A SCALABLE DISTRIBUTED WEB SYNDICATION CRAWLING SOFTWARE ARCHITECTURE

DITESH KUMAR A/L SHASHIKANT

DISSERTATION SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY UNIVERSITY OF MALAYA KUALA LUMPUR

JUNE 2008
Abstract

The availability of high speed affordable access has resulted in a strong growth of user contributed Internet content, much of which is offered in structured web syndication formats. The structured nature of the syndication formats have allowed for easier retrieval and parsing by crawling systems. Nonetheless, such systems typically have had complex and difficult implementation details such as the use of threading methodology which can introduce subtle data corruption bugs as well as high levels of execution coupling and the use of central synchronization servers which can lead to system bottlenecks.

As such, the goal of this dissertation is to investigate a new method of building distributed web syndication crawling architectures which attempt to resolve the problems noted above. Traditional methods of building crawling architectures are first explored in the dissertation and the new methodology for building distributed crawling architectures is introduced whereby direct communication is excluded between individual components of the system by the use of a data driven methodology. The proposed system architecture, algorithms and instrumentation techniques, experimental results and performance tuning approaches are discussed in detail.

The experimental results indicate that a data driven crawling architecture can provide better performance then a traditional distributed crawling architecture, as long as sufficiently strong emphasis is given on the overall design of the system to avoid run time performance issues. This dissertation also contributes more efficient protocols for content retrieval and shows through inspection that data driven distributed crawling architectures ensure data integrity through existing relational database locking mechanisms.
Acknowledgements

I would like to thank my research supervisor, Dr. Rosli, for his generous patience and assistance in helping me complete this thesis.

In addition, I would also like to thank my family whose support has been crucial in completing this dissertation.
# Table of Contents

Chapter 1: Introduction ................................................................. 1  
1.1 Syndication Technologies .................................................. 4  
1.2 Weblogs, The Growth of User Content and The Importance of Crawlers ......................... 6  
1.3 Distributed Crawling Systems ............................................. 8  
1.4 Problem Statement ............................................................ 10  
1.5 Objectives ......................................................................... 12  
1.6 Research Methodology ....................................................... 13  
1.7 Dissertation Organization .................................................. 14  
Chapter 2: Literature Review ..................................................... 16  
2.1 Crawling Architectures ....................................................... 16  
2.2 Coverage, Overhead and Overlap ....................................... 19  
2.3 System Architecture for Transmission of Resource Links .............................................. 20  
2.4 Scalability ....................................................................... 22  
2.5 URL Partitioning .............................................................. 23  
2.6 Description of Really Simple Syndication (RSS) formats .............................................. 25  
2.7 The RSS Specification ....................................................... 25  
2.7.1 RSS 1.0 ................................................................. 27  
2.7.2 Atom ................................................................. 28  
2.8 Process Forking and Threading ........................................... 29  
2.9 Process Forking with PHP .................................................. 29  
2.10 Forking processes and the pcntl_fork() function call .................................................. 30  
2.10.1 Zombie child processes and pcntl_waitpid() function call ....................................... 30  
2.10.2 Installing a signal handle using pcntl_signal() function call ................................... 31  
2.10.3 Exit codes .............................................................. 32  
2.11 Testing and Instrumentation Tools ...................................... 32  
2.11.1 top ................................................................. 33  
2.11.2 vmstat ............................................................ 35  
2.11.3 PostgreSQL Tuning ................................................... 36  
2.11.3.1 Database Indexing ............................................... 36  
2.11.3.2 Analyzing Queries ............................................. 36  
2.12 Conclusion ................................................................. 38  
Chapter 3: Proposed System Architecture .................................... 39  
3.1 Defining Operations ......................................................... 40  
3.2 The Principle of the Independent Execution of Operations .......................................... 41  
3.3 Components ................................................................. 42  
3.4 Proposed System Architecture ......................................... 42  
3.5 Proposed Functional Architecture ..................................... 43  
3.5.1 Retrieving web feed content ........................................ 45  
3.5.2 Retrieving URI link contents ....................................... 46  
3.5.3 Analyzing content for new URI links ........................................ 46  
3.5.4 Feed aging .......................................................... 46  
3.6 Component Architecture ................................................ 46  
3.6.1 Feed Analysis Component ........................................ 47  
3.6.2 Error Handling Component ........................................ 47  
3.6.3 Database Abstraction Component ................................... 47  
3.6.4 Downloader Abstraction Layer .................................... 48  
3.7 Conclusion ................................................................. 49  
Chapter 4: Implementation Details and Algorithmic Techniques ................. 51  
4.1 Base system ................................................................. 51  
4.1.1 Operating System .................................................... 52
List of Tables

Table 2.1:  Key XML Tags in RSS 0.9x  26
Table 4.1:  Cross sectional analysis of popular operating systems  53
Table 4.2:  Data dictionary for feed_content database table  71
Table 4.3:  Data dictionary for feed_url_metadata database table  72
Table 4.4:  Data dictionary for feed_urls database table  72
Table 4.5:  Data dictionary for link_urls database table  73
Table 4.6:  Data dictionary for queue database table  73
Table 5.1:  Data Collection Methodology  76
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Web Connection Speed Trends – Home (US)</td>
<td>1</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Broadband Growth Trend – US Home Users</td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Growth of web servers serving content between August 1995 to February 2007</td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>Exponential growth of weblogs</td>
<td>6</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Illustration of the Google Architecture</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Illustration of the Mercator Architecture</td>
<td>18</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>top in action</td>
<td>34</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>vmstat in action</td>
<td>35</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>High Level View of the Proposed System Architecture</td>
<td>43</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Proposed Distributed System Architecture</td>
<td>43</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Context Diagram of the Proposed Crawling System</td>
<td>44</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Data Flow Diagram of the Proposed Crawling System</td>
<td>45</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Identified System Architecture</td>
<td>49</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Overall System Architecture</td>
<td>52</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>System Architecture with Identified Base Elements</td>
<td>56</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Flowchart of Retrieving Web Feed Content Operation</td>
<td>59</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Flowchart of Retrieving URI Link Content Operation</td>
<td>61</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Flowchart of Analyzing Content for New URI Links Operation</td>
<td>63</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Flowchart of Feed Aging Operation</td>
<td>65</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Flowchart of URL Extractor Component (FeedFinder)</td>
<td>68</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>ER Diagram of the Database for the Crawling System</td>
<td>70</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>System Architecture of Benchmark Crawling System</td>
<td>77</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>New Feeds (Cumulative) vs Time (data sets/crawler = 500)</td>
<td>78</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Average System Load vs Time (data sets/crawler = 500)</td>
<td>79</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>New Feeds (Cumulative) vs Time (data sets/crawler = 1000)</td>
<td>80</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Average System Load vs Time (data sets/crawler = 1000)</td>
<td>80</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>New Feeds (Cumulative) vs Time (data sets/crawler = 1500)</td>
<td>81</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>Average System Load vs Time (data sets/crawler = 1500)</td>
<td>82</td>
</tr>
<tr>
<td>Figure 5.8</td>
<td>New Feeds (Cumulative) vs Time (data sets/crawler = 2000)</td>
<td>83</td>
</tr>
<tr>
<td>Figure 5.9</td>
<td>Average System Load vs Time (data sets/crawler = 2000)</td>
<td>83</td>
</tr>
<tr>
<td>Figure 5.10</td>
<td>Comparison of Data Driven Architecture with Traditional Architecture</td>
<td>85</td>
</tr>
<tr>
<td>Figure 5.11</td>
<td>CPU Usage (PHP) vs Time</td>
<td>86</td>
</tr>
<tr>
<td>Figure 5.12</td>
<td>CPU Usage (PostgreSQL) vs Time</td>
<td>86</td>
</tr>
</tbody>
</table>
Figure 5.13: Memory Usage (PHP) vs Time 87
Figure 5.14: Memory Usage (PostgreSQL) vs Time 87
Figure 5.15: HTTP Response Codes 89
Figure 5.16: Top 10 Feed Generators 89
Figure 5.17: Feed Types 90
Code Listing

Code Listing 2.1: Channel XML Tag 26
Code Listing 2.2: pcntl_waitpid() example 31
Code Listing 2.3: pcntl_signal() example 32
Code Listing 2.4: Debugging PostgreSQL Queries 37
Code Listing 4.1: MagpieRSS Usage 67
Code Listing 4.2: FeedFinder Usage 69
Code Listing 4.3: pg_hba.conf example 73
Code Listing 4.4: Database Creation 74
Code Listing 6.1: Proposed Addition to the Robot Exclusion Protocol 96
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSS</td>
<td>Cascading Style Sheet</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name Server</td>
</tr>
<tr>
<td>ER</td>
<td>Entity Relationship</td>
</tr>
<tr>
<td>ETAG</td>
<td>Entity Tag</td>
</tr>
<tr>
<td>GHz</td>
<td>Gigahertz</td>
</tr>
<tr>
<td>HTML</td>
<td>Hyper Text Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transmission Protocol</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>MD5</td>
<td>Message-Digest Algorithm 5</td>
</tr>
<tr>
<td>PHP</td>
<td>PHP: Hypertext Preprocessor</td>
</tr>
<tr>
<td>PID</td>
<td>Process ID</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface for UNIX</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>RES</td>
<td>Resident Memory</td>
</tr>
<tr>
<td>RSS</td>
<td>Rich Site Summary</td>
</tr>
<tr>
<td>SHR</td>
<td>Shared Memory</td>
</tr>
<tr>
<td>URI</td>
<td>Unique Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Universal Resource Locator</td>
</tr>
<tr>
<td>VIRT</td>
<td>Virtual Memory</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Memory</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

In this day and age, information technology is ubiquitous in people's lives. The recent advent of affordable high speed networks have caused people to share information on the Internet in many ways not foreseen before. Examples include the widespread use of personal websites in the middle of 1990's, the birth of weblogs in the late 1990's, the stunning growth of personal photo and video file sharing websites at the start of the new millennium and many more. As such, we see a clear trend of increasing consumer use of the Internet grow as a directly correlated function of broadband speed.

Figure 1.1 and Figure 1.2 indicate the increasing and widespread use of broadband in the United States (US) from 2005 to 2007 (WebSiteOptimization, 2007):

Figure 1.1
Web Connection Speed Trends – Home (US)
In addition, we see a distinct growth in the number of webservers serving content on the Internet as reported by Netcraft from August 1995 to February 2007 as is indicated in Figure 1.3 (Netcraft, 2007):
As is clearly demonstrated, we can realistically expect that the amount of content existing on the Internet to continue growing substantially in the future given the explosion of commercial low cost Internet access services that are available to consumers today.

Given the high growth of content available on the Internet, users will require a method to search all the information. Search engines generally provide a solution for this problem by downloading the content, categorizing the content and providing search mechanisms for the users. However, search engines have had substantial problems in returning relevant results to the user's query due to improper presentation of content on the Internet. In the past, content has had many different (and often, incompatible) forms of representation (Koch, 2007). As an example, during the 1990's, content was represented in different HTML markup that would display differently on browsers and were often incompatible with each other. Furthermore, content would be interspersed with look and feel, hence making it difficult to separate the actual content from the interface aspect of the content. Finally, even if we could retrieve the actual content itself, we would have trouble interpreting the content due to the fact that the content itself was not represented in a consistent fashion.

These problems are not new and have been addressed in some manner by various standardization and technical bodies. Firstly, the WorldWideWeb Consortium (W3C) worked hard to standardize HTML to prevent fragmentation of the language\(^1\). This simplified the problem of the different representation of HTML. Secondly, to allow for separation of content and look and feel, W3C also standardized Cascading Style Sheets (CSS)\(^2\) and this has been in widespread use across the Internet. Finally, to

---

\(^1\) W3C HTML recommendations can be found at http://www.w3.org/MarkUp/
\(^2\) W3C CSS recommendations can be found at http://www.w3.org/Style/CSS/
help represent content in a consistent fashion, the Semantic Web project\(^3\) was started to persuade web developers to correctly represent their content (Berners-Lee, 2006) so that the problem retrieving and sharing of data is eliminated.

Unfortunately, the Semantic Web project has not had much success (Shadbolt, Hall and Berners-Lee, 2006). Far more successful, in comparison, have been syndication technologies that are different from the Semantic Web because they are substantially simple and allow for content to be represented in the normal form as well as machine consumable form. Syndication technologies are built on the success of XML, are simple to implement and thus are being widely used in weblogs.

