Discovering and Tracking Interesting Web Services

A Thesis
Presented to
The Academic Faculty

by

Daniel J. Rocco

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
December 2004

Discovering and Tracking Interesting Web Services

Approved by:

Dr. Ling Liu, Advisor

Dr. Calton Pu

Dr. Terence Critchlow

Dr. H. Venkateswaran

Dr. Shamkant Navathe

October 6, 2004

To Jenny.

ACKNOWLEDGEMENTS

Show me your ways, O Lord, teach me your paths; guide me in your truth and teach me, for you are God my Savior, and my hope is in you

— Psalm 25:4–5

I would like to thank my wife Jennifer for her constant love and encouragement; her suggestions for improvements and proofreading work enhanced and clarified many of the ideas presented here.

My advisor, Professor Ling Liu, demonstrated exemplary management skills while showing me how to be a researcher. I am especially grateful for her discernment as she simultaneously praised my successes while showing me areas for improvement and pushing me to strive harder and dream bigger.

The members of my committee—Terence Critchlow, Shamkant Navathe, Calton Pu, and H. Venkateswaran—have been instrumental to the success of my graduate career. Their encouragement, project ideas, and career guidance helped shape every stage of this journey.

Finally, special thanks to my family and friends for their love and faith: you have given me more than I could ever hope to repay.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The World Wide Web has become the standard mechanism for information distribution and scientific collaboration on the Internet. Acceptance of the Web as a standard tool has been greatly assisted by the emergence of high quality search engines that allow users to locate documents containing information of potential interest. However, the success of commercial search engines does not solve the problem of information management on the Web. In particular, no universal mechanism exists for automatically discovering and categorizing dynamic Web sources, which comprise an enormous and quickly growing segment of the Web. While some classification schemes exist for these sources, solutions tend to be ad hoc, proprietary, and uncooperative. The continued growth of the Deep Web further exacerbates these problems as Deep Web sources serve dynamically generated data that is beyond the reach of standard search engines.

The rapidly evolving environment of the Web offers many challenges apropos information discovery and dissemination. For example, how can users find applications of interest in the context of the dynamic Web? What technologies exist for keeping abreast of interesting information on the expanding Web? What data handling services are required for efficient management of Web data?

This dissertation research explores a suite of techniques for discovering relevant dynamic sources in a specific domain of interest and for managing Web data effectively. Because the greater level of interactivity on dynamic Web sources invalidates many of the assumptions upon which traditional search engines were built, the first portion of this research explores techniques for discovery and automatic classification of dynamic Web sources. Our initial research focus has been on dynamic sources in the bioinformatics domain, specifically those sources providing genome query services. Our approach utilizes a service class model of the dynamic Web that allows the characteristics of interesting services to be specified using a service class description. These descriptions provide an abstract view of a class of sources

that does not rely on the implementation details of any one class member. This service class model enables generic, probing-based service discovery using a Web crawler approach without requiring service registration or service-specific capability descriptions.

We have constructed a prototype automatic discovery and classification system, DynaBot, to test and promote our service class discovery methodology. Experimental testing of interfaces across the Web that provide genome data via the bioinformatics tool BLAST demonstrate that DynaBot achieves initial recognition rates of up to 75% over a selection of Deep Web sources. This research has demonstrated a framework for building a data source discovery system for the Deep Web that is capable of effectively discovering, identifying, and categorizing a large collection of domain-specific data sources.

Contemplation of the management issues involved when interacting with the Web led to the construction of a novel document encoding system for HTML files, the Page Digest, and a Web change monitoring system built on this encoding scheme. An important problem affecting the scalability of any system that interacts with the Web is the processing of standard Web document markup languages, such as HTML and XML. These Web document formats are not tuned for efficiency and are highly redundant. A second drawback to using standard Web languages in a scalable system is their intermixing of the various aspects of the document, including structure, tag names, attributes, and content. The Page Digest encoding eliminates tag redundancy and places structure, content, tags, and attributes into separate containers, each of which can be referenced in isolation or in conjunction with the other elements of the document. The Page Digest Sentinel system leverages our unique encoding to provide efficient and scalable change monitoring for arbitrary Web documents through document compartmentalization and semantic change request grouping. These improvements have contributed to achieving one to two orders of magnitude decrease in processing time compared with similar Web information monitoring systems.

The final component of this work revisits Page Digest in the context of Web documents encoded as XML. Compressing XML documents is a popular way to mitigate the markup syntax overhead of XML documents, yet a fundamental problem with this approach is its opaque nature: data compressed with standard compressors is only available for use after

being decompressed, a costly overhead step that must be added to the overhead of parsing the text document.

XPack is an XML document compression system that uses a containerized view of an XML document to provide both good compression and efficient querying over compressed documents. XPack's queryable XML compression format is general-purpose, does not rely on domain knowledge or particular document structural characteristics for compression, and achieves better query performance than standard query processors using text-based XML. XPack utilizes the structural features in XML to provide valuable support for path expression queries over compressed documents. By advocating a component-based compression architecture, XPack allows efficient compression of different document components by capitalizing on their structural features and offers opportunities for parallel document compression.

Initial experimental results using a prototype XPack document compressor demonstrate that XPack can reduce the storage requirements for Web documents by up to 20% over previous XML compression techniques. XPack's compression performance stems from its aggressive redundancy elimination techniques. Unlike other widely used XML compression systems, XPack can simultaneously support general query operations over the documents: utilizing compartmentalization, which separates different aspects of Web documents into logical containers, XPack statistics operations achieve up to two orders of magnitude performance improvement when compared to equivalent operations on unencoded XML documents. XPack is also a highly flexible system that allows general path expression queries over compressed documents using the standard XML path query language XPath. XPack's efficient encoding scheme yields significant performance advantages for general path queries over the compressed documents when compared with queries executed over unencoded XML.

Our research expands the capabilities of existing dynamic Web techniques, providing superior service discovery and classification services, efficient change monitoring of Web information, and compartmentalized document handling. DynaBot is the first system to combine a service class view of the Web with a modular crawling architecture to provide automated service discovery and classification. The Page Digest Web document encoding is

the first mechanism for representing Web documents that explicitly separates the individual characteristics of the document to enable scalable and efficient access to information on the Web. The Page Digest Sentinel change monitoring system is the first Web document monitor that utilizes the Page Digest document encoding for scalable change monitoring through efficient change algorithms and intelligent request grouping. Finally, XPack is the first XML compression system that delivers compression rates similar to existing techniques while supporting better query performance than standard query processors using text-based XML. XPack eliminates the need to choose between compact document sizes and efficient document operations. XPack is general-purpose: the compression scheme does not require domain knowledge or particular document structural characteristics for effective compression.

CHAPTER I

INTRODUCTION

1.1 Motivation

The emergence of the World Wide Web ushered in a new era of computing that transformed computers from special-purpose processing tools into general-purpose data assistants used for communications and research, work and play. The Web redefined the concept of data accessibility by providing an infrastructure for accessing data irrespective of temporal and geographical boundaries.

Search engines, which appeared nearly simultaneously with the Web itself [54], quickly developed into a primary technological enabler for the Web. As the coverage and ranking algorithms of the major search engines began to improve, the utility of Web keyword search applications became obvious. Rather than maintaining tedious lists of bookmark URLs or finding sources of interest via word-of-mouth, users could enter key terms on the subject of their search and receive a ranked list of pertinent Web sites. Quality search engines allowed users to search the Web for topics of interest reliably and quickly.

The emergence of the dynamic Web represents a fundamental shift in information distribution on the Web. Web publishing is becoming a more dynamic experience with data-driven server and client applications replacing the more traditional hyperlinked static text documents. These "Deep Web" services provide access to real-time information supplied from large databases or other data repositories. Recent studies suggest that the size and growth rate of the dynamic Web greatly exceed that of the static Web [61, 62]. Estimates suggest that the practical size of the Deep Web may exceed 550 billion individual documents [5].

As the mechanics of information distribution change, so too must the methods used for information retrieval. Some search engine companies have recognized the signs indicating these changes and have produced a host of new applications in an attempt to appeal to a

large, dynamic user base. These applications include geocentric search [47], topic based Web navigation and searching [57, 46, 3], publicly maintained directory services [103, 45, 77], and expert answer forums [2, 3, 44], among many others. However, these new applications do not address the data management problems inherent to the new paradigm of the dynamic Web. Some of these problems include:

- *Service Discovery.* Can users and automated agents discover Web services of interest in the context of the dynamic Web? We hypothesize that finding relevant Web services can be solved with a two-step process. First, we define the concept of a *service class* to describe Web services, implementing a service class-based Web source probing and analysis engine. Second, we design a modular crawler architecture to automate the discovery and classification of interesting Web services. By combining the service probing and analysis engine with the modular crawler, we demonstrate that users can effectively find Web services of interest in the context of the dynamic Web.

- *Change Monitoring.* Can Web sources be monitored for important data updates while managing growing data repositories and user bases? We use change request grouping to leverage expanding populations, combining related requests to reduce network and computation overhead. In conjunction with efficient monitoring functions and our document representation, which separates document structure from content for faster access to interesting document elements, our document monitoring system provides scalable components for tracking interesting changes in Web data.

- *Document Handling.* Can the storage and transmission costs of Web data be minimized while still supporting high performance querying? We use aggressive redundancy elimination and document compartmentalization of data stored in XML to support Web document compression for effective reduction of information storage costs. Careful exposure of document container bookkeeping information allows query functions to access important portions of compressed documents quickly and support general, high performance query operations.

1.2 State-of-the-art and Research Scope

1.2.1 Service Discovery: Existing approaches and DynaBot

The dynamic source discovery problem presents several novel challenges compared with searching static data sources. Static Web sources store data in document files that reside on a server's disk. These sources respond to client requests by locating the requested document and sending it back to the client unmodified. Search engine crawlers operating on the static Web retrieve documents exactly as any other client would. To service user requests, a search engine employs indexing and ranking algorithms on the document's text and hyperlinks to gauge the document's relevance to a particular keyword query relative to other documents in the engine's index.

In contrast to the static Web, dynamic Web sources store data in database systems or other data repositories. The server generates a response to a client request dynamically, composing it from data retrieved from a database, HTML formatting templates, and server executed code. The dynamic system architecture enables an enriched user experience, but the greater level of interactivity invalidates many of the assumptions upon which traditional search engines were built. Service discovery in the dynamic Web involves locating potential sources, determining source capabilities, classifying sources with respect to their relevance for a particular query, and ranking related sources.

There are several possible approaches for users wishing to interact with multiple dynamic Web sources that provide a similar interface but may have access to different data sets. The most rudimentary approach, which corresponds to the current operating paradigm on the Web, forces the user to interact with each source independently. The user chooses a few sources and enters the input data into each, then integrates the results of the various queries by hand. This process is slow, error-prone, and places a significant burden on the user, who must visually inspect each result and decide where each fits in their entire result set. Further, the user is unlikely to have access to an optimal set of data sources and will be unable to query more than a few sources by hand. These concerns, exacerbated by the phenomenal growth rate of the dynamic Web, suggest the use of an automated approach to the problem of discovering and classifying dynamic Web sources.

The availability of a consistently defined registry of services is a common assumption in automated Web service interaction architectures. The most popular of these service registries is Universal Description, Discovery, and Integration of Web services, or UDDI [98, 97]. UDDI defines a registry for descriptions of Web services; each service description provides information on the publishing entity for the service, a textual description of what the service does, and a service API defined in a standard service description formats such as WSDL. Despite efforts in UDDI and WSDL standardization, such a registry is not available for the majority of services.

To address the problems associated with service discovery in the dynamic Web, we have developed a service class model that provides a framework for automated service capability inferencing and service classification. The service class view of the dynamic Web enables the construction of service class descriptions that provide an abstract view of a class of interesting services. DynaBot combines a service class-based service probing and analysis engine with a modular crawler architecture to provide a general purpose automated service discovery system for the dynamic Web.

DynaBot provides a foundation for unifying complex Web data sources using automatic discovery and capability detection; the BLAST family of data sources has provided a test case for our approach [87, 78]. This work addresses the problem of finding and automatically classifying services from an arbitrary set of sites. The service class description format we describe provides greater descriptive power than the domain descriptions used in other systems [35] and can specify complex data types and source control flow information. DynaBot also addresses the problem of locating new services of interest automatically and does not require services to publish explicit interaction instructions or database snapshots.

Several research projects exist related to the service capability detection problem; these projects [17, 66, 50, 32, 49, 56, 73, 18] have examined database selection and classification in the context of keyword queries over document databases. These approaches do not rely on a technical service description but employ service estimation techniques that utilize document term frequencies or other estimation metrics for each document. The QProber work by Ipeirotis, Gravano, and Sahami [49], estimates the contents of text databases

through probing queries but is limited to service classification only; no discovery services are offered. These projects focus on services that expose document databases to the Web through keyword search forms, while DynaBot can be utilized for more complex services and in domains where document term analysis is inappropriate.

1.2.2 Change Monitoring: Existing approaches and Page Digest Sentinels

In order to make effective use of the massive amounts of data on the Web, automated tools must be created to facilitate human interaction with Web sources. As the size of the Web continues to grow, users will become more dependent on change monitoring and update notification systems to sift Web data and alert them to interesting changes in the available information.

The Page Digest Sentinel Web change monitoring system provides robust, targeted, and scalable change detection services for Web information sources. Page Digest Sentinels supplies rich data monitoring services for tracking a variety of interesting Web document characteristics. The system is built on an efficient document representation, the Page Digest, and uses sophisticated grouping techniques to ensure scalable performance.

Examples of similar systems that monitor multiple Web sources for multiple users include WebCQ [70], the NiagaraCQ project [22], and the Change Detector^{TM} system from WhizBang! Labs [7]. The Page Digest Sentinel system improves upon the WebCQ [70] system by using the more efficient Page Digest encoding to improve I/O times, storage space, and to provide a base for more efficient algorithms. NiagaraCQ is focused on continuous queries for XML documents, and supports standard XML query languages for monitoring how query results over a document change over time. NiagaraCQ is more appropriate for monitoring data documents where the document as a whole and each individual element is strongly typed, while the Page Digest sentinels are more appropriate for monitoring changes over documents typically found on the Web. Change Detector^{TM} [7] monitors entire Web sites for business-specific changes. In contrast, our system focuses on efficient algorithms to detect changes in individual Web pages and to leverage large user bases to reduce redundant network requests and processing costs associated with a large group of sentinels.

The change detection components in our Web monitoring system leverage the Page Digest encoding to provide extended capabilities for monitoring diverse facets of Web documents, including changes to content, links, or structure. Typical Web document change detection algorithms, such as AT&T Labs HTML diff algorithm [23], mark only content changes in the entire document. The Page Digest Sentinel system's change detection algorithms allow selective change detection for only portions of a document, and can easily restrict the scope of change to only one aspect of a document (textual content, links, images, attributes, tag names, or structure).

1.2.3 Document Handling: Existing approaches and XPack

The XML document format [8] is a popular document encoding for online information exchange due to its well-defined semantics, strong internationalization support [88], and a plethora of developer tools for managing and exchanging data. XML data is self-describing, allowing clear, descriptive names to ease human cognition. However, XML has two disadvantages that hinder adoption as an information exchange medium: the size penalty and textual representation. While XML provides advantages with its self-describing characteristics and universally recognized format, many applications cannot afford the performance penalty associated with converting to XML.

Compressing XML documents is a popular way to mitigate the markup overhead. The foundation work for modern data compression was done by Lempel and Ziv [105], who proposed the idea of the dictionary compressor, which replace repeated occurrences of a given string with a shorter sequence. Recently, there have been several efforts to design XML-specific compression algorithms. The first, XMill [65], was designed to promote standardized document storage and transmission formats while alleviating the concern of data expansion that is often the penalty of converting data to XML. The XMill compressor achieves this goal by creating a container for each document tag and placing the data values for each tag into the same compressed container. The fundamental problem with the XMill approach is its opaque nature: data compressed with the XMill compressor is only available for use after being decompressed, a costly overhead step that must be added to the overhead of

parsing the text document.

There have been several recent efforts to provide query support over compressed XML documents, typically by making a trade-off between the degree of compression and support for queries. XGRIND [95] compresses XML documents by using Huffman encoding for non-enumerated types and supports exact match and range queries over the compressed XML document. XPRESS [76] maintains the original structure of each XML document to support path queries, but instead uses a technique called reverse arithmetic encoding for compressing labeled paths of the document. In [12], the authors present an XML compression technique that supports path queries over the compressed XML. Their technique relies on the identification of shared subtrees across a single document.

The XPack document compression system supports Web data management by providing an efficient, query capable archive format for Web documents. The XPack document encoding is an extension of our work on the Page Digest system for efficient representation of HTML Web pages [85]. XPack achieves compression performance that is comparable to popular document compression techniques through aggressive document redundancy elimination. Unlike other widely used XML compression systems, XPack supports general query operations over the documents through document compartmentalization, which separates different aspects of Web documents into groups of logical containers. XPack is also a highly flexible system that allows general path expression queries over compressed documents using the standard XML path query language XPath. XPack's efficient encoding scheme yields significant performance advantages for general path queries over the compressed documents when compared with queries executed over unencoded XML.

XPack provides document encoding and compression services, supporting both good compression and efficient querying over compressed documents. XPack's queryable XML compression format is general-purpose, does not rely on domain knowledge or particular document structural characteristics for compression, and achieves better query performance than standard query processors using text-based XML.

1.3 Contribution Summary

This dissertation presents novel approaches for classifying dynamic Web data sources and tracking Web data efficiently. Our research expands the capabilities of existing dynamic Web techniques, providing superior service discovery and classification services, efficient change monitoring of Web information, and compartmentalized document handling.

1. DynaBot is the first system to combine a service class view of the Web with a modular crawling architecture to provide automated service discovery and classification. DynaBot utilizes a service class model of the Web to discover and classify interesting sources without the need for a centralized service registry or human wrapper maintenance.

2. The Page Digest Web document encoding is the first mechanism for representing Web documents that explicitly separates the individual characteristics of the document to enable scalable and efficient access to information on the Web. The Page Digest Sentinel change monitoring system is the first Web document monitor that utilizes the Page Digest document encoding for scalable change monitoring through efficient change algorithms and intelligent request grouping.

3. XPack is the first XML compression system that delivers compression rates similar to existing techniques while supporting better query performance than standard query processors using text-based XML. XPack eliminates the need to choose between compact document sizes and efficient document operations. XPack is general-purpose: the compression scheme does not require domain knowledge or particular document structural characteristics for effective compression.

This dissertation and associated research employ an experimental style and research approach. Our exploration of a problem begins with a research goal, such as integrating autonomous nucleotide BLAST Web sources, and synthesizes iterative discoveries with the current state-of-the-art for the problem to arrive at a novel solution. Where appropriate, we utilize code artifacts to test, validate, and justify design choices. In addition, these artifacts

extend the intellectual merit and potential impact of this work by providing tools for the edification of other researchers and scholars, whom we encourage to download, test, and extend the ideas and systems presented here.

CHAPTER II

DYNABOT

2.1 Introduction

The World Wide Web is the product of two unique approaches to document publication. The traditional or "static" Web consists of documents materialized in the secondary storage of server systems that are hyperlinked to other Web documents. These documents are generally accessible to unauthenticated users and automated agents like search engine crawlers. The dynamic or "Deep Web," in contrast, refers to the dynamic collection of Web documents that are created as a direct response to some user query. Deep Web services provide access to real-time information, like entertainment event listings, or present a Web interface to large databases or other data repositories. Recent estimates place the practical size of the Deep Web at greater than 550 billion individual documents [5] with a growth rate that substantially exceeds that of the static Web [61, 62]. More than half of the content of the Deep Web resides in topic-specific databases, many of which are made available through Web services. A full ninety-five percent of the Deep Web is publicly accessible information that is not subject to fees or subscriptions.

Dynamic content is often ignored by existing search engine indexers owing to the technical challenges that arise when attempting to search the Deep Web. The most significant challenge is the philosophical difference between the static and Deep Web with respect to how data is stored: in the static Web, data is stored in document files while in the dynamic Web, data is stored in databases or produced as the result of a computation. This difference is fundamental and implies that traditional document indexing techniques, which have been applied with extraordinary success on the static Web, are inappropriate for the Deep Web. Related to the data storage issue is the problem of data retrieval, since static Web documents are retrieved via simple HTTP calls while dynamic Web documents often reside behind form interfaces that are impenetrable to traditional crawlers. Finally, Deep Web

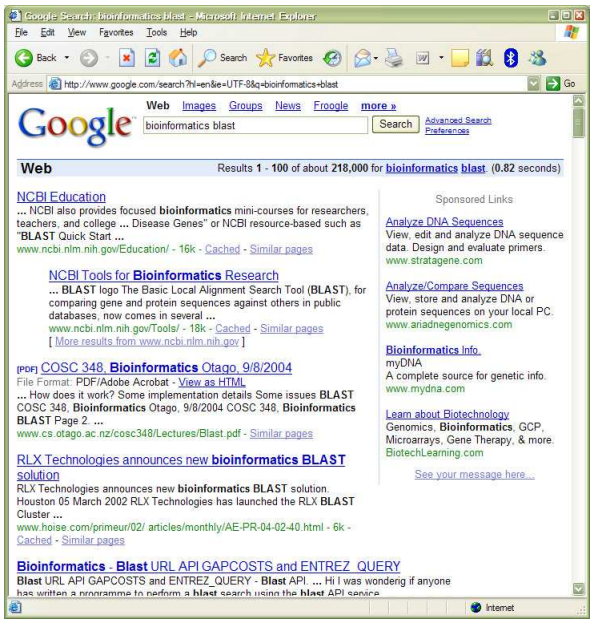


Figure 1: Google results for search 'bioinformatics blast.'

sources tend to be more domain-focused than their static Web counterparts. While there is much to be gained from discovering and clustering Deep Web sources, any significant exploration of the Deep Web will require techniques that exploit service-oriented functionality through intelligent analysis of search forms and result samples.

With these challenges in mind, we present DynaBot, a service-centric crawler for discovering and clustering Deep Web sources. DynaBot has three unique characteristics. First, DynaBot utilizes a service class model of the Web implemented through the construction of service class descriptions (SCD), which provide an abstract representation of a class of services. Second, DynaBot employs a modular, self-tuning system architecture for focused crawling of the Deep Web. Third, DynaBot incorporates methods and algorithms for efficient probing of the Deep Web and for discovering and clustering Deep Web sources and services through SCD-based service matching analysis. Using an appropriate service class description for the domain of interest and a relevant seed list—such as the Google results shown in Figure 1 for the search 'bioinformatics blast' used for finding nucleotide BLAST services in our experimental evaluation—DynaBot can discover, probe, and classify Web services of interest.

The major challenges facing DynaBot are identification of potential services, determining the capabilities of identified services, classifying services, managing data generated throughout the classification process, and ranking of both services and the results they produce. DynaBot employs a modular crawler architecture to address the data management problem and collects potential sources by crawling the Web and identifying servers with a forms-based interface. DynaBot uses its service class model with associated service class descriptions to determine the capabilities of discovered sources and to classify Web sources as members of a service class. Ranking of services and service results remains an open problem.

2.2 Motivation and Related Work

Research on DynaBot for automatically discovering and classifying dynamic Web sources is motivated by the need to fill the gap between the growth rate of the dynamic Web and the rate at which current tools can interact with these sources. More precisely, given a domain of interest with defined operational interface semantics, can we provide superior Web source identification, classification, and integration services than those offered by existing sources which require significant human involvement?

Automated integration of dynamic Web sources requires (1) contact information for each source and (2) a source capability definition that specifies what the source does and how to interact with it. The availability of a consistently defined registry of services is a common assumption in Web service systems [48], where both of these problems are relegated to the service registry. In typical Web service architectures, services publish their location and capabilities to the registry, thereby eliminating the problems of locating sources and discovering their interaction mechanics.

The most popular service registry standard for Web Services is Universal Description, Discovery, and Integration of Web services, or UDDI [98, 97], which, despite broad industry backing, has thus far failed to achieve the critical deployment mass needed for automated interaction with the dynamic Web [99]. UDDI defines a registry for descriptions of Web

services; each service description provides information on the publishing entity for the service, a textual description of what the service does, and a service API defined in a standard service description format such as WSDL. Concerns over security and trust along with lackluster service publication rates limit the utility of UDDI for automated dynamic Web source interaction.

Dynamic Web source integration systems that do not assume the presence of a universal service registry utilize either a wrapper-mediator approach or some form of interface inferencing. Wrapper-mediator systems are by far the most popular of the two and have been used for integration of legacy database sources [29]; several research and commercial systems exist for querying heterogeneous Web data sources. Zadorozhny et al. [104] describe a wrapper and mediator system for limited-capability Web sources that includes query planning and rewriting capabilities. Information Manifold [64] targets the myriad of Web interfaces to general purpose data, using a declarative source description for these sources combined with a set of query planning and optimization algorithms. The TSIMMIS [20] system provides mechanisms for describing and integrating diverse data sources while focusing on assisting humans with information processing and integration tasks. Researchers have also examined heterogeneous data integration in the domain of biological data. DiscoveryLink [51] provides access to wrapped data sources and includes query planning and optimization capabilities. Eckman et al. [37] present a similar system with a comparison to many existing related efforts. BioKleisli [33] provides access to complex sources with structured data but does not include query optimization.

Wrapper mediator systems like these rely on human operators for source discovery and the task of creating and maintaining wrapper programs to interface the mediator system with the target sources. The size of the dynamic Web renders such a labor-intensive process unscalable and impractical. The interface inferencing approach, in contrast, uses automated techniques to discover the interface capabilities of a Web source. Wrapper induction [59, 58, 40] takes a machine learning artificial intelligence approach to interface inferencing. Sources are modeled as a set of tuples; the induction process uses labeled example query responses to learn a wrapper for extracting tuples from pages. Wrapper induction addresses neither

the problem of classifying a source nor that of locating new sources: preselected and labeled sources are hand fed to the wrapper learning system. The wrapper induction process does not provide any mechanism for discovering the semantics of the form inputs to a source.

The ShopBot agent [35] uses an alternate inferencing technique and is designed to assist users in the task of online shopping. Rather than relying on tagged examples, ShopBot uses a domain description that lists useful attributes about the services in question. The authors addressed the problems of learning unknown vendor sites and integrating a set of learned sources into a single interface.

There have been several research projects [17, 66, 50, 32, 49, 56, 73, 18] examining the service selection and classification problem in the context of keyword queries over document databases. Unlike registry-based approaches, these approaches do not rely on a technical service description but employ service estimation techniques that utilize document term frequencies and term weights like TFIDF for each document. Most methods rely on the individual databases exporting a list of terms and their related frequencies although a few, such as the QProber work by Ipeirotis, Gravano, and Sahami [49], estimate the contents of text databases through probing queries. These projects focus on services that expose document databases to the Web through keyword search forms.

