launch the site on the 1st of August

7.5 Objectives, Long Term

- To attract 200,000 visitors a month by the end of the first year of operation.
- To develop the site into the most comprehensive source of Open Source information.

7.6 Key Staff

Etherworks is a partnership of four people whom met whilst studying at the University of Bristol, UK. They all hold equal share of the company, and together personally funded £2,000 for the purchase of a server for prototype work. They all have a technical background, although several have experience of running their own small companies. We are working with the Enterprise Center [3] at the University of Bristol, whom are providing the needed extra business and financial expertise.

7.6.1 Matt Hamilton

Matt, 22, is the main developer of the software behind the search engine. The software was developed whilst studying a Computer Science (MEng) at the University of Bristol. He graduates in June 2000 and intends to work full time on the final development of the OSDigger business.

He has run a part-time computer consulting business whilst at University with an annual turnover of £8000. His main contract, was installation and maintenance of an email system and virtual private network to link the

Moscow and New York offices of Clintondale Aviation, a charter aviation company. He also worked 4 months over summer 1999 with Hewlett Packard in Bristol, UK.

7.6.2 Tim Saigol

Tim, 22, currently works for IPL, a contract programming company in Bath, UK. He graduated with a ii-1 in Computer Science (BSc Hons) in 1999. Tim will be working part-time on the design and coding of various components of the search engine.

7.6.3 Chris Green

Chris, 22, currently works as a graphics designer for Netsight Internet Solutions, an Internet Service Provider in Bristol. He graduated with a 1st in Computer Science (BSc Hons) in 1999. He worked for IPL in Bath, UK for nine months before leaving to work with Netsight. Chris will be working part-time on the graphics design of the web site.

7.6.4 Chris Parsons

Chris, 21, currently works for Elixer Studios a Games Programming company in London. He ran a part-time web design business from University. He graduated with a 1st in Computer Science (BSc Hons) in 1999. Chris will be working part-time on the design of the web site layout and interaction with the search engine.

Chapter 8

Products

8.1 Product Description

The main product being developed by Etherworks is called OSDigger. It is an archive of email messages sent to Open Source Software mailing lists. The form of the archive will be a web site which users can use to browse through the large back catalogue of messages and perform searches to find specific topics. The archive will be updated in real-time as messages are sent out to the lists. This means that the users will be able to read current messages on the lists as easily as if they were subscribed to them.

8.1.1 Searching

The main piece of software being developed is the search system, this is the most complex piece and is the main product differentiator. The search will allow users to type in a query such as *freebsd digital camera* which could return messages about how to use a digital camera with FreeBSD (an alternative operating system).

The search system has been custom written as there is not an existing product on the market to fulfill the needs of the site. We believe that this gives us the competitive advantage. As well as this, the search system incorporates some new features not yet seen in search engines generally.

It is estimated that the archive will hold several million messages, and so the search system must be easy to use and quick.

8.1.2 Browsing

Users will also be able to browse through over 300 mailing lists, grouped by subject. The site will list the messages in chronological order, and users can jump to a particular date should they want to.

It is intended that users will be able to use this feature to read the mailing list messages without having to subscribe themselves. This is useful for people who just want to occasionally want to read what is being posted to the lists. People who mainly read the lists, and do not post many messages will find this very useful.

Users will be able to access the list messages from anywhere using a standard web browser. This paradigm of web-based messaging has been proved to be popular by sites such as HotMail, and Yahoo Mail.

8.2 Readiness for Market

The bulk of the search engine was written by Matt Hamilton as part of his final year undergraduate project at the University of Bristol. A prototype of the site has been created, `http://beta.osdigger.com`, that demonstrates the main working points of the final commercial service.

8.3 Applications

The main application of the software is for the OSDigger web site. However it is envisaged that there may be a market for the back end software to be used for other purposes or maybe sold as a separate product.

8.4 Comparison of Competition

There is one main competitor found in the market, GeoCrawler, whom are described in Section 9.5.

8.5 Legal Protection

Software cannot be patented hence this is not an option. Nor does it use any new algorithms or processes which could be patented. The software for the search engine will be released under the GNU public license. This is an

Open Source license that means that the software can be used for whatever means and that the source code will always remain available. This is in keeping with the subject of the lists being archived, and will provide the Open Source community with a valuable resource.

8.6 Product Differentiation

As mentioned previously the main product differentiator is the custom written search engine. All of the other sites providing search facilities use a generic search engine designed for indexing and searching web pages. These are generally designed for small - medium sized web sites up to a couple of thousand web pages. As the archive will contain several million messages, something more scalable is needed. Another main problem is that the archive is continually expanding, in the order of several thousand messages are added each day. Most already available search systems cannot incrementally update their indexes and hence all of the messages must be re-indexed. This is not such a problem for a site with just the mailing list archives for a handful of projects, but for such a large site, the indexing would take too much time.

Chapter 9

Market and Competitors

9.1 Service Scope

Since the service is offered via the web it is accessible worldwide. The service will be initially based in the UK. The UK has good Internet connectivity on a worldwide scale, so is a fair location for the servers. Future expansion plans may include setting up mirrors of the service in other geographical locations to better service those areas.

9.2 Description of Customers

There will be two main categories of customer: developers and novice users. They will have slightly different uses for the service.

9.2.1 Developers

Developers are the people that create the software. There can be a few or as many as hundreds or thousands of developers working on a project and they use the mailing lists to co-ordinate the work they are doing on the software. Developers will be using the service to keep track of developments of the projects they are working on. They may want to look back at previous discussions and keep a peripheral view on the discussions happening on a related project without flooding their mailbox with email.

9.2.2 Novice Users

With the rise in popularity of alternative operating systems such as Linux, more and more novice users are entering the realm of what has traditionally been considered the domain of experienced users only. Although it is growing there is currently not very much printed documentation for these operating systems and applications – this is mainly due to the rapid development pace of these projects. Consequently often the best place for new users to look for information is on the mailing lists. If a user is just looking for the answer to what is probably a common question, they probably do not want to subscribe to the mailing list just to find out the answer. Also they are unlikely to get a response to their question if it was just asked last week by someone else.

To start with we will be concentrating on the Developers rather than the Novice users. This is simply because the Developers will be looking for a simple clean system that just does the job. The novice users will want more features to help them navigate the vast amounts of data. We could also provide extra services such as collections of other documentation and FAQs [1].

9.3 Customer Needs and Benefits

There are several main needs of the customers:

- To be able to search for answers to questions
- To keep in touch with what is happening on lists
- To use the site as a replacement for reading mailing list messages in their mailbox.

This was found by a survey that was filled in by 50 participants. The survey was 10 short questions about the mailing list usage of the participant. The results were more from the developers than the novice users as they were the easiest to get ahold of and the most willing to fill in the survey.

The survey was announced on a couple of Internet news forums and most of the participants filled in the survey within 6 hours of it being announced. From the web server log files it can be seen that 126 people visited the survey page

[1] Frequently Asked Questions

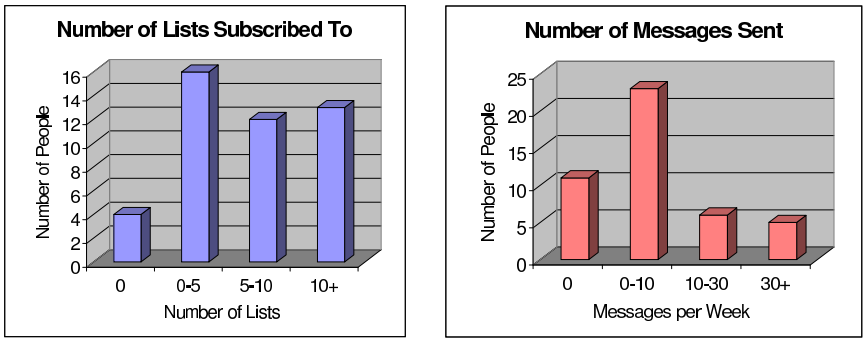


Figure 9.1: Survey Results

The results of the survey were very promising. Over half the recipients that responded indicated that they are subscribed to over 5 lists, showing that indeed the lists are used a lot.

Most people send less than 10 messages a week to the lists. This shows that the emphasis is on reading messages rather than writing them – obviously if everybody wrote as many messages as they read chaos would ensue! However, nearly a third of the respondents send zero messages a week, these people are prime customers. They are only observers on the lists and would probably welcome a less intrusive way of reading all the messages, than subscribe to each list.

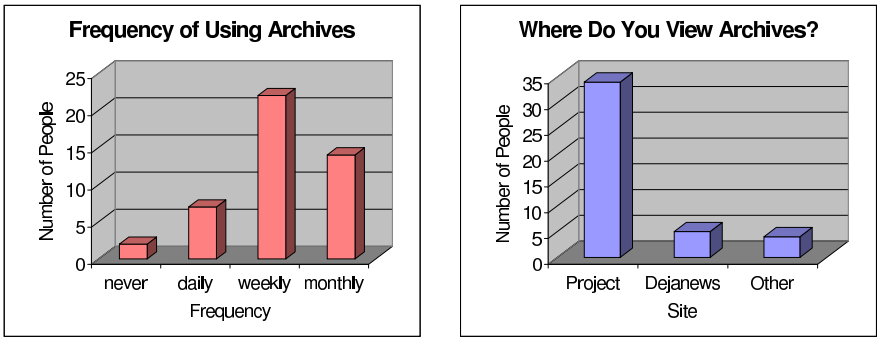


Figure 9.2: Survey Results

The survey also asked how often people use existing archive services for the lists – either to read or to search for messages. The majority of respondents indicated that they used the archives once a week. This shows that we should expect a large number of regular repeat visitors.

Many Open Source projects keep their own mailing list archives on their site, however as noted earlier their searching facilities are generally primitive and archives are not updated as often as one would like. These sites are still the most popular place for people to browse and search the archives. This indicates that any other competitors out there have not made significant inroads into the market. Dejanews is a site that the author uses quite frequently, which indexes newsgroups, not mailing lists, but even it does not seem to be very popular.

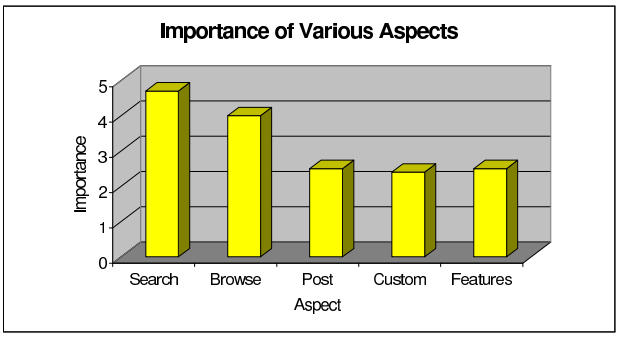


Figure 9.3: Survey Results

The participants were then asked to rate from 0 to 5, five aspects of an archive site as to how important they are. The aspects were:

- Ability to Search
- Ability to Browse
- Ability to post messages to the lists
- Customisability
- Extra Features (eg. news, other documentation)

The results show that searching and browsing are the two most important aspects of the site, the other three all averaged right in the middle at 2.5. Searching had an average score of 4.7 which was the highest, which confirms the need for good searching facilities on the site.