1.1 Syndication Technologies

Web syndication refers to making structured content on a website available for use by other people. Syndication has been traditionally used in news websites but in recent years the popularity of weblogs and its widespread use of syndication as an increasingly important method of news dissemination has brought new importance to the syndication formats. As an indication of the popularity of syndication formats, Feedster (a popular syndication search engine) has logged over half a million syndicated resources and over 50 million content entries in a single year of operation (Gallagher, 2004).

Syndication technology have had two primary technical implementation methodologies – push technology and pull technology (Udell, 2004). Push technology pushes the syndication content to the receiver whereas pull technology requires the receiver to retrieve the syndication content from a common source. Push syndication

\(\text{\footnotesize \(^3\) W3C Semantic Web project can be found at }\) \url{http://www.w3.org/2001/sw/}
has suffered from a poor implementation strategy and as such, today pull syndication technology is dominant in the market. Pull syndication is popularly referred to as web feeds due to the overwhelming use of the technology over HTTP.

Technically, web feeds are structured XML documents which are retrieved over standard HTTP connections. The most widely used formats for web feeds are the Rich Site Summary (RSS) and Atom, and even with these two formats, the differences are minor and several abstraction libraries exist that are generally used to present a consistent interface to the underlying crawling architecture.

Of the two however, the de-facto web syndication format is RSS (King, 2007), a family of XML based communication standards with the following members:

- Rich Site Summary (RSS 0.9x and RSS 2.0)
- RDF Site Summary (RSS 0.9 and 1.0) (RDF: Resource Description Framework)

The Atom format is relatively newer and was created to solve certain technical issues (Ruby, 2003). In essence, however, Atom is conceptually similar to RSS. RSS and Atom allow a web developer to publish content on their website in a format that a computer program can easily understand and digest. This allows users to easily repackage the content on their own websites or blogs, or privately on their own computers (King, 2007).

The importance of RSS and other web feed technologies lie in the fact that the content is structured. Traditionally, web based content has been unstructured and mixed presentation and content, making it difficult to de-mark the boundaries between
presentation and content. Web feed technology solves this problem by delivering all content in a structured form (specifically as XML files). The presentation of the content is up to the client software.

1.2 Weblogs, The Growth of User Content and The Importance of Crawlers

Weblogs are perhaps the largest source of user content available on the Internet today. Technorati is a well known authority on weblogs as they keep track of growth rates of weblogs. As of February 2006, Technorati reported that they were tracking 27.2 million weblogs. This can be clearly seen by the activity chart at Technorati (Figure 1.4) where between March 2003 and January 2006, the growth rate pattern exhibits an exponential growth rate (Sifry, 2007):

Figure 1.4
Exponential growth of weblogs
Hence, it is clear that weblogs are among the most important sources of user generated content on the Internet. Furthermore, given that weblogs implement modern syndication technology and host a substantially large amount of user content, the challenge to build a suitable search engine that support weblogs encompass two main areas of technology:

- building the search engine semantics (for example, to allow for data extraction, data analysis, data aggregation and data indexing).
- building the backend retrieval engine (for example, to retrieve the data efficiently)

Building search engine semantics is a large scale effort that requires significant investment of engineering, design, development and testing resources. This problem has been investigated on many fronts, most famously by Google's PageRank algorithm (Brin and Page, 1998). The academic sector, technology sector as well as the open source and proprietary software sectors have all investigated improving search engine semantics and there exists a large body of literature in this area.

Similarly, backend content retrieval engines have had substantial investigation. Backend content retrieval engines are also known as web crawlers. A crawler is a software entity which serves the role of retrieving, cataloging and storing network content and resources. Crawlers complement search engines as they provide necessary services for searching and indexing technologies to work effectively. The crawling design methodology has proven to be extremely useful in large networks where content storage is not centralized and content ownership is distributed. Without crawling systems, there would be no mechanism available to provide search functionality to users.
which would reduce the usefulness of the network.

### 1.3 Distributed Crawling Systems

There are large number of different crawling systems available and while the operation of the individual crawling systems may differ, the underlying conceptual idea remains the same. Every crawling systems executes the following set of operations (Gathani and Loh, 2004):

a) Accessing a remote resource and downloading the resource  
b) Analyzing the resource  
c) Saving the resource data and result of the analysis

Conceptually, crawling systems may look easy to implement but the converse is usually true. There are large number of significant details that are not be obvious at the outset and often hinder the design and development of a high quality crawling system. Almost all crawling architectures in use today are distributed systems that use several crawlers as part of a coherent crawling architecture. The primary reason for this are the many advantages distributed crawling systems have over a single centralized crawling system, including (Gathani and Loh, 2004):

- Removal of network throughput bottlenecks  
- Removal of processing bottlenecks  
- Removal of database bottlenecks  
- Removal of storage bottlenecks

Given the rapid growth of the weblogs, a centralized crawling architecture will be sooner or later be overloaded with crawling tasks. By decentralizing the tasks,
scalability, failure handling and other pertinent issues in a centralized architecture no longer becomes significant in day to day operations. Additional crawling agents can be seamlessly added into the crawling environment without requiring any major modifications, which clearly is a boon to administration of the crawling architecture, future scalability and failure handling.

Scalability is defined as the capability of a system to increase performance under an increased load (Boldi, Codenotti, Santini and Vigna, 2002). Scalability is important in ensuring that the system remains effective when there is a significant increase in users and resources.

There are several well known ways of achieving scalability. One way is partitioning (Boldi, Codenotti, Santini and Vigna, 2002). Partitioning works by splitting a large system into parts that can operate independently of its component parts. Another approach is replication whereby several copies of components are kept consistent with one another and each can be used in case of failures. Replication works well with a load balancing strategy that keeps the load under reasonable threshold at each component and minimized delay in the communication subsystems.

Hence, to achieve scalability, a distributed crawling architecture is required. A distributed crawling architecture is defined in Ubicrawler (Boldi, Codenotti, Santini and Vigna, 2002) as a software system whereby each crawler in the system is identically programmed and is only distinguished by a unique identifier⁴. A distributed crawling architecture can thus scale as required, and as such, a distributed architecture is suitable for crawling systems.

⁴ The term crawling agent will be used instead of crawler to identify the fact that the crawling agent is only one portion of a distributed crawling architecture.
1.4 Problem Statement

Traditional distributed crawling architectures are architected as singly-threaded crawlers using asynchronous I/O (Burner, 1997) running on different machines within the distributed system and with a central URL server process dispersing URL's to the crawlers (Brin and Page, 1998). Another common approach in building distributed crawling architectures is to build a multi-threaded components within the crawling architecture to achieve high levels of concurrency (Heydon and Najork, 1999).

The research performed in the late 1990's (noted in the paragraph above) have provided a sufficient base for building high performance distributed web crawling architectures. However, there are certain areas of improvement to the methodology proposed in the late 1990's, including:

- there is significant debate on the use of threading methodology (Lee, 2006), as threads can introduce subtle data corruption bugs and difficulty in troubleshooting run-time issues.
- high levels of execution coupling within components of the system lead to system bottlenecks.
- removing the need for the use of a central master server for distribution of URL's.

One method of improving the state of distributed crawling architectures is to introduce a data driven crawling architecture. A data driven system is generally understood as a system whereby the execution path of the system is driven by data structure used, and not by the actual program logic (Raymond, 2003). An example of a data driven system is a state machine.
A data driven architecture in the context of a crawling architecture can thus be visualized as a composition of independent non-communicating parts that act upon the availability or non-availability of data. Data driven architectures are different from traditional architectures as they only execute on the simple principle of the presence and absence of data. Hence, the entire system is built on the principle that operation can only start executing on the availability of data and that each operation is independent of the execution of other operations in the system.

Traditional non-data driven architectures generally have many intersecting components that heavily rely on each other. As such, the successful execution of each component requires the successful execution of each and every child component the parent component depends on (Heydon and Najork, 1997). This execution methodology can lead to system bottlenecks. There has been little progress on this methodology over the years. Object oriented and threaded programming are good examples of such traditional non data driven architectures. While these architectures have their benefits, they also suffer from excessive complexity and bottlenecks. When \( n \) different component in these traditional non data driven architectures interact with each other to perform a task, the complexity of the system is calculated at \( n(n-1) \). This implies that as \( n \) increases in size, the complexity of the system increases as a factor of \( n^2 \). A data driven architecture eliminates the need for this complexity as no components directly interact with each other.

Threading can introduce further issues within components. As multiple components can execute simultaneously within separate threads, the use of global data and race conditions may possibly corrupt program state and lead to difficult to debug
problems. In addition, as each component depends on execution of other components, bottlenecks may arise as some components wait for other components to finish executing. This can lead to problems when scalability is being considered. In data driven architectures, this problem exists to some extent but careful locking and data partitioning is able to resolve these issues.

This dissertation will investigate and build a proof of concept data driven crawling architecture which will attempt to solved the problems noted with traditional crawling architectures. The significance of the proof of concept data driven crawling architecture is to show that execution coupling between components can be reduced and increase performance of the system.

1.5 Objectives

The primary objective of this dissertation is to investigate and build a proof of concept partitioned data driven distributed web feed crawling architecture. Currently, there does not exist any openly accessible work on building a data driven retrieval engine for web feeds. All existing work on building retrieval engines for syndicated feeds are proprietary and are not accessible to external parties for independent implementation.

Given that such a system has not been implemented in the academic environment, there has been no peer review of possible problems and an investigation of the feasibility of such an endeavor.
Thus, the objectives and scope of research of the dissertation is as follows:

- to investigate existing web syndication and traditional crawling architectures and identify implementation and runtime issues in these architectures
- to propose a new crawling architecture which will address the identified problems with existing crawling architectures
- to analyze algorithmic techniques, usage issues and system performance characteristics of the proposed crawling architecture

1.6 Research Methodology

A proof of concept data driven crawling architecture is build and set up on two identical machines with a centralized database. The hardware specifications of these machines are as follows:

- Pentium 4 3.0 GHz
- Hard drive 120GB
- 1GB RAM
- LAN Network connectivity
- 384KBps Internet connectivity

The software specifications are as follows:

- Linux kernel 2.6.20
- PostgreSQL 8.1
- PHP 5.2 CLI
Performance of the system will be determined by base system parameters such as memory user and CPU use will be logged and graphed. Furthermore, overall system efficiency and performance will also be logged and graphed.

In addition, for comparison purposes, a traditional crawling architecture will also be set up for benchmark purposes. This architecture is a simplified derivation of the Google architecture as defined in Figure 2.1 and described in section 2.1.

To collect the data reliably, a test harness will be built to observe the performance characteristics of the system. The test harness will capture an observation point at 10 second intervals and store the results in an external database over a three hour period. Upon the completion of the three hour period, the data is saved and graphed as scatter plots.

### 1.7 Dissertation Organization

This dissertation is organized into the following chapters:

- The second chapter on “Literature Review” will cover past work in the area of distributed crawling architectures as well as a description of technology used in the implementation of the data driven distributed crawling architecture.
- The third chapter on “Proposed System Architecture” will outline and describe the overall proposed system architecture of the data driven distributed crawling architecture.
- The fourth chapter on “Implementation Details and Algorithmic Techniques” will outline and describe the algorithms and implementation issues of the data
Driven distributed crawling architecture.

- The fifth chapter on “Experimental Results and Data Analysis” will describe the experimental methodology and scenarios, present the experiment results and discuss the outcome from the results.
- The sixth chapter on “Discussion” will discuss the overall implementation, issues faced and present proposed solutions.
- The final chapter, “Conclusion”, will discuss the results of the investigation performed in this dissertation, summarize key findings and outline future areas of work.
Chapter 2: Literature Review

This dissertation deals with a wide range of technologies and as such, the literature survey will cover all relevant technologies referenced in this dissertation. This includes:

- crawling architectures
- implementation issues (coverage, overhead and overlap)
- system architecture and link transmission methodologies
- URL partitioning
- web feeds (RSS and Atom)
- process forking and threading
- testing and instrumentation techniques
- database tuning mechanisms
- exit codes

2.1 Crawling Architectures

Every large Internet search engine company runs a crawling architecture. Google, one of the most famous search engines, runs one of the world's largest crawling architectures. Brin and Page published a paper in 1998 whereby they described the use of both a centralized and decentralized crawling system. In this crawling architecture, each crawling agent performed a set of different functions (Brin and Page, 1998)\(^5\). Communication took place between the crawling agent and central server rather than directly between the crawling agents.