DynaBot provides a foundation for unifying complex Web data sources using automatic discovery and capability detection. It addresses the problem of locating new services by building upon a modular crawler architecture that uses static Web information to locate dynamic Web sources. DynaBot utilizes an abstract description of a class of services, the service class description, to classify unknown services and discover their interface semantics [87, 78]. Relative to document database term analysis techniques, DynaBot can be utilized for more complex services and in domains where document term analysis is inappropriate. DynaBot's crawler architecture allows it to locate new services of interest automatically, a feature not present in other automatic wrapper or inferencing approaches. The service class description format we describe provides greater descriptive power than ShopBot's domain descriptions and can specify complex data types and source control flow information.

Web crawlers have been searching and indexing the static Web since nearly the time of its creation. Starting from a set of seed pages, a crawler traverses the Web and processes the sites it encounters while extracting new hyperlinks to crawl from the encountered sites. Crawlers have generated commercial and research interest due to their popularity [81] and the technical challenges involved with scaling a crawler to handle the entire Web [10, 75, 54, 11]. There is active research into topic driven or focused crawlers [19] which crawl the Web looking for sites relevant to a particular topic; Srinivasan et al. [92] present such a crawler for biomedical sources that includes a treatment of related systems.

2.3 The Service Class Model

We introduce the concept of service classes to facilitate the discovery and classification of Deep Web sources with respect to the services that they provide. The service class model views the Deep Web as a collection of *service classes*, which are dynamic sources with related functions.

> A *service class* is a set of Web sources or services that provide similar functionality or data access.

The definition of the desired functionality for a service class is specified in a *service class description*, which defines the relevant elements of the service class without specifying instance-specific details. The service class description articulates an abstract interface and provides a reference for determining the relevance of a particular Deep Web source to a given service class. The service class description is initially composed by a user or service developer and can be further revised via automated learning algorithms embedded in the DynaBot service probing and matching process.

> A *service class description* (SCD) is an abstract description of a service class that specifies the minimum functionality that a Web source must export in order to be classified as a member of the service class.

The service class model supports the dynamic Web source discovery problem by providing a general description of the data or functionality provided. A service class description

encapsulates the defining components that are common to all members of the class and provides a mechanism for hiding insignificant differences between individual sources, including interface discrepancies that have little impact on the functionality of the source. In addition, the service class description provides enough information to differentiate between a set of arbitrary Web sources.

As an example, consider the problem of locating members of the service class of Web keyword search engines such as Google, Yahoo!, and Teoma. The relevant input features in this service class are a text box that accepts descriptive keywords and a button for sending the query to the server. The relevant output features are a set of results, each of which consists of some text containing the query keywords and a hyperlink to a new document. Note that this description says nothing about the implementation details of any particular instance of the service class; rather, it defines a minimum functionality set needed to classify a Web source as a member of the Web keyword search service class.

Our initial prototype of the DynaBot service discovery system utilizes a service class description composed of three building blocks: type definitions, a control pattern, and a set of probing templates. The remainder of this section describes each of these components with illustrative examples.

2.3.1 Type Definitions

The first component of a service class description specifies the data types that are used by members of the service class. Types are used to describe the input and output parameters of a service class and any data elements that may be required during the course of interacting with a source. The DynaBot service discovery system includes a type system that is modeled after the XML Schema [38] type system with constructs for building atomic and complex types. This regular expression-based type system is useful for recognizing and extracting data elements that have a specific format with recognizable characteristics. Since DynaBot is designed with a modular, flexible architecture, the type system is a pluggable component that can be replaced with an alternate implementation if such an implementation is more suitable to a specific service class.

```
<type name="DNASequence"
      type="string"
      pattern="[GCATgcat-]+" />

<type name="AlignmentSequenceFragment" >
    <element name="AlignmentName"
             type="string"
             pattern="[:alpha:]+:" />
    <element type="whitespace" />
    <element name="start-align-pos"
             type="integer" />
    <element type="whitespace" />
    <element name="Sequence"
             type="DNASequence" />
    <element type="whitespace" />
    <element name="end-align-pos"
             type="integer" />
</type>
```

Figure 2: Nucleotide BLAST: type definitions.

The regular expression type system provides two basic types, atomic and complex. *Atomic types* are simple valued data elements such as strings and integers. The type system provides several built-in atomic types that can be used to create user-defined types by restriction. Atomic types can be composed into *complex types*, which are formed by composition of basic types into larger units.

The `DNASequence` type in Figure 2 is an example of an atomic type defined by restriction in the nucleotide BLAST service class description. Each type has a type name that must be unique within the service class description. Atomic types include a base type specification (e.g. `type="string"`) which can reference a system-defined type or an atomic type defined elsewhere in the service class description. The base type determines the characteristics of the type that can be further refined with a regular expression pattern that restricts the range of values acceptable for the new type. More intricate types can be defined using the complex type definition, which is composed of a series of elements. Each element in a complex type can be a reference to another atomic or complex type or the definition of an atomic type. List definitions are also allowed using the constraints `minOccurs` and

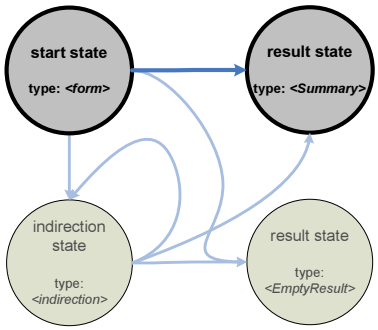
`maxOccurs`, which define the expected cardinality of a particular sub-element within a type. The `choice` operator allows types to contain a set of possible sub-elements from which one will match. Figure 2 shows the declaration for a complex type that recognizes a nucleotide BLAST result alignment sequence fragment, which is a string similar to:

```
Query:  280 TGGCAGGCGTCCT 292
```

The above string in a BLAST result would be recognized as an `AlignmentSequenceFragment` by the type recognition system during service analysis.

2.3.2 Control Flow Graph

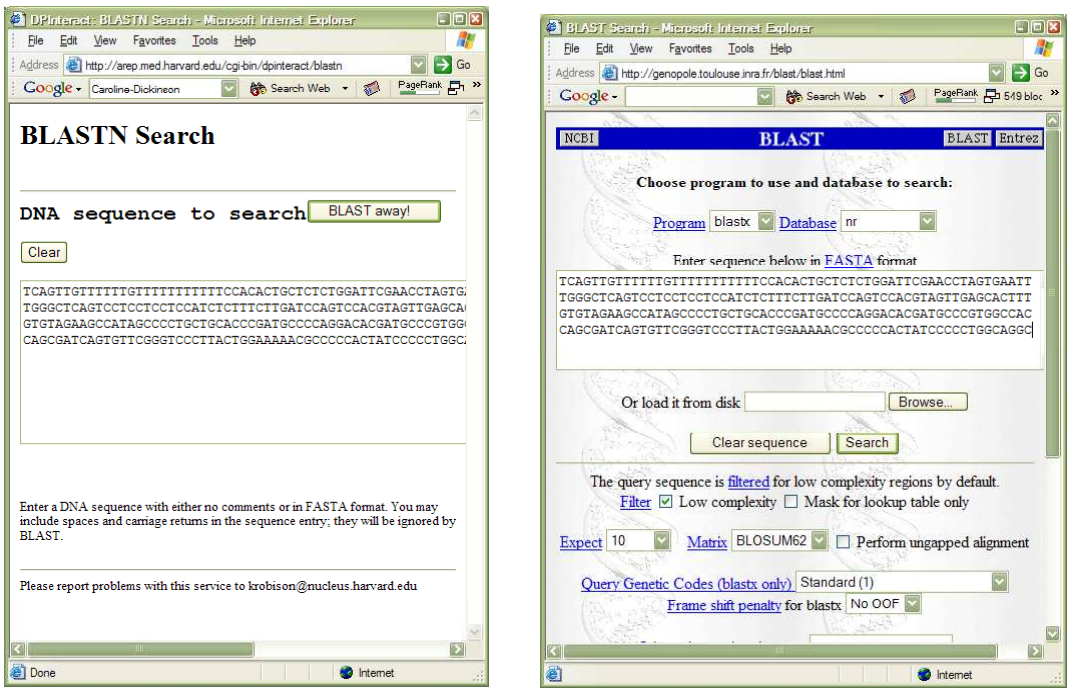
The second component of a service class description is the control flow graph. A service class description's *control flow* graph consists of a set of states connected by edges that reflect the expected navigational paths used by members of the service class. Each state has an associated type; data from a Web source is compared against the type associated with the control flow states to determine the flow of execution of a source from one state to another. Control proceeds from a start state through any intermediate states until a terminal (result) state is reached.



State	Transition States
start	**summary result**
	empty result
	indirection
indirection	indirection
	summary result
	empty result

Figure 3: Nucleotide BLAST: control flow graph.

Figure 3 provides an illustrative example of a service class control flow graph for a nucleotide BLAST Web service. The control flow graph has four state nodes that consist of a state label and a data type. Nodes in the graph represent control points and directed edges depict the possible transition paths between the control states. The state nodes correspond to pages expected to be encountered while interacting with the site. The nucleotide BLAST



(a) (b)

Figure 4: Example nucleotide BLAST simple (a) and complex (b) search forms.

service class description, for example, has a single start state that defines the type of start page a class member must contain: in this case, any member of the nucleotide BLAST service class must have a start page that includes an HTML form with at least one text entry field. Figure 4 shows an example of a simple (a) and a complex (b) entry page to a BLAST service that matched the start state in the control graph.

The control flow graph defines the expected information flow for a service and gives the automated service analyzer, described in Section 2.4.2, a frame of reference for comparing the responses of the candidate service with the expected results for a member of the service class. For example, the most common transition in the nucleotide BLAST service class is from the start state, a page with an HTML form, to the results summary state; this transition is highlighted in Figure 3. In order to declare a candidate service a match for the service class description, the service analyzer must be able to produce a set of valid state transitions in the candidate service that correspond to a path in the control flow graph.

Figure 5 presents a visual example demonstrating how DynaBot uses the nucleotide BLAST service class description control flow graph to match a member of the nucleotide

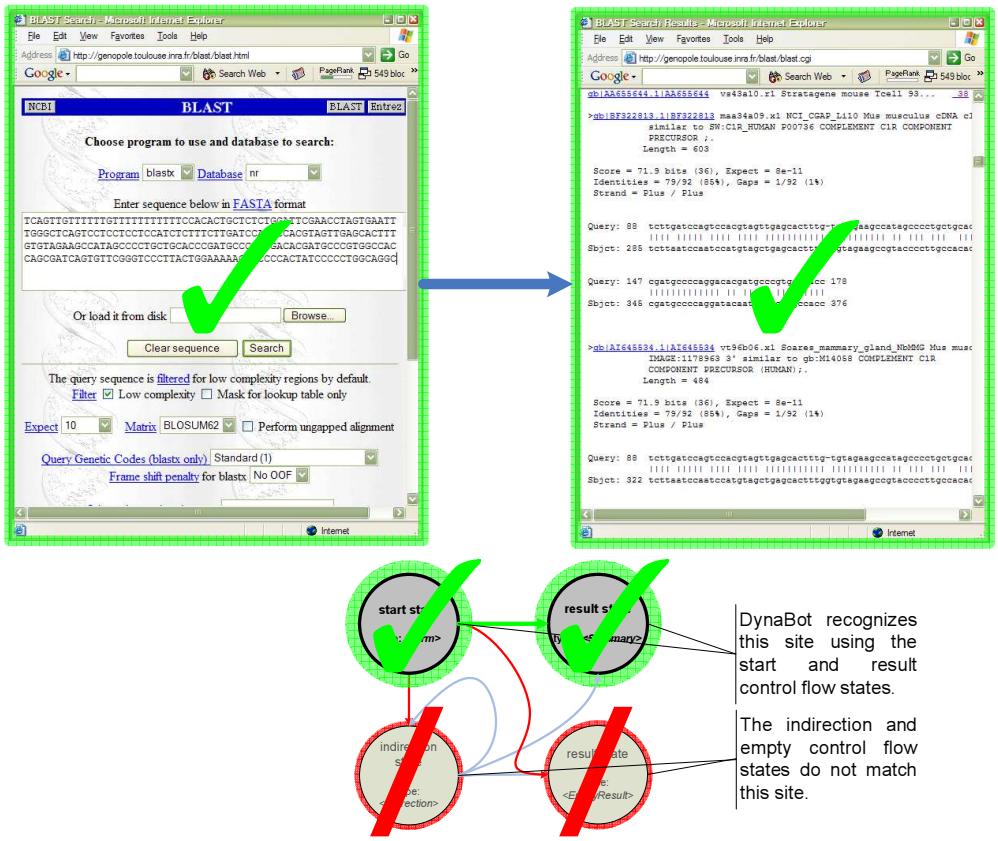


Figure 5: Nucleotide BLAST: example site match with corresponding control flow graph states.

BLAST service class. The top portion of Figure 5 shows the transitions from the Web source: data consisting of a query nucleotide sequence is entered into the form shown on the left and the server's response is shown on the right. The corresponding control flow graph states are shown below the Web forms. Since the start form matches the start state of the control flow graph and the response page matches a terminal result state in the graph, this Web source is classified as a match for the nucleotide BLAST service class description.

Figure 6 shows a contrasting example for a protein BLAST site that does not match the nucleotide BLAST service class description. Although protein and nucleotide BLAST Web sources have similar interfaces—results differ in content but not necessarily in form—the DynaBot service analyzer is able to distinguish between the two types of services using the control flow graph. In Figure 6, the protein BLAST Web source's data entry form resembles the start forms of many nucleotide BLAST sources enough to match the service

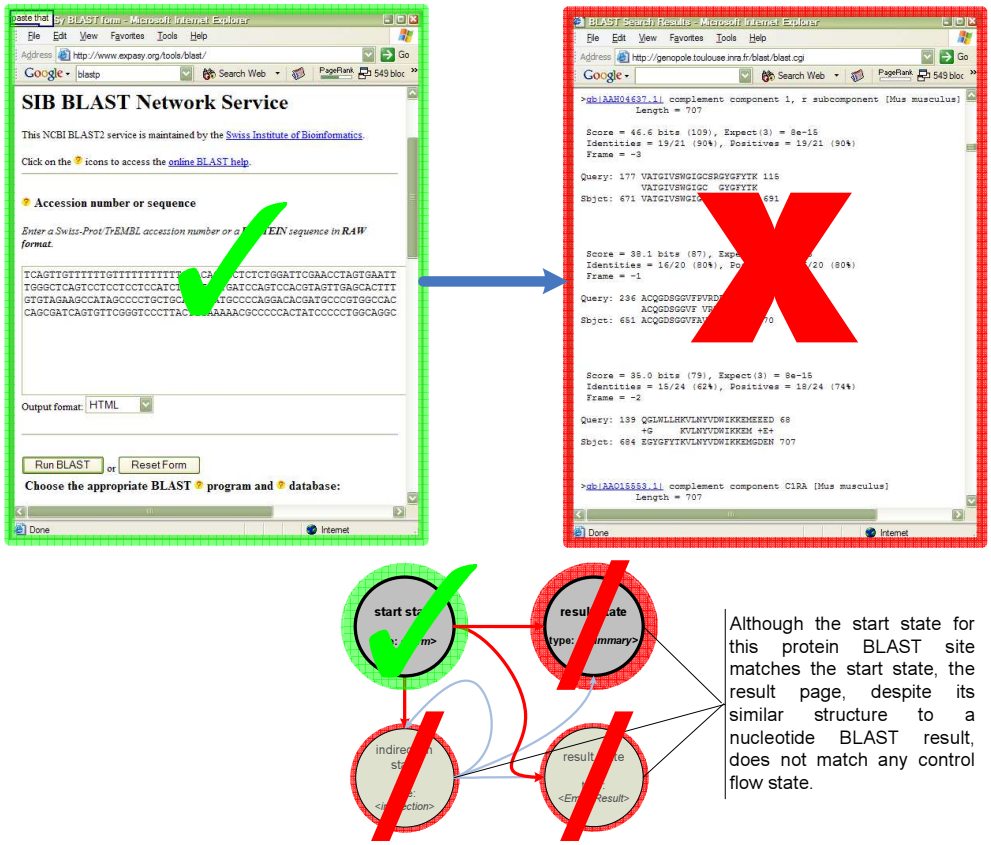


Figure 6: Nucleotide BLAST: example nonmatching site with corresponding control flow graph states.

class description's start state. However, after receiving the result response from the Web source, DynaBot's service analyzer is unable to match the response to any valid transition state. In particular, despite the resemblance between the protein BLAST response and the nucleotide BLAST response, the control flow graph's result state matches the nucleotide type only and therefore rejects the protein BLAST server's response.

2.3.3 Probing Templates

The third component of the service class description is the probing templates, which contain a set of input arguments and can be used to match a candidate Deep Web source against the service class description and determine if it is an instance of the service class. Probing templates are composed of a series of arguments and a single result type. The arguments are used as input to a candidate service's forms while the result type specifies the data type of the expected result. Figure 7 shows an example probing template used in a nucleotide

```
<example>
  <arguments>
    <argument required="true">
      <name>sequence</name>
      <type>DNASequence</type>
      <hints>
        <hint>sequence</hint>
        <inputType>text</inputType>
      </hints>
      <value>TTGCCTCACATTGTCACTGCAAAT
             CGACACCTATTAATGGGTCTCACC
      </value>
    </argument>
  </arguments>

  <result type="SummaryPage" />
</example>
```

Figure 7: Nucleotide BLAST: probing template.

BLAST service class description. The probing template example shows an input argument and a result type specification; multiple input arguments are also allowed. The attribute `required` states whether an argument is a required input for all members of the service class. In this case, all members of the nucleotide BLAST service class are required to accept a DNA sequence as input. The argument lists the type of the input as well as a value that is used during classification. The optional `hints` section of the argument supplies clues to the site classifier that help select the most appropriate input parameters on a Web source to match an argument. Finally, the output result specifies the response type expected from the source. All the types referenced by a probing template must have type definitions defined in the type section of the service class description.

The argument hints specify the expected input parameter type for the argument and a list of likely form parameter names the argument might match. Multiple name hints are allowed, and each hint is treated as a regular expression to be matched against the form parameters. These hints are written by the service class description writer using their observation of typical members of the service class. For example, a DNA sequence is almost

always entered into a text input parameter, usually with "sequence" in its name. The DNA Sequence argument in a nucleotide BLAST service class therefore includes a name hint of "sequence" and an input hint of "text."

2.4 DynaBot *Software Design*

The problem of discovering and analyzing dynamic Web sources consists of locating potential sources and determining their interface and capabilities. There are two basic approaches to service discovery: the registry-based approach and the crawling approach. In the registry-based approach, services advertise their existence and capabilities with a service registry such as the emerging UDDI directory standard. However, registry-based discovery systems have several drawbacks. Many of these technologies are still evolving and have limited deployment. In addition, registry-based discovery relies on services correctly advertising themselves in a known repository, effectively limiting the number of services that can be discovered. Finally, despite the registry approach's ability to avoid interacting with unrelated services, the limited descriptive power in existing registry standards implies that service analysis is still required to ascertain a service's capabilities. The second approach to service discovery is the Web crawling approach, which builds on Web crawler technology to locate candidate services. This approach is widely applicable to the existing Web, removes the burden of registration from service providers, and can be extended to exploit service registries to aid service discovery.

2.4.1 Architecture

The DynaBot crawler is a modular Web crawling platform designed to locate and analyze Web sources relevant to a service class of interest. Like most crawlers, it utilizes the simple but universal Web crawling algorithm that was proposed nearly simultaneously with the Web itself. First, the crawler chooses a URL from the URL frontier [54]—the crawler's list of URLs to visit, which is seeded by hand at the beginning of each crawl run. Next, the crawler fetches the document specified by the chosen URL. Previously unseen links are added to the URL frontier and any further document processing is done. The crawler then returns to the first step. Although a Web crawler is conceptually very simple, a robust crawler must

Table 1: Web crawler feature comparison.

	Network Interaction	**Data Management**	**Processing Modules**
Basic Crawler	name resolver, document retrieval	URL frontier, visited list	link extractor, keyword index builder
Advanced Crawler	caching name resolver, multithreaded document retrieval	URL frontier, visited list, document cache	link extractor, keyword index builder, mirror detector, trap detector
DynaBot **Crawler**	caching name resolver, multithreaded document retrieval	URL frontier, visited list, document cache	link extractor, service class analyzer

handle the immense size of the gathered data, gracefully deal with the numerous errors that can occur, and even avoid malicious servers that attempt to lure the crawler into a trap.

Crawling the Web for dynamic data sources shares many features with standard Web crawling. Table 1 compares the features of three classes of Web crawlers: basic crawlers, advanced crawlers, and our own DynaBot crawler. The components that make up a crawler are divided among three major component groups: network interaction modules, global storage and associated data managers, and document processing modules. The simplest crawlers require mechanisms for retrieving documents and determining if a particular URL has been seen. More advanced crawlers will include features like mirror site detection and trap avoidance algorithms. DynaBot utilizes an advanced crawler architecture for source discovery and adds a document processor that can determine if a dynamic Web source is related to a particular domain of interest. Figure 8 shows the architecture of DynaBot.

Network Interaction. The network interaction modules handle the process of retrieving documents from the Internet, including the resolution of domain names. The significant costs associated with accessing data over the Internet can be amortized using multithreading to handle multiple requests simultaneously. This technique minimizes the penalty incurred when attempting to access a document from a server that is down or extremely slow. A second optimization technique is to cache DNS requests to reduce the number of network interactions needed and thereby improve document throughput. The effectiveness of DNS

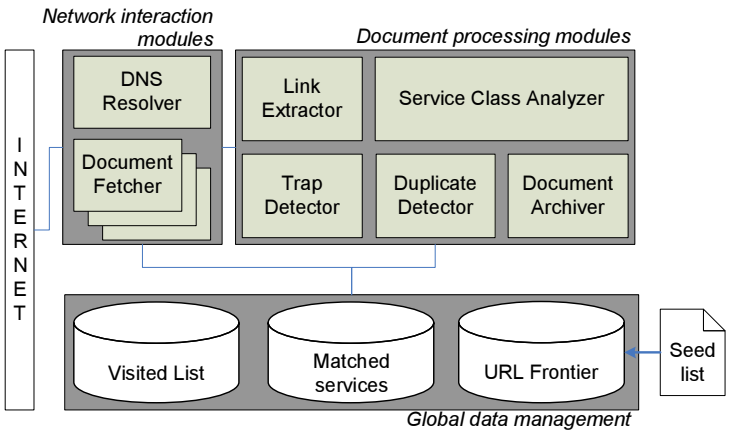


Figure 8: DynaBot System Architecture

caching is due to the high degree of domain locality in Web hyperlinks, which often reference different documents on the same server. In such cases, utilizing a cache insures that DNS name resolution is done only once for all documents in the domain. DynaBot retrieves documents asynchronously using a pool of document fetch modules to maintain high document throughput.

Global Data Management. Managing the immense amount of data that a Web crawler will encounter is a technically interesting problem due to the sheer size of the Web. The crawler's global data management and storage components include the URL frontier and the visited list, which are responsible for tracking URLs the crawler has yet to visit as well as those that have already been processed. DynaBot also maintains a list of services that have been matched to a service class description in its matched services data store. Global data is typically stored on disk with caches used to reduce the performance penalty incurred when moving data to and from core memory.

Processing Modules. The network interaction and global storage components are united by the processing modules, which initiate document retrieval, update the global storage with visited and new links, and perform any document processing required by the crawler's designated task. Processing modules are pluggable components that allow the crawler to be reconfigured for new tasks easily. The DynaBot crawler includes a link extraction module, which extracts hyperlinks from the document, converts any relative links

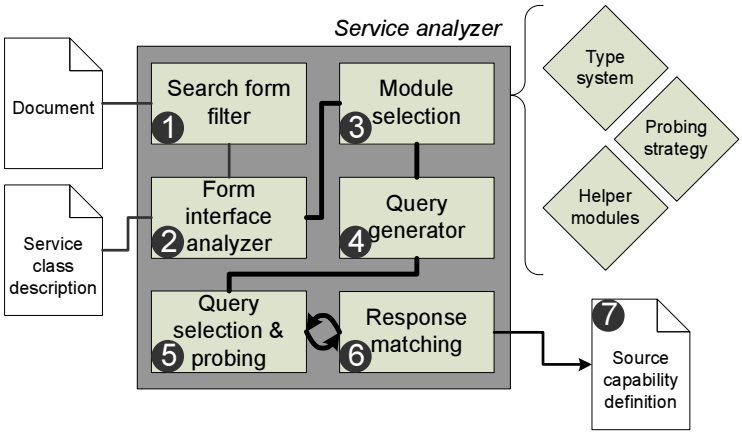


Figure 9: DynaBot Service Analyzer

to their absolute form, and inserts them into the URL frontier. The task of determining the capabilities and interface of dynamic Web sources is assigned to the Service Analyzer, which is the primary processing module of the DynaBot crawler.

2.4.2 Service Analyzer

The process of source discovery begins with the construction of the service class description, which directs the probing operations used by the service analyzer to determine the relevance of a Web source. The *service analyzer* consists of a form filter and analyzer, an extension mechanism, a query generator, a query prober, and a response matcher.

Overview. When the processor encounters a new site to test, its first task is to invoke the *form filter*, which ensures that the candidate source has a form interface (Figure 9(1)). The second step (2) is to extract the set of forms from the page, load the service class description, and load any auxiliary modules specified by the service class description (3). The query generator (4) produces a set of query probes which are fed to the query probing module (5). Responses to the query probes are analyzed by the response matcher (6). If the query response matches the expected result from the service class description, the Web source has matched the service class description and a source capability profile (7) is produced as the output of the analysis process. The capability profile contains the specific steps needed to successfully query the Web source. If the probe was unsuccessful, additional probing queries can be attempted.

Definitions. The process of analyzing a Web source begins when the crawler passes a potential URL for evaluation to the source analysis processing module. A source S for our purposes consists of an initial set of forms F. Each form $f \in F, f = (P, B)$ is composed of a set of parameters $p \in P, p = (t, i, v)$ where t is the *type* of the parameter, such as checkbox or list, i is the parameter's *identifier*, and v is the *value* of the parameter. The form also contains a set of buttons $b \in B$ which trigger form actions such as sending information to the server or clearing the form. The source S may specify a default for each parameter value v.

The process of *query probing* involves manipulating a source's forms to ascertain their purpose with the ultimate goal of determining the function of the source itself. Although the expected inputs and purpose of each of the various parameters and forms on a source is usually intuitive to a human operator, an automated computer system cannot rely on human intuition and must determine the identity and function of the source's forms algorithmically. The query probing component of the DynaBot service analyzer performs this function. Our query prober uses induction-based reasoning with examples: the set of examples $e \in E$ is defined as part of the service class description. Each example e includes a set of arguments $a \in A, a = (r, t, v)$, where r indicates if the example parameter is required or optional, t is the type of the parameter, and v is the parameter's value.