The results obtained from the survey are probably somewhat biased, in the fact that participants answered the survey voluntarily, and hence only those who have some interest in the mailing lists responded. Once the site is up and running we hope to keep in close contact with the visitors by way of polls and surveys to make sure we spend our efforts effectively on developing the site in the right direction for all users.

9.4 Market Size and Growth

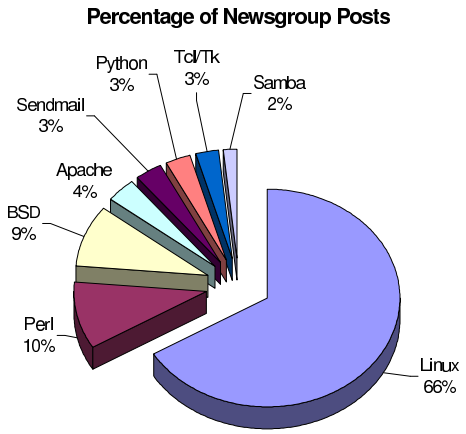
9.4.1 Estimating the Market Size

As mentioned above more and more users are switching away from Microsoft Windows to alternative Operating Systems such as FreeBSD and Linux. These are the biggest and most complex Open Source Projects and have the most numbers of mailing lists and subscribers. Hence this is the area that most of this research has been done.

Estimating the market size of the different open-source user communities can be very difficult as there are no definitive surveys or studies. Also since most can be freely downloaded and copied, we cannot simply ask a single vendor or source for figures.

A prediction was done to try and estimate the size of various communities in 1998. This data is somewhat out of date now, but gives us an idea of what is now probably a minimum figure.

The statistics were calculated by taking ratios of the number of posting to Internet newsgroups associated with each community, as measured by the Netscan analysis tool [2]. From the size of these groups the number of active users was extrapolated using a base of 7 million Linux users.



[2]Smith, Marc. 1997. "Netscan: A tool for measuring and mapping social cyberspaces" http://netscan.research.microsoft.com.

	Number of Posters	Estimated Size of User Community
Linux	13,231	7,000,000
Perl	2,023	1,000,000
BSD	1,820	960,000
Apache	738	400,000
Sendmail	670	350,000
Python	612	325,000
Tcl/tk	563	300,000
Samba	309	160,000

9.4.2 Market Growth

The latest figures show the Linux community to be around 12 - 14 million users [30]. And studies show an astounding 40% growth per year.

9.5 Competitors

There is currently only one direct competitor to OSDigger, that is GeoCrawler. They also are targeting the Open Source community and have quite a large archive already.

9.5.1 GeoCrawler

GeoCrawler has 2,800,000 emails with 5,000 added each day. This means it is roughly the same size the OSDigger will initially be. It allows searching and browsing of the mailing lists and has some customization. According to the web page the archive is created and updated by *custom java spiders*. This shows that they have been writing some software themselves, however the search software used is a public domain piece of software for searching web pages called *HT://Dig*. They only allow searches on a particular list, you cannot search across multiple lists.

They allow posting of messages to lists, which require users to apply for a free account and login before they do so.

From their web page they appear to be sponsored by Linux.com which is a large commercial Linux organization.

9.5.2 MARC

Mailing list **ARC**hive. Was setup initially for internal use by a company, it then evolved into a public service. Again, to search you have to specify a particular list and a particular year first. The data is stored in the same database, mySQL, that OSDigger uses. It allows customization of colour scheme, but not much else. Appears to be funded by its parent company and is run in the creators spare time. It contains many Open Source lists, but does not target them exclusively.

9.5.3 eGroups

From the eGroups website:

> eGroups is a free email group service that allows you to easily create and join email groups. Email groups offer a convenient way to connect with others who share the same interests and ideas

Not a direct competitor as it hosts its own mailing lists, and hence does not cover the multitude of existing mailing lists. It has limited searching capabilities, but offers other services for group collaboration, such as shared calendars. The interface is pretty cluttered and slow to navigate.

9.5.4 DejaNews

The original usenet archive/search system. It has been around for several years now and has evolved into a consumer buying portal. It is probably one of the largest full-text search system on the Internet covering 35,000 usenet newsgroups and 18,000 discussion forums. It only covers newgroups, and not mailing lists, so is not a direct competitor. However many mailing lists are exported to newsgroups, so the messages can still be read via DejaNews.

9.5.5 SourceForge

From the SourceForge website:

> SourceForge's mission is to enrich the Open Source community by providing a centralized place for Open Source Developers to control and manage Open Source Software Development.

Again, Source Forge is not a direct competitor as it does not offer mailing list archives of the existing lists. However it does allow a group of people to easily setup the required development tools and communication structure to work on a new project. It offers hosting of web pages, mailing lists, development feedback and defect management.

The reason they are so interesting is that Tim Perdue, Technical lead of SourceForge is the creator of GeoCrawler. The two compliment each other very well and a merger would make sense for the two sites. Should this happen it would make GeoCrawler/SourceForge a much more popular site as it would offer one-stop access to a wide range of development resources.

9.5.6 Strengths and Weaknesses

The strengths and weaknesses of OSDigger and competitors are summarized below.

9.5.7 Critical Success Factors

In order for OSDigger to be a success it has to be better than its competitors. None of the sites charge any sort of entry fee or subscription fee, so we cannot be cheaper than them. The main factor is the search engine. The only other site with a custom built search engine is DejaNews, and it only works with newsgroups.

The OSDigger search engine has been designed from the ground up to search in mailing lists, and is designed to allow rapid updates of the indexes – much faster than other search products.

Another critical factor is giving the customers what they want. We intend to hold regular polls and/or surveys to make sure we are delivering what they want in terms of an archive site. OSDigger will be customizable so, for instance, people on slow links can load a site without images.

Site	**Strengths**	**Weaknesses**
OSDigger	*Custom designed search engine. Exclusively targeting the Open Source community.*	*Yet unknown, will take a while for the site to become known.*
GeoCrawler	Existing Site. Large database of messages. Links with linux.com and SourceForge.	Cannot search across multiple lists.
MARC	Existing site.	Not much time devoted to it. Restricted search capabilities.
eGroups	Large commercial site. Many extra features.	Awkward interface. Not exclusively Open Source.
DejaNews	Extremely large, and well known.	Only archives newsgroups, not mailing lists. Moving their attention more towards consumer buying portal.
SourceForge	Complete developers portal. Many extra features. Could merge with GeoCrawler.	Does not yet archive mailing lists.

Table 9.1: Competitors Strengths and Weaknesses

Chapter 10

Competetive Business Strategy

10.1 Pricing

The site will be free to access. As with most web sites on the Internet, it is very difficult to charge people for access. Most people will not use the service if they have to directly pay to enter it. It is time consuming and a hassle. Once one user has a username/password to allow them access to the site, then there is nothing to stop them redistributing the credentials to others, so that they may use their account.

10.2 Advertising Revenue

Traditionally most web sites generate revenue through advertising or some kind of sponsorship. The adverts usually take the shape of a 'banner' advertisement along the top of the web page as shown in Figure 10.1.

Advertisers traditionally pay between US$10 - US$30 per thousand visitors who see the banner. The most common way to arrange this is through a third party broker. These brokers are in constant contact with companies wishing to advertise and publishers of web sites and act on behalf of the companies to fill the advertising requirements. This benefits both parties as the advertisers have a large array of web sites that they can display the adverts on so that they get a wide audience, and that the advertisments are targeted at the most relevant viewers. The publishers then have to just deal with one organization to sell their advertising space.

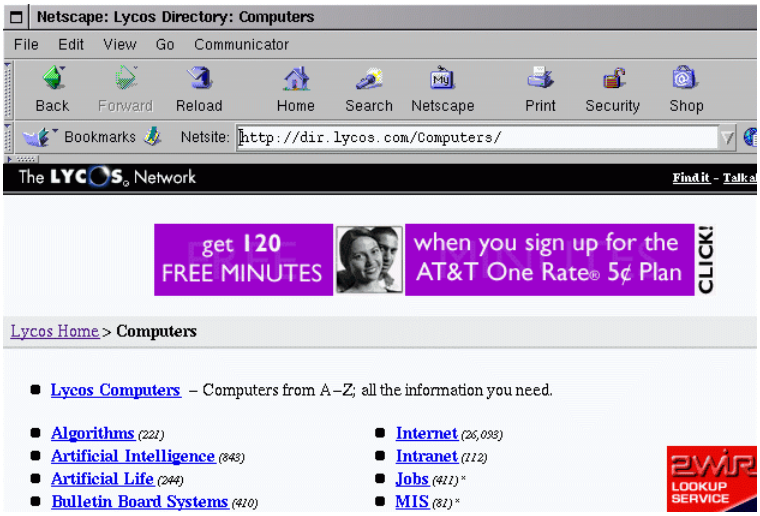


Figure 10.1: Banner Adverts

The advertising brokers also keep track of demographics of the sites in their stable so that they can target specific catagories of adverts at specific sites. They can also give statistics back to the advertisers in terms of number of times the advertisment has been seen, at what times, and by whom.

10.2.1 Advertising Rates

Two of the largest Online Advertising management companies are DoubleClick [13] and Adforce [1]. According to DoubleClick UK's web site, they charge GBP 28 - 34 per thousand impressions (visitors that view the advert) for Technology and Internet sites depending on the site. Large services such as DoubleClick require a site to guarantee at least a million impressions per month, however smaller contracts are available through other smaller brokers.

10.2.2 In-House Advertising

Although it is often simpler to outsource the selling of advertising space, there are benefits of selling directly to advertisers. The obvious benefit is there is no 'middle man' taking a cut of the revenue. However in our case there is a much bigger benefit: since we are a search engine we can tailor the adverts to the search terms. For example, if a user typed in the query 'linux games' we could return a more appropriate advertisement with the results. The user is more likely to be interested in an advert for a games reseller

than, say, for a company selling programming books. This means the user is more likely to click on the advertisement and look at the advertisers web site. This kind of targeting is worth more to the advertiser and hence can be sold at a higher price.