\(^5\) It should be noted that the current Google architecture differs substantially from the original published architecture.
The Google architecture is shown in Figure 2.1 describes a traditional crawling architecture with a central URL server that distributes URL's to multiple crawling agents. The crawling agents store the results in a store server which compresses the pages and stores them to disk (the Repository). An indexer process read the pages back from the disk, extracts all URL links associated with the downloaded web page and sends these URL links to the URL server. The DNS filter resolves the host names into IP addresses while the URL filter removes duplicate URL’s.

Each crawling agent is assigned a set of URL’s and the crawling agent will store all retrieved resources locally. Related resource links will be retrieved by the crawling agent or passed to other crawling agents. URL links are queued in the URL list to be
retrieved at a later time. This is a continually executing process until the URL list queue is empty.

Another crawling architecture which is similar to Google is the Polytechnic crawling architecture (Boswell, 2003). The Polytechnic crawling architecture uses a central server too but differs from the Google architecture by allowing multiple central servers to co-exist. This may be viewed as an improvement to the original Google architecture as it clearly helps in the case of the failure of the central server component.

By comparison, in the Mercator architecture (Heydon and Najork, 1999) and Ubicrawler (Boldi, Codenotti, Santini and Vigna, 2002) architecture, the crawling system is fully distributed. The Mercator architecture's approach is such that each crawling agent is in itself a fully functional system and as such, each crawling agent operates without having to contact central servers. The advantage of this approach is higher tolerance of failures as the system is still functional even if any individual component fails. The distributed architecture is defined in Figure 2.2.

Figure 2.2
Illustration of the Mercator Architecture (Heydon and Najork, 1999)
In the Mercator distributed system as defined in Figure 2.2 (Heydon and Najork, 1999), there are protocol components which allows a crawler to select an appropriate protocol component for downloading online resources stored in queues in the URL frontier. The crawler is multi-threaded and has multiple methods such as Link Extractor (to extract links), Tag Counter (to keep count of tags), GIF stats (to keep track of statistics of GIF image files) and Content-Seen (to determine if the document has already been seen) which facilitates crawler specific tasks. Of particular interest is the RIS module which allows for a stored copy of an online resource to be re-read multiple times by multiple threads. The threaded environment allows for components to have side-effects on the state of the crawler, as well as on their own internal state, which can change run time behavior.

2.2 Coverage, Overhead and Overlap

To ensure proper functioning of the independent components in a fully distributed system as in Mercator or Ubicrawler, assignment of responsibilities become important. Each individual crawling agent must be able to come to a decision on whether to retrieve a URI resource. To make this decision, several factors that need to be considered here which include communication overhead, coverage and the overlap (Najork and Wiener, 2001; Cho and Garcia-Molina, 2002).

Coverage is defined as \( \frac{U}{I} \). \( U \) represents the total number of pages that the overall crawling agents has to download while \( I \) is the number of unique pages retrieved by the crawling agents. Values approaching unity indicates a well functioning crawling architecture whereby most web resources are already being crawled by the crawling agents. Thus, coverage defines the amount of work a crawling architecture is
performing relative to the size of the entire workload. A high coverage value indicates that the crawling architecture is not performing very well.

Overlap is defined as $\frac{N}{U}$. $N$ is the total number of crawled pages while $U$ is the total number of unique pages. Overlap defines the amount of duplicate work individual crawlers are engaging in. A high value of overlap indicates an inefficient system. As such, overlap should be kept at a minimal as a high overlap ratio will consume unnecessary bandwidth and storage.

Finally, communication overhead is considered as a major performance metric for distributed crawling architectures. Communication overhead is defined $\frac{E}{N}$. $E$ is the number of URL's exchanged while $N$ is the number of crawled resources (Najork and Wiener, 2001). Communication overhead is therefore defined as communication that is not concerned with the retrieval of the resource itself. High communication overhead values indicate an inefficient system.

### 2.3 System Architecture for Transmission of Resource Links

In a fully distributed system, it is necessary to send URI resource links to other crawling agents. There are several methods of achieving this as described in Parallel Crawlers, Cho and Garcia-Molina (2002): the firewall architecture, the cross-over architecture and the exchange mode architecture. The firewall architecture essentially ignores resource links assigned to different crawling agents and as such, it discards all non-relevant resource links. The advantage of the firewall architecture is zero communication overhead and overlap, but suffers from the disadvantage of lower
coverage. By comparison, the cross-over model retrieves all resources (even those that are assigned to other crawling agents) and as such, it achieves zero communication overhead and high coverage but suffers from a higher overlap ratio. Finally, the exchange model transmits relevant resource links to other crawling agents and as such, ensures high coverage and low overlap but it also has high communication overhead.

Particularly important is the consideration of the geographic structure of the crawling architecture. In a centralized crawling architecture, all the crawling agents are located physically in the same network. In a distributed crawling architecture, it is probable that crawling agents are placed at geographically different locations. There are some advantages to a geographically distributed crawling architecture (Shkапenyuk and Suel, 2002):

- There is higher fault tolerance when crawling agents are geographically distributed as they become independent of local network failures.
- Geographically separated crawling agents have the advantage of low latency crawls because crawling is performed to closely located resources. For example, a crawling agent in Malaysia would be able to crawl resources in Malaysia faster in comparison to a crawling agent based in England (due to intercontinental latency delays).

In addition, it should be considered that most of the links in a particular web page belong to the same host (Heydon and Najork, 1999). Only a small number of links are located outside the host and as such, only a small number of links need to be sent to other crawling agents. This directly leads to faster retrieval times and imposes minimal communication overhead.
Communications overhead are the same in a distributed and centralized crawling architecture. However, latencies are smaller in a centralized crawling architecture as the crawling agents are located within the same network. However, communications overhead is small compared to the size of the resource itself so communications overhead can be considered to be negligible in practice. Hence, communications overhead is a non-issue for distributed crawling architectures.

2.4 Scalability

Scalability in the context of crawling architecture is defined by the authors of Ubicrawler as the number of resources crawled per second per agent should be (almost) independent of the number of agents and that throughput should grow linearly with the number of agents (Boldi, Codenotti, Santini and Vigna, 2002). In distributed crawling architectures, scalability is achieved by adding crawling agents to the crawling architecture so as to cope with the increases in resources to crawl.

There are several documented approaches in building scalable distributed crawling architectures. The Ubicrawler architecture was designed expressly for scalability because as the number of Ubicrawler agents increase, the number of resources crawled increases in linear tandem. However, scalability in this scenario is a function of available hardware resource because where when hardware resources reach its threshold, performance of the crawling architecture will saturate.

The original Google paper (Brin and Page, 1998) described another way to build scalable crawling architectures. Google's approach was to use both distributed and centralized crawling architectures, whereby each crawling agent would receive URL
links to download from a central server and would pass the retrieved resource back to
the central server. This is a fairly efficient system as the number of crawling agents
increases linearly with the amount of retrievals. However, it has the possibility of the
central server potentially becoming a bottleneck which can affect the scalability of the
system (Heydon and Najork, 1999).

Furthermore, a scalable system allows for the addition of new crawling agents
without affecting the operation of a running system. The reasons for adding new
crawling agents is to reduce and balance the load of the current crawling agents. An
example to illustrate load reduction would be a scenario where a three crawling agent
system having a load of 33% would see their load reduced to 25% each in a four
crawling agent system. With this goal in mind, the new crawling agent should be
architected to be identical to the existing crawling agents with the only distinguishing
factor being configuration options for the new agent.

2.5 URL Partitioning

It is necessary to implement URL partitioning capability in a distributed
crawling architecture and assign the partitioned URL's to individual crawling agents.
The goal of URL partitioning is to reduce overlap and thus increase efficiency. Reduced
overlap implies lower bandwidth, storage and processing usage. The designers of
Ubicrawler (Boldi, Codenotti, Santini and Vigna, 2002) noted that there are three goals
to meet in the URL partitioning scheme:

- Each URL must only be assigned to a single crawling agent. This implies that
  only the crawling agent must be responsible for retrieval of the URL and
  subsequent processing. This goal removes the possibility of multiple retrievals
of the same URL by different crawling agents, and thus ensures high efficiency of the crawling architecture.

● Each crawling agent must be able to independently determine the crawling agent responsible for the given URL and most importantly, the crawling agent must be able to determine this without communicating with other crawling agents. This goal aims to increase coverage of the crawling architecture.

● The assignment of URL's between crawling agents must be equal to ensure equal distribution and thus, equal load between each crawling agent.

The most obvious method of distributing URL assignments is the Google architecture, which is to have a central server to perform the task. A central server approach has the advantage of simplicity and clearly ensures that each crawling agent has a unique URL assignment. However, a central server architecture requires more communication overhead and reduces independence on the part of the crawling agent.

Another popular method that allows for independent URL partitioning by crawling agents in a distributed crawling architecture is the hashing algorithm applied on an aspect of the URL. Each crawling agent is assigned on an equal basis a portion hash space as part of their configuration. The crawling agent then applies a hash to the URL and checks the resultant hash against the assigned hash spaces of the crawling agents. This method meets all the goals stated above and has the advantage of being easy to implement and efficient to execute. There are many hashing algorithms that can be used (such as MD5) and most are widely implemented as optimized third party libraries which can be integrated into the crawling agent implementation without any problems.
2.6 Description of Really Simple Syndication (RSS) formats

All the syndication formats repackage content as a list of data items, such as a list of news stories or a list of latest blog posts. The Really Simple Syndication (RSS) format defines the structure of the data items. At the top level of a RSS document is a <rss> element, with a mandatory attribute called version, that specifies the version of RSS that the document conforms to. Subordinate to the <rss> element is a single <channel> element, which contains information about the channel (metadata) and its contents. Within the <rss> element are also content items, each of them referred to as <item>. Within each <item>, the structured content is presented along with the relevant metadata. An example of an RSS file is presented in Appendix A.

Using widely available XML parsing tools, the file can be easily represented as an associative array, and using direct object model mapping techniques, it can be stored in the database for further manipulation.

2.7 The RSS Specification

The original version of RSS was version 0.90 and was introduced by Netscape (King, 2007). Userland introduced version 0.91 which was widely adopted for basic syndication. Versions 0.92, 0.93, 0.94 which followed had richer metadata and defined the basic syndication metadata that all succeeding syndication formats inherited (King, 2007).

The key XML tags of interest in the 0.9x RSS formats are listed in Table 2.1 (Winer, 2000):
### Table 2.1
Key XML Tags in RSS 0.9x

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>The title refers to the name of the channel.</td>
</tr>
<tr>
<td>Link</td>
<td>The link is the URL to the HTML website corresponding to the channel.</td>
</tr>
<tr>
<td>Description</td>
<td>The description is would be a phrase or sentence describing the channel.</td>
</tr>
<tr>
<td>Language</td>
<td>The language tag specifies the language the channel is written in.</td>
</tr>
<tr>
<td>Publication Date</td>
<td>The publication date refers to the publication date of the content</td>
</tr>
<tr>
<td></td>
<td>in the channel. The pubDate conforms to the Date and Time Specification of</td>
</tr>
<tr>
<td></td>
<td>RFC 822 with the exception that the year may be expressed with two</td>
</tr>
<tr>
<td></td>
<td>characters or four characters (four preferred).</td>
</tr>
<tr>
<td>Items</td>
<td>A channel contains any number of &lt;item&gt;s. An item effectively represents</td>
</tr>
<tr>
<td></td>
<td>textual content elements. An item has its own set of tags, the most</td>
</tr>
<tr>
<td></td>
<td>important of which are description (which contains the text). It also</td>
</tr>
<tr>
<td></td>
<td>optionally contains the link to the item and a title.</td>
</tr>
</tbody>
</table>

These items are wrapped in the channel XML tag as described in Code Listing 2.1 (Winer, 2000):

```
<channel>
  <title>Example Channel</title>
  <link>http://example.com/</link>
  <description>an example feed</description>
  <language>en</language>
  <rating>(PICS-1.1 "http://www.classify.org/safesurf/" l r (SS--000 1))</rating>
  <textinput>
    <title>Search this site:</title>
    <description>Find:</description>
    <name>q</name>
    <link>http://example.com/search</link>
  </textinput>
  <skipHours>
    <hour>0</hour>
  </skipHours>
  <item>
    <title>1 &lt; 2</title>
    <link>http://example.com/1_less_than_2.html</link>
  </item>
</channel>
```
RSS 2.0 is syntactically and semantically similar to RSS 0.9x. However, it removed several elements that were confusing and not widely used. Several minor semantic changes were also made in the format (Winer, 2003).