Form Filter and Analyzer. The *form filter* processing step helps to reduce the service search space by eliminating any source S that cannot possibly match the current service class description. In the filtration step, shown in step 1 of Figure 9, the form filter eliminates any source S from consideration if the source's form set is empty, that is $F = \emptyset$. In form analysis, shown in step 2, the service class description will be compared with the source, allowing the service analyzer to eliminate any forms that are incompatible with the service class description. Algorithm 1 sketches the steps involved in the form filter process.

Module Selection. The modular design of the service class description framework and the DynaBot discovery and analysis system allows many of the system components to be extended or replaced with expansion modules. For example, a service class description may reference an alternate type system or a different querying strategy than the included

Algorithm 1 Form Filter

Let $S \leftarrow$ source with forms $f \in F, f = (P, B)$
Let $D \leftarrow$ the service class description with examples $e \in E$

for all $f = (P, B) \in F$ **do**
  for all $e \in E$ **do**
    for all $a = (r, t, v)$ s.t. $required(a) =$ true **do**
      if $\nexists\, p = (t, i, v) \in P$ s.t. $a_t = p_t$ **then**
        $F = F - f$
if $F \neq \emptyset$ **then**
  $processForms(F)$

versions. Step 3 in the service analysis process resolves any external references that may be defined in the service class description or configuration files and loads the appropriate code components.

Query Generation. The heart of the service analysis process is the query generation, probing, and matching loop shown in steps 4, 5, and 6 of Figure 9. Generating high quality queries is a critical component of the service analysis process, as low-quality queries will result in incorrect classifications and increased processing overhead. DynaBot's query generation component is directed by the service class description to ensure relevance of the queries to the service class. Queries are produced by matching the probing templates from the service class description with the form parameters in the source's forms; Figure 7 shows a fragment of the probing template for the nucleotide BLAST service class description.

Probing and Matching. Once the queries have been generated, the service analyzer proceeds by selecting a query, sending it to the target source, and checking the response against the result type specified in the service class description. This process is repeated until a successful match is made or the set of query probes is exhausted. On a match, the service analyzer produces a source capability profile of the target source, including the steps needed to produce a successful query.

Figure 10 shows the probing results from two different services analyzed with the same nucleotide BLAST service class description. Source (a) is a member of the nucleotide BLAST service class while source (b) is a member of the protein BLAST service class, a related type of service that uses a similar interface to nucleotide BLAST but performs a

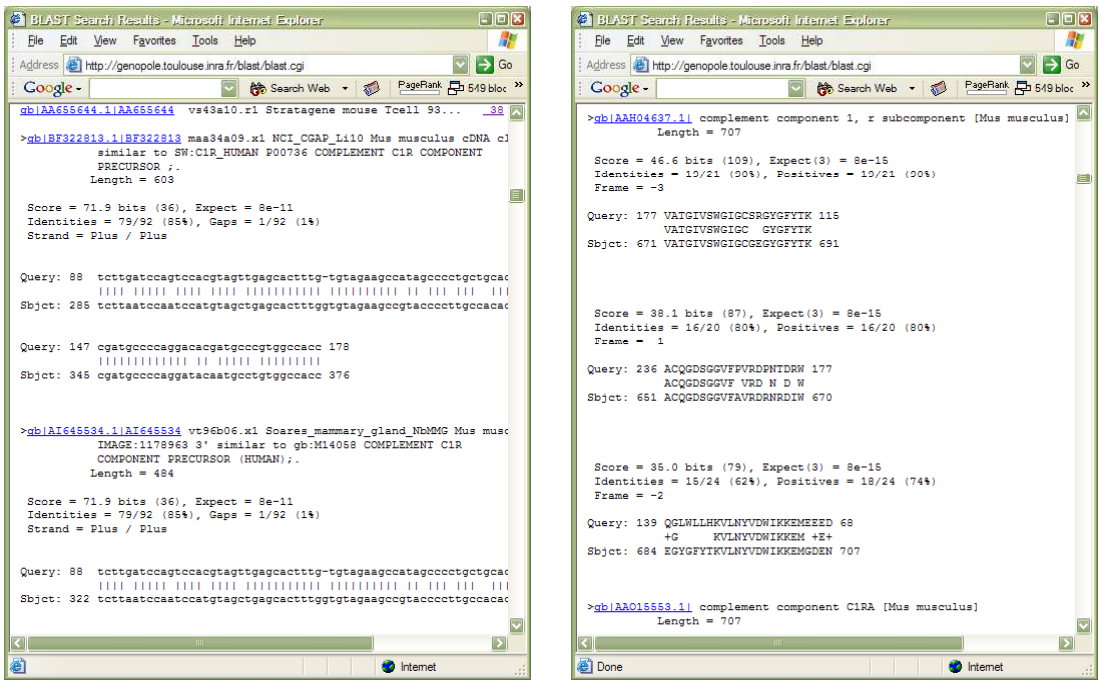


Figure 10: Example matching (a) and nonmatching (b) search results

different function. Using the type definitions from the service class description, the service analyzer is able to determine that (a) is an appropriate response for a member of the nucleotide BLAST service class while (b), although structurally similar, is not an appropriate response. This information allows the service analyzer to correctly classify these two sources despite the similarity of their responses: source (a) is declared a match while source (b) is not.

Algorithm 2 presents a sketch of the query probing and matching process. Our prototype implementation includes invalid query filtering and some heuristic optimizations that are omitted from the algorithm presented here for clarity's sake. These optimizations utilize the hints specified in the probing template section of the service class description to match probing arguments with the most likely candidate form parameter. For instance, the nucleotide BLAST service class description specifies that form parameters named "sequence" that accept text input are very likely to be the appropriate parameter for the `DNASequence` probe argument. These hints are static and must be selected by the service class description author; our ongoing research includes a study of the effectiveness of learning techniques for

matching template arguments to the correct form parameters. We expect that the system should be able to deduce a set of analysis hints from successfully matched sources which can then be used to enhance the query selection process.

Algorithm 2 Query Probing

```
Let S ← source with forms f ∈ F, f = (P, B)
Let D ← the service class description with examples e ∈ E

for all f ∈ F do
  Let Q ← E × P
  for all q ∈ Q do
    Let r ← executeQuery(q)
    if responseMatches(r, D) then
      processMatch(r, q, D)
```

2.5 Experimental Results

We have developed a set of experiments based on the DynaBot prototype service discovery system to test the validity of our approach. The experiments were designed to test the accuracy and efficiency of DynaBot and the service probing and matching techniques. We have divided our tests into three experiments. The first experiment is designed to test only the probing and matching components of the crawler without the confounding influence of an actual Web crawl. Experiment 2 tests the performance of the entire DynaBot system by performing a Web crawl and analyzing the potential sources it encounters. Experiment 3 shows the effectiveness of pruning the search space of possible sources by comparing an undirected crawler with one using a more focused methodology.

The DynaBot prototype is implemented in Java and can examine a set of supplied URLs or crawl the Web looking for sources matching a supplied service class description. All experiments were executed on a Sun Enterprise 420R server with four 450 MHz UltraSPARC-II processors and 4 GB memory. The server runs SunOS 5.8 and the Solaris Java virtual machine version 1.4.1.

Crawler Configuration. The DynaBot configuration for these experiments utilized several modular components to vary the conditions for each test. All of the configurations used the same network interaction subsystem, in which domain name resolution, document

retrieval, and form submission are handled by the HttpUnit user agent library [43]. The experiments utilized the *service analyzer* document processing module for service probing and matching. Service analysis employed the same static service class description in all the tests, fragments of which have been shown in Figures 2 and 7. All of the configurations also included the *trace generator* module which records statistics about the crawl, including URL retrieval order, server response codes, document download time, and content length. 32 crawling threads were used in each run.

We utilized two configuration variations in these experiments: the trace configuration and the random walk configuration. The trace configuration is designed to follow a predetermined path across the Web and utilizes the trace URL frontier implementation to achieve this goal. This frontier accepts a seed list in which any URLs found are crawled in the order that they appear in the list. These seed lists can be either hand generated or generated from previous crawls using the trace generator. In the trace configuration, no URLs can be added to the frontier and no attempt is made to prevent the crawler from retrieving the same URL multiple times.

The random walk configuration mimics more traditional Web crawlers but attempts to minimize the load directed at any one server. In this configuration, the *link extractor* module was employed to extract hyperlinks from retrieved documents and insert them into the URL frontier. The random walk frontier implementation uses an in-memory data structure to hold the list of URLs that have yet to be crawled, from which it selects one at random when a new URL is requested. This configuration also includes a visited list, which stores hash codes of URLs that have been visited which the crawler can check to avoid reacquiring documents that have already been seen.

2.5.1 Experiment 1: BLAST Classification

The first experiment tested the service analyzer processing module only and demonstrates its effectiveness quantitatively, providing a benchmark for analyzing the results of our subsequent experiments. In order to test the service analyzer, the crawler was configured to utilize the trace frontier with a hand-selected seed list.

Table 2: Sites classified using the nucleotide BLAST service class description.

Crawl Statistics	
Number of BLAST sources analyzed	74
Total number of forms	79
Total number of form parameters	913
Total of forms submitted	1456
Maximum submissions per form	60
Average submissions per form	18.43
Number of matched sources	53
Success rate	**72.97%**

Aggregate Probe Times	
Minimum probe time	3 ms
Minimum fail time (post FormFilter)	189 s
Maximum fail time (post FormFilter)	11823 s
Average fail time (post FormFilter)	2807 s
Minimum match time (post FormFilter)	2.3 s
Maximum match time (post FormFilter)	2713 s
Average match time (post FormFilter)	284 s

The data for this experiment consists of 74 URLs that provide a nucleotide BLAST gene database search interface; this collection of URLs was gathered from the results of several manual Web searches. The sites vary widely in complexity: some have forms with fewer than 5 input parameters, while others have many form parameters that allow minute control over many of the options of the BLAST algorithm. Some of the sources include an intermediate step, called an indirection, in the query submission process. A significant minority of the sources use JavaScript to validate user input or modify parameters based on other choices in the form. Despite the wide variety of styles found in these sources, the DynaBot service analyzer is able to recognize a large number of the sites using a nucleotide BLAST service class description of approximately 150 lines.

Tables 2 and 3 show the results of Experiment 1. Sites listed as successes are those that can be correctly queried by the analyzer to produce an appropriate result, either a set of alignments or an empty BLAST result. An empty result indicates that the site was queried correctly but did not contain any results for the input query used. Since all of the URLs in this experiment were manually verified to be operational members of the service class, a perfect classifier would achieved a success rate of 100%; Table 2 demonstrates that the

Table 3: Experiment 1 probing statistics.

Number of probes	Frequency
0	12
1–10	46
11–20	1
21–30	2
31–40	2
41–50	1
51–60	10

Probe time (sec.)	Frequency
<0.5	3
0.5–1	1
1–5	11
5–10	5
10–50	10
50–100	2
100–500	31
>500	11

DynaBot service analyzer achieves an overall success rate of 73%.

There are several other notable results in the data presented in Tables 2 and 3. The relatively low number of forms per source—79 forms for 74 sources—indicates that most of these sources use single-form entry pages. However, the average number of parameters per form is over 11 (913 parameters / 79 forms = 11.56), indicating that these forms are fairly complex. We are currently exploring form complexity analysis and comparison to determine the extent to which the structure of a source's forms can be used to estimate the likelihood that the source matches a service class description.

Source form complexity directly impacts the query probing component of the service analyzer, including the time and number of queries needed to recognize a source. To grasp the scaling problem with respect to the number of form parameters and the complexity of the service class description, consider a Web source with a single form f containing 20 parameters, that is $|P| = 20$. Further suppose that the service class description being used to analyze the source contains a single probing template with two arguments, $|A| = 2$, and that all of the arguments are required. The number of combinations of arguments with parameters is then $\binom{|P|}{|A|} = \binom{20}{2} = 190$, a large but perhaps manageable number of queries to send to a source. The number of combinations quickly spirals out of control as more example arguments are added, however: with a three-argument example the number of combinations is 1140, four arguments yields 4845, and testing a five argument example would require 15,504 potential combinations to be examined!

Despite the scalability concerns, Table 3 demonstrates the effectiveness of the SCD-directed probing strategy: most of the sources were classified with less than 10 probes (58) in less than 500 seconds (63). These results indicate the effectiveness of the static optimizations employed by the service analyzer such as the probing template hints. Our ongoing research includes an investigation of the use of learning techniques and more sophisticated query scoring and ranking to reduce these requirements further and improve the efficiency of the service analyzer.

Failed sites (27%) are all false negatives that fall into two categories: indirection sources and processing failures. An indirection source is one that interposes some form of intermediate step between entering the query and receiving the result summary. For example, NCBI's [79] BLAST server contains a formatting page after the query entry page that allows a user to tune the results of their query. Simpler indirection mechanisms include intermediate pages that contain hyperlinks to the results. We do not consider server-side or client-side redirection to fall into this category as these mechanisms are standardized and are handled automatically by Web user agents. Recognizing and moving past indirection pages presents several interesting challenges because of their free-form nature. Incorporating a general solution to complex, multi-step Web sources is part of our ongoing work [78].

Processing errors indicate problems emulating the behavior of standard Web browsers. For example, some Web design idioms, such as providing results in a new window or multi-frame interfaces, are not yet handled by the prototype. Support for sources that employ JavaScript is also incomplete. We are working to make our implementation more compliant with standard Web browser behavior. The main challenge in dealing with processing failures is accounting for them in a way that is generic and does not unnecessarily tie site analysis to the implementation details of particular sources.

2.5.2 Experiment 2: BLAST Crawl

Our second experiment tested the performance characteristics of the entire DynaBot crawling, probing, and matching system. The main purpose of this experiment is to demonstrate the need for a directed approach to service discovery. Intuitively, the problem stems from

Table 4: Results from 6/2/2004 crawl, Google 100 BLAST seed, random walk URL frontier.

Crawl Statistics	
Number of URLs crawled	1349
Number of sites with forms	467
Total number of forms	686
Total number of form parameters	2837
Total of forms submitted	4032
Maximum submissions per form	10
Average submissions per form	5.88
Number of matched sources	**2**

Response Code	**Frequency**
200	1212
30x	114
404	18
50x	6

Content Type	**Frequency**
text/html	1238
application/pdf	36
text/plain	23
other	52

the characteristics of the service Web environment: instances of a particular service class, such as nucleotide BLAST, will make up a small fraction of the sites related to the relevant domain, e.g. bioinformatics. Likewise, the sites belonging to any particular domain will constitute a small portion of the complete Web. Experiment 2 provides evidence to support this conjecture and demonstrates the need for intelligent service discovery and resource allocation. An effective service discovery mechanism must use its resources wisely by spending available processing power on sources that are more likely to belong to the target set.

The results of this experiment are presented in Table 4. For this test, the crawler was configured utilizing the random walk URL frontier with link extraction and service analysis. The initial seed for the frontier was the URLs contained in the first 100 results returned by Google for the search "bioinformatics BLAST." URLs were returned from the frontier at random and all retrieved pages had their links inserted into the frontier before the next document was retrieved. These results are not representative of the Web as a whole, but rather provide insight into the characteristics of the environment encountered by the DynaBot crawler during a domain-focused crawl. The most important feature of these results is the relatively small number of matched sources: despite the high relevance of the seed and subsequently discovered URLs to the search domain, only a small fraction of the services encountered matched the service class description. The results from Experiment 1 demonstrate that the success rate of the service analyzer is very high, leading us to believe that the nucleotide BLAST services make up only a small percentage of the bioinformatics

Table 5: Results from 6/2/2004 crawl, Google 500 BLAST seed.

Crawl Statistics	
Number of URLs crawled	174
Number of sites with forms	74
Total number of forms	108
Total number of form parameters	348
Total of forms submitted	2996
Maximum submissions per form	60
Average submissions per form	27.74
Number of matched sources	**0**

(a) Random walk URL frontier.

Crawl Statistics	
Number of URLs crawled	182
Number of sites with forms	71
Total number of forms	137
Total number of form parameters	1038
Total of forms submitted	3340
Maximum submissions per form	60
Average submissions per form	24.38
Number of matched sources	**12**

(b) LinkHint "blast" frontier.

sites on the Web. This discovery does not run counter to our intuition; rather, it suggests that successful and efficient discovery of domain-related services hinges on the ability of the discovery agent to reduce the search space by pruning out candidates that are unlikely to match the service class description.

2.5.3 Experiment 3: Directed Discovery

Given the small number of relevant Web services related to our service class description, Experiment 3 further demonstrates the effectiveness of pruning the discovery search space in order to find high quality candidates for probing and matching. One important mechanism for document pruning is the ability to recognize documents and links that are relevant or point to relevant sources before invoking the expensive probing and matching algorithms. Using the random walk crawler configuration as a control, this experiment tests the effectiveness of using link hints to guide the crawler toward more relevant sources. The link hint frontier is a priority-based depth-first exploration mechanism in which hyperlinks that match the frontier's hint list are explored before nonmatching URLs. For this experiment, we employed a static hint list using a simple string containment test for the keyword "blast" in the URL.

Table 5 presents the results. The seed lists for the URL frontiers in this experiment were similar to those used in Experiment 2 except that 500 Google results were retrieved and all the Google cache links were removed. The link hint focused crawler discovered and matched 12 Web sources with a fewer number of trials per form then its random walk counterpart.

Although the number of URLs crawled in the both tests was roughly equivalent, the link hint crawler found sources of much higher complexity as indicated by the total number of form parameters found: 1038 for the link hint crawler versus 348 for the random walk crawler.

The results of Experiment 3 suggest a simple mechanism for selecting links from the URL frontier to move the crawler toward high quality candidate sources quickly: given a hint word, say "blast," first evaluate all URLs that contain the hint word, proceeding to evaluate URLs that do not contain the hint word only after the others have been exhausted. This scheme can be quite easily implemented using a priority queue. However, the hint list is static and must be selected manually. We are investigating the effectiveness of learning algorithms and URL ranking algorithms for URL selection. This URL selection system would utilize a feedback loop in which the "words" contained in URLs would be used to prioritize the extraction of URLs from the frontier. Words contained in URLs that produced service class matches would increase the priority of any URLs in the frontier that contained those words, while words that appeared in nonmatching URLs would likewise decrease their priority. In order to be effective, this learning mechanism would also need a word discrimination component, such as term frequency inverse document frequency (TFIDF) measure, so that common words like "http" would have little effect on the URL scoring.

2.6 DynaBot *Summary*

DynaBot is a crawler designed to discover and analyze dynamic Web data sources relevant to a domain of interest. DynaBot leverages a service class model of the Web that groups sources based on related functionality. This model is implemented through the construction of service class descriptions (SCDs), which capture the domain knowledge fundamental to analyzing and classifying sources. For service discovery, DynaBot employs a modular, self-tuning system architecture for focused crawling of the Deep Web using service class descriptions. DynaBot's service analyzer incorporates methods and algorithms for efficient probing of the Deep Web and for discovering and clustering Deep Web sources and services through SCD-based service matching analysis.

DynaBot's use of the service class model of the Web, through the construction of service class descriptions, allows an abstract rendition of the target domain to guide the crawler toward relevant sources and probe them for their capabilities. Our experimental results demonstrate the effectiveness of the service class discovery mechanism which achieves recognition rates of up to 73%. Experimental testing has also shown the effectiveness of incorporating service clues into the search process for improved service matching throughput. These results offer techniques for efficiently managing service discovery in the face of the immense scale of the Deep Web.

CHAPTER III

PAGE DIGEST SENTINELS

3.1 Introduction

The World Wide Web offers an incredible communication medium that eliminates many barriers to information broadcast. The relative ease with which new ventures can be created has fostered growth of online communities that publish information on a near limitless array of topics. A significant and continually expanding community of Internet users rely on the Web daily for access to information about all facets of life. Web content delivery mechanisms are nearly as varied as the data, ranging from simple text files served from desktop computers to database-driven dynamic Web applications powered by enterprise computers in distributed cluster environments. Likewise, the rate of change of published data is highly variable: some pages—e.g. stock quote services—are updated frequently, requiring those who need the latest information to constantly check them. Other, more static data sources may only update a few times per year on an irregular basis.

The promise of Web services offers an exciting new opportunity for harnessing the immense collection of information present on the Internet. Full utilization of the Web as an advanced data repository will require sophisticated data management techniques providing mechanisms for discovering new information and integrating that knowledge into existing stores. Current search engines provide core discovery and classification services, but foundational work is still needed to make large-scale Web services a reality.

One important consideration for large Web services is data storage and processing efficiency. Much of the data on the Internet is contained in HTML documents that are useful for human browsing but incur significant drawbacks from a data management perspective. HTML has no real type information aside from layout instructions, so any data contained in the document is mixed with formatting and layout constructs intended to help browser software render pages on screen. Consequently, automated data extraction or comparison

of Web pages is expensive.

We introduce a new document encoding scheme, the Page Digest, to address some of the problems associated with storage and processing of Web documents that enable Web service applications to operate efficiently on a large scale. A Page Digest of a Web document is more compact than HTML or XML format but preserves the original structural and semantic information contained in the source document. Unlike schemes using general compression algorithms, a Page Digest is not compressed from the source nor does it need to be decompressed to be used, which minimizes the processing needed to convert a document from its native format. Further, the digest encoding highlights the tree structure of the original document, greatly simplifying automated processing of digests. Finally, the Page Digest is structurally aware and exposes the information contained in a document's tree shape to applications.

Using the Page Digest as a foundation, we present an automatic Web change detection system that provides a mechanism for monitoring Web information sources. Rather than expending energy checking sites of interest for changes, the system allows users to concentrate on finding innovative applications for monitored information. Our system also offers semantically rich data processing services that provide fine granularity change detection with more expressive power than simple Boolean change tests. This chapter describes our system architecture, specifically addressing data management features that offer opportunities for optimization. The design provides a framework for flexible and scalable Web change monitoring through the use of:

- *Efficient Data Management.* We encode Web documents in the Page Digest format, described in Section 3.3. The Page Digest format encodes the structure and content of a document efficiently for fast load times and efficient evaluation. The major characteristics of the Page Digest are summarized below.
  - *Separate structure and content.* Documents on the Web—such as HTML or XML—can be modeled as ordered trees, which provide a more powerful abstraction of the document than plain text. The Page Digest uses this tree model and

explicitly separates the structural elements of the document from its content. This feature allows many useful operations on the document to be performed more efficiently than operating on the plain text.

  - *Comparable.* Page Digests can be compared directly to each other. Subsections of the digest can also be compared, which provides the means for semantically richer document comparisons such as resemblance.

  - *Invertible.* Page Digests can be efficiently converted back to the original document. Since the digest encoding is significantly smaller than the original document, it provides a scalable solution for large-scale Web services. The digest is an excellent storage mechanism for a Web document repository.

- *Rich Processing Constructs.* Individual change monitoring requests focus on only interesting changes, which provides more utility for users while supplying opportunities for processing optimizations.

- *Grouping.* Certain pages attract attention and large groups of interested users. Scalable systems must analyze and combine compatible monitoring requests to actively reduce computation, network usage, and local I/O.

3.2 Related Work

Web Document Monitoring. Examples of other systems that monitor multiple Web pages for multiple users include WebCQ [70], the NiagaraCQ project [22], and the Change Detector™ system from WhizBang! Labs [7]. WebCQ [70] was implemented as an adaptation of the Continual Query system for the monitoring Web pages. The Page Digest Sentinel system improves upon WebCQ by using the more efficient Page Digest encoding to improve I/O times, storage space, and to provide a base for more efficient algorithms. NiagaraCQ is focused on continuous queries for XML documents, and supports standard XML query languages for monitoring how query results over a document change over time. NiagaraCQ is more appropriate for monitoring data documents where the document as a whole and each individual element is strongly typed, while the Page Digest sentinels are

more appropriate for monitoring changes over documents typically found on the Web.

Change Detector^{TM} [7] employs intelligent learning algorithms that monitor entire Web sites for business-specific changes (such as changes to senior company executives). In contrast, our system focuses on efficient algorithms to detect changes in individual Web pages and to leverage large user bases to reduce redundant network requests and processing costs associated with a large group of sentinels.

Change Detection and Diff Services. Traditional change detection and difference algorithms [36, 23, 72, 100] detect modifications to documents represented as either character strings or in a tree structured format. Past research focused on establishing a set of change operators over the domain of discourse and a cost model for each of the defined operators. Once these parameters were established, the research sought, given two elements from the domain, to construct an algorithm that would produce a *minimum-cost edit script* that described the changes between the two elements with the smallest overall cost of operations. Barnard et al. [4] present a summary of several string-to-string and tree-to-tree correction algorithms. Dennis Shasha et al. [90] summarize their work on various problems related to tree-to-tree correction and pattern matching in trees. Chawathe and Garcia-Molina [21] extend portions of this work with additional semantic operators to make the difference report more meaningful: in addition to insert, delete, and update, they introduce the operators move, copy, and glue to impart greater semantic meaning to the generated diffs. Change detection using these algorithms consists of generating a minimum-cost edit script and testing to see if the script is empty. Algorithms for generating minimum-cost edit scripts for trees are computationally expensive, while string difference algorithms miss many of the nuances of tree-structured data formats.

Others have also recognized the importance of fast change detection that targets tree-structured documents, such as HTML and XML. Khan et al. [60] compute signatures for each node in a tree, allowing fast identification of only those subtrees which have changed. The main cost here is the computation of the signature of each node. The drawback to this approach is that false negatives—documents which change, but the signature of the root node does not change—are possible. Many applications cannot tolerate false negatives,

making this approach unsuitable or undesirable for such systems.

XDiff [100], presents a document-to-document change detection algorithm for unordered XML documents. While this may be preferable for database-derived documents where order is incidental, many Web documents rely on the implicit order in the structure of the document. Further, the expense of present unordered tree-to-tree change detection algorithms would hinder the performance of a large-scale Web change monitoring system. XyDiff [28] is another document-to-document change detection system designed for XML documents. Their algorithms are based on computing MD5 hashes to identify repeated subtrees between two documents. This technique relies on the implicit order of elements in a document as any permutation in the order of elements will cause all ancestor elements to change. This technique is also sensitive to changes in any aspect of the document: structure, content, attributes, comments, etc. Using the Page Digest encoding we can isolate changes in one aspect of the document from other aspects, allowing much faster processing of documents when no interesting changes have occurred.