10.2.3 Other Revenue

Other sources of revenue have also been investigated. The main other viable revenue stream is via licensing out the back end of the search system to affiliate sites. Say, for instance, a Value Added Reseller of some Open Source software wanted to provide a branded version of our archive/search site as part of their web site. They could integrate some code into their web site that would allow visitors to their site to browse and search the archives of their product mailing lists. The actual mailing lists would be held on our servers and that is where the searches would be performed, but the user would still appear to be at the VARs site. The VAR would pay us a licensing fee dependant on the number of searches made per month. This is how search engines such as Google are funded [26].

Chapter 11

Forcasts and Financial Data

11.1 Sales Forcasts

The actual entity being sold in this business is the advertising space on the web site. The value of this space is directly proportional to the number of visitors to the site that view the advertisments. Hence this section concentrates on *page views*, that is the number of pages of the site viewed by the visitors.

To get an idea of the number of people we can expect to visit the site we contacted the maintainers of several other internet sites to ask them for their visitor figures. The sites contacted were all sites that we believe that our target audience may also visit, this was to get an idea of the popularity of the sites in this sector.

The first two people contacted were the maintainers of the FreeBSD and Debian Linux web sites. Both of these web sites have mailing list archives that can be browsed and searched (with limited capability).

Site	Search	Search + Browse
FreeBSD	51,000	255,000
Debian Linux	24,000	-

Table 11.1: Page views per month

In the case of FreeBSD we found that 51,000 page views per month purely for the search page. However, including browsing the messages and viewing the messages returned by the search results, results in 255,000 page views per month. This means that for each for each visitor they view on average 5 pages. If there is one advert per page, this means that there will be 5

impressions of an advertisment per visitor.

If these figures are extrapolated using the percentages found in Section 9.4.1, which show BSD to be about 10% of the Open Source sector, that results in 2.5 million page views per month. In fact BSD is a term describing three projects FreeBSD, NetBSD and OpenBSD, and we have only used the numbers for FreeBSD as a base figure. FreeBSD has the largest share of the three BSDs but this means that the potential market may be higher.

We are aiming to reach the growth phase by 6 months and occupy a 10% market share by the end of the first year. This means attracting 250,000 page views a month. We feel this is an appropriate target based upon the positive reaction from the survey we carried out, detailed in Section 9.3.

The reachability of this target is strengthened by the statistics from a new news site, *Daily Daemon News*, relating specifically to BSD news. The site has been running for six months and, on contacting the author, we found that the site has grown from an initial 75,000 to 200,000 page views per months in that time.

Part IV

Implementation

Chapter 12

Related Work

Information Retrieval (IR) is a well established research area within Computer Science. There are several regular conferences and working groups on IR topics: ACM SIGIR, TREC, etc.

IR combines the needs for better algorithms for searching, storing, and displaying data, with aspects of high performance computing and large data storage.

12.1 Digital Libraries

With the growing increase of information globally, the need for better ways of managing the data are evolving. Digital Libraries are emerging which are being investigated to see if they can replace their conventional counterparts. One of the main aspects of digital libraries is the loading of information into the digital library, most of the data is still in paper format, which has to be converted to digital forms. Even data already in digital format may have to be converted into a format suitable for storage and retrieval.

As said before the amount of data to be stored is ever increasing, in the order of Gigabytes or even Terabytes are not uncommon. Storing and searching through this information is a major task.

The New Zealand Digital Library [37] has been a good source of much of the research into the different aspects of IR.

12.2 Information Storage

There are several methods for storing the information, from flat files stored on file systems to large complex RDBMS. As well as storing the actual data, ancillary data, such as indexes, also have to be stored. The main goal is to allow efficient access to this data. Although computers are getting faster and faster, disk accesses still take several milliseconds. Hence, one of the main aims is to reduce the number of disk accesses to retrieve a document.

12.3 Indexing Methods

The main way of retrieving a document is to specify some sort of search criteria, most often words that appear in the document. With text collections of this size, linear searches are not yet possible without specialist hardware. Generally some sort of index is needed to narrow the search space. These indexes can take several forms:

12.3.1 Inverted Indexes

One of the most popular ways of constructing and index is a structure called a an *inverted index*. This is a word-oriented mechanism for indexing a text collection in order to speed up the searching task. A set of *inverted lists*, one for each distinct word in the text collection, is constructed. It closely resembles the index in the back of a book, that lists page numbers for each word occurrence.

The index is composed of the *vocabulary* and the *occurrences*. The vocabulary or lexicon is a list of all distinct words in the text. For each word in the vocabulary, a list of document ids, called the *occurrences* is also stored.

Along with each occurrence, the location of the word in the document may also be stored. This aids proximity queries in which the word location is important. The *granularity* of the index can vary and each occurrence may indicate a paragraph, word, or character position.

The space required for the vocabulary storage is a small fraction of the size of the text. According to Heaps Law [27], it grows as $O(n^{\beta})$ where β is a constant between 0.4 and 0.6 in practice, depending on the type of text being stored.

The space required for the occurrences is traditionally $O(n)$, and in practice is 30 - 40% of the size of the text collection. However Witten et Al [29] suggest a compression method that can reduce the size of the occurrences to

just a few percent of the size of the text collection. This is discussed further is Section 12.6.1.

Index Construction

Building an inverted index is generally $O(n)$ in complexity, and is a fairly straight forward task. The documents are parsed, and each word occurrence is stored in a tree or hash table. Each word is searched for in the tree, if it is found the document number is added to the end of the word list. If the word is not found a new node is added to the tree or hash table, and a new word list started.

Once all the documents have been processed, the index is written to disk. Each word list is written sequentially to disk, in practise the vocabulary and occurrences are written to separate files. The vocabulary file is normally small enough to be stored in main memory during the query process, which speeds up searches. Each word in the vocabulary has a pointer with it that points the the location of the word list in the occurrences file.

There are two main problems with this approach – index size and ease of updates. If the document collection is large, then the entire index may not fit in main memory during construction. This can be solved by using a paging mechanism, or compressing the word lists during construction. The text collection may be partitioned into smaller sections, each of which is processed in turn. The resulting indexes are then merged together. The resultant occurrences file cannot be easily updated as it requires inserting extra word occurrences into the middle of the file. This is why most public domain indexers require the entire collection to be re-indexed after changes to the text collection. This may be addressed by using a storage manager of some sort to allow paged updates.

Searches

To search the collection, the search terms are first looked up in the vocabulary. The word list for each search term is then read from the occurrences file. The document ids are read from each word list and are combined in some way (logical AND, OR, NOT, or some ranked method).

12.3.2 Signatures

Signatures are another word oriented approach to indexing based upon hashing. It imposes a smaller, 10 - 20%, overhead than inverted files, but requires

linear searching. However the data to be searched in smaller, and the searching not as complex (bitwise comparisons, rather than string comparisons).

A hash function creates a bitmap or *signature* of each document (or text block within a document). The function maps each word to a bit mask of B bits. The signature for the document, is the logical OR of the bit masks for each word occurring in the document.

Although signatures only return a binary answer to whether a word appears in a document, Croft and Savino propose a ranking method for signatures in [11].

Searching

If a word is present in a document, then the bits set in the words bit mask, will also be set in the documents signature. To find a document containing a set of search terms, a search signature is constructed by OR'ing all of the search terms. This search signature is then AND'ed with each document signature in turn. The down side of this, is that the matches may result in false positives. This is because the hash function will map more than one word to each possible combination of bits. Because of this each match much be verified by a brute force search method to ensure it is actually a proper match.

Index Construction

The construction of the signatures is very simple, for each document, the bit masks for each word in the document are OR'ed together. Documents can be easily added as signatures are simply appended to the end of the existing ones. They could even be stored in an RDBMS as a field along with the text.

This method is actually used in several RDBMS to allow the rapid searching of large text fields.

12.3.3 Suffix Tries

Inverted indexes and signature files assume the text is composed of words. This is not always the case, such as in searching through strings of genetic sequences.

A suffix tree assumes the text is one large string. Each position in the text is considered a suffix, that is a string that stretches from that position to the end of the text. The start of each suffix is lexographically different. Thus,

each suffix is then identified by its position. Not all text positions need to be indexed, just specific *index points*, such as word boundaries. Positions other than these index points cannot be retrieved (just as in an inverted index, the middle of a word may not be retrieved).

Each suffix is stored in a tree with pointers to the suffixes stored at the leaf nodes. To conserve space the tree is converted into a Patricia Tree, in which unary paths are compressed into a single node.

The advantage of this method is that phrases are very easy to match, the disadvantage is the storage requirements are high – 120 - 240% of the text collection size.

12.3.4 Sequential Searching

Although sequential searching is too slow to cover the entire text, it may be used in part, once the search space has been narrowed down by other means. The details of these methods are beyond the scope of this report, but include: Brute Force, Knuth-Morris-Pratt, Boyer-Moore, Shift-Or, and Suffix Automation.

12.4 Compression

In dealing with such large amounts of data, compression can save large amounts of storage space. Not only that, but can often increase performance too. Processors and memories are getting faster and faster, whilst secondary storage is still comparatively slow. Hence time saved on reading data from disk can be greater than time taken to decompress it.

12.4.1 Text Compression

There are a variety of means of compressing text which can result in 30 - 70% space savings. Generally however:

> The better the compression, the slower the program runs or the more memory it uses.

Again the details of the compression methods are out of the scope of this report, but further details can be found in [59, 6, 12, 23, 35] and a summary of routines can be found in [29].

One important feature is the synchronisation of the compression methods. Most compression schemes depend upon previous data, and hence reading cannot start midway through the compression text. Synchronisation points need to be added into the middle of the compression scheme, this is where the coding is reset to a know state at certain points. An alternative is self-synchronising codes, in which the decoder will automatically come into synchronisation after a while.

Additionally, some interesting work has been done in the above papers on pattern matching in compressed text. The idea being that a search pattern can be compressed and then matched against the compressed text collection. This avoids the need to decompress the text first, and also results in less data to actually try the pattern against.

12.4.2 Index Compression

The indexes used to speed up searching can take up a significant proportion of the overall storage. By compressing the indexes, the storage requirements can be reduced, in some cases to the extent that most of the index can be cached in main memory. This can prove an enormous performance gain.

The occurrences in an inverted list are stored in order of document id. This means that the ids may be represented by the intervals between successive ids. These intervals can be stored using fewer bits, hence less storage. An upside of this is, the more frequently a word appears, the smaller the intervals in the word list, and hence the few bits needed to represent each occurrence.

It is common practice to remove all *stopwords* from the text before indexing. These words are frequently occurring words that do not contribute much to the value of the index. Articles, prepositions, and conjunctions are natural candidates for stopwords. By omitting these words the index can be reduced by up to 40%. In [16], a list of 425 stopwords is presented.