2.7.1 RSS 1.0

The RSS 0.9x and the RSS 2.0 versions are fairly similar. However, there is another version of RSS that uses Resource Description Framework (RDF), termed RSS 1.0 (RSS in the 1.x version is a short form for RDF Site Summary). The RSS 1.x series aimed to solve the problem of ad-hoc extensions to the RSS format. It does so by the compartmentalization specific functionality into the pluggable RSS modules. This is achieved by using XML namespaces for vocabulary partitioning. This ensures that the core specification is not continually reinvented. It also does not bloat the RSS feed with elements that are not used. XML namespaces also ensure that collisions do not exist.

The core syntax of RSS 1.x is backwards compatible with RSS 0.9x. This is achieved in the following manner: firstly, no RSS 0.9x attributes are outside of the RDF namespace declaration; secondly, elements of modularized extensions are outside of the default namespace. In addition, schemes that are compatible with RSS 0.9x are allowed, including http, https, ftp and mailto.

The mimetype of RSS 1.0 is set to application/xml and file extensions to RSS 1.0 feeds are not required. It differs from RSS 0.9x by assuming a UTF-8 encoding.
whereas RSS 0.9x only supports ASCII encoding.

2.7.2 Atom

In 2003, a new syndication format was proposed as there was some level of unhappiness over the RSS specification (Ruby, 2003). In particular, the following objections (Trott, 2003) were made against the RSS specification:

- No way to specifying encoding mechanisms. Non English content can be represented in Unicode formats but RSS 0.9x had no standardized mechanism to store Unicode information. In particular, the feeds may have non-textual elements and content type declarations become of utmost importance. Atom was created, in part, to allow for a standardized mechanism to represent content encoding mechanisms.

- In addition, there was a confusion over the semantic of elements. For example, the <link> element was not clearly defined as some treated it as a permanent link and others as a temporary link. In addition, in some cases, namespaces were used and others, relative items were used. Atom seeked to standardize the semantics of the tags.

- Finally, there was no universally accepted mechanism in RSS to be able to define and include extensions. This was addressed to some extent by RSS 1.x but was not adopted by RSS 0.9x and RSS 2.x series.

Atom fixes these issues and provides a more generic API for the manipulation of web resources. However, at the core of the API and Atom feed, the same set of
attributes that are present in RSS are used. Hence, using an abstraction layer, the same set of attributes present in RSS and Atom can be mapped to a common naming scheme and be accessed accordingly.

2.8 Process Forking and Threading

Threads and processes are two methods of achieving concurrency in programming. The primary difference between threads and processes is that processes are independent, have separate address spaces and communicate via inter process communication mechanisms. As threads run in the context of a single process, they have the same address space and share the memory directly.

There is significant debate as to using processes over threads (Lee, 2006). Among the issues considered is that the usage of threads include the need for the careful use of shared resources as concurrent execution of threads accessing a shared data source leads to race conditions, deadlocks and other problems. In addition, threads do not allow the use of hardware memory protection systems (because threads run within a single address space) and this can cause substantial runtime issues.

2.9 Process Forking with PHP

PHP provides an ability to use process control functions via the PCNTL extension (PHP Documentation Group, 2007). Function calls in the PCNTL extension are prefixed with “pcntl_”. Process control support in PHP is implemented as in the Unix style of program execution, process control, process termination and signal handling. As such, process control is not available on non-Unix platforms such as MS-Windows. Furthermore, process control functionality is to be used only in the context of
the standalone SAPI PHP client and not when mod_php is used in the context of the webserver.

PCNTL uses the concept of ticks as the signal handle callback mechanism. The declare() statement is used to specify locations in the program where callbacks are allowed to occur. This minimizes the overhead of handling asynchronous events.

2.10 Forking processes and the pcntl_fork() function call

The pcntl_fork() function call is probably the most important of all the PCNTL functions provided. This function call effectively forks the current process and the child process only has a different process ID (PID) and parent process ID (PPID). If the fork was successful, the PID of the child process is set to an integer 0 in the context of the child and the PID of the child in the parent. On failure, an integer value of -1 is returned to the parent and a PHP error is raised.

The child process is exactly the same as the parent process and continues processing after the function call. A copy of the parent process's memory is made for the child process and as such, all variables, objects and resources of the parent are available to the child process.

2.10.1 Zombie child processes and pcntl_waitpid() function call

If the parent process exits before the child processes, the child processes will be known as zombie processes as they do not belong to any particular process. As such, the parent process needs to wait until all the child processes have finished running before exiting itself.
The `pcntl_waitpid()` function will wait and return the status of a forked child process (PHP Documentation Group, 2007). Any system resources used by the child process are freed. An example of doing this is noted as in Code Listing 2.2:

```
Code Listing 2.2
pcntl_waitpid() example

$pid = pcntl_fork();
if ($pid) {
    pcntl_waitpid($pid, $status, WUNTRACED);
    exit;
}
```

### 2.10.2 Installing a signal handle using `pcntl_signal()` function call

A software signal is a software based interrupt delivered to a process (Free Software Foundation, 2007). Signals can be used to report errors, exceptional situations or even custom defined events. As such, signal handling support in a process allows system administrators to control process behavior. For example, if a process needs to be terminated, a termination signal can be sent to the process by the system administrator. The process receiving this signal is able to capture the signal and take appropriate steps (for example, saving important data) before terminating. Signal handling is crucial in a distributed crawling architecture to ensure processes can be terminated without causing data corruption.

The PCNTL extension allows for a custom signal handler be set up in PHP as is illustrated in Code Listing 2.3 (PHP Documentation Group, 2007):
Code Listing 2.3
pcntl_signal() example

```
declare (ticks = 1);

function signal_handler($signo) {
    switch ($signo) {
        case SIGTERM: // Handle shutdown tasks
            break;
        case SIGINT: // Handle interrupt signal
            break;
        default: // All other signals – do nothing
            break
    }
}

pcntl_signal(SIGTERM, “signal_handler”);
pcntl_signal(SIGINT, “signal_handler”);
```

2.10.3 Exit codes

A child can return an exit value what can be captured by the `pcntl_waitpid()` function call. This is accomplished by setting the status variable (the second argument) in the `pcntl_waitpid()` function call.

2.11 Testing and Instrumentation Tools

Testing involves identifying the bottlenecks in the system and the precise use of
instrumentation is required to identify bottlenecks. The bottlenecks are generally either CPU, RAM or disk based I/O. Several common POSIX compatible tools can be used to identify these bottlenecks in the execution of the crawling architecture as a whole.

2.11.1 top

The top program provides a dynamic real-time view of a running system (Linux User's Manual, 2007). It can display system summary information as well as a list of tasks currently being managed by the Linux kernel. The types of system summary information shown and the types, order and size of information displayed for tasks are all user configurable and that configuration can be made persistent across restarts. The program provides a limited interactive interface for process manipulation as well as a much more extensive interface for personal configuration.

For monitoring purposes, it is important to have the following information of currently executing processes:

- PID: process ID
- VIRT: VIRT refers to the total amount of virtual memory used. Virtual memory is defined as SWAP + RES. VIRT includes all code, data, shared libraries and pages that may have been swapped out
- RES: RES refers to the physical memory that has not been swapped out
- SWAP: SWAP refers to the portion of memory that have been swapped out in VIRT
- SHR: SHR refers to the shared memory that is used. This generally includes dynamically loadable libraries.
- CODE: CODE refers to the memory used that is executable code.
- DATA: DATA refers to memory used that is marked non-executable.
- %CPU: CPU usage in percentage of total CPU capabilities.
- %MEM: Memory usage in percentage of total memory.
- TIME+: CPU time used in hundreds.

\textit{top} is set up to reflect this information using a configuration file as noted in Appendix 2. A screenshot of top in action is shown below in Figure 2.3:

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{top_in_action.png}
\caption{\textit{top} in action}
\end{figure}
2.11.2 vmstat

*vmstat* is a very important tool that is used for reporting virtual memory statistics (Linux User's Manual, 2007). It can report statistics on processes, memory, paging, block I/O, traps and CPU activity. In addition, vmstat can also give a running real-time statistics on the system. The first report that it gives has the average statistics since the last reboot. All additional reports give real-time statistics after that point.

*vmstat* is particularly useful in displaying information about disk I/O. *vmstat* reports on the rate of blocks received from a device per second (labelled as “bi”) and the rate of blocks sent to a device per second (labelled as “bo”). A screenshot of vmstat in action is shown below in Figure 2.4:

![Figure 2.4](image.png)

**Figure 2.4**

*vmstat* in action
2.11.3 PostgreSQL Tuning

In a distributed crawling system, a storage system is required. This dissertation uses a relational database as a storage system, in particular the PostgreSQL database server. To ensure that SQL queries that are run are suitably optimized, it is necessary to analyse the SQL query and the execution plan created by PostgreSQL when executing the query, and generate necessary indexes to ensure optimum execution speed.

2.11.3.1 Database Indexing

Indexes are created by using the CREATE INDEX command (PostgreSQL Global Development Group, 2007). With this command, the key fields for the index are specified as column names and multiple fields are specified in the cases where multicoloumn indexes were required. This allows for fast access to data fields in the system. While there were several indexing methods provided by PostgreSQL, including B-tree index, R-tree index, hash index and GiST index, the B-tree index method is generally sufficient for traditional database applications.

2.11.3.2 Analyzing Queries

The `explain` command works by displaying the execution plan generated by PostgreSQL's planner for the SQL query given to it (PostgreSQL Global Development Group, 2007). The PostgreSQL planner has several methods of scanning the database tables such as plain sequential scan and the index scan.

A sequential scan is a sequential lookup of all the records in a table to see if each
record matches the query constraints. An indexed scan, on the other hand, stores indexes for certain columns and if the query constraints match the indexed column, PostgreSQL can skip non-relevant records and directly read the relevant records. The planner will reveal whether PostgreSQL will execute a sequential scan or an index scan.

The most important portion of the output from the planner is the estimated statement execution cost. This allows the user to determine the length of the execution of the SQL query. This value is measured in units of disk page fetches. There are two numbers that are shown, start-up time and total time to execute the query, as illustrated in Code Listing 2.4:

```
Code Listing 2.4
Debugging PostgreSQL Queries

psql> EXPLAIN SELECT * FROM feeddata id = 4;

QUERY PLAN
---------------------------------------------------------
Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)

psql> EXPLAIN SELECT * FROM feedcontent WHERE id = 4;

QUERY PLAN
--------------------------------------------------------------
Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
  Index Cond: (id = 4)
(2 rows)
```

As can be seen, the first query had no constraints and the PostgreSQL planner used a sequential scan. In the second query, PostgreSQL used an index scan as the constraint column id was already being indexed.
The numbers quoted by the explain command reveals cost estimates by the PostgreSQL planner. \( cost=0.00..155 \) means that the start up cost is 0.00 and the total cost is approximate 155 disk fetches. As can clearly be seen, in the case of sequential scans, more disk fetches are required then indexed scans. As such, indexed scans should be used by the PostgreSQL planner whenever possible if the tables are indexed correctly. The explain command can thus help the implementator to index the correct columns.

### 2.12 Conclusion

From the literature survey conducted of traditional crawling architectures, it is clear that there are some limitations with existing crawling architectures. In particular, the centralized crawling architecture leads to high communication overheads whereas the distributed crawling architectures make extensive use of threading methodologies between components which lead to data corruption bugs and other run time problems. A process forking methodology was introduced which allows for concurrent execution without the problems introduced by threading methodologies.