We have written a change detection algorithm, Sdiff, that utilizes the structural characteristics exposed by the Page Digest encoding to report changes between Web documents. This algorithm deviates from the focus of earlier work by leveraging the unique encoding of the Page Digest to detect semantically meaningful changes. Sdiff explicitly supports many different types of change detection, providing a smooth trade-off between cost and the level of change detail extracted. The supported types of change detection include simple changed versus not changed, efficient markup of exactly which elements have changed between two versions of a document, and computation of a minimal edit script that transforms one version into another. Many applications can use Sdiff's flexibility to achieve better performance and semantic interpretation of data. Using Sdiff, a Web service can compare documents quickly and with more control than text diff algorithms allow, examining particular facets of the document that are interesting and ignoring changes in uninteresting portions. For example, applications such as Web change monitoring services need to detect interesting changes to a specific aspect or subset of a page, but do not need to compute edit scripts. If a Web service requires full edit scripts for proper operation, Sdiff can invoke a powerful

tree-diff algorithm that focuses computing resources on only those documents that have been changed.

The change detection components in our Web monitoring system leverage the Page Digest encoding to provide extended capabilities for monitoring diverse facets of Web documents, including changes to content, links, or structure. Typical Web document change detection algorithms, such as AT&T Labs HTML diff algorithm [23], mark only content changes in the entire document. Our algorithms allow selective change detection for only portions of a document, and can easily restrict the scope of change to only one aspect of a document (textual content, links, images, attributes, tag names, or structure).

Digest. There is a large body of existing work on comparison and digest mechanisms for general text and binary strings. One of the earliest applications of these algorithms is in network transmission protocols where they are used to detect and possibly correct transmission errors in blocks of data.

A simple scheme is to append a *parity bit* [93] to each block that is chosen to force the number of "1"s in a bit string to be even or odd; the choice of even or odd is agreed upon beforehand. Hamming [53] proposed the notion of the *Hamming distance* between two bit strings and devised a mechanism for using this metric to produce protocols that could correct single bit errors in a block; this technique has been extended to correct burst errors as well [93]. Another family of error detecting algorithms is the cyclic redundancy check (or cyclic redundancy code) [93, 6], which produces a "check word" for a given string. The check word is computed by the sender and appended to the transmitted block. On receiving the block, the receiver repeats the checksum computation and compares the result with the received checksum; if they differ, the block is assumed to be corrupt and the receiver typically requests a retransmission.

Another application of digest algorithms is in the area of information security where they are used to protect passwords, cryptographic keys, and software packages. Perhaps the most popular of the *cryptographic hash functions* is the MD5 [83] algorithm. Other examples of similar hash functions include SHA-1 [39] variants and the RIPEMD [34] family. Although the mechanics of these algorithms differ, they have similar design goals.

The three important features for cryptographic applications are pseudo-unique hashing, randomization, and computationally difficult inversion. Ideally, if two documents x and y exist such that $x = y$, then using a hash function H, $H(x) = H(y)$; conversely, $x \neq y \Rightarrow H(x) \neq H(y)$. However, these hash algorithms produce fixed length results (that is, $|H(x)|$ is constant for any x), so guaranteeing a unique hash for any arbitrary input is impossible. Rather, these algorithms are designed for computationally difficult inversion: for a given $H(x)$, finding an x that generates $H(x)$ is computationally difficult by current standards. The randomization property of these algorithms means that small changes in the input document result in dissimilar hash values. Given x and x_1, where x_1 is a single-bit modification of x, $H(x)$ and $H(x_1)$ will be totally distinct.

In designing the Page Digest encoding, we attempted to capture the utility of string digest mechanisms while leveraging the unique features of Web documents. The Page Digest design diverges from more traditional digest mechanisms with respect to its intended application domain, focusing on efficient processing, comparison, and storage utility rather than cryptographic security or error correction.

3.3 Page Digest Overview

An important problem affecting the scalability of any system that interacts with the Web is the processing of standard Web document markup languages, such as HTML and XML. Present Web document formats are not tuned for efficiency and are highly redundant. A second drawback to using standard Web languages in a scalable system is their intermixing of the various aspects of the document, including structure, tag names, attributes, and content. For example, table data in HTML will appear in line with the HTML code that defines the table's structure. These drawbacks lead us to consider an alternate data encoding that focuses on efficiency. To effectively support change monitoring over various semantic components of a Web document, we desire a document format that will group the related components of the document into logical containers and eliminate data redundancy while preserving the meaning of the original document.

The Page Digest Web document encoding [85] increases processing efficiency in our Web

document monitoring system by providing access to semantically interesting characteristics of a Web document. Web documents are typically modeled as tag-trees, in which each tag in the document's text is represented as a tree node; tag nesting levels determine the node hierarchy. Standard tree model implementations represent tree nodes as data objects; document operations like traversal and search are implemented by following object references through the in-memory tree structure. The Page Digest encoding eliminates tag redundancy and places structure, content, tags, and attributes into separate containers, each of which can be referenced in isolation or in conjunction with the other elements of the document. Many basic operations over document trees take considerably less time using a Page Digest encoding of the document since these operations operate over arrays. Loading a Page Digest encoded document into memory is more efficient because parsing is much simpler. In our prototype Web change monitoring system, all documents in the local data cache are stored in the Page Digest format; we demonstrate that using this encoding significantly reduces the time required to transfer a document from disk to memory.

The Page Digest is a straightforward encoding that can be efficiently computed from an ordered tree of a source document, such as an HTML or XML document. Figure 11 shows an example of a rendered HTML page and a visualization of its tag-tree representation. The Page Digest encoding consists of three components: node count, depth first child enumeration, and content encoding. We selected these particular elements for inclusion in the format based on careful evaluation of Web documents and the characteristics of those documents that are important for many applications. For example, the tree structure of a document can help pinpoint data object locations for object extraction applications.

Node Count. The first component of a Page Digest is a count of the number of nodes in the document's tree representation. The main purpose for this count is for fast change detection: if two documents W_1 and W_2 have different node counts then $W_1 \neq W_2$. Inclusion of this count can eliminate costly difference computations for change detection applications and provides a size estimation for storage management.

Depth-First Child Enumeration. The second component of the digest is an enumeration of the number of children for each node in depth-first order. In a typical Web document

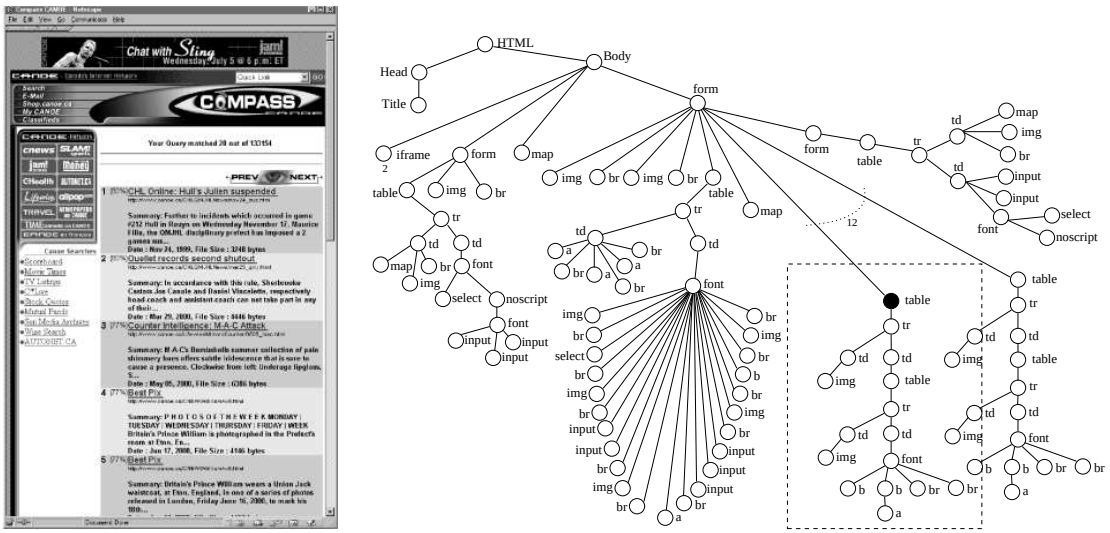


Figure 11: Sample HTML Page and Associated Tree

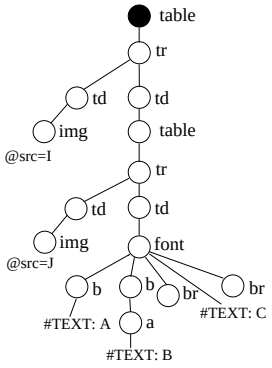
encoding such as HTML, the structure of the document is entwined with its content. We maintain that separating the content and structure of the document provides opportunities for more efficient processing of the document for many applications; examples include using structural cues to extract data objects and enhancing change detection efficiency.

Content Encoding and Map. Finally, the Page Digest encodes the content of each node and provides a mapping from a node to its content. Using the content map, we can quickly determine which nodes contain content information. The content encoding preserves the "natural order" of the document as it would be rendered on screen or in print, which can be used to produce text summaries of the document.

3.3.1 Page Digest Example

Consider the example Web document in Figure 11, which shows the tree representation of a page from a Canadian news site with text nodes omitted for clarity of the diagram. This example page is typical of Web pages modeled as trees. It contains two subtrees—rooted at nodes "head" and "body"—that are children of the root HTML node. The head of the document contains the title tag and text of the title, while the body of the page comprises the bulk of the document and contains the content of the page enclosed in several tables, plus a table for the navigation menu and a few images and forms.

The tree fragment in Figure 12 shows the boxed table subtree from Figure 11 with the addition of the node attributes and text nodes. The node types—or tag names—appear to the right of the node. Attributes appear below the node to which they belong and are denoted with the "@" symbol. All nodes except for text nodes are shown as circles on the graph; text nodes are shown via the label "#TEXT" followed by the text content of the node represented with a single-letter variable. The document fragment represented by this subtree is a table that contains one row with two columns—child "tr" with two "td" children. The first column of the table contains an image while the second column contains another table containing two columns with an image and some text.



Page Digest	
node count (N)	19
structure (ST)	`1,2,1,0,1,1,2,1,0,1,5,1,0,1,1,0,0,0,0`
type map (TM)	`1,2,3,4,3,1,2,3,4,3,5,6,0,6,7,0,8,0,8`
attribute map (AM)	`-,-,-,0,-,-,-,-,1,-,-,-,-,-,-,-,-,-,-`
content map (CM)	`-,-,-,-,-,-,-,-,-,-,-,-,0,-,-,1,-,2,-`
type list (TL)	#TEXT, table,tr,td,img,font,b,a,br
attribute list (AL)	I,J
content list (CL)	A,B,C

Figure 12: Tree fragment with corresponding Page Digest

To the right of the subtree is its Page Digest representation including the HTML tag and attribute extensions. The first component of a Page Digest is a count of the number of nodes in the document's tree representation. Inclusion of this count can eliminate costly difference computations for change detection applications and provides a size estimate for storage management.

The representation of the tree structure of the document is contained in the list ST, whose elements are number of children at each node in depth-first traversal order. Notice that the subtree root node "table" has one child: ST[0] therefore has the value '1'. Continuing in depth-first order, "tr" has two children (ST[1] = '2'), the leftmost child "td" has one child (ST[2] = '1'), and the node "img" contains no children (ST[3] = '0'). The encoding continues in a similar fashion along the rest of the subtree in depth-first order.

The array ST has several useful properties. It encodes the structure of the tree and can

be used to re-construct the source document or generate the path to a particular node. It also provides a compact representation of the tree. ST can in most cases be easily encoded in only N characters without any bit-packing or other compression techniques. Each node can be identified via the ST array index, which indicates the node's position in the array: the root is "0," the root child node "tr" is "1" and so on. Finally, ST can be used to quickly identify areas of interest such as leaf nodes or large subtree nodes in the tree. The subtree anchored at the "font" node, for example, contains 5 children while every other node has 2 or fewer.

Node labels are encoded as a mapping from the tree nodes, TM to a set of labels, TL; attributes are similarly encoded in AM and AL. Finally, the Page Digest encodes the content of each node in CL and provides a mapping from a node to its content, CM. Using the content map, we can quickly determine which nodes contain content information. The content encoding preserves the "natural order" of the document as it would be rendered on screen or in print; this property is useful for producing text summaries of the document.

Figure 12 shows TL, AL, and CL for the highlighted subtree, which denote the tag list, attribute list, and content list respectively. TL is the set of all node types found in the subtree. AL is the set of all the attributes encountered. CL is a set containing any content from #TEXT nodes. TM, AM, and CM denote tag mapping, attribute mapping, and content mapping respectively. They map the node array ST to the type, attribute, and content lists. The nodes are matched to their types, attributes, and contents in the same order they appear in ST. Thus, the root node, 0, is mapped to TM[0] = 1 indicating it has type 1, AM[0] = '-' and CM[0] = '-' signifying that this node has no attributes or content. Looking closer, we observe that TL[TM[0]] = TL[1] = type "table," exactly as we would expect given that the root node 0 is of type "table."

Every node must have an entry in TM since every node has a type. For HTML, the size of TL is typically less than N since tags are usually repeated: in any given page, there is usually more than one node of the same type, be it a text section, paragraph, table row, or section break. In contrast, having an entry in AM or CM depends entirely upon whether the node has attributes or content. Observe that text content is always found at the leaves

of the tree: CM[x] $\neq$ '-' implies that ST[x] = 0, where x is the index on the number of nodes in the subtree table ranging from 0 to $(N-1)$.

The Page Digest format splits structure from content and provides the opportunity to only load the structure of a page for performing structural change detection; similar optimizations can be used for size, attribute and type changes. Experimental performance results, shown in Figure 13, indicate that creating a Page Digest from HTML is roughly equivalent in cost to using standard HTML or XML parsers over a Web document, while loading a serialized version of the Page Digest is orders of magnitude faster than loading serialized XML. Both of these properties are desirable for system scalability: the small extra cost of reencoding a Web document is insignificant compared to the savings afforded by loading the more efficient encoding from disk.

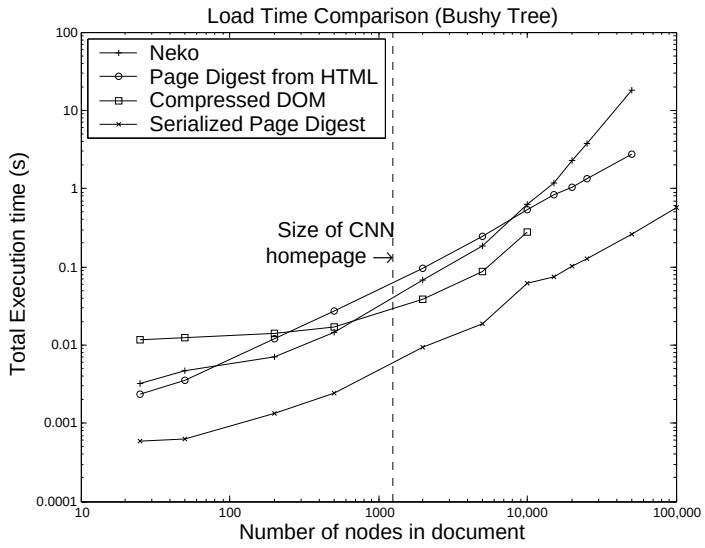


Figure 13: Comparison of various loading techniques. The vertical line indicates the approximate size (1250 nodes) of cnn.com.

3.3.2 Specialized Digest Encodings

In the previous section we examined the general Page Digest encoding that can be applied to many different types of documents. However, if more specific information is available about a particular type of document to be encoded, it is possible to construct a specialized digest format that takes advantage of the nuances of that type. For example, TEX documents and

HTML both have additional document features that can be encoded and used.

We have constructed a specialized digest for HTML pages. In addition to the three components of the general digest format, the HTML encoding includes two additional aspects: tag type information and attributes. All HTML nodes are tagged with a particular type such as "BODY" or "P." While we could encode this information as a form of content, it is advantageous to use a specialized encoding to exploit the characteristics of HTML. Tags tend to be quite repetitive, so using a tag encoding and a map from nodes to their tag allows us to store each unique tag name only once for the entire document, thus saving storage and memory costs. This also provides an efficient mechanism for locating nodes of a particular type.

The specialized Page Digest encoding can also handle XML documents, which are more data-centric than HTML and tend to have longer, more descriptive tag names. Given that tags in XML documents tend to comprise a higher percentage of the overall document, Page Digests for XML documents can present significant saving in terms of storage and memory costs.

3.3.2.1 Page Digest Encode

Algorithm 3 shows a sketch of the algorithm for converting a source document `W` into a Page Digest. The algorithm parses the text of the document into start- and end-tag events which are returned in depth-first order. These tag events are tracked via a node stack to determine the child counts for the structure array. For this algorithm, we assume that a document is *well-formed* [102] with properly nested tags. For clarity of presentation but without loss of generality we only describe the construction of ST array and omit the algorithm detail for constructing tag name (TL, TM), attribute (AL, AM), and content (CL, CM) arrays from these algorithms. An overview of the impact of these facets on the time and space complexity of our algorithms is reported in Section 3.3.3.

Algorithm 3 begins by initializing its data structures, including the stack that is used to retain node ancestry and a `PARENT` array that tracks each node's parent index. These data structures monitor every node's child count as the document is being parsed, eliminating

the need for multiple passes or an intermediate in memory tree materialization. The main loop of the algorithm reads the characters of the input document W until it reaches the end of the input. While it is reading, the algorithm scans the document for start and end tags, which determine the structural information and ancestor context for each node. The end result of Algorithm 3 is the structural array (ST) with its node count information.

```
Algorithm 3 Page Digest
  W ← <character array of source document>
  S ← stack(), PARENT ← [], ST ← [], index ← 0

  PARENT[0] ← '-' // root has no parent
5 push(0, S)

  while W ≠ EOF do
    tag ← readNextTag(W)
    if isStartTag(tag) then
10    if index ≠ 0 then
        PARENT[index] ← peek(S)
        ST[PARENT[index]]++
        ST[index] ← 0
      push(index, S)
15  else
      pop(S)
    index++
```

Analysis of Algorithm 3. The algorithm reads the characters of W and uses the stack to retain the ancestor hierarchy of the current node. A newly encountered tag indicates a descent down the tree, while an end tag indicates a return up the tree. The time complexity of the algorithm is dominated by line 8, which scans part of the source document W at each pass. Each scan is non-overlapping with all other iterations, and each part of W is read only once, giving a total time for all iterations of $O(k)$, where $k = |W|$. The other steps inside the loop are all $O(1)$ operations to within a constant factor, so the remaining steps of the loop require $O(n)$, where n is the number of nodes in W; we thus have $O(k+n)$. However, we note that n is at most $\frac{k}{7}$ since each node in W occupies a minimum of 7 characters, which yields a final time complexity of $O(k)$ for Algorithm 3.

The space required by the algorithm is dominated by the PARENT and ST arrays, each of which occupy $O(n)$ characters. In addition, the space required by the stack S is bounded

by the depth of the tree representation of W; in the worst case where W is a linked list, S occupies $O(n)$ space during the course of the algorithm.

Algorithm 4 Page Digest to original document

```
P ← Digest of Page W, ST ← P.ST
V ← {}, S ← stack(), CHILDREN ← children(P)

/* push the root onto the stack */
push(0, S)
while |S| > 0 do
  current ← pop(S)
  if current ∉ V then
    push(current, S)
    V ← current ∪ V
    print('<' + type(current) + '>')
    for c ∈ reverse(CHILDREN[current]) do
      push(c, S)
  else
    print('</' + type(current) + '>')

/* subroutine to find children of all nodes */
N ← nodeCount(P), ST ← P.ST
CS ← stack(), CHILD ← []

for index = 0 to (N - 1) do
  CHILD[index] ← {}

/* push the root onto the stack */
push(0, CS)
for index = 1 to (N - 1) do
  parent ← peek(S)
  add(index, CHILD[parent])
  if ST[index] ≠ 0 then
    push(index, CS)
  else
    CHILD[index] ← '-'
    if ST[parent] = |CHILD[parent]| then
      pop(CS)
```

Algorithm 4 shows the process of converting from a Page Digest back to the original document. This process is modularized into two components: the main node processing routine and the *children* subroutine. The main routine maintains a visited list V to track nodes that have already been seen during the tree traversal. The stack S maintains the ancestor hierarchy and preserves sibling order for the traversal. The CHILDREN array keeps

a list of each node's children, which is obtained from the *children* subroutine. Initially, the stack contains only the root node. The main loop of the algorithm proceeds by popping the top node from the stack; this is the current node. If the current node has not yet been visited, three actions occur: the open tag for that node is output, the node is pushed back on the stack and the visited list, and the node's children are pushed onto the stack from right to left so they will be visited in left to right order. If the current node has already been visited when it is first popped off the stack, then the algorithm has finished traversing that node's children so it outputs the node's closing tag.

The *children* subroutine takes a Page Digest and constructs a list of the indices of each node's children. To accomplish this task, it requires the stack `CS` to maintain the node hierarchy. The subroutine first constructs an array of empty lists to hold the children of each node. Then, starting from the root node, it visits each node in `ST` keeping the current node's parent as the top element of the stack. Every node is added to its parent's `CHILD` list. If the current node is an internal node, the next nodes in the depth-first ordered `ST` array will be the children of the current node, so the subroutine pushes the current node onto the stack. For leaf nodes, the subroutine checks the top node on the stack to see if the current node is its last child; if so, the top node is popped off the stack and the routine continues.

Analysis of Algorithm 4. We first consider the complexity of the main portion of the algorithm. Initialization of variables—excepting the `CHILDREN` array, which we will discuss shortly—can be done in constant time. Although a cursory inspection of the body of the algorithm suggests $O(n^2)$ operations due to the nested loops, careful inspection reveals that the inner `for`-loop will visit each non-root node exactly once, thereby executing exactly $n-1$ push operations over the course of all iterations of the outer loop. Additionally, we can ensure that the *reverse* operation incurs no additional cost by traversing the `CHILDREN` lists from the end rather than the front. For the visited list check, if `V` is handled naïvely as a list, the algorithm will execute approximately $2n$ searches of `V`, a list of average size $\frac{n}{2}$. However, using a hash table for `V` will yield constant time searches and inserts. The other operations contained in the outer loop execute in constant time. Thus, the complexity of

the algorithm arises in part from the cost of pushing all $n-1$ non-root nodes onto the stack in the inner loop. We have an additional cost of $c \cdot 2n$ for the outer loop where c is a constant representing the total cost of operations for the outer loop. This yields a total cost for both loops of $(n-1) + c \cdot 2n$.

Computation of the `CHILDREN` lists involves several constant time variable initializations plus n empty list creations in the first `for` loop. The body of the algorithm loops $n-1$ times over the nodes of the Page Digest, performing several constant time stack manipulations, comparisons, and list additions. Total cost for the loop is $d \cdot (n-1)$, where d represents the cost of the operations of the loop.

Our final cost for Algorithm 4 is the cost of computing the `CHILDREN` list plus the cost of the output routine. The total cost is $n + d \cdot (n-1) + (n-1) + c \cdot 2n$, yielding a final time complexity of $O(n)$.

3.3.3 Further Analysis

Algorithm 3 shows Page Digest conversion on simplified documents for clarity of presentation, but converting actual Web documents adds only a few additional processing requirements. The Page Digest shown in Figure 12 captures all the components of the document by encoding the type, attribute, and content lists. These mappings can be built up at the same time that the structure array `ST` is constructed with little additional computation overhead. The space overhead for the mappings `AM`, `TM`, and `CM` is $3n$. Redundant tag names are eliminated, so each unique tag name in the document is stored once; we will demonstrate the savings afforded by this technique in the experimental section. The attribute and content lists consume the same amount of space as in the source document.

We assume in the above algorithms that Web documents are well-formed: proper nesting and closing tags are assumed to be in place. Although proper nesting is an absolute requirement, it is possible to relax the strict requirement for closing tags in some circumstances. For instance, HTML specifications have historically allowed tags such as "br" that do not have an end tag. In such cases, we can generate an implied closing tag using rules that are specific to each tag type; it is also possible to correct some HTML errors this way.

3.4 Experiments

We report two sets of experiments. The first set of experiments is designed to test various characteristics of the Page Digest when performing some common operations over Web documents. The second set of experiments tested the first version of our change detection implementation on several data sets with the goal of evaluating the performance of change detection over Page Digest characteristics relative to other tools. We will first discuss our experimental setup by noting the equipment and language features used to run our experiments. We also discuss the test data including how we obtained the data set and what the test data is intended to model. Finally, we outline our experimental goals and present the results.

3.4.1 Experimental Setup

Equipment and Test Data. All measurements were taken on a SunFire 280 with 2 733-MHz processors and 4 GB of RAM running Solaris 8. The software was implemented in Java and run on the Java HotSpot Server virtual machine (build 1.4.0, mixed mode). The algorithms were run ten times on each input; we averaged the results of the ten executions to produce the times shown.

We used two sources of data for our experiments: data gathered from the Web and generated test data. The Web data consists of time-series snapshots of certain sites and pages collected from a distributed set of sites. For the generated data, we built a custom generation tool that can create XML documents containing an arbitrary number of nodes with a user specified tree structure. The data used in these experiments consists of documents of various sizes with three basic tree shapes: bushy, mixed, and deep. We used data with different shapes to model the variety of data found in different application scenarios and to test the sensitivity of tree processing applications to tree shape.

3.4.2 Page Digest Experiments

The suite of experiments outlined here are designed to test various characteristics of the Page Digest when performing some common operations over Web documents. Our experimental

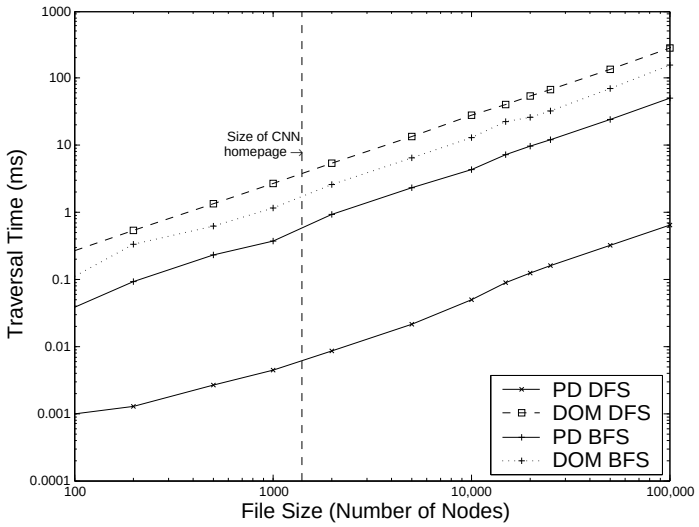


Figure 14: Page Digest Traversal

goals were to model real Web data, to test the strengths and weaknesses of the Page Digest algorithms, and to gauge performance relative to similar tools.

Experiment 1: Tree Traversal

Figure 14 shows a comparison of the cost of traversing an entire document tree for both the Page Digest and DOM. The Page Digest is able to complete traversals faster than the DOM implementation used in our tests; using the Page Digest for depth-first search (DFS) traversals is equivalent to scanning an array of size n, which accounts for the Digest's huge performance advantage for DFS. For breadth-first search, the Page Digest still performs faster than DOM, but the additional complexity required to reorder the tree into breadth-first order incurs a performance penealty compared to DFS.