Since these words occur so frequently, their wordlists may be compressed very effectively. For example on TREC, the inverted list for the 33 most common words are stored using just one bit per occurrence, resulting in a maximum of 741,856 bits long – about 91KB. The saving generated by stopping all 33 words is less than 3MB, which is less than 3% of the total 93MB size of the compressed index.

This shows the benefit of eliminating the stopwords before indexing when using a suitable compression scheme is minimal.

The workings of the compression schemes are shown in Section 12.6.1.

12.5 Storage

12.5.1 Index Storage

The method of storage of indexes on disk is a major factor in the performance of an indexing system. Using inverted files it is generally not possible to efficiently update the index and insert more data into the middle of the index. Several papers have been written on how to overcome this problem by using a persistent object store. This is some method of ensuring the permanence of objects in memory. Brown et Al [9] describe the way they took an existing information retrieval system, INQUERY, and replaced its custom data management system with a persistent object store, Mneme [34]. Their positive results showed that this may be an area worth looking into in greater depth in the future.

12.5.2 Updating Indexes

Most IR systems do not support the addition of new documents to the indexes due to the problems outlined above. However modern applications like OSDigger or News filters work in a dynamic environment and require frequent updates of the indexes. Tomasic [55] describes a method of overcoming this problem by using a dual-structure index. Words that are very common are stored in a different structure to those that are not so common. This allows different policies to be employed as to how the indexes are updated. Using persistent object stores as described above, Brown et Al [8] further show that using persistent object stores abstracts the need for special treatment of updating indexes. Their experimental results show much improved performance over traditional methods of updating indexes.

12.5.3 Distributed Searches

Research has been done into using networks of computers to run in parallel for Information Retrieval. The extra power and memory of using multiple computers can speed up both the indexing process and also the query process. The main area of research is how to efficiently break the task up into subtasks to be executed in parallel. The methods used in the OSDigger system increase the efficiency of the indexing and searching to a point where the tasks can be run on a single computer. Stanfill, Thau and Waltz describe an algorithm for constructing indexes in parallel in [53]. Lu and McKinley describe a method of searching a terabyte of text using partial replication on a large parallel computer in [64]. Large web search engines such as AltaVista and Infoseek handle their enormous loads by using large farms of

machines in parallel. The overhead of communication and synchronisation to get the whole system to function may actually degrade performance as described by Lu, McKinley and Cahoon in further work [62].

12.6 Algorithms

12.6.1 Compression Schemes

The inverted file compression plays a large part of this project and is what enables the search system to perform so well on modest hardware. Firstly, an inverted index is composed of an inverted list for each word that occurs in the lexicon. Suppose a term t appears in five documents, those numbered 3,5,20,21,76. This term is represented in the index by a list:

$$\langle 5; 3, 5, 20, 21, 76 \rangle$$

The offset of this list in the inverted file is often stored in the lexicon, so that the list can be found without searching. In general the list is of the form:

$$\langle f_t; d_1, d_2, d_3, ..., d_{f_t} \rangle$$

Since the list of document numbers is in ascending order the document numbers could be represented by the intervals, or *d-gaps*, between each successive document number. Hence the previous list becomes:

$$\langle 5; 3, 2, 15, 1, 50 \rangle$$

The original document numbers can be re-constructed by calculating the cumulative sums of the d-gaps.

Although the largest d-gap is still of the same magnitude as the original list, the distribution of numbers has changed. There will be far more smaller numbers than before. This is especially so for frequent words which will have more entries, each of a smaller d-gap.

Specific models have been developed to take advantage of this shift in probability distribution. They can be roughly grouped into two classes: *global* methods, in which each list is encoded with the same parameters, and *local* methods which use different encoding parameters for each list, depending upon the values being coded. A list of some of these models is shown in

Method	Reference
Global Methods	
Nonparameterized	
Unary	
Binary	
γ	[14, 4]
δ	[14, 4]
Parameterised	
Bernoulli	[22, 24]
Observed frequency	
Local Methods	
Bernoulli	[59, 6]
Skewed Bernoulli	[54, 32]
Hyperbolic	[52]
Observed frequency	
Batched frequency	[32]
Interpolative	

Table 12.1: Some methods for compressing inverted files

Table 12.1. In general local methods will always outperform global methods, and they are not any more computationally expensive, but maybe slight more complicated to implement. Of the local methods, the most suitable for general text is the Local Bernoulli Method, using Golomb coding. As this is the actual coding method used in the final project, it will be explained here.

Golomb Coding

This method was first described by Solomon Golomb in 1966 [24]. For some parameter b any number $x > 0$ is coded in two parts: first, $q + 1$ in unary, where the quotient $q = \left\lfloor \frac{(x-1)}{b} \right\rfloor$. The remainder $r = x - qb - 1$ is coded in binary, requiring $\log b$ bits.

For example, with $b = 3$ there are three possible remainders $r = 0...2$, coded 0, 10 and 11, respectively. Similarly for $b = 6$ there are six possible remainders $r = 0...5$, coded 0, 01, 100, 101, 110, and 111, respectively. If the value $x = 9$ is to be coded with $b = 3$, calculation gives: $q = 2$ and $r = 2$ because $9 - 1 = 2 \times 3 + 2$. Thus, the encoding is 110 followed by 11. If coded with $b = 3$, the values calculated are $q = 1$ and $r = 2$, resulting in the encoding 10 followed by 100. Other codings for small values of x can be seen in Table 12.2.

Gap x	Coding Method				
	Unary	γ	δ	Golomb	
				$b=3$	$b=6$
1	0	0	0	00	000
2	10	100	1000	010	001
3	110	101	1001	011	0100
4	1110	11000	10100	100	0101
5	11110	11001	10101	1010	0110
6	111110	11010	10110	1011	0111
7	1111110	11011	10111	1100	1000
8	11111110	1110000	11000000	11010	1001
9	111111110	1110001	11000001	11011	10100
10	1111111110	1110010	11000010	11100	10101

Table 12.2: Example codes for integers

Gallager and Van Voorhis [22] showed that if b is chosen to satisfy

$$(1-p)^b + (1-p)^{b+1} \leq 1 \leq (1-p)^{b-1} + (1-p)^b$$

this coding generates an optimal prefix-free code for the geometric distribution corresponding to Bernoulli trials with the probability of success given by p.

Witten et al [29] show that assuming, $p = \frac{f}{N \times n} \ll 1$, a useful simplification is

$$b^A \approx \frac{\log_e 2}{p} \approx 0.69 \times \frac{N \times n}{f}$$

Where N is the total number of documents, n is the number of distinct terms, and f is the number of index pointers.

Local Bernoulli Model

The value of b described above is calculated from the statistics of the entire text collection. However some words are more common than others, in which case their word lists will contain many more terms in them with much smaller d-gaps.

If the frequency, f_t, of term t, is known, then a Bernoulli model on each individual list can be used. Using this method, frequent words are encoded

with a smaller value of b than infrequent words. Very common words are encoded with $b = 1$ which causes the code to degenerate to a set of unary codes for the gap sizes with no binary components. This is equivalent to storing the word list as a *bitvector*, which is a binary vector in which each document is represented by a bit, the bit being set if the word appears in that document.

For frequent words, this can result in a word list compressed to just 3-4% of its original size, compared to storing each document number as an uncompressed 32-bit integer.

12.6.2 Document Ranking

Once the index has been constructed and a query is issued to be tried on the index, the outcome is a set of documents that somehow match the query terms. In order to determine what is matched and what is not (and to what degree it matches) a similarity measure is needed.

Boolean Queries

These queries are the simplest to implement. Either a term appears in a document or it does not. A term is constructed of a set of queries terms with some logical operators eg. FreeBSD *and* scsi, linux *or* FreeBSD, linux *not* debian. These terms are matched against the index and matching documents are returned in no particular order.

Ranked Queries

These queries are more complicated, they order the resultant set of documents in some order with the most likely candidates at the top of the results. This is intuitively how mast users expect a search system to work. However, how does the search system determine how *relevant* a document is? This has been studied in great depth and a number of methods have emerged. The details of each of these methods is beyond the scope of this report, however some of them are listed with references in Table 12.3

Each of these methods has its good and bad points, and as always a trade-off between complexity, processing speed, and accuracy.

Model	Reference
Classic Models	
Boolean Model	[29, 2]
Vector Model	[49, 51]
Probabilistic Model	[48, 19]
Alternative Set Theoretic Models	
Fuzzy Set Model	[43]
Extended Boolean Model	[50]
Alternative Algebraic Models	
Generalised Vector Space Model	[60]
Latent Semantic Indexing Model	[21, 28]
Neural Network Model	[58]
Alternative Probabilistic Models	
Bayesian Networks	[40, 20]
Inference Network Model	[57, 56, 63]
Belief Network Model	[47]

Table 12.3: Ranking Models

Document and Term Weights

When trying to rank documents one of the main problems is trying to work out how relevant a document is to a query. The main intuitive way of thinking of this, is that a document that contains more occurrence of the query term is more important than one that contains less. However, what happens when a query consists of several terms? Should each term have the same *weight* in the query? If the query *java database* was issued, should a document that mentions *java* lots of times, but *database* only once, be ranked more or less relevant than one than mentions *database* many times and *software* only once? What is needed is some way of determining the usefulness or weight of each term in a document (or query).

George Zipf noted in his book *Human Behaviour and the Principle of Least Effort* [65] that the frequency of an item tends to be inversely proportional to its rank. That is, the weight w_t of a term t might be calculated as:

$$w_t = \frac{1}{f_t}$$

where f_t is the frequency of the term in the text collection.

This can be combined with a *relative term frequency*, denoted $r_{d,t}$ to give the document-term weight $w_{d,t}$ and/or the query-term weight $w_{q,t}$.

Other methods of calculating w_t are mentioned in [29] however the most usual is

$$w_t = \log_e \left(1 + \frac{N}{f_t}\right)$$

where N is the number of documents in the collection. The logarithm is added to prevent a term with $f_t = 1$ from being regarded as twice as important as $f_t = 2$.

The relative term frequency factor, $r_{d,t}$ can also be calculated in several ways, the most usual being

$$r_{d,t} = 1 + \log_e f_{d,t}$$

The logarithm here prevents the number of occurrences of the term in a document from contributing too much as the number of occurrences increase.

The document vectors can then be calculated as:

$$w_{d,t} = r_{d,t} \cdot w_t$$

These style of weightings are loosely known as TD × IDF, term frequency times inverse document frequency, and play an important part of search ranking.

Cosine Measure

The ranking system used in OSDigger is a vector space model known as the Cosine Measure.

Each document is represented as an vector in n-dimensional space in which each dimension represents a different term in our lexicon. If we also represent the query terms in this fashion, as a vector, then the similarity between the two is proportional to the similarity in direction of the two vectors.