The RSS and Atom syndication formats were also outlined in detail. These syndication formats will be the target of a process forking data driven crawling architecture (to be introduced in Chapter 3), whose objective is to solve the traditional crawling architecture problems noted above. Several instrumentation techniques were also introduced in the Literature Survey. The instrumentation tools allow for fine grain monitoring, debugging and optimization. Such techniques are integral to architecting and building the process forking data driven crawling architecture.
Chapter 3: Proposed System Architecture

The proposed distributed crawling architecture is structured as a data driven architecture. The traditional definition of a data driven architecture is that the architecture and execution path is driven by data structures used, and not the actual program logic. This is a wide, encompassing definition of a data driven architecture but for the purposes of this dissertation, a more concise definition is required in the context of distributed crawling architectures.

Thus, this dissertation contributes a new narrow definition of a data driven architecture to be more suitable for distributed crawling architectures:

A data driven architecture is a software architecture whereby each constituent functional element of the system is expressly designed to execute independently without any form of direct communication (such as IPC, RPC shared memory etc) with other elements in the system, and that each element only executes on the existence (or, as the case may be, the non existence) of assigned and partitioned data sets.

As per the definition, the crawling architecture must be architected to have independent elements which encapsulate islands of functionality which can be executed autonomously with only a single dependence on access to data sets. This dissertation defines such islands of functionality as operations and the importance of independent execution as the Principle of Independent Execution of Operations.

To ensure that the definition above takes into account the architecture of
distributed systems, this dissertation requires that the data sets be able to be partitioned cleanly to each operation. This requirement is necessary as it allows for data sets to be consumed without conflicts by operations in a distributed crawling architecture.

3.1 Defining Operations

As provided in the narrow definition in the introduction to Chapter 3, the data driven distributed crawling system is structured around a set of independent operations that are continually executing. This definition of operations is directly derived from the well accepted Principle of Modularity (McIlroy, Pinson and Tague, 1978) on Unix systems which states that:

“Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.”

The objectives for each operation is for the independent execution as this will ensure that:

- operations can focus on only a single task (leading to clarity of purpose in the system architecture)
- operations can execute without run time dependencies on any other operation in any machine in distributed crawling system
- operations can asynchronously execute without waiting for the completion of another operation (this removes much of the delay inherent in synchronous mechanisms such as IPC or RPC)
- the run time logic of operations can be modified without impacting another
static code coupling and execution coupling between different parts of the system is eliminated

the lack of global shared memory removes subtle data corruption issues common in a threading methodology (Lee, 2006)

Hence, each operation is defined to only execute on a single task on a set of partitioned data.

3.2 The Principle of the Independent Execution of Operations

The proposed system is designed around the fundamental Principle of Independent Execution of Operations. Every operation in the system is designed so as to remove (wherever possible) and reduce coupling as well as dependencies on the completion of other operations. To achieve this, the following system design criteria is proposed:

- Operations cannot explicitly or implicitly call other operations in the system. This includes direct function calls, remote procedure calls and direct execution of binaries. However, operations that are external to the system and do not access the data sets can be executed directly.

- Data sets that operations work on must be cleanly partitionable and not be assigned to any other operations at the same time. Wherever possible, operations should not access data sets external to the operation's assigned data sets as there cannot be a guarantee on the integrity of the external data sets. However, if the operation is required to access data in external data sets, the operation should restrict its access, wherever possible, to non blocking read access only. If write
access is required, necessary locks must be set correctly to avoid data corruption.

- The physical location where operation is being executed is irrelevant, as long as it is able to allow external operations to access its data in an equitable manner. This allows for geographically distributed systems across different networks.

- The actual deployed environment where the operation is deployed is irrelevant, as long as it is able to allow external operations to access its data in an equitable manner. This implies that there are no run time restrictions in the distributed system.

### 3.3 Components

Components are objects that are written to a specification and are encapsulated as objects. Components have to fulfill certain criteria (Szyperski, 2002) such as composability, independence, non-context specific and encapsulation.

Given the noted criteria for components, this dissertation identifies components as non-contextual objects providing necessary functionality to all operations. Components provide operation-agnostic functionality to all components and do not participate in operation specific functionality. This allows for components to be re-used across operations.

### 3.4 Proposed System Architecture

The high level view of the proposed layered system architecture is outlined as in Figure 3.1:
As can be seen in Figure 3.1, operations run in parallel and utilize components which in turn rely on the base system. The base system is composed of the database, operating environment and operating systems that supports the execution of the operations and components. This system can be distributed by running each operation on a separate machine on the network as indicated in Figure 3.2:

### Figure 3.2
Proposed Distributed System Architecture

#### 3.5 Proposed Functional Architecture

As the proposed architecture of the distributed crawling system is built around
operations, this dissertation identifies the following operations which are required for running the crawling system:

- Retrieving web feed content
- Analyzing content for new URI links
- Retrieving URI link contents
- Feed aging

The context diagram for the proposed crawling system can be defined as in Figure 3.3:
A data flow diagram can be derived from the context diagram, outlining the overall proposed functional structure as shown in Figure 3.4:

Figure 3.4
Data Flow Diagram of the Proposed Crawling System

In Figure 3.4, operations interact with the database to get a list of data set to operate on and save relevant results back into the database. In addition, there are two operations which interact with feeds and URI links on the Internet (shown as a cloud bubble).

3.5.1 Retrieving web feed content

The retrieval of web feed content is a key operation in the distributed crawling
system. The distributed system defines a feed queue which stores a set of feeds that need to be retrieved. The feeds are retrieved using the downloader component.

### 3.5.2 Retrieving URI link contents

URI links are websites which need to be parsed in order to discover new feed URL's. The distributed system defines a link queue which stores a set of links that need to be retrieved. The links are retrieved using the downloader component.

### 3.5.3 Analyzing content for new URI links

Feed content will be parsed for URI links (in most cases, web pages) and new URI links will be added to the link queue.

### 3.5.4 Feed aging

As feed content is updated frequently, feeds must be aged such that they are inserted back for retrieval into the feed queue on a periodic basis. The feed aging operation achieves this by using a configurable time-out value and re-inserts feeds into the queue when the time out value is reached. The time-out value should not be too short so as to avoid aging the feed prematurely.

### 3.6 Component Architecture

This dissertation also identifies the following components required by the operations:
3.6.1 Feed Analysis Component

The feed analysis component encapsulates a URL extractor subcomponent and a feed auto discovery protocol subcomponent. The URL extractor component is used to extract URL's from a web page or feed content using regular expressions. The feed auto discovery protocol discovers feeds defined on web pages. Both these components are used by the operations to auto detect new feeds and to retrieve URL's.

3.6.2 Error Handling Component

The distributed crawling system requires a specific error handling component for all operations to report errors encountered in a systematic fashion, consolidates all error handling in a consistent fashion and as such, allows for error recovery in a consistent fashion too. From a software engineering perspective, this functionality allows for more efficient debugging capabilities.

3.6.3 Database Abstraction Component

The distributed crawling system specifies a database abstraction layer which encapsulates differences in database handling of standard SQL and allows for single consistent API. The database abstraction layer is designed to fulfill the following
considerations:

- does not impose unnecessary overhead. As every database query will be channeled through the database abstraction layer, it is necessary that the overhead imposed by the database abstraction layer is minimal.
- mature and widely used. As the database abstraction layer impacts every operation, it is necessary that the database abstraction layer is stable and bug-free. Maturity is generally a pre-condition to this.
- portable to different databases. To allow for the possibility that the system may be ported to a different database at a later point, it is necessary for the database abstraction layer to provide the capability to change databases with minimal trouble.

The database abstraction works on two levels:

i. On a lower level, it encapsulates the database queries (such as starting and ending transactions, executing INSERT/UPDATE operations via wrapper functions).

ii. On a higher level, it exposes non-stateful objects which are used by the operations to query, add, update and remove rows from the database.

3.6.4 Downloader Abstraction Layer

The downloader abstraction layer component is used to retrieve web feeds, parse the web feeds and present the parsed data in a readily usable manner to operations. The downloader component must fulfill the following considerations:
● support for RSS and Atom feeds
● in line with Postel's Law\(^6\), strict validation of the RSS feeds is not required.
● ease of integration with the proposed architecture.
● to remove possible network bottlenecks:
  ○ automatic caching of parsed RSS is required.
  ○ support of conditional GET with Last-Modified, and Etag HTTP headers.

The Last-Modified HTTP element in the response headers allow us to decide whether the server from which the RSS feed is being retrieved has an updated feed.

### 3.7 Conclusion

The individual elements outlined in the overall system architecture defined in Figure 3.1 has been defined in this chapter. As such, the defined elements are entered into the system architecture as in Figure 3.5:

---

6 Postel's Law (Postel, 1981) (also known as the Robustness Principle) argues that programs should always be liberal in accepting input and be strict in its output. This principle ensures that programs will be able to handle improper input.
The implementation details of all operations, components and the base system will be looked into more detail in the next chapter.
Chapter 4: Implementation Details and Algorithmic Techniques

Based on the system architecture outlined in Chapter 3, a proof of concept distributed crawling system was built using open source technology. This chapter outlines the selection criteria of software used in the base system, the architecture of the base system itself as well as the implementation strategy and algorithms used for the operations and components.

4.1 Base system

The base system requires the following architectural components:

- Language and operating environment
- Database
- Operating System

A graphical illustration of the architecture is indicated in Figure 4.1:
4.1.1 Operating System

The consideration used for the selection of an operating system were as follows:

- strong and mature support for process forking instead of process threading. This is necessary as outlined in Section 2.9.

- strong and mature networking library. The operating system must provide a sufficiently mature and bug free networking stack that can scale as the system grows in size.

- Mature and useful tools to debug use of disk and network I/O as well as process CPU and memory use.
- POSIX system calls. We adhere to POSIX system calls that ensures that we can port the system to other operating systems that support POSIX if required.

Based on this criteria, Table 4.1 outlines a cross sectional analysis of popular operating systems available today:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MS Windows 2003 Server</td>
<td>Yes</td>
<td>Immature</td>
<td>Good</td>
<td>Average</td>
<td>Not built in</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>Yes</td>
<td>Mature</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Built in</td>
</tr>
<tr>
<td>OpenBSD</td>
<td>No</td>
<td>Mature</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Built in</td>
</tr>
<tr>
<td>NetBSD</td>
<td>No</td>
<td>Mature</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Built in</td>
</tr>
<tr>
<td>Linux</td>
<td>Yes</td>
<td>Mature</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Built in</td>
</tr>
</tbody>
</table>

From the table, it is clear that FreeBSD and Linux are suitable to the requirements and as such, given Linux's popular use in the academic sector, the most current version was chosen at the time of implementation (2.6.20) for implementation purposes.

### 4.1.2 Operating Environment

An operating environment differs from an operating system in the context of running applications. For example, an operating environment for a systems application would be the C language, necessary development environment (such as the compiler, linker and assembler), the C library as well as any other necessary libraries as well as support for POSIX system calls. Hence, for the purposes of the operating environment

---

7 POSIX stands for Portable Operating System Interface is defined by the IEEE for a standard API of operating systems.
for the crawling system, the following criteria was used:

- Support for POSIX system calls. It may be required to port the crawling system to another operating system at some point in the future and the only way to retain compatibility with the new system is to ensure that full POSIX support is always required in the design criteria.

- Quick and rapid prototyping support. To allow for rapid prototyping, it is necessary to use scripting languages to highlight feasibility of a particular feature. As an example of use of scripting languages in the industry, Google has noted its use of the Python scripting language (Python Foundation, 2007). Rapid prototyping necessarily rules out the use of compiled languages.

- Strong database support. The operating environment should provide stable and mature database binding libraries to the database of choice.

- Strong text processing and regular expression support. As the crawling system will be primarily processing text retrieved from web feeds, strong and efficient text processing functionality is required. Text processing also include regular expression support.

- Comprehensive online documentation.

- Open source language. This is necessary to ensure interoperability, lack of vendor lock in and the ability to extend the language platform if required.

There are three languages that fit this criteria: Python, PHP and Ruby. Of the three, PHP was chosen as:

- Its online documentation is far more comprehensive compared to Python and Ruby.
- It has powerful extensions that are not necessarily found in the other languages.
- It is the most mature of all the three languages. This translates into stability, high performance and an extensive feature set.
- Ease of use. PHP has the lowest barrier of entry compared to the other languages.