Experiment 2: Object Extraction Performance

Object extraction from Web documents occurs in four phases: read file, parse, identify subtree, and determine object delimiters. This experiment uses the Omini [14] object extraction library, which uses a document tree model for its processing routines. In this experiment, we show the object extraction performance using an off-the-shelf DOM implementation compared with a re-implementation using Page Digest as the processing format.

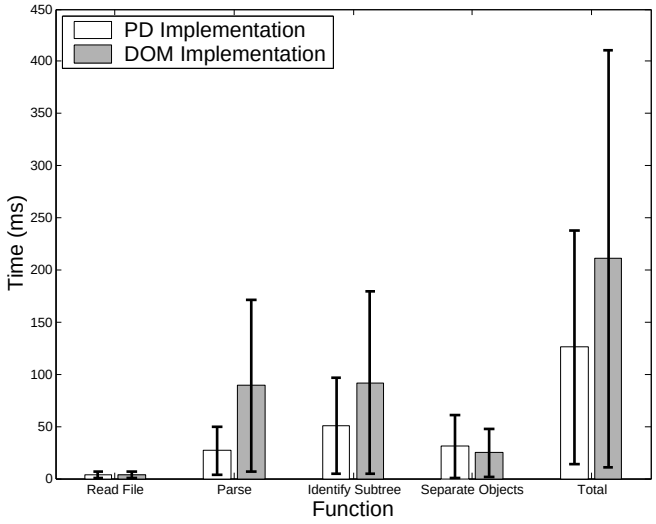


Figure 15: Object Extraction

Of the four phases, we expect the reading time to be constant for both systems since they both use the same IO subsystem. The parsing and subtree identification phases rely on tree traversals to function, which we expect to be an advantage for the Page Digest encoding. Object delimiter determination operates on nodes of the already-traversed tree, which does not immediately present an advantage to either approach.

Figure 15 shows the average and maximum performance times for Omini using Page Digest and DOM. The graph bars show the average performance time for all data samples in the experiment. We have included error bars that show the range of times obtained using the wide range of pages in our experimental data set. As we expected, the time to read the page was identical between the two implementations. The DOM implementation edged out the Page Digest version for object separation minimum, maximum, and average case performance, while the Page Digest implementation performed markedly better in most cases for parsing and subtree identification.

Experiment 3: Size Comparison

Experiment 3 presents a size comparison between two documents in XML format and their equivalent Page Digests. The Page Digest encoding, while not a traditional compression algorithm, can reduce the size of documents by eliminating redundant storage of tags

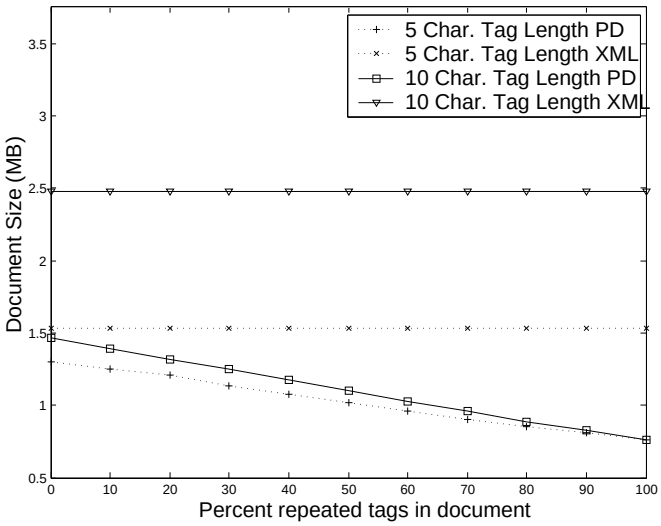


Figure 16: Page Digest Size Comparison

in the source, present in both HTML and XML. Naturally, the savings afforded by this feature vary with the type of data being digested: content-heavy documents with few tags will benefit the least, while highly tag-repetitive data-centric documents will enjoy the largest reduction in size.

This experiment examines the potential reduction offered by the Page Digest; Figure 16 shows the results. All tag sizes in the data were fixed at 5 or 10 characters. The x-axis in the graph is the percentage of tag repetitions: the 50% mark, for example, shows the savings when 50% of the document's tags are the same and the other 50% are all unique.

Experiment 4: Page Digest Generation and Load Time

This experiment examines the time required to convert a Web document from its native format to the Page Digest for various document sizes. Figure 17 shows the performance of loading both parsed and pre-computed Page Digest encodings. For applications that can make use of cached data, such as change monitoring applications, caching Page Digest representations of pages is a clear improvement in storage space and load times that does not sacrifice any flexibility: the source HTML can always be reconstructed from the Page Digest if needed.

In addition, the graph shows the time needed to create a Page Digest contrasted with

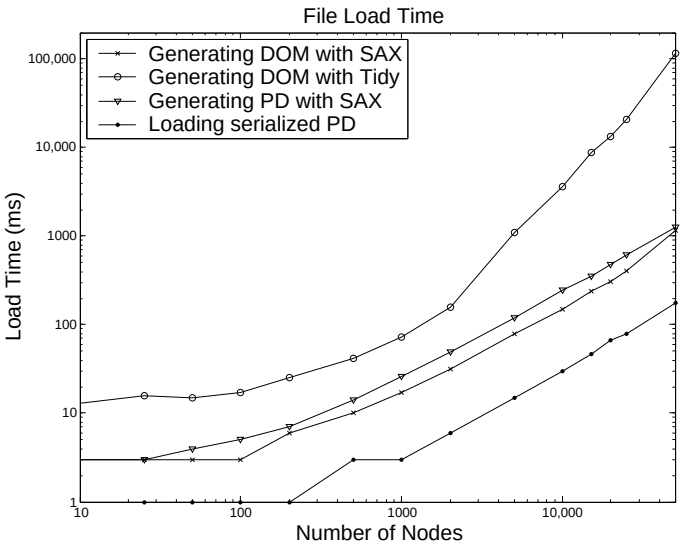


Figure 17: Page Digest to Web Document Conversion Performance

the time needed to construct an in-memory DOM tree of the same document. This test compares the JTidy [82] DOM parser for HTML DOM tree generation, the Xerces [94] SAX parser for HTML DOM, and the Page Digest generator which also uses the Xerces SAX parser. Although the SAX-based HTML to Page Digest converter outperforms the error-correcting DOM parser, loading the native Page Digest stored on disk demonstrates the greatest performance gain since it requires minimal parsing to return an in-memory tree representation.

We note that the standard SAX to DOM converter performs slightly better that the Page Digest generator. This is due to the minimal overhead needed to organize the Page Digest after parsing, a cost which the DOM generator does not incur. However, this performance hit is mitigated by the substantial savings in traversal time afforded by the Page Digest, as shown in Experiment 1.

3.4.3 Change Detection Experiments

We tested Page Digest's change detection performance relative to other document comparison approaches. Our goal with the performance tests was to emphasize the benefit of separating structure and content in Web documents, which is a core component in the Sdiff design.

Experiment 5: Page Digest Change Detection

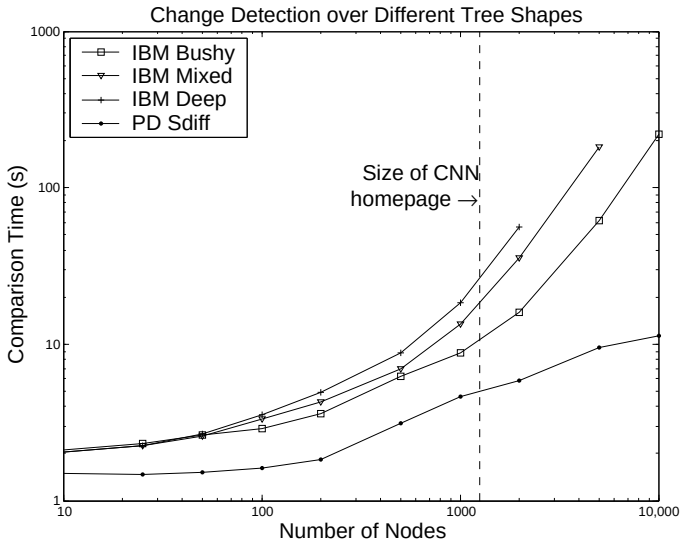


Figure 18: Page Digest Tree Shape Performance Comparison with IBM Corporation XML TreeDiff

Figure 18 compares Sdiff's performance with another tree difference algorithm, IBM Corporation's XML TreeDiff [55]. The times in this experiment are measured from the command line, and the trials for both tools cover all three test data sets: bushy, deep, and mixed. Note that the graphs are plotted in a log log scale. Also, although we ran all three tests for Sdiff and the Page Digest, we have only plotted the data for the mixed data set as the performance of the Page Digest trials were not significantly affected by changes in tree shape, with the result on this graph of all three lines being plotted in approximately the same place. For clarity, we omit the Page Digest deep and bushy lines.

Figure 19 emphasizes the Page Digest's ability to do meaningful document comparisons along different facets of the document. The IBM trial and the Page Digest Sdiff trial show the performance of the two tree difference algorithms. Applications that only require structural comparison, however, can realize a significant performance increase by using the Page Digest, which the Page Digest Structure trial reveals.

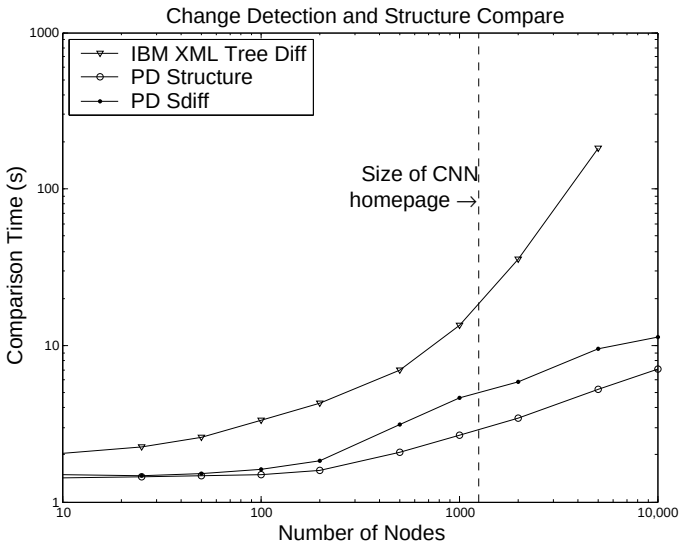


Figure 19: Page Digest Semantic Comparison Performance Comparison with IBM Corporation XML TreeDiff

Experiment 6: Comparison with GNU Diff Algorithm

Figure 20 shows a comparison between the performance of change detection over Page Digest and an implementation of the standard GNU diff algorithm [52] in Java [41]. This experiment is designed to show Page Digest's change detection performance compared with a traditional text based diff algorithm. We used the Java implementation to eliminate the confounding factor of different implementation languages from the experiment.

The results of this experiment show that diff performs better than the full Sdiff algorithm but performs worse than the Sdiff structural change detection. We note that diff is a text based algorithm that does not account for the tree structure or semantics of the documents it is comparing. The structurally-aware Page Digest change detection algorithms can detect structural modifications very quickly. They also provides more control over the semantics of document comparison than is possible with diff, making it more appropriate for Web services that operate on tree structured documents. Many Web services requires change detection that is specific to a single aspect of a document—such as tag names, attribute values, or text content. The diff algorithm detects any changes whether they are interesting to the service or not.

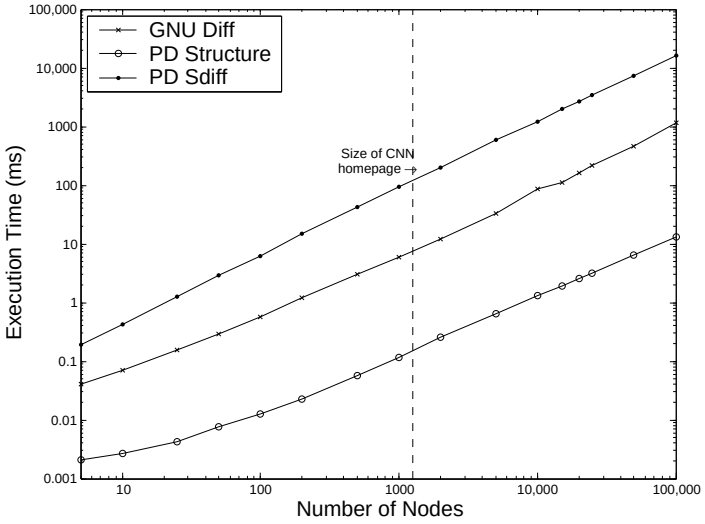


Figure 20: Page Digest Performance Comparison with GNU Diff

3.5 Architecture

Figure 21 shows the general architecture of our Page Digest sentinel system, which shares many components with other similar systems [96, 70, 68, 7]. These *third-party* Web change

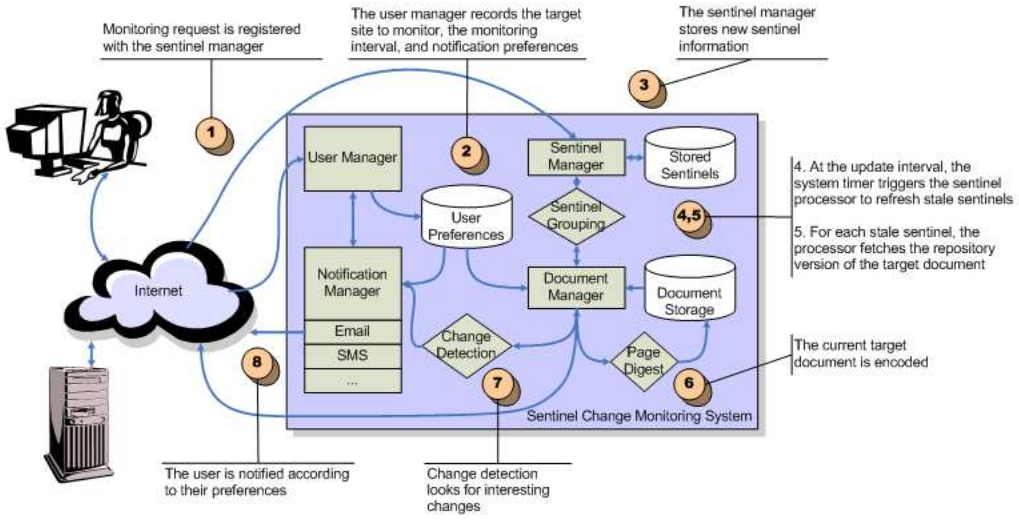


Figure 21: Page Digest Sentinel System Architecture

monitors—so called because they are independent of both the client and the target site—service target queries and handle data management on behalf of users. Users interact with the system through a user interface, typically Web based, which is responsible for

managing user notification preferences and assisting the user in articulating queries that will be executed against monitoring targets. One distinguishing characteristic in the architecture of our system is the *sentinel manager*, which is responsible for processing change queries against target sites. Conceptually, the sentinel manager keeps track of a set of user agents called *sentinels*, which carry out the tasks of querying a particular document, called the target, and notifying the user if any changes of interest have occurred. The sentinel manager installs and removes sentinels, triggers sentinel evaluation at the user specified interval, and intelligently batches queries to maximize processing efficiency over popular targets.

The sentinel manager architecture and implementation are of critical importance to the efficiency of our Web change monitoring system. The sentinel manager consists of a data manager and associated storage, a document manager and associated storage, a change detection engine, the sentinel processing engine, and an update trigger timer. When a user wishes to monitor a target, the request is registered with the sentinel data manager. The user supplies information about what components of the target site they wish to monitor, the change monitoring interval, and their notification preferences. The sentinel data manager stores this information in its sentinel store and requests that the document manager fetch the initial snapshot of the target site. At the system-wide update interval, the system timer triggers the sentinel processor, which requests the set of sentinels that have become stale and need to be refreshed from the sentinel data manager. For each stale sentinel, the processor fetches the target document, which is encoded by the document manager and fed to the change detection engine. For the documents that have interesting changes, the sentinel processor fires a notification event that will alert the user according to their preferences.

The sentinel manager's central role in our Web change monitoring system mandates efficient processing of sentinels to ensure system scalability. One of the most important optimizations in the sentinel manager is the grouping of related sentinels together to minimize redundant processing and I/O. The next few sections will explore the various parts of the sentinel manager in detail, focusing on optimizations we have used to increase the performance of our Web change monitoring system.

3.5.1 Web Document Monitoring

The Page Digest encoding forms the basis for our monitoring system as the storage and comparison format for Web documents. We now consider the challenges that must be addressed by any Web document monitoring system, focusing on our use of the Page Digest encoding to enhance the system's efficiency. The first issue is that of user interaction, which can be divided into two subproblems: query specification and user notification. The second issue is that of data management, which includes document storage, change evaluation, versioning, and effective use of compute resources. This section will examine the query specification and data management problems. Notification is the means by which the system informs users about changes they feel are important; a detailed analysis of the user notification problem is beyond the scope of this work.

Update Semantics There are two elemental types of changes with respect to Web documents: content changes and structural changes. Intuitively, content changes include any change to text that appears in the rendered version of a Web document, including changes to hyperlinks or images. Content changes in words and phrases can be detected via keyword searches or through complex regular expression matching. In contrast, structural changes modify the tag structure of the document and alter the relationships between document elements. Examples of structural changes include modifications to a document's tag names and attribute values, page organization, and alteration of document annotations such as comments and `meta` tag values.

We can expand the semantic flexibility of these two basic change types along several refinement axes. First, structural changes can be restricted to a particular logical group of structural elements, such as attribute alterations. Second, users will typically not be interested in changes across an entire document; rather, the desire to "hone in" on an area of interest leads to refinement by location. Third, an update may be triggered only if it satisfies some regular expression pattern or structure expression. Finally, an interesting change may be defined in terms of combinations of the two basic change types with any of the refinement modifications present.

Web Change Sentinels The notion of a Web change sentinel is that of an autonomous software agent that actively monitors a user-supplied Web document and sends notification when the document has changed in some way that is interesting to the user. In most instances, the user is interested in changes affecting some characteristic of the document independent of its representation. Typical user queries might include monitoring an information technology news site for mention of "IBM" or monitoring a popular Web site for major design changes. Table 6 shows some example monitoring requests and highlights the various document components that each request considers.

Table 6: Examples of Various Update Semantics and Refinements

#	**Example Monitoring Request**	**Type**	**Refinement**	**Location**
1	any change	—	—	—
2	any content change	content	—	—
3	any structure change	structure	—	—
4	any link target change	structure	links	—
5	text of the second ¶ changes	content	—	second ¶
6	new rows added to third table	structure	table rows	third table
7	new ¶ containing the word "foo"	combination	¶, keyword	—

Monitoring requests over Web documents are converted into Page Digest sentinels by the Sentinel manager. Sentinels operate exclusively over the Page Digest encoding that provide each sentinel direct access to the parts of the document it needs. Page Digest sentinels require type information, which is inferred from the user's monitoring request. Figure 22 displays the various sentinel types and the organization of sentinel types into a hierarchy, enabling short-circuit evaluation in sentinel groups. Intuitively, the hierarchy enforces the rule that no sentinel type can be triggered unless its parent has also been triggered, eliminating processing overhead when it is impossible for certain types of changes to have occurred. Any of the sentinels may be modified to check for changes in only specified portions of a document to focus change detection on the interesting parts of the document.

The two main types of sentinels are content and structure sentinels, corresponding to the two major types of updates identified earlier. Content sentinels are text-based and monitor for any kind of change to the content of a document. Content sentinels can be

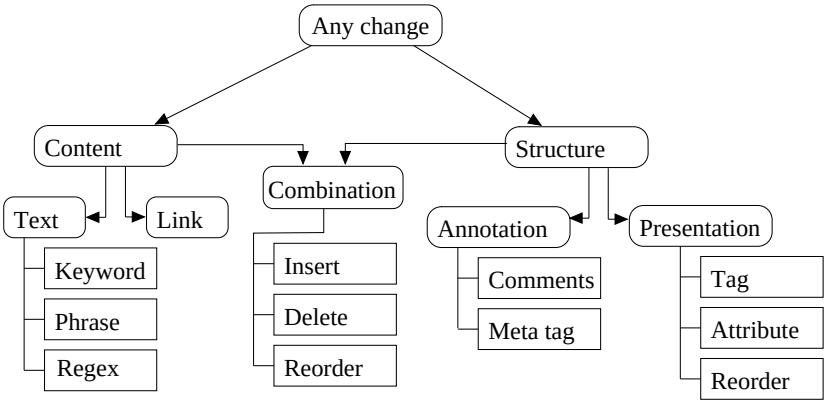


Figure 22: Sentinel Hierarchy

targeted to a specific portion of a document using a location specification. They can also be refined using keywords or phrases to monitor for the addition or deletion of specific strings to a document. For more advanced users, content sentinels support regular expressions for monitoring content in a more semantically powerful way.

Structure sentinels monitor a page for changes in layout, changes in the types of tags used, and changes in the order of the document tree. The sentinels monitor parts of Web document that are "invisible" when rendered to a display terminal or other media. Like content sentinels, structure sentinels can be targeted to specific areas of a document. In addition, they can also be restricted to search for characteristics like the addition of new tag names or attribute values.

The *any change* sentinel is a special entity that is treated separately from both content and structure. Any change sentinels are triggered, as their name implies, after any change to the document no matter how minor. Like content and structure sentinels, any change sentinels can be restricted to look for changes in a specific location.

In addition to its type, each sentinel has a firing interval, which specifies how often the document is to be monitored. Our implementation restricts this value to multiples of a system-wide minimum time interval to maximize grouping potential. Since most Web documents are served using a client pull model, sentinels must poll the target document periodically to determine if any interesting changes have occurred. By default, the firing interval is set for the minimum time interval allowed by the system, although users may

specify any other interval that is a positive integer multiple of the minimum. Restricting the allowable time intervals greatly increases the ability of the system to optimize execution strategies. The user may also specify a longer time interval if they wish to throttle notifications from a particularly active document.

3.6 Page Digest Sentinel Processing

The challenge in optimizing a monitoring system is in determining the primary costs and implementing effective schemes to minimize those costs. We categorize the problems facing the sentinel system and describe how we leverage the Page Digest data structure to alleviate the costs.

The primary costs in processing sentinels are network costs, data management costs, and processing costs. Network costs are incurred during the retrieval of monitored data. The total network cost is derived from three factors: incoming bandwidth to the change server, transient available bandwidth over the network, and processing and transmission at the data source. We assume that Web document servers are autonomous, so any costs due to server latency or processing are beyond our control. Similarly, transient network bandwidth is highly variable. Purchasing a faster connection to the ISP can increase incoming bandwidth, but for most commercial network connections this is the last link to become saturated.

Data management costs are also a major consideration in designing a Web change monitoring system. At present, it is not economically practical to maintain even a minimal history for managed Web documents in primary memory for large numbers of sentinels due to the size and growth rate of the data. This implies that previous document versions should be kept in secondary storage and only brought into memory when needed for comparison with new versions. While using the Page Digest encoding allows us to dramatically reduce local I/O cost, we do not address storage cost concerns here as they were not found to be the primary bottleneck in processing large numbers of sentinels.

This work focuses on local processing costs incurred during the evaluation of sentinels. In particular, we address techniques to reduce the cost of detecting interesting changes between two versions of a Web document and to optimize processing of large groups of sentinels. We

employ several methods to achieve more efficient operation. First, our processing algorithms reduce the amount of computation used for a single sentinel by employing low cost change indicators such as page signatures. Second, Web documents are encoded in the Page Digest format, which captures the structure and content of a document efficiently for fast load times and efficient in-memory algorithm evaluation. Third, sentinels are grouped together to reduce redundant computation. Certain documents attract many users that are interested in changes to the document: by combining related sentinels we can reduce processing costs, network transmission time, and local I/O.

3.6.1 Single Sentinel Processing

The implementation of sentinels in our Web change monitoring system deviates from the intuitive notion of a sentinel presented above. Rather than sentinels existing as autonomous agents, our system employs a sentinel manager which is responsible for processing sentinel queries. Since the goal of our system is scalability, our implementation seeks to maximize throughput and may sacrifice individual sentinel latency if necessary. We maximize throughput in two main ways: minimizing redundant network access and minimizing local processing.

Preprocessing All sentinels are first checked to see if applying a signature to the new document will reduce overall computation. This is determined by comparing the probability that a document has changed with the cost of computing the signature. If the probability that the page has changed is high, then the cost of the signature computation is not worth the benefits and the signature computation is skipped. If there is a change in the signature, certain sentinels (e.g. *any change*) may be notified immediately without further processing.

Location masking A sentinel will not necessarily watch an entire page for changes. Location masks mark those sections of a page that are interesting for a sentinel, reducing the computation required for all change detection algorithms. A location mask is an array that marks document nodes that are of interest to a particular sentinel.

Content Sentinels Text change detection operates over the document's content list. The algorithms match user-defined regular expressions or keyword phrases directly over the content container, automatically bypassing other parts of the document. The separation of document components in the Page Digest encoding allows text change detection to operate efficiently over the text without introducing extraneous computational overhead for parsing different document elements during the search.

Structure Sentinels The Page Digest encoding allows structure sentinels, which ignore the text content of Web documents, to avoid loading the entire document into memory for processing. In many cases, the structure of an HTML document may be small enough to maintain in memory with the sentinel. For example, the CNN home page contains approximately 1200 nodes; the Page Digest encoding of the structure, tag map, and tag names requires less than 4 KB of memory storage. Experimental evaluation demonstrates that the largest cost in sentinel evaluation is the local I/O; thus, structure sentinels can save over 50% of the time required for content sentinels before any compuation is done.

Structure information can be used to dramatically speed up the execution of generic tree-to-tree change detection algorithms in the case where there are few or no differences between the document's structure. Algorithm 5 demonstrates how a tree change detection algorithm can be optimized to simple leaf comparisons for matching subtrees whose structure has not changed. The algorithm performs no worse than the underlying tree-to-tree algorithm in the case where there are many structural differences between the documents being compared.

Combination Sentinels Combination sentinels monitor changes to both the content and structure of the target document. Combination changes include adding items to lists, inserting rows or columns in a table, or moving document sections from one location to another. Combination sentinels are initially processed identically to structure sentinels since significantly less of the document's Page Digest must be loaded to determine if there is a structural change. If a structural change has occurred, the combination sentinel is then processed like the content sentinel.