This is a well understood concept in vector algebra – it is the angle between the two vectors. Simple algebra yields:

$$X \cdot Y = |X||Y| \cos\theta$$

where $X \cdot Y$ is the vector inner product and

$$|X| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

is the Euclidean length of X. The angle θ can be calculated from

$$\cos\theta = \frac{X \cdot Y}{|X||Y|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$$

This makes sense as by dividing by the Euclidean length of the vectors we normalise their lengths. This removes the effect of the length of the document, in that long documents are not favoured over shorter ones.

If the documents are each vectors in the positive region of an n-dimensional space, then the query term can be imagined as a ray that emanates from the origin in a particular direction. The document vectors that lay closest to the this ray in an angular sense are the most relevant.

This leads to the *cosine rule* for ranking

$$\begin{aligned} \text{cosine}(Q, D_d) &= \frac{Q \cdot D_d}{|Q| \cdot |D_d|} \\ &= \frac{1}{W_q W_d} \sum_{t=1}^{n} w_{q,t} \cdot w_{d,t} \end{aligned}$$

where

$$W_d = \sqrt{\sum_{t=1}^{n} w_{d,t}^2}$$

is the Euclidean length – ie. the *weight* – of document d and

$$W_q = \sqrt{\sum_{t=1}^{n} w_{q,t}^2}$$

is the weight of the query.

Chapter 13

Technical Basis

13.1 OSDigger Implementation

As seen in the previous chapter there are lots of possible methods of indexing and searching, below is presented the technologies and algorithms that OSDigger uses and why.

13.1.1 Storage

The storage of the messages themselves and the storage of the indexes are two different matters. Although it is possible that they could both be stored in the same manner, this would not be optimal for either of them.

The messages are stored in MySQL, an Open Source RDBMS [1]. This was chosen as it allows very fast read access to the data, and allows storage of large objects. MySQL does not support server-side procedures or fine-grained locking as Oracle or PostgreSQL [42] do, but these features are not that important for this application. MySQL does support user defined functions (UDFs) that can be written in C and could be used to perform task such as on-the-fly compression/decompression or parsing of a field. An RDBMS was used so that various headers in the messages could be stored as separate fields in the database and searched on in a relational way. eg. Find all messages last year from Jordan Hubbard on the FreeBSD Questions list.

Several methods were investigated for storing the indexes. Initially for the prototype the indexes were stored as a binary file to disk in some manner. The indexer and search programs knew how to handle the files internally.

[1] Relational Database Management System

One of the main problems with this approach was managing concurrent access, and how to find a particular inverted list in the inverted file. Initially when the inverted file was written to disk, a second file – an index was also written. This listed the offsets of the word lists for each word in the inverted file. This still did not address the concurrency issue.

Several persistent object stores were looked at: Generic Object Oriented Database System (GOODS) and Persistent Object Storage for C++ (POST++) [25] , both written by Konstantin Knizhnik of the DEC Moscow Software Center, however both required either C++ or Java knowledge which I did not have at the time of reviewing them. If more time was available I would like to have investigated these further as previous research into using Persistent Object Stores has been optimistic.

In the end the BerkeleyDB [38, 5] embedded database was used. This is a key/value pair database, that provides exceptional speed, and is simple to integrate. It has no arbitrary record limits, so scales to the size needed. The DB supports B+tree, Extended Linear Hashing, Fixed and variable-length records, all with the same interface. It also has caching, locking and transactional subsystems.

13.1.2 Word Stemming

Before words in the documents are indexed, the suffixes are removed, this is known as *stemming*. This is to reduce the number of different variations of a word. As an effect of this it normally reduces words to a single tense. eg. *run*, *runs*, *running* are all reduced to *run*.

There are several stemming algorithms around, the two most common are the Porter stemming algorithm [41] and the Iterative Lovins algorithm [31]. A comparison of these and other algorithms are are detailed in [15].

Due to its simplicity OSDigger uses Porters algorithm.

13.1.3 Index Compression

The indexes are compressed using the Local Bernoulli model with Golomb Coding, as described in Section 12.6.1. This scheme turned to to be extremely fast and the indexes used in the prototype were compressed to just over 4 bits per pointer.

13.1.4 Index Construction

There are two methods of dealing with updating indexes: Creating a dynamic index structure that can be expanded, or re-indexing the entire collection. The former traditionally takes too long to be done very often, and the latter requires complex data structures and better underlying storage management.

Profiling the prototype indexer highlighted an important fact. The majority of the time spent constructing the index, is actually spent parsing the documents and stemming the words. Hence if the index was going to be rebuilt from scratch each time it would take quite a while. To solve this an intermediate *forward* index is needed. This index is not inverted and so can be appended to when new documents arrive. The inversion of this index is very efficient as there is no parsing or stemming to do.

> This gives us a unique hybrid approach to updating the indexes. The forward index is incrementally updated; and the inverted index can be re-created very often as it can be constructed quickly from the forward index.

The indexes are constructed in two phases. First the documents are extracted from the MySQL database and parsed. Each word is looked up in a lexicon (stored in a BerkeleyDB), and a new wordid is assigned if the word does not yet exist in the lexicon. A forward index is constructed which details words are in a document. This is stored compressed. Each list is then stored in a BerkeleyDB with the document number as a key. Note: this has not yet been inverted. It is just a more efficient representation of what is in the messages.

Key	*Value*		
Docnum (4)	Numwords (4)	d (4)	wordids (variable)
Docnum (4)	Numwords (4)	d (4)	wordids (variable)
Docnum (4)	Numwords (4)	d (4)	wordids (variable)
Docnum (4)	Numwords (4)	d (4)	wordids (variable)

Figure 13.1: Format of Forward Index

The format of the forward index can be seen in Figure 13.1. The numbers in parenthesis are the length of each field in bytes. D is the local Bernoulli parameter used to encode and decode that list, *Numwords* is the number of words encoded in that list.

The forward index is actually partitioned into multiple indexes. Each index contains a subset of the total terms in the lexicon. A hash value is used to calculate which word goes in which index. This results in several entries for each document, each in a different index. This adds a slight bit to the overhead, but means that then terms are now partially ordered. This concept was taken from the method used for the indexes of Google [7], a large web search engine.

An inverted index is create from each forward index. This results in each inverted index containing complete lists for for a subset of the words. The reason for doing this, is that each forward index can be processed independently, requiring less memory. The inverted indexes are a similar format to the forward indexes and is shown in Figure 13.2. The inverted indexes are again stored in a BerkeleyDB database with the wordid as the key.

Key	*Value*		
Wordid (4)	Numdocs (4)	d (4)	docnums (variable)
Wordid (4)	Numdocs (4)	d (4)	docnums (variable)
Wordid (4)	Numdocs (4)	d (4)	docnums (variable)
Wordid (4)	Numdocs (4)	d (4)	docnums (variable)

Figure 13.2: Format of Inverted Index

13.1.5 Searches

Two searching methods are supported by OSDigger:

Onestep Searches

The Onestep search works like most other search engines in that the user enters a set of search terms and a list of documents are returned.

Twostep Searches

The Twostep search tries to overcome the main problem of searching – the user. Often a user does not exactly know what they are searching for and prefer to enter in a couple of terms and then look at the results. They then refine their query, adding terms the saw from the first search and try again. Research shows that he average user enters only 2 query terms on average. There is no way a search engine can determine exactly what a user is looking for with just 2 terms. eg. a child entering in *dinosour food* will

most likely be looking for something entirely different to a paleontologist entering in *dinosaur food*. The only way a search engine can return the correct results is to somehow get more information from the user as to what they are searching for. Often this is done by presenting the user with a set of sample documents and asking them to indicate which are most like what they are looking for. The search process is the re-iterated.

OSDigger supports another method. With the Twostep search a user enters in their query as normal, but instead of presented with the results, they are presented with a list of additional query terms related to what they first entered. This is to encourage the user to enter in more terms. Even if none of the terms presented are exactly what they had in mind, they may jog the user into thinking more about specific terms to aid the query.

The process works by looking into the forward indexes created by the indexer, and making note of all the words that have appear in the top 15 documents returned by the Onestep process. Each word is then looked up in the lexicon to find out the total number of times it occurs. Using this information a TD $\times$ IDF score can be computed for each word. Any word which appears often in the set of results relative to how often it appears in the entire text collection is considered a related word and is returned.

Results show this method in many cases to be quite effective. Results from the query *FreeBSD sound card* are shown in Table 13.1

where *Score* is how relevant a word is, *Wordlog* is the TD $\times$ IDF for that word, F is how often that word appears throughout the collection and f is how often the word appeared in the top 15 documents returned by the Onestep query.

As expected *sound* and *card* are near the top as they are the actual search terms. Other notable terms are *blaster* coming from Sound Blaster a make of sound card; *oss* the Open Sound System; *pcm0* and *snd0* are audio devices under unix; *x11amp* is an audio player application; *soundcard* and *soundblaster* are variations on above.

13.2 User Interface

One of the main goals of software engineers is to separate the logic and the presentation of a system. To this end we have decided to use Extensible Markup Language (XML) [44] to pass the data from the search server to the web server for presentation to the user. A java servlet running on the web server then applies an Extensible Style Sheet Transformation (XSLT) [45] to the XML data to convert it into the HTML page.

Score	Wordlog	F	f	word
22.70	6.61	1944	30	sound
17.69	5.64	4011	22	card
16.90	9.43	63	5	blaster
16.72	6.97	1424	10	3.1
15.40	7.91	558	6	pnp
15.17	8.46	291	5	oss
14.74	9.16	105	4	theme
14.58	6.33	2429	9	pci
14.54	7.47	875	6	dwhite
13.16	8.17	414	4	128
13.14	3.76	12917	32	freebsd
13.04	9.41	66	3	ess
12.68	9.14	108	3	soundcard
12.58	9.08	121	3	pcm0
12.51	9.03	131	3	snd0
12.40	5.64	4150	8	driver
12.37	5.37	4883	9	under
12.26	8.84	174	3	x11amp
12.19	8.79	188	3	maker
12.14	8.76	197	3	soundblaster

Table 13.1: Twostep results for *freebsd sound card*

This has the advantage of separating the logic and the presentation in such a way that a web designer can work on the layout of the web site, whilst a software engineer can work on the backend search server.