4.1.3 Database

The database is the central repository of information. As it is used widely in every aspect of the system, the database play an important and central role in the crawling system. The design criteria for the selection of the database is:

- ACID compliance (Gray, 1981). ACID compliance refers to the support for Atomicity, Consistency, Isolation and Durability which are the basic building blocks in ensuring that the database executes SQL queries in safe transactional manner.
- Stable and mature. It is important that the database is capable of handling high load environments without crashing.
- Open source. This is necessary to ensure interoperability, lack of vendor lock in and the ability to extend the platform if required.

While there are many open source databases, there are only two databases that are open source and are sufficiently mature in terms of developing history: MySQL and PostgreSQL. However, MySQL is known to crash under high load environments and as such, PostgreSQL was chosen as the database platform.
4.2 Overall System Architecture

Given the identification of base elements in Figure 4.1, the overall system architecture is outlined as in Figure 4.2:

![Figure 4.2](image)

System Architecture with Identified Base Elements

<table>
<thead>
<tr>
<th>Retrieving web feed content</th>
<th>Analyzing content for new URI links</th>
<th>Retrieving URI link contents</th>
<th>Feed Aging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feed Analysis</td>
<td>Download Abstraction Layer</td>
<td>Downloader Abstraction Layer</td>
<td>Error Handling</td>
</tr>
<tr>
<td>Database Abstraction Layer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operating Environment (PHP 5.2 CLI)

Database (PostgreSQL 8.1)

Operating System (Linux kernel 2.6.20)

4.3 Implementation of Operations

The implementation of operations follows the Principle of Independent Execution of Operations as elucidated in Chapter 3. To ensure independent execution, the following execution model will be adopted for all operations:

1. Data sets are divided among machines on the distributed system using URL partitioning methodology.
2. Parent process retrieves list of data sets from database.
3. Parent process forks child processes and assigns the child processes a subset of
4. Child process operates on the assigned subset of data sets and exits when completed.

4.3.1 **URL partitioning methodology**

The URL partitioning methodology is built on the following algorithm:

1. The MD5 hash is a hexadecimal string with 32 hexadecimal characters. This means that it can represent $2^{128}$ unique hash values. This is a sufficient number to represent hosts on the Internet uniquely.

2. The MD5 hash space is divided equally to all hosts on the distributed network and each host is assigned a unique MD5 hash space.

3. Each URL that the crawler encounters is divided into its constituent parts:
   - protocol
   - host
   - port
   - user
   - password
   - full path and query

4. Only the host portion of the URL is hashed using the MD5 algorithm and all other portions are discarded.

5. The resultant MD5 hash is converted from hexadecimal to decimal.

6. The decimal number is stored in the database as a reference to the URL in question.

In this manner, each and every URL encountered will be partitioned into one of
the unique partitioned hash areas. As such, it is possible for operations to
deterministically retrieve data sets whose URL's are within the hash space assigned to
the crawler.

4.3.2 Retrieving web feed content operation

This operation works on a queue which stores a set of feeds which need to be retrieved and stored. The algorithm for this operation is as follows:

1. The global configuration is attempted to be read. If all relevant configuration parameters are present, the processing continues.
2. A database connection is attempted. If the database connection is successful, the processing continues.
3. A list of feeds to be processed is retrieved. The list of feeds is matched to assigned hash values of the crawler.
4. A child process is forked for each data set.
5. The parent process keeps looping and forking off child processes as soon as there are sufficient number of feeds for the child process.
6. For each feed, the downloader component is used to retrieve the feed content.
7. The feed content is then saved in a database transaction.

This algorithm is outlined in a flowchart in Figure 4.3.
4.3.3 Retrieving URI link contents operation

URI links are websites which need to be parsed in order to discover new feed URL's. Links have a queue of their own which is emptied by the link emptying
operation algorithm as follows:

1. The global configuration is attempted to be read. If all relevant configuration parameters are present, the processing continues.

2. A database connection is attempted. If the database connection is successful, the processing continues.

3. A list of links to be processed is retrieved. The list of links is matched to assigned hash values of the crawler.

4. A child process is forked off for each data set.

5. The parent process keeps looping and forking off child processes as soon as there are sufficient number of links for the child process.

6. For each links, the Feedfinder component is used to retrieve the new links in the content.

7. New links are then saved in a database transaction.

The algorithm is outlined in a flowchart in Figure 4.4.
4.3.4 Analyzing content for new URI links operation

Feed contents are analyzed to extract URI links. URI links are added to a link queue which are parsed by the “Retrieving URI link contents” operation. The algorithm of this operation is as follows:

1. The global configuration is attempted to be read. If all relevant configuration parameters are present, the processing continues.
2. A database connection is attempted. If the database connection is successful, the processing continues.
3. A list of feed contents to be processed is retrieved. The list of feeds contents is matched to assigned hash values of the crawler.
4. A child process is forked off for each data set.
5. The parent process keeps looping and forking off child processes as soon as there are sufficient number of feed contents for the child process.
6. For each content item, the Feedfinder component is used to retrieve the new links in the content.
7. New links are then saved in a database transaction.

The algorithm is outlined in a flowchart in Figure 4.5.
Figure 4.5
Flowchart of Analyzing Content for New URI Links Operation
4.3.5 Feed aging operation

Given that feed content is updated frequently, feeds are aged so that after a certain period of time, they are re-inserted back into the queue for retrieval. The algorithm of this operation is as follows:

1. The global configuration is attempted to be read. If all relevant configuration parameters are present, the processing continues.
2. A database connection is attempted. If the database connection is successful, the processing continues.
3. A list of feed which are older then a defined time value are retrieved.
4. The feeds are re-inserted into the queue if they do not exist in the queue.
5. The new queue items are saved in a database transaction.

The algorithm is outlined in a flowchart in Figure 4.6.
Figure 4.6
Flowchart of Feed Aging Operation

1. Read Configuration File
2. If Configuration File Read and Parsed Successfully:
   a. Open database connection
   b. If Database connection successful:
      i. List of feeds older than a defined time value are retrieved
      ii. Retrieve Feed Contents
      iii. Suspend execution for predefined number of seconds
      iv. Is the TERM signal received?
      v. Yes: Exit
      vi. No: Continue
   c. No: Exit
3. No: Exit
4.4 Components

As previously defined in Section 3.3, components must be non context specific and must be designed in such a manner so as to allow usage by all operations. As such, all components are wrapped around a custom wrapper to allow for easy access by operations.

4.4.1 Downloader Component: Magpie RSS

The downloader component is Magpie RSS which is encapsulated in an object oriented manner. MagpieRSS is an open source XML-based RSS parser in PHP which is used to provide the abstraction layer. MagpieRSS provides the following functionality:

- supports RSS 0.9 and RSS 1.0, with limited RSS 2.0 support. The limited RSS 2.0 support is not an issue as the required content for this system is covered by the 2.0 support.
- does not provide strict validation of the RSS feed retrieved. To be completely inclusive, this dissertation requires the exclude validation of the RSS feed.
- simple, functional interface, to object oriented backend parser. The interface is fairly easy to use and follows the PHP naming methodology, which makes building the wrapper code for MagpieRSS very simple.
- automatic caching of parsed RSS objects makes its easy to integrate. RSS feeds retrieved are cached to ensure that the load on the servers are kept to a minimum.
- supports conditional GET with Last-Modified, and Etag. The Last Modified HTTP element in the response headers allows for a decision on whether the
server from which the RSS feed is being retrieved has an updated feed.

4.4.1.1 Using MagpieRSS

The code snippet in Code Listing 4.1 illustrates the use of MagpieRSS:

```
require_once(rss_fetch.inc);
$url = $_GET['url'];
$rss = fetch_rss( $url );
```

fetch_rss() returns an object wrapper to an associative array which contains the RSS feed values.

4.4.2 URL Extractor Component: FeedFinder

Feedfinder is a Python based application used to find the RSS or Atom feed for a Web page. The algorithm for FeedFinder is as follows:

1. At every step, feeds are minimally verified to ensure that they are really feeds.
2. If the URI supplied is a feed, it is returned. Else, the page given is downloaded for analysis.
3. Using the auto discovery protocol, feeds which are pointed to by LINK tags in the header of the page are considered.
4. The `<A>` links to feeds on the same server ending in ".rss", ".rdf", ".xml", or ".atom" are considered as feeds.
5. The `<A>` links to feeds on the same server containing "rss", "rdf", "xml", or "atom" are considered as feeds.
6. `<A>` links to feeds on external servers ending in ".rss", ".rdf", ".xml", or ".atom" are considered as feeds.

7. `<A>` links to feeds on external servers containing "rss", "rdf", "xml", or "atom" are considered as feeds.

8. Some guesses are tried about common places for feeds (index.xml, atom.xml, etc.).

9. Finally, Syndic8\(^8\) is searched for feeds matching the URI

The flowchart for FeedFinder is described in Figure 4.7.

---

\(^8\) Syndic8.com is a community driven effort to gather syndicated news headlines
4.4.2.1 Using FeedFinder

feedfinder.py is executed via an external exec() PHP function call and its results are used to find new feeds. Code Listing 4.2 outlines the usage of FeedFinder.

```
exec("python feedfinder.py diveintomark.org")
```

and the return value is would be “http://diveintomark.org/xml/atom.xml”

4.5 Database Component

The database is a crucial base component in a database driven system. The database must be set up to allow multiple machines to access the system and must allow multiple connections even from one machine.

4.5.1 Database ER Diagram

The ER diagram of the overall system is as defined in Figure 4.8.
Figure 4.8
ER Diagram of the Database for the Crawling System
4.5.2 **Data Dictionary**

The ER diagram in Figure 4.8 outlined the relationship between the tables. The data dictionary of the database tables are as follows.

**Table:** feed_content

**Description:** This table stores all feed related content.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Integer</td>
<td>Primary key</td>
</tr>
<tr>
<td>url_id</td>
<td>Integer</td>
<td>Foreign key of feed_urls.id</td>
</tr>
<tr>
<td>channel_title</td>
<td>Text</td>
<td>Title of the channel</td>
</tr>
<tr>
<td>channel_description</td>
<td>Text</td>
<td>Description of the channel</td>
</tr>
<tr>
<td>channel_link</td>
<td>Text</td>
<td>Link to the channel</td>
</tr>
<tr>
<td>channel_generator</td>
<td>Text</td>
<td>Generator of the RSS feed</td>
</tr>
<tr>
<td>channel_language</td>
<td>Text</td>
<td>Language of the RSS feed</td>
</tr>
<tr>
<td>item_title</td>
<td>Text</td>
<td>Title of the feed item</td>
</tr>
<tr>
<td>item_link</td>
<td>Text</td>
<td>Link to the feed item</td>
</tr>
<tr>
<td>item_description</td>
<td>Text</td>
<td>Description of the feed item</td>
</tr>
<tr>
<td>item_content</td>
<td>Text</td>
<td>Content of the feed item</td>
</tr>
<tr>
<td>item_timestamp</td>
<td>Character Varying</td>
<td>Timestamp of the feed item</td>
</tr>
<tr>
<td>feed_type</td>
<td>Character Varying</td>
<td>Type of feed</td>
</tr>
<tr>
<td>feed_version</td>
<td>Character Varying</td>
<td>Feed version</td>
</tr>
<tr>
<td>encoding</td>
<td>Character Varying</td>
<td>Encoding used by the feed</td>
</tr>
<tr>
<td>parsed_for_links</td>
<td>Boolean</td>
<td>Whether this link has been parsed for links</td>
</tr>
<tr>
<td>hash</td>
<td>numeric</td>
<td>Decimal value of a MD5 hash of the normalized URL of the feed</td>
</tr>
</tbody>
</table>
**Table: feed_url_metadata**

**Description:** This table stores all the web feed URL metadata.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Integer</td>
<td>Primary key</td>
</tr>
<tr>
<td>crawl_timestamp</td>
<td>Integer</td>
<td>Unix timestamp of time the feed was crawled</td>
</tr>
<tr>
<td>http_response</td>
<td>Character Varying</td>
<td>HTTP response code from the HTTP server</td>
</tr>
<tr>
<td>feed_url_id</td>
<td>Integer</td>
<td>Foreign key of feed_urls.id</td>
</tr>
<tr>
<td>retrieval_time</td>
<td>Numeric</td>
<td>Retrieval time of the feed (in seconds)</td>
</tr>
<tr>
<td>number_of_feed_items</td>
<td>Integer</td>
<td>Number of feed items in the feed</td>
</tr>
</tbody>
</table>

**Table: feed_urls**

**Description:** This table stores all the web feed URL's.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Integer</td>
<td>Primary key</td>
</tr>
<tr>
<td>url</td>
<td>Text</td>
<td>Web feed URL to be parsed</td>
</tr>
<tr>
<td>hash</td>
<td>Numeric</td>
<td>Decimal value of a MD5 hash of the normalized URL of the feed</td>
</tr>
</tbody>
</table>
Table: link_urls

Description: This table stores all HTML link URL’s.