Processing Specific Structural Elements Structure sentinels may employ refinements that limit what structural elements the sentinel considers. In web documents, this can be used to detect changes that are restricted to hyperlinks or in-line images. Since the actual link itself is embedded in a node attribute (of the `a` and `img` nodes, respectively), data location is simply a matter of traversing the tag and attribute lists and determining if changes have occurred in the appropriate node types. The Page Digest encoding makes this a very efficient array scan operation thereby enabling fast change detection for these types of changes.

Algorithm 5 Smart Difference Compare

```
S = Original PageDigest of page at time t_o (loaded from disk)
W = Page at time t_1

let D = EncodeToPageDigest(W)
let sIndex = 0
let dIndex = 0
for sIndex < S.size do
  if S.ST[sIndex] = D.ST[dIndex] = 0 AND S.TL[S.TM[sIndex]] =
  D.TL[D.TM[dIndex]] then
    if S.ST[sIndex].getContent() ≠ D.ST[dIndex].getContent() then
      /* mark node as content update */
    sIndex++; dIndex++;
  else if S.ST[sIndex] ≠ D.ST[dIndex] OR S.TL[S.TM[sIndex]] ≠
  D.TL[D.TM[dIndex]] then
    /* structural change, mark subtrees for examination in general diff
    algorithm */

    /* skip the subtrees just compared */
    sIndex += S.sizeOfSubtreeAt(sIndex)
    dIndex += D.sizeOfSubtreeAt(dIndex)
  else
    sIndex++; dIndex++;
```

3.6.2 Multiple Sentinel Processing

We expect that many users will want to monitor popular documents for changes. This overlap in interest presents opportunities for dramatically increasing efficiency by semantically grouping related change requests. Grouping provides many potential advantages for

optimizing sentinel processing. By combining related requests, we are able to minimize network costs and reduce processing costs with only a modest processing overhead for group maintenance. Sentinel grouping allows the system to execute a single document fetch for all sentinels over a certain document. This optimization assumes that all sentinels over the document expect updates at the same interval. For groups where this is not the case, the sentinel grouping process temporarily removes sentinels that are not due for notification from the group. Coupled with the Page Digest encoding, grouping allows the system to perform change detection only once in many cases. Grouping also eliminates redundant change detection over documents that have not changed by executing more general sentinels in the group before their more specific counterparts. For instance, if an *any change* sentinel detects no change to the document, there is no reason to process more specific sentinels since it is impossible for anything of interest to them to have changed.

Group Types The sentinel composition of the group will determine the types of optimizations that can be exploited for that group. At a minimum, all groups share a common document; therefore, in the worst case the only optimization the system can perform is to combine the group's fetch operations into a single network access. Since the network is often a major bottleneck in systems that deal with the Web, any reduction in network traffic can produce dramatic results for the scalability of the system by reducing I/O blocking and allocating available bandwidth more effectively. The opposite extreme is found in groups where all of the constituent sentinels are identical. In this case, document fetch and change detection are performed once for the entire group.

We expect that most groups will fall somewhere between these two extremes. An important class of sentinel groups are those having location subset relationships in which the monitored regions are contained within one another. Figure 23 shows two sentinels monitoring a portion of a Web document. The *attribute* sentinel's monitored location is a subset of the *any change* sentinel's location. In this case, the system processes the *any change* sentinel first: if it fails to find a change, then evaluation of the *attribute* sentinel is skipped.

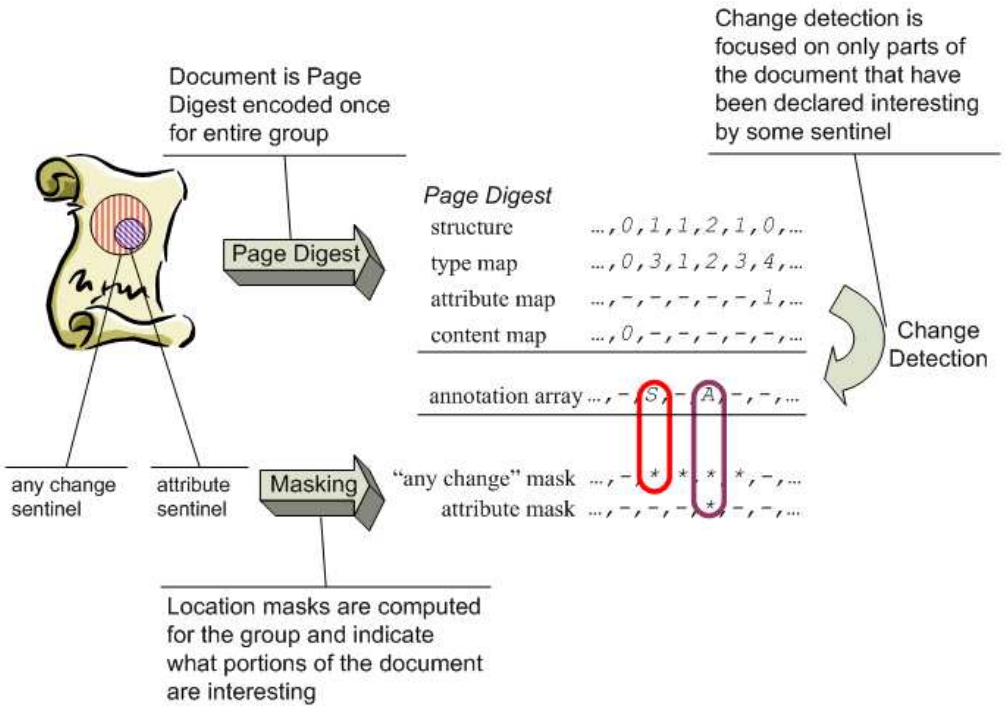


Figure 23: Grouping Process Overview

Grouping Process Figure 23 shows the grouping process. Group processing begins when the sentinel manager is triggered to check for changes that affect the group. The sentinel manager retrieves all sentinels that are installed on the same Web document from which it selects the current active set as determined by each sentinel's firing interval. The new version of the Web document is retrieved from its source once, encoded to the Page Digest encoding, and maintained in memory. The previous version of the document is then loaded from disk. This minimizes the network bandwidth required to evaluate sentinels and reduces local I/O.

Once the sentinels and documents have been loaded, the sentinel manager checks for an *any change* sentinel that covers the entire page. If present, this sentinel is run first: if it cannot find a change, no further sentinels are executed. After the *any change* sentinel has run, each sentinels location mask, which is computed when the sentinel is installed, is loaded from disk. From this set of location masks, the sentinel manager constructs a group location mask which focuses change detection on only the parts of the document that are of interest to the group.

The change detection process is sketched in Algorithm 5. The output of this algorithm is an *annotation array*, which corresponds directly to the structural array of the Page Digest and marks each node in the document with the type of change that has occurred at that node. In addition, each array position also signals the change state for all descendents of the current node.

Once an annotation array has been computed, the final step in sentinel group processing examines the locations in the annotation array specified in each sentinel's location mask; Figure 24 shows an example of this process. Comparison begins with the sentinels in the minimal covering subset, which is the smallest subgroup of sentinels that covers every location of interest to the group for each particular change type. If none of the sentinels in this subgroup detects a change, no further processing is needed. Sentinels in a particular location execute in order from more generic to more specific, as shown in the sentinel hierarchy of Figure 22. A sentinel over the content executes before more specific sentinels over keywords, phrases, or regular expressions. Generic structural sentinels that monitor the entire document structure are processed before refined structure based sentinels (layout, attributes, comments, and type).

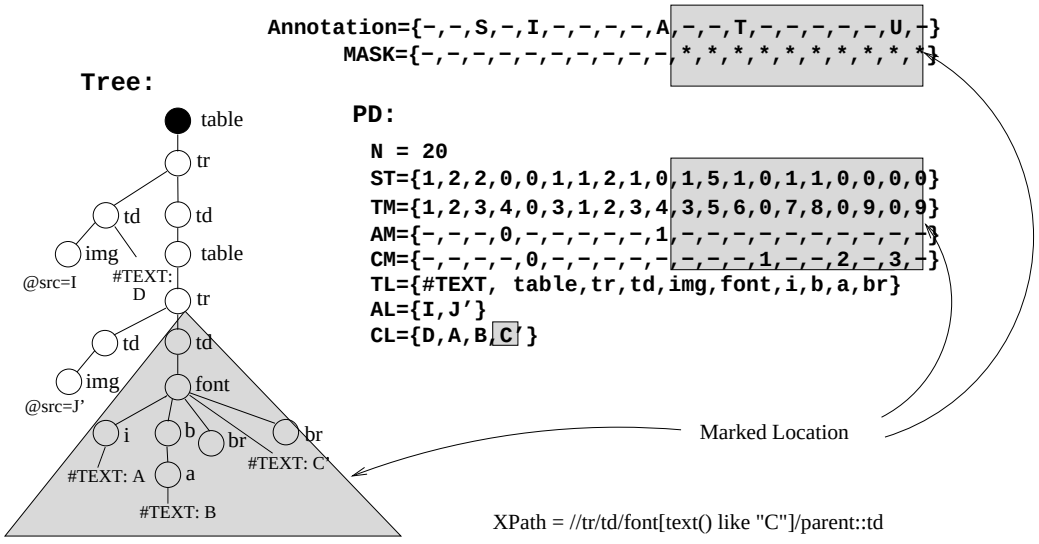


Figure 24: Location Mask applied to Page Digest and Change Annotation

3.7 Experimental Evaluation

In this section we report three sets of experiments that evaluate the performance of our algorithms and grouping techniques. These experiments evaluate sentinel performance with respect to processing time, grouping, and data management.

Experimental Environment. All experiments were run on a SunFire 280 dual 733 MHz processor server with 8GB RAM, 72 GB disk on a RAID 5 controller, and gigabit local Ethernet. The software was implemented in Java and run with the J2SE 1.4.0 from Sun using the server virtual machine. Experiments were averaged over at least ten execution runs.

Test Data. We used two different sets of test data to validate our system. Our first source of test data was a set of randomly generated HTML files that explicitly includes documents with varying numbers of nodes, page sizes, and tree shapes. We wanted to ensure that we adequately tested the performance of the various parts of our system relative to data sets having widely different characteristics. In order to construct this data, we built a custom generation tool that can create HTML documents containing an arbitrary number of nodes with a variety of content, tree shapes, attributes, and tags.

The second set of data used for these experiments were pages gathered from the *Yahoo!* news portal from December of 2001 to October of 2002; these pages are used as a representative of a relatively complex type of Web page that can support a large variety of trigger types. Page size in this set varies between 30KB and 60KB, averaging 47KB; the number of nodes in each page varies from 1400 nodes to 2171 nodes, averaging 1672 nodes. For the first few months we collected a snapshot every hour, and for the last six months a snapshot every day, for a sample set of over 1300 pages.

3.7.1 Sentinel Performance

3.7.1.1 Overall Performance Improvement

Our first experiment tests the overall performance of the sentinel processing system. We compare the cost of processing sentinels with the WebCQ system, also developed at Georgia Tech. WebCQ was first implemented as an adaptation of the Continual Query system

for monitoring distributed databases [69] to the monitoring of Web pages. The original implementation was based on an Oracle database system for storing the historical cache and meta-data for users and sentinels. A careful examination of the running system revealed serious bottlenecks in accessing the database to retrieve sentinels and previous versions of Web pages. To address these issues, the Page Digest Sentinel system implemented three specific changes: (1) moved the page cache from database into the file-system; (2) moved the user sentinels to main memory with logs to provide failure protection; (3) more efficient Web page encoding (Page Digest) to improve I/O times, storage space, and to provide a a base for more efficient algorithms. On top of those improvements, we implemented highly effective grouping techniques to reduce redundant computation. These improvements have contributed to achieving one to two orders of magnitude speedup in processing time. Figure 25 shows the performance of the WebCQ implementation compared to the Page Digest enhanced sentinel processing system with and without grouping.

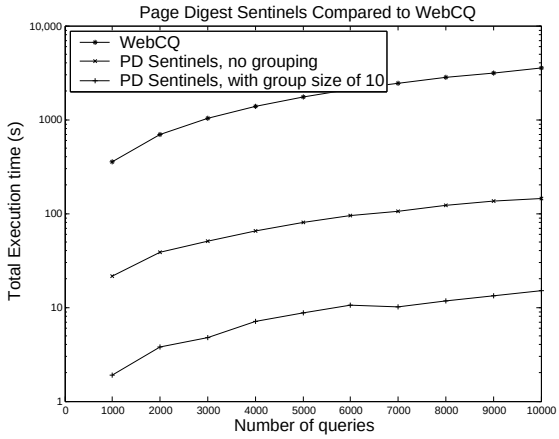


Figure 25: Comparison of Page Digest Sentinels with WebCQ

3.7.1.2 Execution Performance for Zipf-like Sentinel Distribution

In general, we expect sentinels to be distributed in a Zipf-like pattern, which suggests that a few pages will be highly popular while most pages will draw only a few users. Zipf-like distributions are expressed by the equation Ω/i^{α} describing the popularity of page i, where Ω is a normalizing constant, i is the i^{th} most popular page, and the exponent α

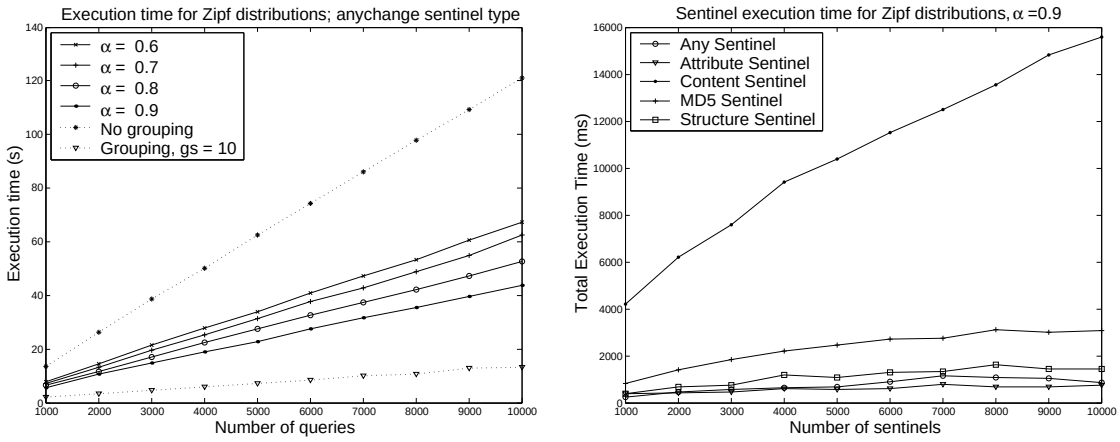


Figure 26: Zipf-like distribution of sentinels and execution time

describes the slope of the curve. Low values of α indicate that many of the sites have similar popularity, while higher values of α indicate that a few sites are drawing most of the users. Zipf-like distributions have been used to model the page popularity for Web proxy caches. Experimenters report α values between 0.6 and 0.9 [9]; the distribution of sentinels over Web pages should show a similar distribution.

The left graph in Figure 26 shows the execution time for groups distributed in a Zipf-like fashion with four different α values. This experiment used the cached Web pages from the *Yahoo!* data set. For reference, we show the execution times for sentinels that have not been grouped and sentinels in groups of size 10. This graph provides an approximate measure of the expected performance for processing sentinels in a typical operating environment. The no grouping line approximates worst-case performance while the line show a fixed 10 sentinel group size shows the system in a simplified, optimistic environment. The four Zipf-like lines reflect the observed popularity distributions in Web proxy caches.

The right graph in Figure 26 divides the execution costs of various sentinel types for a typical Zipf distribution of $\alpha = 0.9$. Since the *Yahoo!* data set was used, pages changed significantly between different versions of the documents, eliminating short-circuit evaluation. In addition, each of the change detection algorithms, except the MD5 sentinel, marked specifically which part of the document changed. This graph highlights the effects of the Page Digest on change detection efficiency: using the Page Digest dramatically improves

the performance of sentinels over pages where either no changes have occurred (identified using the MD5 sentinel) or where users are not interested in textual changes.

3.7.1.3 Cost for grouping sentinels

While grouping sentinels together can save memory and execution time, there is some overhead for creating sentinel groups. Grouping sentinels requires three steps: creating a group, inserting new sentinels into a group, and loading and computing the MD5 hash for the group. Figure 27 demonstrates the relative costs of grouping sentinels depending on the group size. The pages used for this experiment were based on the data collected from *Yahoo!*.

The cost of grouping sentinels decreases for larger group sizes because adding a sentinel to a group is the cheapest of the three steps. Every other step only needs to be performed once per group; the cost of these steps can be amortized over the size of the group so that the larger the group the more savings can be realized. In other words, grouping 50,000 sentinels into groups of 100 requires the creation of only 500 groups, 500 I/O requests to load the page into memory, and 500 MD5 computations. Putting these same sentinels into groups of size 1 means creating 50,000 groups.

There are five lines in Figure 27: three lines list the average cost of grouping sentinels into groups of different sizes. The remaining two lines showing the average cost of the local I/O required to retrieve a document and the cost to compute an MD5 signature for a document. As expected, the average time required to group sentinels increases only slightly as the total number of sentinels increases. Group time for a particular sentinel is dependent on the average number of sentinels per group and not on the total number of sentinels in the system.

3.7.2 Component evaluation comparing Page Digest vs. DOM Tree

In this section we compare basic operations between the Page Digest and DOM formats. Load time and simple traversal time comparisons between Page Digest and HTML and DOM parsers was already discussed in Section 3.3. Here we compare the time required to perform equivalent change detection algorithms over the two formats.

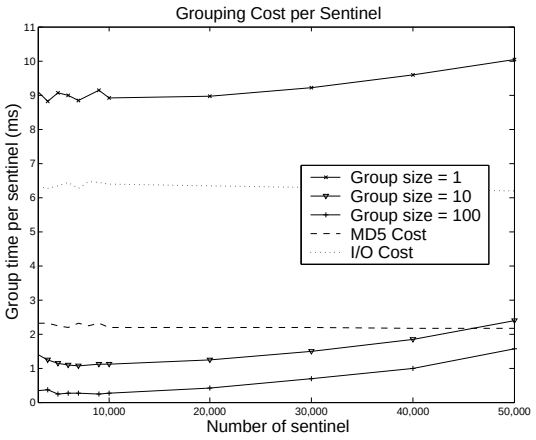


Figure 27: Grouping cost per sentinel

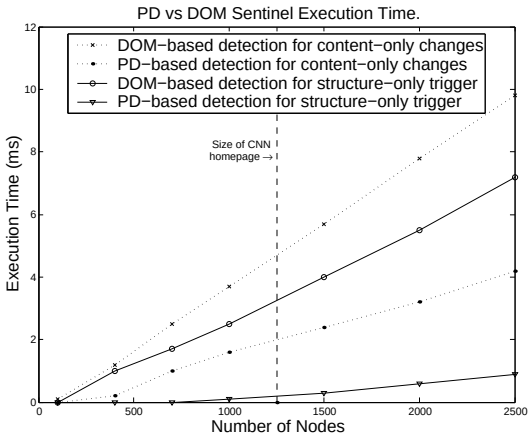


Figure 28: Sentinel Cost

Figure 28 compares execution times for the two basic sentinel types of content and structure change. This experiment used the randomly generated data files to examine performance over a range of document sizes. Experiments on data gathered from *Yahoo!*, *CNN*, and other popular Internet sites confirm these results. As designed, the Page Digest encoding performs much better than equivalent algorithms over a conventional DOM tree. For average sized documents, the content and structural change detection algorithms perform at least twice as fast.

3.8 Page Digest Sentinel Summary

We have introduced Page Digest, a mechanism for efficient storage and processing of Web documents, and shown how a clean separation of the structural elements of Web documents from their content promotes efficient operations over the document. Our experimental results show that Page Digest encoding can provide an order of magnitude speedup when traversing a Web document and up to a 50% reduction in the document size. We have seen the impact of Page Digest on the performance of the Sdiff information monitoring Web service and shown how Page Digest's driving principle of separation of data and content provides the ability to perform very efficient operations for Web documents. Our analytical and experimental results indicate that the Page Digest encoding can play a foundational role in large scale Web services by enabling efficient processing and storage of Web data.

Our research on Page Digest continues along two dimensions. On the theoretical side,

we are interested in studying the formal properties of Page Digest and its role in enhancing tree-based graph processing. On the practical side, we are interested in deploying the Page Digest encoding into a wider range of enabling technologies in the electronic commerce domain, including Web service discovery through similarity and Web page compression for efficient storage of Web data.

In addition to the Page Digest, this work has demonstrated a scalable system for Web Change monitoring that uses efficient data management and request grouping to achieve good performance. We are currently extending the basic sentinels by adding support for groups of related Web pages and dynamic content pages that require session management or authentication. We also plan to combine the lessons we have learned from our experience with sentinels and those from conventional continual queries over databases to produce an efficient and data-centric change detection system over XML.

CHAPTER IV

XPACK

4.1 Introduction

The XML document format [8] is emerging as a popular document encoding for online information exchange. Standardized Web document formats like XML are advantageous for a variety of reasons. XML has well-defined semantics, strong internationalization support [88], and a plethora of developer tools for managing and exchanging data. In addition, XML derived languages, such as WSDL [27] and SOAP [91], provide higher level interaction standards that leverage existing XML technology. XML data is self-describing and authors are encouraged to use clear entity names to assist other users in understanding the data [8]. Since many parties interested in data exchange interact with different entities during the course of a transaction, predefined data exchange standards are a must. In the highly dynamic world of the Web, the set of data exchange partners an entity may use will evolve over time, which provides a strong argument for the use of industry-standard communication technologies rather than ad hoc solutions.

However, XML has two disadvantages that present obstacles to widespread adoption as an information exchange medium for many applications: the size penalty and textual representation. Many entities that might consider XML would need to convert existing proprietary document formats into XML, which typically produces an undesirable and dramatic increase in the size of the stored data [65]. Another concern stems from the fact that XML is stored in a text document encoding like ASCII, which incurs significant computational costs for parsing and validation.

Many applications exist that could benefit from the standardization of XML but require more efficient document representation. For example, data distribution and routing applications require large numbers of documents to be handled quickly [1], while today's mobile devices have limited processing power and storage capacity. For these and other

applications, it is advantageous to minimize the storage space required by documents and to provide efficient access to application-specific areas of interest. While XML provides advantages with its self-describing characteristics and universally recognized format, these applications cannot afford the performance penalty that has previously been the price of converting to XML.

There have been several efforts to reduce the storage impact of converting to or otherwise utilizing XML. XMill [65] is an XML document compressor designed to alleviate the concern of data expansion due to XML conversion. The WAP Binary XML Format [31] and its extension Millau [42] have been introduced for the efficient encoding and streaming of XML structure, particularly in a wireless environment. These compression schemes addressed the data expansion concern but do not provide explicit support for querying compressed documents. There have been subsequent efforts to provide query support over compressed XML documents with systems that compress the content of the document [76] or its structure [12].

XPack was conceived as a document compression system providing both good compression and strong query support for compressed documents. The design goal of XPack is to support the acceptance of XML as a viable data exchange mechanism by minimizing the performance penalty incurred by applications that use it. XPack's compression and query techniques are built upon the concepts of redundancy elimination and compartmentalization.

Redundancy elimination allows XPack to significantly reduce the size of Web documents. The XML format is highly redundant by design. For example, document nodes are specified with an opening and closing tag in order to enhance human readability of the document. Therefore, most document nodes occupy a minimum of $2 * |tagName| + 5$ characters, including 5 characters for tag delimiters. XPack's compression engine relies on techniques that reduce or eliminate this and other types of redundancy often found in Web documents.

Compartmentalization separates the components of a document into logical containers: aspects of the document that are normally intermingled, such as structure and content, are stored independently in the XPack system. Document compartmentalization provides

the basis for building efficient query operators over XML documents. For instance, a high-volume Web service could validate incoming SOAP messages quickly by simply examining the document's tag set to verify the presence of the appropriate SOAP envelope and method name tags. Path expression based document dissemination can also benefit from compartmentalization, since only the document's tag dictionary and structure must be examined to determine the routing for a document. In general, compartmentalization provides the ability for applications to operate over only the portion of the document in which they are interested.

This chapter presents XPack, a compression system for XML documents that provides four main contributions:

1. *Redundancy Elimination.* XPack reduces much of the redundancy found in XML documents, yielding a smaller document footprint.

2. *Binary Format.* XPack's binary encoding requires no parsing when loading a document from disk. Document parsing and verification is a resource intensive operation, so applications using XPack can expect better performance when loading documents.

3. *Compartmentalization.* XPack separates various document components to provide faster access to interesting aspects of a document, such as its tree structure information or node tag list.

4. *Compressed Data Access.* Unlike other widely used document compression systems, XPack provides general query facilities that operate over the compressed documents. Compressed data access allows applications to store data in a compact, space-saving format without sacrificing the ability to do ad hoc querying.

4.2 Related Work

The XPack document encoding is informed by previous work on the Page Digest system [85] for efficient representation of HTML web pages. The Page Digest was designed to support efficient document processing in applications such as Web change monitoring [16, 15]. XPack is an extension of this work that targets XML documents, in particular, by providing a

more flexible framework that supports containerized document compression and efficient path querying.

The foundation work for modern data compression was done by Ziv and Lempel [105], who proposed the idea of the dictionary compressor. Dictionary compressors operate by substituting repeated occurrences of a given string with a shorter sequence; the original file can be reconstructed by reversing the substitution. This technique and its variants have become integral components of many standard computing tools such as the Gzip [71] file compression tool. For a more general introduction to data compression, we refer the reader to [89, 101].

Recently, there have been several efforts to design XML-specific compression algorithms. The first, XMill [65], was designed to promote standardized document storage and transmission formats while alleviating the concern of data expansion that is often the penalty of converting data to XML. The XMill compressor achieves this goal by creating a container for each document tag and placing the data values for each tag into the same container. The containers are then compressed using a standard dictionary compression library. The intuition behind this approach is that grouping values by their tag names would arrange repetitious sections of the document closer to each other, which would yield greater compression efficiency from the dictionary compressor. A second contribution is the incorporation of meta-information about the semantics of the data to further reduce the storage requirements for known data types. For instance, knowing the data type of an "IPaddress" tag would allow the compressor to store the value for that element as four bytes, which could produce significant savings over data stored as longer character strings. The fundamental problem with the XMill approach is its opaque nature: data compressed with the XMill compressor is only available for use after being decompressed, a costly overhead step that must be added to the overhead of parsing the text document.

The Millau [42] binary format—an extension to the WAP Binary XML Format [31]—has been introduced for the efficient encoding and streaming of XML structure, particularly in a wireless environment. A technique using multiplexed hierarchical PPM models was introduced in [24]. It has been shown to be slower than XMill, but in some cases achieves

a higher degree of compression.