A very simple example of an XSLT transformation is shown below. Assume we had some XML document representing a business card:

```
<card type="simple">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 456-1414</phone>
</card>
```

We define the XHTML[2] rendering semantics for our business-card markup language using an XSLT stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform"
                xmlns="http://www.w3.org/1999/xhtml">
<xsl:output doctype-system=
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

  <xsl:template match="card[@type='simple']">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <title>business card</title><body>
        <xsl:apply-templates select="name"/>
        <xsl:apply-templates select="title"/>
        <xsl:apply-templates select="email"/>
        <xsl:apply-templates select="phone"/>
    </body></html>
  </xsl:template>

  <xsl:template match="card/name">
    <h1><xsl:value-of select="text()"/></h1>
  </xsl:template>

  <xsl:template match="email">
    <p>email: <a href="mailto:{text()}"><tt>
      <xsl:value-of select="text()"/>
    </tt></a></p>
  </xsl:template>
```

[2] A well-formed XML version of HTML

```
  ...
</xsl:stylesheet>
```

After combining the two we get the resulting XHTML document:

```
<!DOCTYPE html SYSTEM
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<title>business card</title>
<body><h1>John Doe</h1><h3><i>CEO, Widget Inc.</i></h3>
<p>email: <a href="mailto:john.doe@widget.com">
<tt>john.doe@widget.com</tt></a></p>
<p>phone: (202) 456-1414</p>
</body></html>
```

which a web browser might display like:

This also allows us to easily construct multiple front ends for the site. These could be different HTML styles, eg. a low bandwidth version with less graphics, or even another XML format, such as WML[3] for use on WAP[4] mobile phones and handheld PCs.

[3]Wireless Markup Language
[4]Wireless Application Protocol

Chapter 14

Specification and Design

This chapter details the final specifications for the components needed for OSDigger to function.

14.1 Overall Design

The overall design here has been expanded from the initial design to take into account the results from the research into related work and the prototype. One of the main changes has been the split of the indexer into two halves: the *indexer* and the *inverter*.

The flow of data through the system is shown in Figure 14.1. Note that the web server communicates directly with the Query server, and not the Database server. This is to reduce the number of communication paths in the system. Also note that all information passed from the Query server to the Web server is encoded in XML.

14.2 Database

MySQL will be used as the database to store the messages. The messages will be parsed by a perl script called from the mail server and inserted into the `messages` table of the database. The *To*, *from*, *subject*, *cc*, *date*, *message-id*, and *in-reply-to* or *references* headers are all stored as separate fields in the database. The table scheme is shown in Table 14.1.

The message bodies are to be compressed/decompressed by a User Defined Function (UDF) using the zlib compression library.

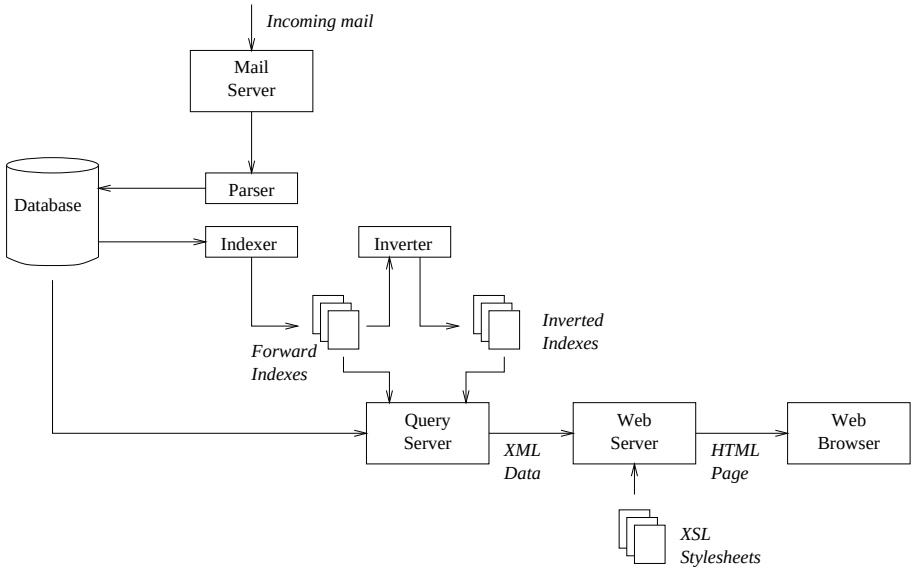


Figure 14.1: Data flow in OSDigger

Field	Type	Null	Key	Default	Extra
id	int(10) unsigned		PRI	0	auto_increment
hmessage_id	varchar(255)		MUL		
hrefs	varchar(255)		MUL		
hdate	datetime		MUL	0000-00-00	
hsubject	varchar(255)	YES		NULL	
hfrom	varchar(255)	YES		NULL	
hto	varchar(255)	YES		NULL	
hcc	varchar(255)	YES		NULL	
list	int(10) unsigned		MUL	0	
body	text	YES		NULL	

Table 14.1: `messages` table scheme

An additional table *lists*, Table 14.2 stores information about the lists themselves, such as the list number, subscription address, posting address, type.

Field	Type	Null	Key	Default	Extra
listnum	int(10) unsigned		PRI	0	
subaddr	varchar(255)				
addr	varchar(255)				
type	int(11)			0	

Table 14.2: `lists` table scheme

14.3 Indexer

The indexer is to be split into two parts. The *indexer* and the *inverter*. Both are to be written in C.

14.3.1 Indexer

The indexer is to extract the message bodies, address and subject fields from the database using the MySQL C API. The indexer should then parse the message text and stem all words using the Porter Stemming algorithm [41]. The words in each document are then looked up in the lexicon to find their wordids. Words not found in the lexicon are added. Each time a word is looked up a count of the number of occurrences of that word in the lexicon should be incremented.

The words are partitioned into 8 sets using the first 3 bits of their wordid. Each set is sorted into ascending order then stored as a compressed list using the Local Bernoulli method with Golomb coding as described in Section 12.6.1. The local parameter d is calculated as

$$0.69 \times \frac{\text{the total number of words in the lexicon}}{\text{the number of words in the list}}$$

and stored at the start of each list, along with the number of words in that list. The format of each record is shown in Figure 13.1.

Each time the indexer is run, it must append the new messages in the database to the end of the existing forward indexes.

14.3.2 Inverter

The inverted takes each one of the forward indexes in turn and inverts it into an inverted index. The inversion is to take place in memory. The inverted indexes are to be stored in BerkeleyDB databases with the wordid in the key. The number of documents, the local Bernoulli parameter d and the compressed list of document numbers are stored as the value. The format is shown in Figure 13.2.

The inverter should replace each existing entry in the database with the updated version whenever it is run.

14.4 Query Server

The Query server is to be written in C. It is a daemon process that listens on TCP port 3333. The daemon uses a simple protocol to communicate with the web server. The web server can issue one of three requests:

Command	*Description*
`onestep` *start num term* [*term* ...]	Perform a onestep search and return *num* results starting from *start* in XML
`twostep` *term* [*term* ...]	Perform a twostep search and return the results in XML
`retrieve` *docid*	Fetch a message and return it encoded in XML

The server computes the results for the queries passed to it.

14.4.1 Onestep Queries

To compute a onestep query, the server first stems the query terms and looks up the corresponding wordid from the lexicon. It then retrieves the compressed list of document numbers from the appropriate inverted index for the word. The lists are then decompressed and ranked together using the Cosine Ranking rule. The resulting scores are sorted and *num* results are returned starting at *start*.

14.4.2 Twostep Queries

To compute a twostep query, the server first proceeds as a onestep query and returns the top 15 documents. The documents are then looked up in

the forward indexes to find out which words occur in them. A tally is kept of each word and how often it appears. The final tally is then multiplied by the TF×IDF value for that word and the resultant scores sorted. The server then returns the top 20 words.

14.5 Web Site

The interaction between the user and the query server is via a Java Servlet. Hence the web server must support servlets. The servlet will be responsible for taking the input from the user and formatting it and sending the query to the query server. The format of the queries is described in Section 14.4.

The servlet will take the XML data returned by the query server and apply an XSLT stylesheet (see Section 13.2 to convert it to the desired output format (initially HTML, but other formats may follow). The servlets is also responsible for handling any customisation the user wants. This will be done by storing preferences in a MySQL database, keyed by an HTTP cookie, held by the users browser.

14.6 Parser

The parser will be written in Perl and will take messages passed to it from the mail server and parse them. Any non-text MIME parts will be discarded, uuencoded and Base64 text parts will be decoded and converted to the host character set.

The message will then be inserted into the database and compressed using a User Defined Function in the database. The headers of the message will be parsed and inserted as fields described in Section 14.2.

Chapter 15

Software Guide

15.1 Software Availability

The home page for the project is `http://beta.osdigger.com/osdigger/`.

Available from there is:

- The source code as a tarball
- CVSWeb access to the source code
- A Library of various Information Retrieval papers
- Various forms of this thesis (, DVI, PS, HTML)

15.2 Choice of Programming Languages

Programming languages each have their particular features that make them good for particular tasks. Hence in this project several languages were used for different parts, as shown in Table 15.1

Half way through the project, I decided to try and code the indexer in Java to see how it would compare. Although the speed was too slow for production use, the object oriented nature was well suited for the task. The may be worth looking into again at a later date when Java is a bit quicker.

Language	*Section*	*Reason*
C	Indexer	Speed
	Inverter	Speed
	Query Server	Speed
Java	Web scripts	Good XML/XSL support
Perl	Parser	Great at parsing text. Has existing MIME modules

Table 15.1: Programming Languages Used

15.3 Software Structure

The code is quite diverse with several sections in several languages. As this software is not intended at the moment to be distributed to users and installed on their machines, no effort has yet been made no neatly package up the code for easy installation.

Below is a breakdown of the sections and the files in those sections and what they do.

15.3.1 Indexer

The indexer is actually three parts, the indexer, the inverter and the query server, all written in C. A description of the source files is in Table 15.2.

Indexer

The indexer retrieves documents from the MySQL database and parses them using a zero-copy parser. The parsing and stemming was one of the main bottlenecks of the system to start with, hence quite a bit of work was put in to eliminating all of the memory allocation and copying routines. This gave the resultant parser a significant performance boost. The parsed and stemmed words are then looked up in the lexicon and stored in compressed for as a record in a BerkeleyDB database.

Inverter

The inverter is run on each of the forward indexes. It reads and decompresses the word list for each document and then constructs a hash table in memory of words. Each time a word appears in a document, that document number is added to that words document list. This can require quite a large amount

File	*Description*
indexer	
indexer.c	The main file for the indexer. It retrieves documents from the MySQL database and parses them
indexer.h	Header file for above
parse.c	Implements a zero-copy parser for the indexer
parse.h	Header file for above
processword.c	Processes each word, stemming it, looking it up in the lexicon and adding it to the forward index
processword.h	Header file for above
inverter	
inverter.c	Reads a forward index and inverts it in memory and writes an inverted index to disk
query server	
search.c	Main query server daemon. Listens for queries on 3333/tcp and processes them.
search.h	Header file for above
searchset.c	Implements a 'set' of documents for the query server
searchset.h	Header file for above
net.c	Implements the network function for the search server (from Comer & Stevens, TCP/IP Vol. III)
common	
inverted.c	Creates and inverted index object, has methods to index a word
inverted.h	Header file for above
forward.c	Creates and forward index object, has methods to index a word
forward.h	Header file for above
lexicon.c	Creates a Lexicon object, has methods to lookup and add words to the lexicon
lexicon.h	Header file for above
stem.c	Implements the Porter Stemming Algorithm (B. Frakes and C. Cox, 1986)
bitio*.h	Bitwise IO and compression routines. (N. Sharman and A. Moffat, 1994)

Table 15.2: Source files for the Indexer

of memory, hence why the forward indexes are split into eight. Each one can be processed separately.

The document lists are stored in compressed form in memory, which reduces the memory needs of the inverter considerably. Once the entire forward file has been processed the inverted file is written to disk as a BerkeleyDB file with each word as a separate record.

Query Server

The query server opens all of the inverted and forward indexes along with the forward and reverse lexicons at startup. It then listens on TCP port 3333 for queries. It is currently a single threaded server and processes requests from the java servlet sequentially.

When a query comes in the server stems each query term and looks its wordid up in the lexicon. It then retrieves the document list for each term from the inverted indexes. The first three bits of the wordid indicate which of the eight inverted indexes the word is in.

A searchset is then constructed of each search term. The sets are in order and hence a merge sort can be performed to find any document ids that appear in all of the word lists. When one is found it is added to an array of results, with the score of that result being computed by the accumulation of the TF×IDF scores for each word.

Once the smallest document list has been exhausted the merge stops and the results array is sorted by descending score using the system qsort function.

The document ids are then looked up in the MySQL database and the subject, date, list and author are returned. The results are formatted as XML and passed back down the network socket to the servlet.

15.3.2 Parser

The parser is contained in a single Perl script, *load.pl*. It is called by the mail server when each new message arrives. The script makes a connection to the MySQL database and uses the MIME::Head and MIME::Parser modules to parse the message passed on standard input. Various headers are extracted from the message and an SQL query constructed. The text parts of the message body are extracted and concatenated together.

The message is checked to ensure that it has a message id and a To or Cc header, if not it is discarded. The parser also checks to see if the message has an X-No-Archive header set. If so the message is also discarded.

The query is then executed to insert the message into the database. The User Defined Function `Pack()` is called on the message body to compress it before it is inserted.

If the query fails due to the database not accessible then the script fails with an error code of 75. This is picked up by the mail server during delivery and causes the delivery to fail temporarily. The mail server will queue the message and try again later.

Any error messages during parsing are directed away from the sender to an admin address such that a failed parse will not cause the message to bounce, and the address to be removed from the mailing list.

15.3.3 Java Servlets

There are three Java servlets currently implemented:

Servlet	*Description*
Query.class	Takes a query from an HTML form and returns the results
Browse.class	Takes a list name and displays the most current messages on that list
Message.class	Takes a message id and displays the corresponding message

The servlets open a socket to the query server and send their query to it. The query server then pass the results back formatted in XML. The servlets use the Xerces XML parser and the Xalan XSLT processor to transform it with a specified stylesheet. The resultant HTML it then passed back to the users web browser. Most of the work is done by the XSLT processor hence there is really not much to the servlets themselves.

15.4 Code Samples

These are sample sections taken from the code that demonstrate particular parts of the code that I am most proud of.

15.4.1 Parser

The following fragment is part of the zero-copy parser in `parse.c`. It was profiled quite a bit to reduce as many performance hindrances as possible. Procedures `tolower2()`, `isalnum2()` and `ispunct2()` are macros and inline functions to speed things up.

The parser works through a text string (a document) from the start, `s` until the end, `e`. A word may have the punctuation characters [_-.@] provided only one occurs in a row and that it is surrounded by two alphanumeric characters (line 28). This picks up email addresses and version numbers that may want preserving.

Each valid word is then passed to `processword()` to be processed (line 110).