Table 4.5
Data dictionary for link_urls database table

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Integer</td>
<td>Primary key</td>
</tr>
<tr>
<td>url</td>
<td>Text</td>
<td>Link URL to be parsed</td>
</tr>
<tr>
<td>hash</td>
<td>Numeric</td>
<td>Decimal value of a MD5 hash of the normalized URL of the feed</td>
</tr>
</tbody>
</table>

Table: queue

Description: This table stores the active web feed queue to be parsed

Table 4.6
Data dictionary for queue database table

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Integer</td>
<td>Primary key</td>
</tr>
<tr>
<td>url_id</td>
<td>Text</td>
<td>Foreign key of feed_urls.id</td>
</tr>
</tbody>
</table>

4.5.3 Access and Authorization

The file `pg_hba.conf` stores all the authentication information that is used to control which hosts are allowed to connect and how users are authenticated on that host as well as the databases that are allowed to be access by the host. The file is a simple space delimited text file and its format is as in Code Listing 4.3.

Code Listing 4.3

`pg_hba.conf` example

```
#TYPE    DATABASE  USER   CIDR-ADDRESS   METHOD    sameuser
local    all       all     ident         sameuser
host     all       all     192.168.1.0/24  md5
host     all       all     ::1/128        md5
```
The type is set to be “local” or “host”. “local” refers to using Unix domain sockets and host uses TCP/IP. The database field refers to the database in question. The user field are the users allowed to authenticate to access the database. The CIDR-Address field notes the IP addresses allowed to access the database. This field values are the IP addresses and netmask. Finally, the Method field is the method of authentication. The possible values for the fifth field are “trust”, “md5”, “password”, “krb5”, “ident” or “pam”.

There is only a single database that is used by all the operations in the system. To ensure security of the system, the settings are set to only allow the database to be accessed via TCP/IP with the password hashed via MD5 hashing algorithm.

4.5.4 Database Encoding

It is important to store the content downloaded in the correct encoding method. By default, PostgreSQL creates a database with ASCII encoding. However, this is not suitable as there are many multibyte encoding feeds available. As such, UTF-8 is set as the default encoding format. The database is created as in Code Listing 4.4:

Code Listing 4.4
Database Creation

```bash
createdb -E UNICODE
```
Chapter 5: Experimental Results and Data Analysis

This chapter defines the approach used in testing the data driven crawling architecture. It outlines the system parameters selected, the process of building a test harness which allows for a standardized manner of running test cases and finally, the results are presented.

There is very little academic data published that shows overall system performance of a traditional crawling architecture. As such, for benchmark purposes, a basic crawling architecture was built for purposes of comparing the methodology between the data driven architecture and traditional architectures. Results are displayed for the traditional architecture as well as the data driven distributed crawling architecture.

5.1 System Parameters

A two crawler data driven crawling system was set up on identical machines with a centralized database. In order to determine the performance of the data driven distributed system, it is first necessary to determine the system parameters. The following system parameters were determined to be of interest:

- New Feeds vs Time: to identify the system performance characteristics when discovering new feeds
- Average System Load vs Time: to identify overall system load of the data driven distributed system
- Memory Use vs Time: to identify overall memory use of the operating
environment and database

- CPU Use vs Time: to identify overall CPU use of the operating environment and database

The data collection methodology is outlined as in Table 5.1:

<table>
<thead>
<tr>
<th>Data</th>
<th>Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Feeds vs Time</td>
<td>Execute a COUNT SQL query on the relevant table for each data item required</td>
</tr>
<tr>
<td>Feed Generators</td>
<td></td>
</tr>
<tr>
<td>Feed Types</td>
<td></td>
</tr>
<tr>
<td>HTTP Response Code</td>
<td></td>
</tr>
<tr>
<td>CPU Usage</td>
<td>Execute an external command to read operating system statistics through the command <code>top</code></td>
</tr>
<tr>
<td>Memory usage</td>
<td></td>
</tr>
<tr>
<td>System Load</td>
<td>Execute an external command to read operating system statistics through the command <code>uptime</code></td>
</tr>
</tbody>
</table>

5.2 Traditional Crawling Architecture Benchmark

A traditional crawling architecture was set up for benchmark purposes. This architecture is a simplified derivation of the Google architecture as defined in Figure 2.1 and described in section 2.1. The system architecture of the benchmark crawling system is described in Figure 5.1:
As can be seen in Figure 5.1, there is a central URL server which distributes URL's to multiple crawling agents by storing the URL's in a file on a networked file system. The singly-threaded crawling agents retrieve the URL's, perform analysis to extract links and auto discover feeds, and finally store the results on network filesystem. The central URL server then reads the results, partitions the results appropriately and stores the results in the database. This process continually repeats.
5.3 Test Harness

To collect the data reliably, a test harness was built to observe the performance characteristics of the system. The test harness captures an observation point at 10 second intervals and stores the results in an external database over a three hour period. The data captured during the observation point uses the data collection methodology as listed in Table 5.1. Upon the completion of the three hour period, the data was saved and graphed as scatter plots.

5.4 Experimental Results

The data driven crawling system is run with number of data sets per crawler set to 500. The result was obtained is described in Figure 5.2:

Figure 5.2
New Feeds (Cumulative) vs Time (data sets/crawler = 500)

![New Feeds (Cumulative) vs Time (Data Sets/Crawler = 500)](image)

The average system load vs time for the machines is described in Figure 5.3 as
follows:

Figure 5.3
Average System Load vs Time (data sets/crawler = 500)

(overall average load=7.525; standard deviation=7.50)

While the overall average is 7.525, it is evident from Figure 5.3 that the system load is increasing exponentially as a function of time, at one point reaching 45. The high value for the standard deviation indicates a non stable system.

The high load resulted in a non responsive system and as such, the distributed crawling system needed to be shut down before the completion of the experiment. High system loads can be caused by too many crawlers in the system and as such, to reduce the number of crawlers the number of data sets per crawler was increased to 1000 data sets/crawler. The result is shown below in Figure 5.4 and Figure 5.5:
Figure 5.4
New Feeds (Cumulative) vs Time (data sets/crawler = 1000)

Figure 5.5
Average System Load vs Time (data sets/crawler = 1000)

(overall average load=4.17; standard deviation=4.20)
From Figure 5.5, it can be seen that the system load is slightly more stable with a maximum load of 27.55. However, this load was still too high and caused the system to be unresponsive, hence the experiment was terminated.

The data sets/crawler setting was increased to 1500 and the results can be seen in Figure 5.6 and Figure 5.7.

Figure 5.6
New Feeds (Cumulative) vs Time (data sets/crawler = 1500)
Figure 5.7
Average System Load vs Time (data sets/crawler = 1500)

(overall average load=3.87; standard deviation=3.60)

From Figure 5.7, it is possible to see continuous improvement of the system load. However, the system load is still too high and the experiment was halted before the end of the experimental period due to the system being non-responsive. As such, the data sets/crawler setting was set to 2000. The results are show in Figure 5.8 and Figure 5.9.
Figure 5.8
New Feeds (Cumulative) vs Time (data sets/crawler = 2000)

Figure 5.9
Average System Load vs Time (data sets/crawler = 2000)

(overall average load=2.63; standard deviation=1.56)
In Figure 5.9, the overall system load shows spikes but low value of standard deviation of 1.56 indicates a stable system. As such, 2000 data sets per crawler was chosen as it provided for a sufficiently stable system.

5.5 Comparison Between Data Driven Architecture and Traditional Architecture

As described in Section 5.2, a simplified derivation of the Google architecture crawling architecture was used as a benchmark to compare the data driven architecture against the data driven crawling architecture.

Figure 5.10 clearly shows that the traditional architecture reaches saturation point at approximately 200 feed URL's. This is due to the limitation of the centralized URL distribution server which is unable to keep up with the task of distributing URL's, and as such, the performance of the system as a whole suffers. The step wise increase in the performance of the traditional crawling system in Figure 5.10 indicates the bottleneck of retrieval resources in strict sequence in a singly-threaded crawling system. By comparison, the process forking architecture in the data driven crawling system shows a linear increase over time as it is able to retrieve resources in parallel.
5.6 System Performance Characteristics

To ensure the distributed system is efficient in the consumption of CPU and memory, CPU and memory usage for the operating environment (PHP) and the database environment (PostgreSQL) as a function of time was captured for a single system. The CPU performance is shown in Figure 5.11 and Figure 5.12. The memory usage is shown in Figure 5.13 and Figure 5.14.
Figure 5.11
CPU Usage (PHP) vs Time

Figure 5.12
CPU Usage (PostgreSQL) vs Time
Figure 5.13
Memory Usage (PHP) vs Time

Figure 5.14
Memory Usage (PostgreSQL) vs Time
Figure 5.11 and 5.12 indicates the CPU usage of the operating environment (PHP) and database environment (PostgreSQL) relatively stable with occasional spikes. Figure 5.13 and 5.14 show the memory usage increasing linearly at the start and stabilizing subsequently. The memory graphs show stepwise increases and decreases, which can be explained by the batch memory allocation mechanism of the Linux operating systems (Lee, 2000).

5.7 Web Feed Statistics

Based on the web feeds retrieved from remote servers, details on HTTP response codes, web feed generators and feed types (either RSS or Atom). Figure 5.15 indicates the HTTP response codes from the remote servers. As can be seen, an overwhelming 98.43% of servers returned a 200 HTTP response code and the web feed returned was well formed. This value is favorably compared to the 87.03% returned by the Mercator crawler in 1999 (Heydon and Najork, 1999) In comparison, 0.08% of servers returned web feeds which could not be parsed. This is likely the case when the remote server has been misconfigured and is returning incorrect results.
Figure 5.15
HTTP Response Codes

Figure 5.16
Top 10 Feed Generators
Figure 5.16 clear indicate that WordPress is the most popular web publishing tool with it being responsible for 25% of the web feeds. This is followed by TypePad and Blogger, both being hosted blogging environments online. Flickr is a very popular online photo site which allows for photos to be syndicated as web feeds. Typo, Movable Type, Apache Roller, Plone, Mephisto and Community Server are all web publishing tools. From Figure 5.17, it is obvious that RSS 2.0 is the most popular RSS feed with 59% of web feeds polled using it. In total, the RSS feed type adds up to almost 75% of all web feeds. With 25% of web feeds not identifying themselves as either RSS or Atom, it is not possible to determine the popularity of the Atom feed type.
Chapter 6: Discussion

The implementation of the proof of concept data driven crawling architecture indicated that building such a system based on a data driven architecture is feasible and provides better performance than a traditional crawling architecture. This chapter discusses how the data driven system successfully handled locking behavior, run time implementation issues and improvements to protocols which would allow for distributed crawling systems to operate more efficiently.

6.1 Data Locking in a Concurrent Environment

A major issue in concurrent execution environments is correctly implementing locking behavior for commonly accessed data sets. Locks are known to have problems with blocking and also cause other run time concurrency issues such as deadlocks and race conditions. This section describes how locking issues were avoided by the use of a multiversion concurrency control system in the backend PostgreSQL database storage system.

6.1.1 Reading Data

Traditional database locking mechanisms allow for locked and unlocked SQL queries. An exclusive lock prevents read access on the same data set when another thread of execution is in the critical section whereas a shared lock allows for other threads of execution read access. In the PostgreSQL concurrency model, data readers never wait for data writers. As such, when a SELECT query starts executing, any subsequent INSERT and UPDATES never affect the dataset of the SELECT query. This
ensures that all data reads through SQL selects do not require locking and do not cause any form of data corruption.