There have been several recent efforts to provide query support over compressed XML documents, typically by making a trade-off between the degree of compression and support for queries. The first of these techniques, XGRIND [95], compresses XML documents by using Huffman encoding for non-enumerated types. If a document conforms to a known DTD, additional compression may be achieved by encoding the enumerated types listed in the DTD. XGRIND supports exact match and range queries over the compressed XML document. Similarly, XPRESS [76] maintains the original structure of each XML document to support path queries, but instead uses a technique called reverse arithmetic encoding for compressing labeled paths of the document. In experiments, the XPRESS system is shown to be faster than XGRIND for compression and query resolution. In [12], the authors present an XML compression technique that supports path queries over the compressed XML. Their technique relies on the identification of shared subtrees across a single document. Recently, the XCQ compression and querying system was introduced for documents that conform to a particular DTD, though the compression scheme is not clearly articulated in the paper.

4.3 XPack Document Encoding

XPack is a document encoding and compression system designed to operate over XML data. In this section, we construct a formal model of an XML document to facilitate the explanation of the techniques that XPack employs. Using this document model, we define a set of *document operators*, which are reversible functions that transform the XML data to reduce representation redundancy and provide efficient access to interesting document components.

4.3.1 Reference Document Model and Design Concepts

We model an XML document as an ordered tree of nodes where each node has a name and optionally contains a namespace qualifier, attributes, and text content. More formally, an XML document D is a set of tags $\{t_1, \ldots, t_{2n}\}$; each tag t_i is a string of characters denoting the tag's name and value. We say that tag t_i is a *closing tag* if the tag name begins with the slash character '/'; otherwise the tag is an *opening tag*. A document D is required to

have the same number of opening and closing tags, which occur in tag pairs[1]. A tag pair describes a *node* used in a document's tree model: a node is a descendent of the tag pairs that enclose it and an ancestor of the tag pairs that it encloses. The document D's tag set $\{t_1, \ldots, t_{2n}\}$ must satisfy the following invariants:

1. $\forall$ opening tags t_i, $\exists\, t_j$ s.t.
   (a) $i < j$
   (b) t_j is a closing tag for t_i
   (c) $\nexists$ a tag pair t_k, t_l s.t. $i < k < j < l$, and
   (d) $\nexists$ a tag pair t_k, t_l s.t. $k < i < l < j$
2. D contains the same number of opening and closing tags
3. t_1 is an opening tag
4. by extension of the above, t_{2n} is the closing tag for t_1

Invariants 1 and 2 require all documents to contain a balanced tag set, i.e. each open tag must have a corresponding close tag. (a) and (b) state that open tags are required to preceed their closing tag. (c) and (d) are concerned with the proper nesting of tags; Figure 29 demonstrates proper nesting along with two examples of improper nesting. Finally, invariants 3 and 4 state that the first tag must be an open tag and that the last tag must be the corresponding closing tag.

4.3.2 XPack System Overview

The XPack encoding system operates over this document model to produce an XPack-encoded version of the document that retains the same structural elements and semantic meaning as the original document represented in a more efficient format. XPack espouses a container-oriented document structure that is created and modified by a set of unary document operators:

[1] Actual XML documents can also have "self closing" tags which combine the opening and closing tag. We model such tags as a tag pair t_i, t_{i+1}; in the ASCII version the closing tag t_{i+1} is implied. All nodes specified by self-closing tags are leaf nodes in the document tree.

```
...
<i>
  <k>
</i>
  </k>
...
```
(a)

```
...
<k>
  <i>
</k>
  </i>
...
```
(b)

```
...
<i>
  <k>
  </k>
</i>
...
```
(c)

Figure 29: Examples of improperly nested tags (a) and (b) along with a properly nested example (c).

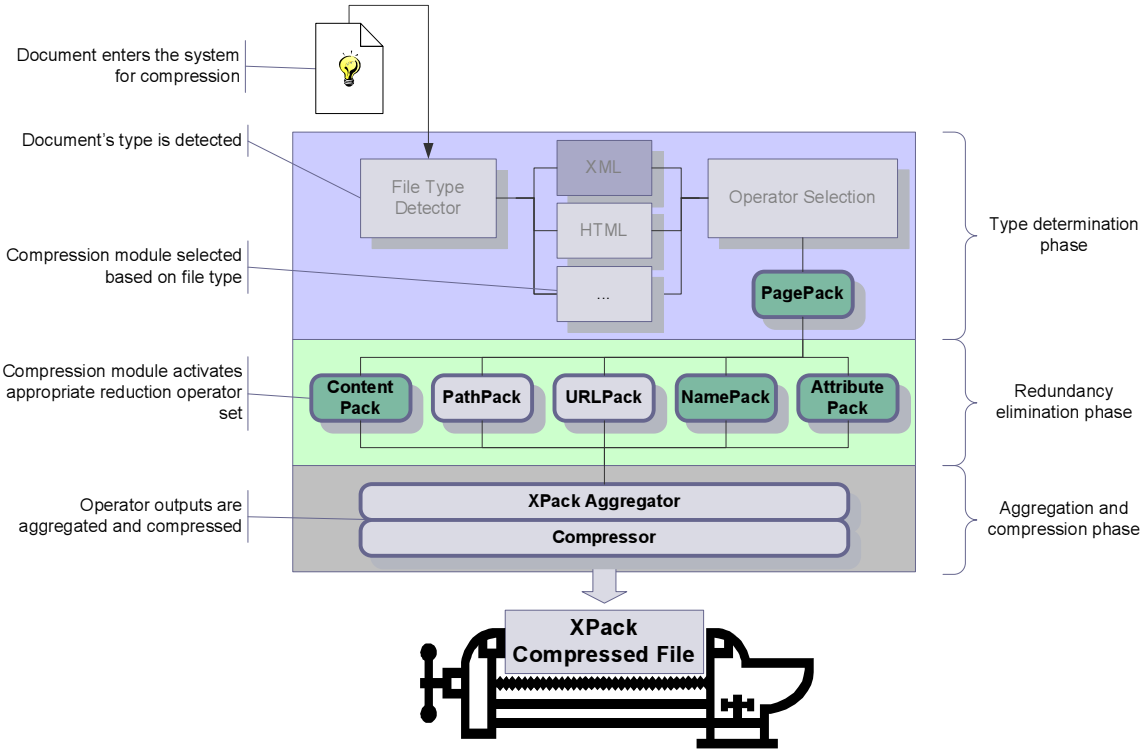


Figure 30: XPack system overview.

- PagePack (ϕ): document container encoding
- PathPack (ψ): path structure encoding
- NamePack (ρ): node tag name encoding
- URLPack (γ): document URL encoding
- AttributePack (α): attribute encoding
- ContentPack (χ): content encoding

XPack's document operators are designed to support flexible, orthogonal redundancy reduction. The PagePack operator is unique in that it operates over the original XML, while the remaining operators take a document that has already been containerized as their input. PagePack's purpose is to transform the text-oriented representation of an XML document into a compact tree-oriented representation of the document's structure augmented by a series of content containers. These containers can then be transformed, augmented, or replaced by subsequent operators. To as great an extent as possible, the remaining operators are designed to work in parallel so that overlapping computation can be used on parallel machines.

Figure 30 shows the XPack document compression process. When a document enters the system, a type detector module determines the document's type and loads a document type profile. The document type profile determines the default set of operators XPack will use in the redundancy elimination phase and also specifies how the document is split into components. Next, the document enters the redundancy elimination phase, which uses the selected XPack document operators to reduce the redundancy, and therefore the size, of the document. The minimized document components are then passed to the aggregator for reassembly and compressed to yield the final XPack-encoded document.

The heart of the XPack system is the redundancy elimination operators. The PathPack operator tries to reduce the space consumed by the document's tree structure. This encoding works best on documents that utilize structure more than content to convey meaning.

PathPack is also designed to provide efficient access to the paths between document nodes for faster path matching and query operations.

The NamePack operator utilizes observations about the tag names in XML documents to reduce their size. XML tag names tend to be repetitive and in certain cases, such as namespace qualified documents like SOAP messages, are extremely long. NamePack eliminates this redundancy by creating a numerical identifier for each unique tag name and substituting this identifier for the tag name throughout the document. NamePack is most effective on documents containing long, often repeated tag names.

The URLPack operator can reduce the space consumed by URLs. As an example, consider the XHTML documents that make up many Web sites. These documents utilize URL hyperlinks, many of which share common prefixes. In addition, these links will sometimes be placed in multiple locations on a single page to provide easier navigation for users. URLPack performs identifier substitution, like NamePack, to reduce the space consumed by these duplicated URLs. URLPack accounts for the similarities found in many document URLs by storing common URL prefixes only once. URLPack is most effective on documents having a large number of similar or repeated URLs.

The AttributePack operator combines the concepts from the NamePack and URLPack operators and reduces the space consumed by element attribute names and values. AttributePack performs identifier substitution on attribute names and substitution and prefix production on attribute values. AttributePack is most effective on documents that have many repeated attributes with similar names and values.

4.3.3 PagePack

The PagePack operator ϕ creates a *node structure container* S by assigning a unique identifier to each node in the document and extracting the structure information inherent in the opening and closing tag sequence. $\phi(D) = (S, M)$ is a reversible function mapping an XML document D to a compact tree-structure representation and a set of containers for the document's content. $S = \{a_1, \ldots, a_n\}$ where a_i is the number of child nodes under the opening tag t_i. S preserves the document's tree structure by recording the relationship

between opening tags. Node content is placed into a list of containers $M = \{m_1, \ldots, m_n\}$, which retain the information from the original document regarding each node's name, associated attributes, and content. The PagePack transformation stands as the basis upon which the other document operators are constructed.

Figure 31 demonstrates the intuition behind PagePack by illustrating its operation over an XML document, which is shown in its tree representation. Conceptually, PagePack encodes the document in three steps. First, the document tree is traversed in depth-first order and each node in the tree is assigned a unique identifier. For the sake of convenience, we choose an identifier equal to the visit order of each node; the root of the tree is assigned identifier "1." After each node has an ID, PagePack constructs a structure array that succinctly encodes the relationships between document nodes. Finally, node containers are constructed for each node in the document.

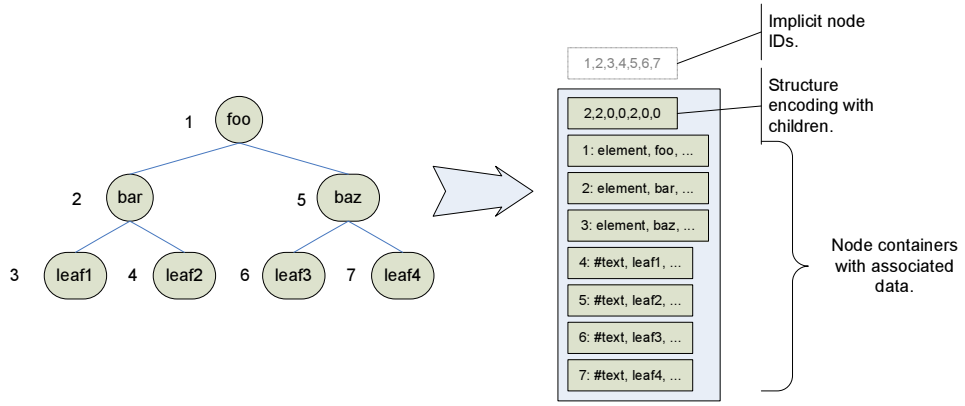


Figure 31: Effect of PagePack Operation.

While Figure 31 explicitly lists the node identifiers, a convenient byproduct of our chosen identifier structure is that the identifier array can be eliminated from the document encoding as it can be reconstructed from the structure array. Aside from recording node IDs—via the array indices—the structure array contains a count of the number of children for each node. The array is stored in depth-first traversal order. In the example above, the root node is stored as the first element in the array, where it is recorded that the node has two children. Because of the depth-first ordering, the second array position holds the root node's leftmost child, which itself has two leaf children. Thus the entire leftmost subtree is stored before

any of the root node's subsequent children.

The remaining containers in the PagePack structure hold the information contained in the original nodes of the document tree, such as each node's tag name, attributes, and any associated content. Node containers are stored in index order, so the first node container holds the data belonging to the node with index "1," the root node.

Algorithm 6 provides an algorithm sketch of the process of converting an XML document into its PagePack encoded form. The algorithm makes use of a number of functions. The function *stack*() returns an empty stack data structure that can be manipulated or tested with the operations *empty*(<*stack*>), *push*(<*stack*>, <*item*>), and *pop*(<*stack*>). The function *increment*(<*item*>) is a convenience function that adds one to a data element. Finally, the function *attributes_and_content*(< *input_stream* >) consumes data from an input stream up to the start of the next tag and returns it.

The algorithm begins by initializing its data structures. Then, for each opening tag, the tag's parent is popped off the stack, the parent's child count is incremented, and the parent is returned to the top of the stack. The child count of the current node is set to zero and its attributes and content are stored in node content array M. Finally, the node is pushed to the top of the stack to give a proper frame of reference for any of its children.

> *Algorithm 6 is linear in k, the number of characters used to specify the XML document D.*

Proof. Construction of the structure and node arrays S and M requires time proportional to the number of nodes in the document n; the other initialization steps require constant time. The algorithm's main for-loop examines each opening and closing tag once for $2n$ total iterations. The stack operations *empty*, *push*, and *pop* and the other operations in the for-loop, excepting the function *attributes_and_content*, require constant time. Computing *attributes_and_content* for each node requires time proportional to the number of characters in the attributes and value for that node.

To characterize the relationship between the number of characters k in the document and its node count n, we consider a "minimum-node-case" XML document with the minimum

number of nodes for a given k, which would contain a single root node with a single character name. The value of this node would occupy $k-7$ characters with the remaining characters being used to specify the node itself. In the "maximum-node-case," the document would have $(n-1)$ leaf nodes united under a single root and all tag names would be of length 1 to minimize the overhead from XML tag delimiters. In this case, $k = 7 + 4(n-1)$ yielding $n = \frac{k-3}{4}$. For all cases, $1 \leq n \leq \frac{k-3}{4}$. The algorithm is therefore $O(k)$. □

Algorithm 6 PagePack ϕ

```
Let D = {t_1, ..., t_2n} be the source XML document
Let S, M be arrays of size m = n
Let c = stack()
Let node = 0

for all t_i ∈ D do
  if opening_tag(t_i) then
    if ¬empty(c) then
      Let parent ← pop(c)
      increment(a_parent)
      push(parent)

    increment(node)
    a_node ← 0
    m_node ← attributes_and_content(t_i)
    push(c, node)
  else
    pop(c)
```

4.3.4 PathPack

Path structure encoding transforms the tree structure of an XML document into a sequence that encodes the paths found in the document tree. Given a node structure container S, PathPack $\psi(S)$ is a reversible function that generates a sequence $x_j, 1 \leq j \leq m$, where $m \leq n$ and each x_j is a subpath tuple of the form $< start_j, end_j, parent_j >$. Each subpath in the sequence represents a nonbranching fragment of a root-leaf path; later paths in the sequence are further to the right and further down the tree than earlier paths in the sequence. For example, the subpath $< 2, 3, 1 >$ represents a two-node nonbranching path between nodes 2 and 3; the start node of path, 2, is a child of node 1. Given a node

```
<entry>
  <host>202.239.238.16</host>
  <request>GET / HTTP/1.0</request>
  <contenttype>text/html</contenttype>
  <status>200</status>
  <date>1997/10/01-00:00:02</date>
  <bytecount>4478</bytecount>
  <refer>http://www.net.jp</refer>
  <useragent>Mozilla/3.1 [ja.] (I)</useragent>
</entry>
```

Figure 32: Example XML Document

structure container S for document D, the following algorithm sketch highlights the main components of the PathPack operation:

1. set the start of the next path to the root node $a_1 \in S$
2. for each node $a_j \in S$
   (a) if a_j is a branch or leaf, output the current path. for each child c, set c to the start of the next path and run PathPack with it as the root.
   (b) otherwise continue.

The PathPack encoding is designed to meet several criteria. First, it needs to provide a succinct representation of the tree structure of the original document. Inherent in this design goal is the ability to convert the encoded path structure back to the original tag tree representation. Second, the encoding represents the possible paths through the document tree. Finally, the encoding supports efficient execution of path-style queries.

Figure 32 shows an example XML document with the tag-tree representation depicted in Figure 33. To generate the PathPack encoding for this tree the PathPack operation traverses the tree in depth-first order beginning from the root node "entry." Each node is assigned a unique identifier which is its visitation order in the traversal. During the traversal, we discover any nonbranching paths which form the basis for the path structure encoding.

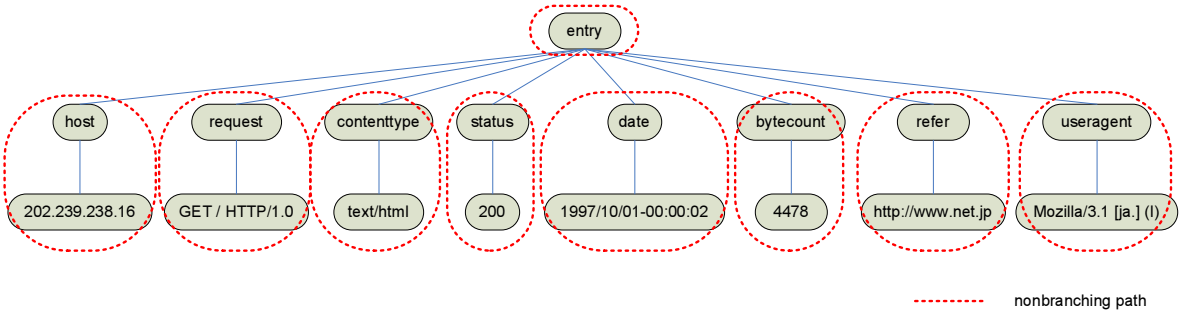


Figure 33: Tag-tree XML representation with nonbranching paths.

Algorithm 7 defines the formal process of transforming the paths of an XML document into the PathPack encoding. The function *stack*() returns an empty stack data structure that can be manipulated or tested with the operations *empty*(<*stack*>), *push*(<*stack*>, <*item*>), and *pop*(<*stack*>). The functions *isBranch*(<*item*>) and *isBranchOrLeaf*(<*item*>) are convenience functions that check if the given node is a branch or a leaf node. A node is not a leaf if it has children; it is a branch or a leaf if its child count $\neq 1$.

Initialization begins in lines 2 through 7 with the creation of two stacks: one to track the parent of each node as it is being processed and another to track the number of children remaining to be processed for each node. The two stacks begin with the information for the root node as their only entry. The algorithm then loops over each node, tracking its path through the tree and constructing a path whenever a branch or leaf is encountered.

Algorithm 7 is linear in n, the number of nodes of D.

Proof. The initialization steps require constant time. The main for-loop executes n times, once for each node in S. The test functions *isBranchOrLeaf* and *isBranch* operate in constant time by checking each node's child count in S. The stack operations *push* and *pop* and the other operations in the for-loop also require constant time. Thus the time complexity of the algorithm is $O(n)$. □

4.3.5 NamePack

The NamePack operator ρ eliminates this redundancy by storing each unique tag name once and assigning a short reference to each name. For a document D with node container

Algorithm 7 PathPack ψ

```
Let D be the XML document with node structure container S
Let parentNodeStack ← stack()
Let parentChildrenRemainingStack ← stack()
Let pathStart ← '-'

push(parentNodeStack, '-')
push(parentChildrenRemainingStack, '-')
for all a_i ∈ S do
  if pathStart = '-' then
    pathStart ← a_i
  if isBranchOrLeaf(a_i) then
    parentNode ← pop(parentNodeStack)
    parentChildrenRemaining ← pop(parentChildrenRemainingStack)
    if parentChildrenRemaining ≠ 0 then
      push(parentNode, parentNodeStack)
      push(parentChildrenRemainingStack, parentChildrenRemaining −1)
    if isBranch(a_i) then
      push(parentNodeStack, i)
      push(parentChildrenRemainingStack, a_i)
    print(pathStart,a_i,parentNode)
    pathStart ← '-'
```

$M = \{m_1, \ldots, m_n\}$, $\rho(M) = (I, M')$ is a reversible function that generates a set of tag name identifiers $I = \{i|$ unique tag names in $M\}$, stored in lexical order for efficient tag name searching.

NamePack reduces the redundancy of a Web document by generating tag name references and substituting the shorter references for the occurrences of the name in the document, eliminating the extra characters needed to store long and repeated tag names. NamePack is effective because the opening and closing tags that are the main structural feature of XML documents require tag names to be repeated; this necessity stems from the design of the document storage model but is not fundamental to representing structured documents in a computer system. Repetition of tag names can amount to a considerable proportion of the document's size.

NamePack collects the tag names from the node container M and stores each unique name in a new container. Occurrences of the names in the document are then replaced with a name reference that indicates which container and what name from that container is being referenced. M' is the new node container for D where tag names have been replaced

with the appropriate index into the tag name container I.

Algorithm 8 NamePack ρ

Let M be the document node container
Let I, M' be arrays

for all $m_i \in M$ **do**
  name $\leftarrow tagName(m_i)$
  index $\leftarrow insertIntoSortedList(I, name)$
  $m'_i \leftarrow replaceName(m_i, name, index)$

Algorithm 8 shows the NamePack conversion process.

Algorithm 8 has time complexity $O(n \lg n)$, where n is the number of nodes of the XML document D.

Proof. The initialization steps require constant time. The main for-loop executes n times, once for each node in M. Retrieving and replacing the tag name in the node container can be done with a constant amount of overhead. The function *insertIntoSortedList* inserts a tag name into a sorted list if the name is not already present and returns the name's list index. Finding the insertion point or name index in a sorted list l is known to require $lg(|l|)$, while the actual insertion can be done in constant time. Since the number of elements in list I is at most i, the number of nodes that have been seen, $|I| \leq n$ in every iteration. Thus the time complexity of the algorithm is $O(n \lg n)$. □

4.3.6 URLPack

A common feature found in many Web documents is the hyperlink reference, which directs the application processing the document to further information or provides additional material for an end-user to explore. Hyperlinks are described via a URL that typically specifies a protocol, a target site, and a document reference. Figure 34 shows an example of a SOAP message used to invoke a remote Web service. SOAP messages make heavy usage of namespace references, which are externally linked documents that define the semantics of the call. The example message contains three such references.

We have observed that many URLs contain common prefixes. For example, two of the URLs in the SOAP message in Figure 34 start with the prefix "http://example.org/alert"