```
70  void parse(unsigned int docnum, char *s, char *e) {
71    char *w;  // working pointer
72    int len;  // length of the word
73    int i;
74
75    // Create a new document entry in each forward index
76    for(i=0; i<8; i++)
77      f[i]->newdocnum(f[i], docnum);
78
79    w=s;          // set the pointer to the start
80    while(w<e) {  // while not at the end of the doc
81      // eat up non alphanumeric space
82      while(!isalnum2(*w) && w<e)
83        w++;
84      s=w;
85
86    start:
87      // move pointer along word until we reach the end
88      // or reach non-alphanumeric characters
89      while(isalnum2(*w) && w<e) {
90        *w = tolower2(*w);
91        w++;
92      }
93
94      // punctuation support
95      // if the character after the punctuation character
96      // is alphanumeric then continue parsing
97      if (ispunct2(*w) && w+1<e) {
98        if(isalnum2(*(w+1))) {
99          w++;
100         goto start;
101       }
102     }
103
104     *w = 0;       // mark the end of the string
105     len =  w - s; // calculate the length of the string
```

```
106
107       // Only accept words greater than 2 and less than 15 letters
108       if(len > 2 && len < 15) {
109         // process the word
110         processword(docnum, s);
111       }
112     }
113     // Close all of the forward indexes
114     for(i=0; i<8; i++)
115       f[i]->enddocnum(f[i]);
116   }
117
```

15.4.2 Query Server Ranking

The query server does a merge sort on all of the query terms in order to find document numbers that appear in the document lists for all of the terms. The sample below is taken from `search.c`.

First an array of search sets are created one for each term:

```
108     // Construct search sets
109     for(t=0; t<numterms; t++) {
110       Stem(terms[t]); // stem the term
111       // lookup the wordid in the lexicon
112       wordid = lex->getwordid(lex, terms[t]);
113       // Construct the set
114       set[t] = getwordlist(wordid);
115     }
```

The first document number is then fetched from each set and the count of number of results is reset:

```
117     // Get first docnum in each list
118     for(t=0; t<numterms; t++)
119       set[t]->getnext(set[t]);
120
121     numresults = 0;
```

The array of sets is then sorted so that the set with the lowest current document number is at position zero. The current document number of each set is then compared to the next set, if they do not match then the comparison ends and the `match` variable set to zero:

```
123     // Merge all of the results looking for matches
124     do {
125       int match = 1; // match by default
126
127       // Sort the sets
128       qsort(set, numterms, sizeof(struct searchset *), compar);
129
130       // Check if all the docnums match
131       for(t=0; t<numterms-1; t++) {
132         if(set[t]->docnum != set[t+1]->docnum) {
133           match = 0;
134           break;
135         }
136       }
```

If all of the document numbers are the same then all of the query terms appear in that document. The score for each word is calculated in line 145. The log of the number of times the word appears is used to prevent a document with too many occurrences getting scored too high. The document weight is returned in line 149, it is calculated as the number of words in the document

```
138       // If there is a match...
139       if(match) {
140         float score = 0;
141         unsigned int wd;
142
143         // Calculate the cumulative scores for the words
144         for(t=0; t<numterms; t++) {
145           score += log(1 + set[t]->count) * set[t]->wt;
146         }
147
148         // Get the document weight
149         wd = getwd(set[0]->docnum);
150         numresults++;
151
152         // Allocate some more memory for the new result
153         *results = realloc(*results, numresults * sizeof(struct score));
154         if(results == NULL) {
155           fprintf(stderr, "Malloc error\n");
156           exit(1);
157         }
158
```

```
159         // Insert the result into the array
160         (*results + numresults - 1)->docnum = set[0]->docnum;
161         (*results + numresults - 1)->score = score/wd*100;
162       }
```

This process loops, retrieving the next document number from the list with the lowest current document number, `set[0]`, or until the end is reached on a list – returns -1.

```
164       // increment the lowest list
165     } while (set[0]->getnext(set[0]) != -1);
```

The final results are then sorted by score, and the number of results is returned.

```
167     // Sort the final scores
168     qsort(*results, numresults, sizeof(struct score), rescompar);
169
170     // Return the number of results
171     return (numresults);
172   }
```

15.5 Third Party Software Components

Several existing pieces of software were incorporated into OSDigger. This saved quite a bit of development time. Below is a *brief* synopsis of the software, the author and the license.

15.5.1 Compression routines

Author: *Neil Sharman and Alistair Moffat*
License: *GNU General Public License*

These routines are C macros used in the indexer, inverter and query server. They allow writing and reading of integers coded in various forms: unary, binary, delta, gamma, bernoulli. They were taken from the software mg-1.3x (Managing Gigabytes [29]).

15.5.2 MIME-Tools

Author: *Eryq*
License: *GNU General Public License*

This is a perl module that simplifies the decoding of MIME Internet mail messages [17]. It is used in the parser to strip away unwanted binary attachments, and decode base64 and uuencoded text parts.

15.5.3 Xerces XML Parser

Author: *The Apache Foundation*
License: *Apache Software License*

An XML parser derived from IBMs XML4J. It is used in the Java Servlet running on the web server to parse the XML data read from the query server.

15.5.4 Xalan XSLT Stylesheet Processor

Author: *The Apache Foundation*
License: *Apache Software License*

An XSLT processor, it is used in the Java Servlet to apply stylesheets to the XML data parsed by Xerces.

15.5.5 BerkeleyDB 3.x

Author: *Sleepycat Software*
License: *Free re-distribution with Open Source Software*

An embedded database, it is used to store the forward and reverse indexes produced and read by the indexer, inverter and query server.

15.5.6 MySQL

Author: *TcX AB*
License: *Free for use, provided it is not sold as part of a product*

An SQL database, used to store the mail messages in once parsed by the parser.

15.5.7 ZLib

Author: *Jean-loup Gailly and Mark Adler*
License: *May be freely redistributed provided it is not altered*

A compression library used to compress the bodies of the mail messages as they are stored in the database.

15.5.8 MM MySQL JDBC Drivers

Author: *Mark Matthews*
License: *GNU General Public License*

Java Database Connectivity drivers to allow Java programs to access the MySQL database. These are used for the servlet to store/retrieve preferences from the database.

Chapter 16

Maintenance Guide

The code for this project has evolved in many different directions. Many different approaches were tried and by using source control the project remained manageable.

16.1 Source Control

The entire source for the project is kept under revision control using Concurrent Versioning System (CVS). This system allows multiple users controlled access to a set of cental source files. It's main feature is that it keeps different revisions of a file, so at any time a previous version of the code can be checked out.

One of the main reasons for using source control, is that there are external parties needing to keep track of what happens with this project. The final web pages will be designed by one of the other members of Etherworks, and once the project is finished other members of Etherworks will be working on bringing the project to life as a full commercial site. Hence they all need to be kept in the loop as to what was happening with the development. The CVS system was set to email the commit log, entered whenever a revision was committed to the repository, to all of the members of Etherworks.

Not only is the code kept in the CVS repository, but also this documentation, and many of the research papers referenced here.

The code is available via anonymous CVS to others wishing to download it. To do this under unix you need to first set the CVSROOT variable:

```
% setenv CVSROOT :pserver:anon@beta.osdigger.com:/home/cvs
```

The login (there is no password, just hit return):

```
% cvs login
```

Then checkout the sources:

```
% cvs -z3 checkout indexer scripts servlets
```

There is also a CVSweb interface to the CVS repository allowing access to the various revisions and the commit logs via a web browser. This can be found from the project page at:

`http://beta.osdigger.com/osdigger`

16.2 Makefiles

Makefiles aid the compilation of programs as they can check the dependencies and avoid having to recompile all of the sources for each change. There are makefiles for the C code and for the documentation.

16.2.1 Documentation

This documentation can be compiled into various forms from the `thesis` directory.

Command	*Description*
`make thesis.dvi`	Compiles a DVI version of the thesis
`make thesis.ps`	Compiles a Postscript version of the thesis
`make thesis.pdf`	Compiles a PDF version of the thesis
`make html`	Converts the thesis to HTML using latex2html

16.2.2 Indexer

The Indexer, Inverter and Query server can be compiled using the makefile from the `indexer` directory.

Command	*Description*
`make indexer`	Compiles the indexer
`make inverter`	Compiles the inverter
`make search`	Compiles the query server

16.3 Portability

The code has been mostly written on FreeBSD, a freely available Unix. This is the platform that the OSDigger server runs. The compression routines used in the code do not convert the data to any specific byte order, so although the code may run on other platforms, the indexes produced cannot be read on platforms with a different byte order.

The java servlets used for the web interface to the indexes, being written in java, are cross platform. The servlets do not have to run on the same machine as the index server or the database server.

The system requires quite a few third party modules and libraries installed at run-time in order to work, however since this system is intended for use as a service, rather than being installed on hundreds or thousands of users PCs, this is not considered a problem.

Part V

Operations

Chapter 17

Security

Chapter 18

Testing

Chapter 19

Integration

Part VI

Project Review

Chapter 20

Project Review

The project has been a very valuable experience. One of the main aspects that I have learned the most from has been the Business Plan. This is a real venture that I am working on, and although I know what I am trying to do, it has been a great challenge to try an articulate this to a business audience. On the one hand I have been reading in-depth papers into the latest Information Retrieval algorithms. On the other hand I have been trying to convince an investor that the Return on Investment is high enough and that a market exists for the product.

This wide range of tasks has broadened my experience considerably. I have already tried to take the idea to a Bank Manager to ask for a loan, but the ratio of loan capital to investment capitol was too high. This forced me to look at the business plan again and work out how we could re-finance it in a better way. I now feel confident to try again and see if we can get the loan!

During the development I changed directions quite a few times, and there was parallel development of both a Java and C version of the Indexers for quite a while. There seems to be no one simple perfect solution and as with most things in life a mix is needed. I learned a great deal from the many different partial solutions I tried to develop. Many of them did not prove satisfactory, but I learnt as much from the ideas that did not work as I did from the ideas that did.

Information Retrieval is a very wide topic, and many of the ideas proposed in the papers I have read make most existing search engines seem extremely primitive. I would like to take some of these ideas further and implement them in a production system.

Chapter 21

Future Work

21.1 Tilebars

21.2 LSI

Latent Semantic Indexing (LSI) is described earlier and is a way of automatically building up associations between related words. The Twostep search process currently implemented in OSDigger uses a similar method to find related terms to a search query. The method currently used works out the related terms on the fly, however it would be possible to batch process the entire collection to precompute this. Since the relationships between words is unlikely to change in the short term the mappings would not need to be updated that often. As it is not being computed in real time, much more sophisticated techniques could be used, such as using the spacing between terms – terms closer together are more likely to be related than terms further apart.

Bibliography

[1] Adforce. http://www.adforce.com.

[2] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley Longman, 1999.

[3] Bristol Enterprise Centre. http://www.bris.ac.uk/enterprise.

[4] J. Bentley and A. Yao. An almost optimal algorithm for unbounded searching, 1976.

[5] The sleepycat software home page. http://www.sleepycat.com.

[6] A. Bookstein, S. Klein, and T. Raita. Model based concordance compression, 1992.

[7] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. 7th Int. World Wide Web Conf.*, 14–18 April 1998.

[8] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast Incremental Indexing for Full-Text Information Retrieval. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 192–202, Santiago, Chile, 1994.

[9] Eric W. Brown, James P. Callan, W. Bruce Croft, and J. E. B. Moss. Supporting full-text information retrieval with a persistent object store. Technical Report UM-CS-1993-067, University of Massachusetts, Amherst, Computer Science, August, 1993.

[10] D. Crocker. RFC 822: Standard for the format of ARPA Internet text messages, August 1982.

[11] W. Bruce Croft and Pasquale Savino. Implementing ranking strategies using text signatures. *ACM Transactions on Office Information Systems*, 6(1):42–62, 1988.

[12] Edleno S. de Moura, Gonzalo Navarro, and Ricardo Baeza-Yates Nivio Ziviani. Compressed pattern matching, approximate compressed pattern matching, direct compressed pattern matching. 1998.

[13] DoubleClick. http://www.doubleclick.com.

[14] P. Elias. Universal codeword sets and the representation of the integers, 1975.

[15] W. Frakes and R. Baeza-Yates. Stemming algorithms.

[16] W. Francis and H. Kucera. Frequency analysis of english usage: Lexicon and grammar, 1982.

[17] N. Freed and N. Borenstein. RFC 2045: Multipurpose Internet Mail Extensions (MIME) part one: Format of Internet message bodies, November 1996.

[18] N. Freed and N. Borenstein. RFC 2046: Multipurpose Internet Mail Extensions (MIME) part two: Media types, November 1996.

[19] N. Fuhr. Probabilistic models in information retrieval, 1992.

[20] R. Fung and B. Favero. Applying bayesian networks to information retrieval, 1995.

[21] G. Furnas, S. Deerwester, S. Dumais, T. Landauer, R. Harshman, L. Streeter, and K. Lochbaum. Information retrieval using a singular value decomposition model of latent semantic structure, 1988.

[22] R. Gallager and D. Van Voorhis. Optimal source codes for geometrically distributed integer alphabets, 1975.

[23] Leszek Gasieniec and Wojciech Rytter. Almost optimal fully lzw-compressed pattern matching. 1999.

[24] S. Golomb. Run-length encodings, 1966.

[25] Generic object oriented database system (goods) and persistent object storage for c++ (post++). http://www.ispras.ru/ knizhnik.

[26] Google affiliate programs. http://www.google.com/websearch_programs.html.

[27] H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects.* Academic Press, New York, 1978.

[28] D. Hull. Improving text retrieval for the routing problem using latent semantic indexing, 1994.

[29] Timothy C. Bell Ian H. Witten, Alistair Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Inc., second edition, 1999.

[30] The Linux Counter Project. http://counter.li.org.

[31] J. Lovins and o algorithm. Mechanical translation and computational linguistics, 1968.

[32] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps, 1992.

[33] K. Moore. RFC 2047: MIME (Multipurpose Internet Mail Extensions) part three: Message header extensions for non-ASCII text, November 1996.

[34] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.

[35] Gonzallo Navarro and Mathaeu Raffinot. A general practical approach to pattern matching over ziv-lempel compressed text. 1999.

[36] Netcraft Web Server Survey. http://www.netcraft.com/survey/.

[37] New Zealand Digital Library. http://www.nzdl.org.

[38] M. Olson, K. Bostic, and M. Seltzer. Berkeley db, 1999.

[39] The Open Source Definition. http://www.opensource.org/osd.html.

[40] J. Pearl. Probabilistic reasoning in intelligent systems: Networks of plausible inference, 1988.

[41] M.F. Porter. An algorithm for suffix stripping. Program 14 (3), July 1980.

[42] PostgreSQL. http://www.postgresql.org.

[43] T. Radecki. Fuzzy set theoretical approach to document retrieval, 1979.

[44] W3C Recommendation. Extenisble markup language (xml) 1.0. http://www.w3.org/TR/REC-xml, Feb 1999.

[45] W3C Recommendation. Xsl transormations (xslt) 1.0. http://www.w3.org/TR/xslt, Nov 1999.

[46] P. Resnick and A. Walker. RFC 1896: The text/enriched MIME content-type, February 1996.

[47] B. Ribeiro and R. Muntz. A belief network model for ir, 1996.

[48] S. Robertson and K. Sparck-Jones. Relevance weighting of search terms, 1976.

[49] G. Salton. The smart retrieval system: experiments in automatic document processing, 1971.

[50] G. Salton, E. Fox, and H. Wu. Extended boolean information retrieval, 1983.

[51] G. Salton and M. Lesk. Computer evaluation of indexing and text processing, 1968.

[52] E. Schuegraf. Compression of large inverted files with hyperbolic term distribution, 1976.

[53] Craig Stanfill, Robert Thau, and David Waltz. A parallel indexed algorithm for information retrieval. In *Proceedings of the Twelfth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Architectures, pages 88–97, 1989.

[54] J. Teuhola. A compression method for clustered bit-vectors, 1978.

[55] A. Tomasic, H. García-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):289–300, June 1994.

[56] H. Turtle and W. Croft. Inference networks for document retrieval, 1990.

[57] H. Turtle and W. Croft. Evaluation of an inference network-based retrieval model, 1991.

[58] R. Wilkinson and P. Hingston. Using the cosine measure in a neural network for document retrieval, 1991.

[59] I. Witten, T. Bell, and C. Nevill. Indexing and compressing full-text databases for cd-rom, 1991.

[60] S. Wong, W. Ziarko, and P. Wong. Generalized vector space model in information retrieval, 1985.

[61] Yahoo. http://www.yahoo.com.

[62] Brendon Cahoon Zhihong Lu, Kathryn S. McKinley. The hardware/software balancing act for information retrieval on symmetric multiprocessors. 1998.

[63] James P. Callan Zhihong Lu and W. Bruce Croft. Applying inference networks to multiple collection searching. 1991.

[64] Kathryn S. McKinley Zhihong Lu. Searching a terabyte of text using partial replication.

[65] G. K. Zipf. *Human Behavior and the Principle of Least-Effort.* Addison-Wesley, Cambridge, MA, 1949.