6.1.2 Inserting new data sets

As inserting new data sets do not affect UPDATE and REMOVE SQL queries, these can be safely be removed from consideration. In addition, concurrency measures at the database layer safely ensure that transactions involving SELECT SQL queries are not affected by INSERT SQL queries, thus allowing for all database queries to safely insert new data sets without affecting the integrity of the data.

6.1.3 Updating data sets

Updates of data sets do not affect read and write queries. Due to the concurrency model, updates also do not conflict with each other. Hence, all database queries that update data sets do not affect the integrity of the data.

6.1.4 Removing data sets

Removal of data sets do not affect reads and writes and are therefore safe to be used in all database queries.
6.2 Implementation and Design Strategies

The experience gained in writing the data driven crawling architecture in a distributed fashion has indicated that while simplicity is gained by removing the problem of locking, architecting the system as a whole is difficult for an inexperienced system designer.

The success of building such a distributed system is based on a well defined design and given that, by definition, a distributed system tends to have many independent components. As such, much thought and perspective must be considered before building a distributed system. This section outlines some of the design considerations undertaken and issues faced in the scope of building the data driven distributed crawling architecture.

6.2.1 Efficient parallel processing through process forks

Forking a process involves copying over the address space, file descriptors as well as other resources of the parent for the child process. As such, if not properly implemented, forking may implement a high penalty. On POSIX compliant operating systems, forking tends to be an efficient system call. On the Linux operating system, fork() is implemented as a copy-on-write operation which is relatively efficient.

The process forking methodology to parallel processing adopted in this dissertation resulted in a problem with the Linux kernel implementation of copy on write. Copy on write refers to a kernel level optimization whereby given a situation in which multiple child process forks of a single parent process, all the child processes will share the same pointers to established resources until a write is attempted to the resource in question, in which case the resource is duplicated transparently by the kernel to the
This posed a problem as the database connections would fail when two child processes which were forked from the same parent process attempted to simultaneously access the database using the same resource pointer. This problem was solved by explicitly destroying the resource pointer in a forked child process and reconnecting to the database. This required that the database be correctly set to be able to accept large number of simultaneous connections as the connections cannot be shared among multiple child processes.

6.2.2 Clearly divisionable data space

The Principle of Independent Execution of Operations introduced in Chapter 3.2 was presented as a method of reducing interprocess communication overhead and as a method of reducing the possibility of runtime data corruption bugs inherent in concurrent processing applications. To that end, it is necessary for parent and child processes to exclude all forms of direct communication and thus, a clearly divisionable data space was required. It is important for child processes to operate on data spaces that have been cleanly partitioned and exclusively assigned to only one child process to operate on.

Partitioned data space is possible through several means which have been explored in Section 2.5. URL partitioning through a hashing algorithm was used a method for data partitioning. This approach proved to be suitable in segregating the data space efficiently between crawlers.
6.2.3 Instrumentation Methodology

Given the process forking methodology adopted, it was important to measure resource usage of the child processes so as to ensure high levels of system productivity, reliability, efficiency and stability. Proper instrumentation techniques were used to measure resource utilization and circumvent any possible memory leaks. The instrumentation methodology used was to capture of a snapshot of the complete system state every 10 seconds. This provides the ability to graph the overall performance of the system over a period of time and note emergent trends that may not be obvious at the outset. The system data which was captured included memory usage, CPU usage, disk I/O, network I/O as well as feed queue and link queue back logs. This data was sufficient to graph the performance characteristics of the overall system.

6.3 Robot Exclusion Protocol

Based on the experimental results of the data driven crawling system, it was discovered that the crawling system makes a substantial number of content retrievals and such activity can be extremely bandwidth intensive. It is indeed possible for the system to make a large number of queries to a single host within a short period of time leading to situations whereby the host may get overwhelmed and choose to block all incoming connections from the crawler.

As such, it was necessary to determine a suitable re-crawling interval for feeds which does not overwhelm the host. In line with this, this dissertation proposes that the Robots Exclusion Protocol be extended to include a minimum timeout field which can be used by crawling agents to implement throttling behavior. The minimum timeout period is defined in seconds after a single resource retrieval.
An example of such an implementation is defined in Code Listing 6.1:

**Code Listing 6.1**

**Proposed Addition to the Robot Exclusion Protocol**

<table>
<thead>
<tr>
<th>User-agent: *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recrawling-interval: 3600</td>
</tr>
</tbody>
</table>

As can be seen, the recrawling interval is set to one hour (3600 seconds) through the proposed new field *Recrawling-interval*.

The existence of an explicit timeout has several benefits:

1. it allows system administrators to choose a recrawling interval that crawling agents should follow.
2. it allows easier abuse administration by systems administrators.
3. retrieval algorithms can be implemented by straightforward methods as opposed to having to make educated guesses on the type of throttling behavior to implement.
4. crawling systems do not need to make an excessive number of queries to a host within a short period of time, thus saving bandwidth.

Given that most web feeds are automatically generated by web feed generators, it is feasible for generators to update the recrawling interval field based on usage statistics. This leads to a more reliable result which can be used by crawlers to schedule crawling intervals accurately.
Chapter 7: Conclusion

The investigation of this dissertation focused on building a data driven distributed crawling architecture, with a particular focus on web feed technology. The two major protocols for web feed technology were explored and the algorithmic methods of web feed auto discovery protocol were investigated and implemented successfully. To implement the crawling architecture, a data driven methodology was explored and implemented successfully.

Experimental results for the web feed retrieval statistics have indicated a high level of performance. The experimental results also showed that RSS 2.0 was the most popular web feed technology currently in use and the most widely used software used to generate web feeds is the open source WordPress software. In addition, the top 10 feed generators list clearly indicated that web feeds are most popularly used in web-logging software.

The contribution of this dissertation is summarized below:

- a new method of architecting and building data driven distributed crawling architectures was introduced.
- it was shown through experimental results that a data driven distributed crawling architecture can provide better performance than a traditional distributed crawling architecture, as long as sufficiently strong emphasis is given on the overall design of the system to avoid run time performance issues.
- more efficient protocols were provided for content retrieval.
- it was shown through inspection that a data driven distributed crawling
architecture ensures data integrity through existing relational database locking mechanisms.

7.1 Design and Implementation Strategy

Existing crawling architectures and traditional methods of building crawling architectures were also extensively explored in this dissertation. The methodology of parallel execution as well as the overall design of traditional crawling architectures were discussed and the identified shortcomings were noted.

Building a production crawling system is a considerable task that requires substantial investment of time and effort. Thus, certain design considerations were closely adhered too during the development of the data driven crawling architecture. Instead of the use of a singly-threaded or multi-threaded methodology to achieve parallel execution, this dissertation proposed the use of process forking to simplify the implementation. A methodology for building efficient distributed crawling architectures by eliminating direct communication between individual components of the system through the use of a data driven methodology was also introduced. The design, algorithmic approach and implementation details of the data driven crawling architecture were outlined in detail in this dissertation.

The process of designing and implementing the data driven distributed crawling architecture indicated that much of the complexity inherent in building concurrent systems can be reduced through the adoption of a data driven architecture. The data driven methodology introduced in this dissertation reduced overall complexity of the system by segregating data sets between individual executing operations. This methodology also reduced interprocess communication overhead by not allowing
parallel executing operations to communicate directly to one another.

It was also observed that once the data model was architected in detail, building each operation within the crawling system was relatively simple as the relational database backend handled data locking and thus relieved the implementor from having to correctly handle locking issues which is traditionally difficult in the implementation of parallel execution.

7.2 Experimental Results and Contributions

The experimental results indicated that a data driven crawling architecture provided better performance then a traditional distributed crawling architecture. It was also shown through inspection that data integrity could be ensured through existing locking mechanisms. Whilst the process forking methodology proved to be successful in parallelizing content retrieval, implementation and run time issues such as copy-on-write behavior and system load needed to be correctly addressed.

A further contribution of this dissertation was in providing a feedback mechanism for web hosts to indicate to crawlers as to optimal re-crawling intervals. Low values for crawling intervals clearly increases bandwidth usage as well as increases scalability issues, whereas higher values for crawling intervals can lead to possible data loss. Recognizing that most web feeds are generated by software, an update to the Robot Exclusion Protocol was suggested which explicitly specifies retrieval time out between re-crawling intervals which crawling systems can use as an accurate guide to scheduling re-crawling times.
7.3 Future Work

The biggest advantage of using structured content for syndicated content is the ease of accessing the actual content instead of building logic to differentiate the content from presentation. Hence, the quality of the content is significantly higher compared to traditional web crawling architectures and it is possible to use this higher quality of data to build interesting applications.

One interesting application of web syndication crawling engines is to group and aggregate the structured content according to commonality. To give an example of a real life situation, let's assume that a significantly newsworthy item takes place - for example, a major flood occurs in Kuala Lumpur. Reports and discussions of the flood will begin to spread across the news syndication and weblog sites. The aggregation logic should note the similarity between the issues of discussion and group them together. Google News\(^9\) is an example of such a technology that uses an algorithmic driven methodology of grouping news items from multiple news sources.

Another area of future work is to develop algorithms that would attempt to schedule the best time to crawl a RSS feed. It is important that the algorithm be able to change predictions of crawling times based on the latest timing information. There is little information on when the timing information of when an existing webpage has been updated. As such, it is difficult to predict when webpages are expected to change next. This information is useful as a method of optimization i.e. in cases of limited bandwidth, it allows the crawler to build a crawling schedule that favors webpages that are updated more often. The areas of future work should also focus on evaluating techniques of estimating the update intervals for webpages and building estimation models that can be empirically compared to real data.

---

9  Google News is available at http://news.google.com
In conclusion, web feed technology is a very popular method of serving content on the Internet and this dissertation shows that web feed auto discovery protocols have worked well in enabling crawling systems to discover new feeds on the Internet. In addition, experimental results have indicated that a data driven distributed crawling architecture can be efficiently implemented through a process forking methodology and data partitioning techniques.
Chapter 8: References


[38] Lee, E. A., “The Problem With Threads” [online], Available: http://www.computer.org/portal/site/computer/menuitem.5d61c1d591162e4b0ef1bd108bcd45f3/index.jsp?&pName=computer_level1_article&TheCat=1005&path=computer/homepage/0506&file=cover.xml&xsl=article.xsl&;jsessionid=G9nWv6DTHj21JJhrjlx2Gh0Gw7SipST713qmTY45dY1wXfns0XF1-1182277384, Accessed: 1st April 2007


Appendix A: Example of an RSS Feed (Winer, 2000)

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN" "http://my.netscape.com/publish/formats/rss-0.91.dtd">
<rss version="0.91">
  <channel>
    <title>The Walrus's Blog.</title>
    <link>http://gathani.org/</link>
    <description>Gathani.org</description>
    <language>en-us</language>
    <pubDate>Thu, 18 Jun 2004 18:00:50 +0000</pubDate>
    <item>
      <title>Item 2</title>
      <link>http://gathani.org/1</link>
      <pubDate>Thu, 17 Jun 2004 18:00:50 +0000</pubDate>
      <description>
        Content.
      </description>
    </item>
    <item>
      <title>Item 1</title>
      <link>http://gathani.org/2</link>
      <pubDate>Thu, 18 Jun 2004 18:00:50 +0000</pubDate>
      <description>
        Content.
      </description>
    </item>
  </channel>
</rss>
Appendix B: .toprc file

RCfile for "top with windows"
Id: a, Mode_altscr=0, Mode_rixps=1, Delay_time=3.000, Curwin=0
Def   fieldscur=AWXEOQPTRShiKNMbcdfgjluvyz
      winflags=62777, sortindx=15, maxtasks=0
      summclr=1, msgsclr=1, headclr=3, taskclr=1
Job   fieldscur=ABcefgjlrstuvyzMKNIHWOQDX
      winflags=62777, sortindx=0, maxtasks=0
      summclr=6, msgsclr=6, headclr=7, taskclr=6
Mem    fieldscur=ANOPQRSTUVbcdefgjlmzyWHIKX
      winflags=62777, sortindx=13, maxtasks=0
      summclr=5, msgsclr=5, headclr=4, taskclr=5
Usr    fieldscur=ABDECGfhijloqrstuvwxyzMNWX
       winflags=62777, sortindx=4, maxtasks=0
       summclr=3, msgsclr=3, headclr=2, taskclr=3