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
   <n:priority>1</n:priority>
   <n:expires>2001-06-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
 </env:Header>
 <env:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:msg>Pick up Mary at school at 2pm</m:msg>
  </m:alert>
 </env:Body>
</env:Envelope>
```

Figure 34: Example SOAP Message

and all three have the same "http://" protocol specifier. The URLPack operator γ is designed to eliminate repetition in a Web document's URLs by factoring out these common prefixes. URLPack uses a modified dictionary substitution mechanism for encoding URLs that is restricted to start of string prefixes to maximize efficiency.

In general, $\gamma(M) = (U, M')$ where $U = \{u | u \in extractURLs(M)\}$; $extractURLs$ encodes the document's URLs through the following steps:

1. Retrieve the document's URLs from the node container M
2. Sort the URLs into lexical order, remove duplicates
3. For each u_i, replace the prefix shared with the expanded form of u_{i-1} with the count of shared characters

Figure 35 depicts these three steps operating over the URLs in the example SOAP message. First, the URLs are extracted and placed into the URL container. This container is then sorted so that the URLs are in lexical order, which maximizes the potential for prefix factoring by ensuring that URLs with common prefixes appear near each other in the container. Finally, URLPack moves through the container and replaces common prefixes with a shared character count. In Figure 35, the common prefix "http://example.org/alert" is replaced with a character count of "24" in the second URL. When this URL is retrieved

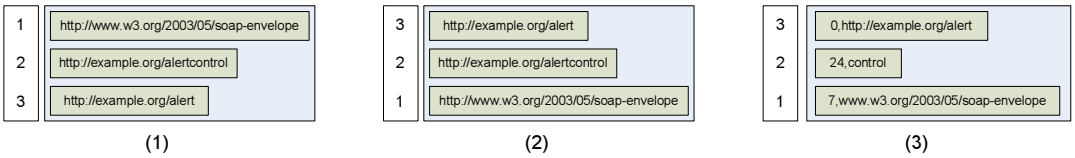


Figure 35: URLPack steps: (1) Extract URLs, (2) Sort URL container, (3) Reduce common prefixes.

from the container, the character count will be replaced with the first 24 characters from the previous URL, which will be expanded if necessary.

4.3.7 AttributePack

The AttributePack operator α is a logical combination of the ideas found in the NamePack and URLPack operators that is used to compress document attributes and expose them for faster processing. Attributes are associated with nodes: a single node can have zero or more attributes, each of which consists of a name and a possibly empty value. Attribute names are frequently reused throughout the document; certain attribute values—hyperlink reference URLs, for instance—will also appear multiple times in a document's attributes.

Consistent with the approach we have espoused with the other XPack operators, AttributePack extracts attribute names and values from a document D's node containers M. For a document D with node container $M = \{m_1, \ldots, m_n\}$, $\alpha(M) = (X, Y, M')$ where $X = \{x|$ unique attribute names in $M\}$, stored in lexical order for efficient searching, and $Y = \{y|y \in extractAttributeValues(M)$; $extractAttributeValues$ operates identically to the function $extractURLs$ used in the URLPack operator γ.

4.3.8 ContentPack

The ContentPack operator χ eliminates duplication of document content and organizes the content to achieve better document compression. ContentPack $\chi(M) = (C, M')$ is a reversible function mapping a document D's node containers M to a content list C and an updated list of node containers M'. The updated node containers replace each node's content with a reference to the appropriate entry in the content list for that node. Any duplicated text nodes in the original document will receive references to the same entry in

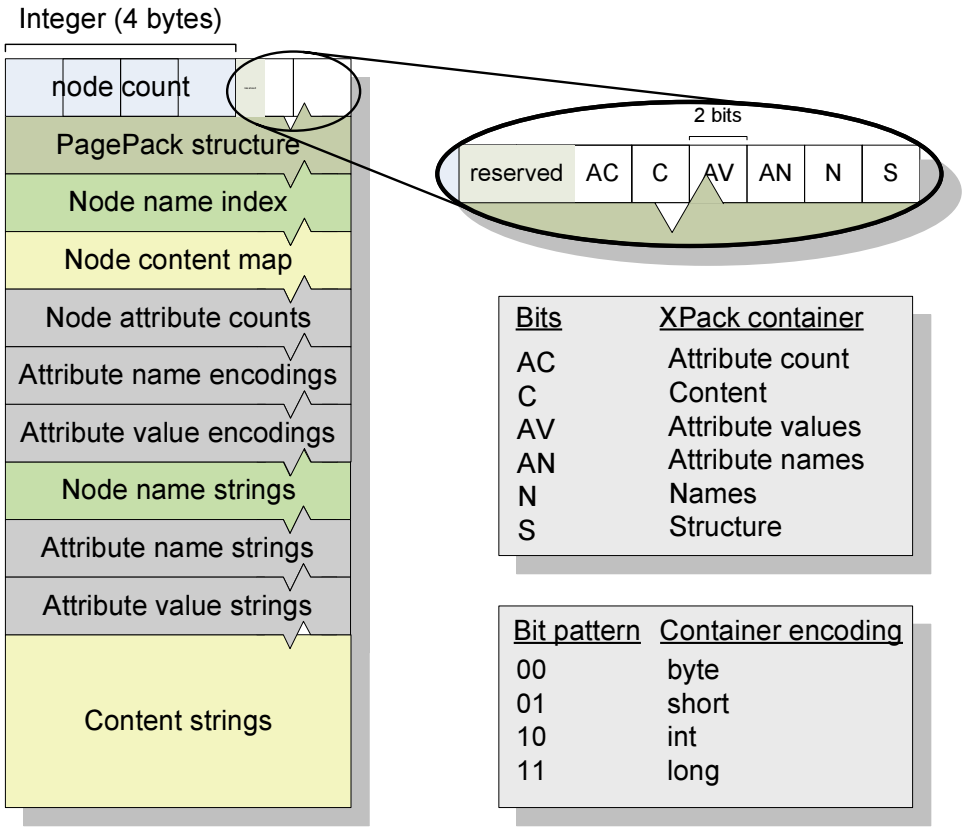


Figure 36: XPack binary disk layout scheme.

C.

4.3.9 XPack Aggregation and Binary Disk Format

After the selected packing operators have completed their operations over the input document, XPack collects the various document subcomponents and merges them to produce the compressed document output. Figure 36 shows the format our XPack prototype system uses when saving the aggregated components of a compressed XML document.

The first six bytes contain header information that is used as a guide when reading the document. The remainder of the file is the XPack document structures that were created by the packing operators. First, the document structure container holds the structure array created by the PagePack operator. Next, the document contains several containers for mapping tag names, node content, and attribute names and values to the appropriate nodes. The XPack file is completed with the document's tag names, attribute names and values, and text content. The mapping containers have an entry for each node that contains

an index into the corresponding value container for the map. For example, to determine the tag name of a node, XPack retrieves the value from the node's position in the node name index; this value is the identifier for the node's tag name in the node name string container.

In the six byte document header, the first four bytes contain the count of the document's nodes, which is used to compute the length of the mapping data structures in the XPack document. The next two bytes after the node count specify the bit width which, when combined with the node count, determine the length of each mapping container. For example, the length of the PagePack document structure container is the node count times the number of bytes used to encode the structure as specified by the two "S" bits in the header. A value of "00" indicates that the structure uses one byte per node, "01" indicates two bytes, "10" indicates an integer width of four bytes, and "11" indicates an encoding width of eight bytes.

The encoding width for each container is determined when the document is created by examining the output of each packing operator. For PagePack, the structure is a list of child counts, so an encoding width of one byte per node (S = "00") can be used if every node in the document has less than 256 children. For the name index, attribute name encoding, attribute value encoding, and content map, the encoding width is determined by the number of unique tag names, document attribute names, document attribute values, and unique document content strings, respectively. Finally, for the attribute count structure, the bit width can be determined by examining the node that has the largest number of attributes.

4.4 Experimental Evaluation

The experiments in this section are designed to test the features of the XPack system. The experiments are presented in three groups. The first group of experiments explores the effect of the individual packing operators that form the basis of XPack compression, demonstrating the effect each of these operators has on the data that it operates over. These tests are designed to provide insight into how the individual packing operators contribute to the overall compression achieved by XPack. The second set of experiments demonstrate

the compression performance of XPack over several document sets. Finally, the third set of experiments test the query performance of XPack and demonstrate that it is possible to achieve both good compression and performance. These experiments test the efficiency of document characteristic operations over compressed documents as well as more general path expression queries. The experiments show the power of the XPack encoding system: many interesting document operations can be completed with only a partial decompression of the document.

4.4.1 Experimental Environment

To test the XPack design experimentally, we have implemented a prototype XPack encoder and query processor. The prototype is implemented in Java. All experimental results presented here were conducted on a PC with an AMD Athlon XP processor at 1.4GHz with 512MB RAM running Windows XP Professional. The experiments were run on Sun's Java virtual machine for Windows, version 1.4.2. The Apache Foundation's Xerces XML parser was used for testing SAX and DOM document materialization performance and the Xalan XSLT library provided XPath query evaluation support for DOM documents.

4.4.1.1 Experimental Data

Random Walk. The random walk data set consists of approximately 2000 Web pages gathered by an automated crawler. To gather the data, the crawler's URL frontier was seeded with a small set of start pages. At each iteration, a URL was chosen at random from the frontier and the corresponding document retrieved. The document was then converted from HTML into XML and annotated with a single comment tag recording the originating URL of the document and a timestamp. The document's links were then added to the URL frontier for possible selection in the next iteration. The average size of the documents collected after normalization to XML was 43,888 bytes with a minimum of 161 bytes, maximum of 898,924 bytes, and standard deviation of 37,150 bytes.

Structure. The structure data set is a transformation of the random walk data set in which all of the nonstructural elements in the document have been stripped out. In addition,

all tag names have been replaced with a string of the form _x where x is the depth-first visitation order of the node. The '_' character is necessary for the document to be valid XML, since XML tag names are forbidden to start with a number. This data set was designed to test the effect of the PagePack and PathPack operations on the structure of XML documents and allow the results to be meaningfully compared with the performance of XMill and gzip.

Shakespeare. The Shakespeare data set is a public domain collection of Shakespeare's plays that have been converted into XML. The tag set is small and consists of such tags as LINE, SPEECH, and SPEAKER. None of the nodes have attributes. The data set contains 37 documents whose average size is 213,448 bytes with a minimum of 141,345 bytes, maximum of 288,735 bytes, and standard deviation of 36,345 bytes.

SwissProt. The SwissProt data set is an XML document containing protein sequence specifications coupled with a large amount of annotation data. The original document was obtained from the University of Washington's XML repository and consisted of a single file approximately 109MB in size. Entries from this document were split into 23 files of approximately equal size. The tag set in these documents is relatively small and there is a significant amount of text content. Attributes are common and tend to have short names and values. The documents have an average size of 5,145,228 bytes with a minimum size of 5,032,079 bytes, a maximum size of 5,155,767 bytes and a standard deviation of 24,735 bytes.

DBLP. The DBLP data set contains a computer science bibliography centered around database research that contains references to dissertations, masters theses, articles in journals and conferences, and technical reports. The original document was obtained from the DBLP Web server and consisted of a single file approximately 219MB in size. This single file was then split into 69 files where each file grouped articles published in the same year. These files contain many short content sections, short node names, and a few short attributes. The documents have an average size of 3,362,037 bytes with a minimum size

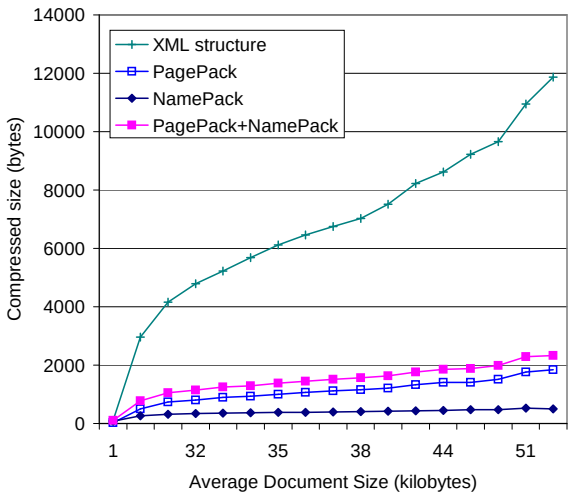


Figure 37: Size of document structure.

of 1,723 bytes, a maximum size of 24,597,150 bytes, and a standard deviation of 5,899,654 bytes.

4.4.2 Packing Operator Experiments

The first group of experiments test the performance of the individual packing operators that comprise the XPack system. Each packing operator is designed to reduce the redundancy present in XML documents. These operators work by removing repetitive strings from the document, possibly substituting a numerical placeholder as a reference to the replaced string. The experiments in this section show the performance of the PagePack, PathPack, NamePack, and URLPack operations.

4.4.2.1 PagePack

The PagePack operator encodes the document's structure into an efficient, array-based representation that is used to organize the remaining transformation operations. Figure 37 illustrates the savings achieved using PagePack in a comparison over the structure data set. The graph plots the size of the document's structure, as measured using the structure data set; the x-axis provides the size of the original, complete document. Thus, the XML structure line provides an estimate of the space consumed by structural elements in the original XML documents from the random walk data set. In order to keep the graph

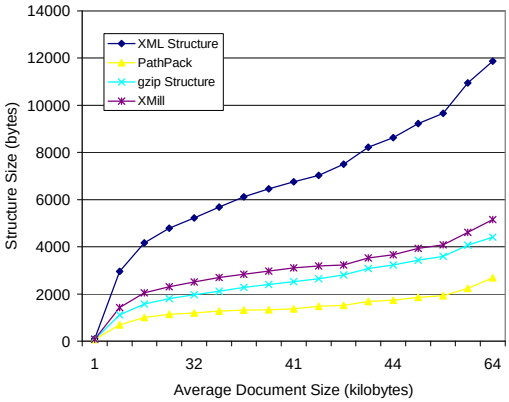


Figure 38: Size of document structure relative to the size of the original document.

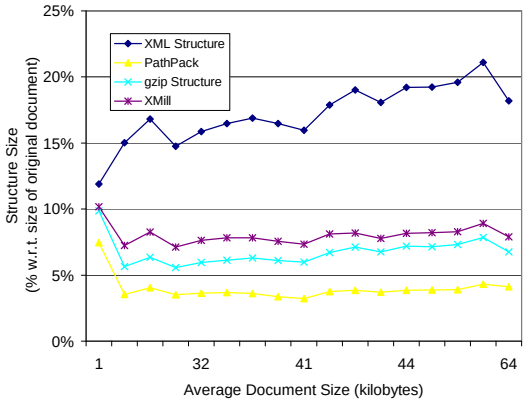


Figure 39: Size of document structure as a percentage of the original document size.

manageable in the face of the large number of documents in the data set, the plot points show the average sizes for groups of 100 documents.

The remaining three lines in the graph show the space consumed by the PagePack and NamePack operators along with the sum of these two values for the structure data set. The XPack encoding of a document's structure consumes much less space than its XML equivalent due to the efficient binary representation of the structure and the elimination of delimiters. The sum line (PagePack+NamePack) shows the space needed to store all of the data represented in the XML structure line, including both the document's structure and its tag names; this line therefore provides the most useful comparison with the XML structure line. PagePack eliminates the structural storage inefficiencies present in XML while preserving the original structure of the document.

4.4.2.2 PathPack

The PathPack encoding reduces the amount of space occupied by the structure of Web document while retaining the ability to execute traversals and queries over the structure. Although document structure is preserved with the PagePack structure encoding, PathPack provides an enhanced representation of the constituent paths in the document for faster path queries.

Figures 38 and 39 show the effects of the PathPack operation on the structure of an

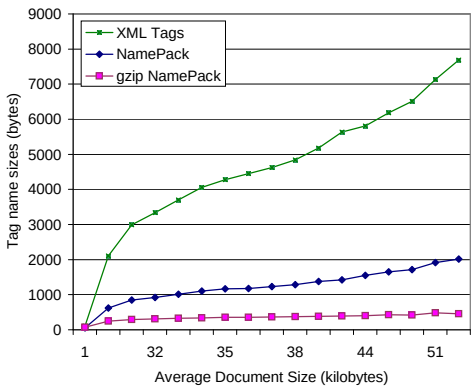


Figure 40: Effect of NamePack on document tag names.

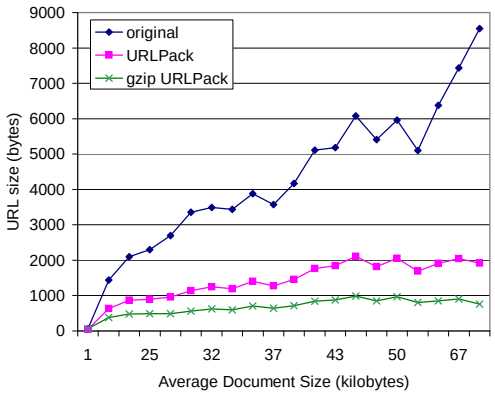


Figure 41: Effect of URLPack on document URLs.

XML document. The data in this test is the structural data set described in Section 4.4.1.1. PathPack achieves excellent compression of the tree structure of a document by reducing the amount of redundant information in the representation. In addition, XPack's compartmentalization of document components allows the PathPack encoding to be used for efficient path operations by simply extracting the PathPack structure from the compressed document: a complete decompress of the document is not required.

4.4.2.3 NamePack

Another XPack document operator, NamePack, removes the repetition of tag names from XML documents. NamePack creates a tag dictionary for the document and replaces occurrences of the tag name with a shorter numerical representation. Figure 40 shows the effect of this redundancy elimination on the size of the documents in the random walk dataset. The graph plots the size occupied by tag names in the XML documents encoded as ASCII text in comparison with the space occupied by the tag names and index references after NamePack encoding the document. Since the documents in this data set are derived from HTML, the tag names are short, typically less than seven characters. In data-centric XML, tag names tend to be considerably longer which would give NamePack more opportunity for reducing the space occupied by the tags.

4.4.2.4 URLPack

Figure 41 shows the average amount of space saved by URLPack in terms of the number of bytes saved relative to the original URLs' sizes. To gather the data in this graph, we used the random walk data set, described in Section 4.4.1.1. The documents were grouped into collections of 100 documents selected in nondecreasing order of the byte size of the URLs in the original document. Thus the first group—documents 1 to 100—contains the documents with the smallest number of bytes occupied by URLs. The figures plot group averages.

4.4.3 XPack Compression Performance

The next few experiments test the aggregate performance of the XPack compression system. The data sets that we have selected are intended to represent a broad cross-section of the types of XML documents typically used by applications. The random walk data set contains documents representative of the XHTML format used by modern, standards compliant Web applications. An important characteristic of this data set is the large number of attributes containing long, somewhat similar values. In contrast, the Shakespeare data set is characterized by a large amount of text content with no attributes and a small, terse tag set. The DBLP data set is similar to the Shakespeare data set but contains a few attributes and more balance between document structure and content. The SwissProt data set is highly structural in nature, containing more attributes and document nodes than content. These data sets provide a balanced means for comparison of different compression schemes similar to the varied conditions encountered by XML applications.

The documents in these experiments were converted from XML to gzip compressed XML, XMill, and XPack. For the XPack encoding, the documents were passed through the PagePack, NamePack, and AttributePack operators. The resulting document containers, including the node containers holding the text content of the document, were sent through a gzip compression filter and written to disk.

The random walk data set contains documents having a relatively small tag dictionary consisting of short names—each document employs a subset of the HTML tag set. These documents contain many attributes with repeated or similar values, such as hyperlink URLs.

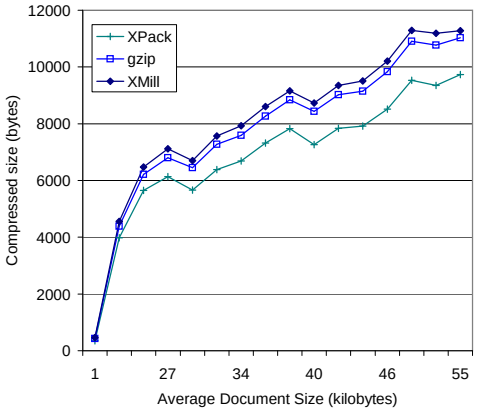


Figure 42: XPack document compression, random walk data set.

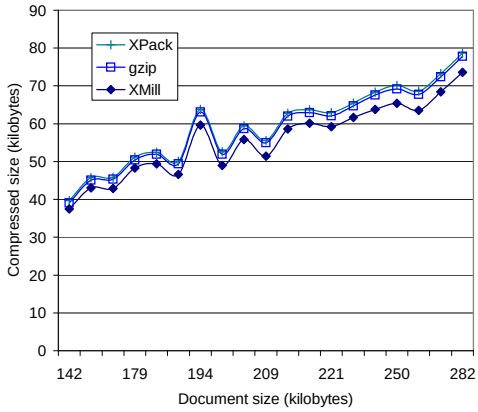


Figure 43: XPack document compression, Shakespeare data set.

The documents also have significant text content but are not dominated by it as in the Shakespeare data set. Figure 42 demonstrates the effectiveness of the XPack encoding system for compressing the random walk data set. The graph shows the size of the XMill, gzip, and XPack compressed versions of the documents, whose original sizes appear on the x-axis. The documents have many repetitive attributes, such as navigation URLs, that provide a significant opportunity for redundancy elimination by AttributePack, allowing XPack to achieve compression rates up to 20% better than the other systems' rates.

In contrast with the random walk dataset, the Shakespeare data set is content-heavy, with the majority of the size of the documents occupied by actors' lines. None of the documents contained node attributes, eliminating any potential savings from AttributePack. The tag dictionaries are composed of a few short names. Even in the Shakespeare data set, with its limited redundancy, XPack achieves compression rates comparable to the other systems as shown in Figure 43. These results demonstrate that XPack can achieve compression rates comparable to XMill and gzip while maintaining the ability to perform meaningful operations over the document without first decompressing it.

The next experiment tests the performance of XPack, XMill, and gzip on the SwissProt data set. Figures 44 present the results of this experiment. The bars in the graph are cumulative and are intended to show the relative differences between the three compression mechanisms. Thus, the XPack documents are approximately 75KB larger than the XMill

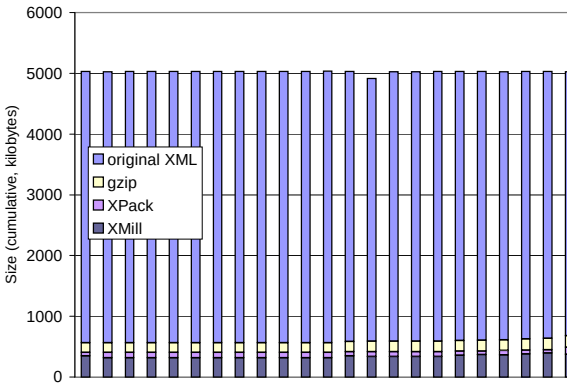


Figure 44: SwissProt compression relative to original size (cumulative sizes).

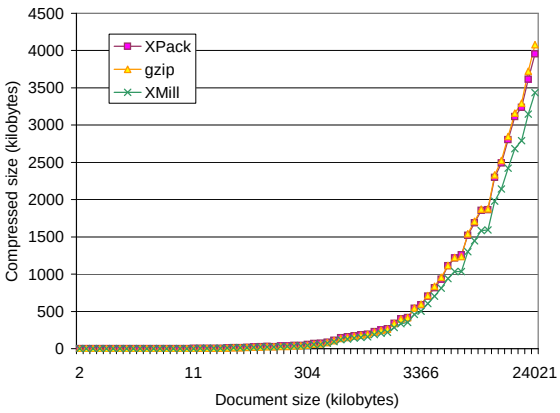


Figure 45: DBLP compression.

versions and the gzip files are approximately 150KB larger than XPack. The original files are all approximately the same size at about 5MB, yielding a difference in compression ratios between XMill and XPack of around 2%: XMill compresses the documents to 6% of original size while XPack compresses them to 8%. This slight difference in compression ratios is offset by XPack's ability to query the compressed documents, allowing applications to store documents efficiently without sacrificing utility of the information they contain.

Figure 45 presents the results of the final compression experiment that examines the compression results of the three compression systems over the DBLP data set. The DBLP data is highly structured but uses a small, terse tag dictionary, which minimizes the redundancy elimination gains that can be achieved with NamePack. There are also few attributes to reduce. As expected, the characteristics of the DBLP data provide little material for the space reduction techniques used by XPack and XMill, resulting in similar performance for all three systems.

4.4.4 Query Performance Experiments

The query performance experiments test the utility of the XPack system when performing data extraction operations over compressed documents. The XPack encoding is explicitly designed to support a wide variety of interesting query operations that operate over the document in its compressed form. These experiments demonstrate the wide range of

Table 7: Time to obtain node count and tag list, Shakespeare data set.

Node Count

Time (ms)	DOM	gzDOM	SAX	gzSAX	XPack
≤10	0	0	0	0	100%
10–500	0	0	0	0	0
501–750	0	0	100%	100%	0
>750	100%	100%	0	0	0

Tag List

≤ 10	0	0	0	0	5%
11–50	0	0	0	0	95%
51–750	0	0	100%	97%	0
>750	100%	100%	0	3%	0

query capabilities XPack documents support and show that the system achieves good query performance.

For all the query performance experiments, we report results utilizing XPack and DOM parsers to read in the document. For some of the tests, we also include measurements using a SAX XML parser. The test classes measure the time needed to load the document from disk, execute the query, and return the results. The startup time of the JVM and any time taken during cleanup and shutdown are not included.

We also provide query performance measurements using gzip compressed XML documents with both DOM and SAX parsers. The mechanism used to execute these tests is identical to the uncompressed counterparts except that the Java file input object is wrapped in a gzip decompression filter class before being passed to the XML parser, which treats it as any other input stream.

4.4.4.1 Document Calculation and Data Extraction

Table 7 presents the results of an experiment that measured the time needed to obtain a count of the nodes in the Shakespeare document data set. The table shows the percentage of documents for which the query results was obtained in the given time frame. For example, XPack retrieved the node count from 100% (37) of the Shakespeare documents in 10ms or less. The test is designed to highlight one of the important features of the XPack encoding system: compartmentalization. The system's design allows an application using

Table 8: Time to obtain node count and attribute list, Random walk data set.

Node Count

Time (ms)	DOM	gzDOM	SAX	gzSAX	XPack
≤ 10	0	0	0	0	1722
11–50	0	0	0	0	104
51–500	0	24	1584	1938	0
501–750	1733	1784	429	75	0
> 750	93	18	0	0	0

Attribute List

≤ 10	0	0	0	0	3
11–50	0	0	0	0	1446
51–100	0	0	0	0	307
101–500	0	0	1328	1625	3
501–750	1369	1557	431	134	0
> 750	390	202	0	0	0

XPack-encoded documents to access different facets of the document quickly and independently. This result demonstrates the time required to obtain a node count for various size documents; such information can be used, for example, in high-volume document dissemination services for prefiltering documents sent to mobile clients. Due to its separation of various document components, XPack is able to report the count several orders of magnitude faster than standard XML parsers. XPack reads the pertinent information for a query from the appropriate node container, while an XML parser must read and validate the entire document before it is able to produce a response.

Table 7 also reports the time needed to obtain a complete tag set from the documents in the Shakespeare data set. This result shows the benefit of compartmentalization by demonstrating the efficiency of the XPack document system. Since a complete tag set can be obtained from an XPack document very quickly, applications could use a tag set inclusion test as a filter for executing path queries. If a tag name appears in a path query but not in the target document, the query will not return any results and no further processing is needed. Like the node count results, XPack is able to process the tag list response very quickly by reading the appropriate container from the compressed document, which is much faster than processing an entire XML document, much of which is irrelevant to the query.

Table 9: XPath Evaluation Time, Shakespeare data set.

//PERSONA

Time (ms)	**DOM**	**gzDOM**	**XPack**
≤ 100	0	0	15
101–200	0	0	22
201–500	0	0	0
501–600	28	8	0
601–700	9	29	0

/PLAY/ACT/SCENE

≤ 100	0	0	15
101–200	0	0	22
201–500	0	0	0
501–600	29	5	0
601–700	8	32	0

Table 8 presents the results of a similar experiment conducted over the random walk data set. The tables show an account of the number of documents for which a node count and attribute list were obtained in the given time interval. Since the random walk data uses the HTML tag set, we elected to replace the tag set test from the previous experiment with an attribute list test that measures the time required to retrieve all attribute names and values from the document. Such an operation would be useful for extracting the hyperlink URLs from the documents, for example. The results in Table 8 demonstrate similar performance characteristics to those in Table 7 with XPack holding a commanding performance advantage over the other document types.

4.4.4.2 XPath Path Expressions

The next experiment evaluates the XPack system's ability to provide query support over compressed documents by testing the performance of XPath path expression queries on XPack documents. Path queries over XML documents using the XPath node selection language are a popular means for extracting data from XML documents. Table 9 presents the performance results obtained for two test queries over the Shakespeare data set and compares these results with the performance of the same queries over standard and compressed XML. The first query, "//PERSONA," selects all the nodes in the document with

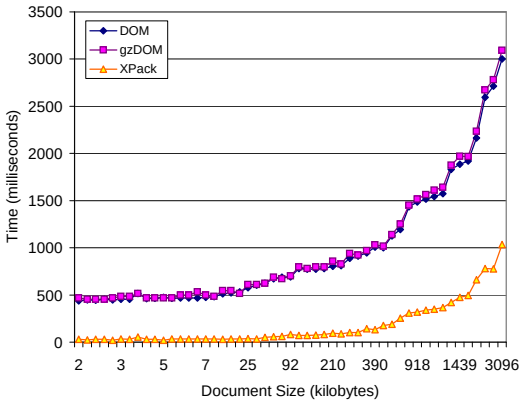


Figure 46: DBLP path query performance, query '//inproceedings/author', small files (original size <3MB).

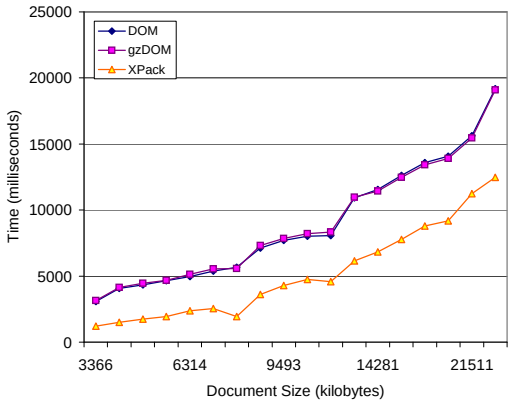


Figure 47: DBLP path query performance, query '//inproceedings/author', large files (original size >3MB).

the name "PERSONA" while the second selects all SCENE nodes with a parent ACT and a root grandparent PLAY. For each of the three document types used in the experiment—DOM, compressed DOM, and XPack—Table 9 lists the frequencies of query execution times obtained executing the query over each of the 37 documents in the Shakespeare data set. Note that these are "cold start" measurements: the time recorded for each test is the sum of the time needed to load the document from disk, parse it into an internal memory representation, and execute the query.

These results indicate that XPack can support efficient evaluation of path queries while simultaneously reducing the storage and materialization costs for XML documents. For many types of interesting queries, XPack's compartmentalization and separation of document components supports faster querying than the original documents: as shown in Table 9, document queries requiring more than 500 ms using a standard XML parsing and querying library can be executed in less than 200 ms with XPack. Because the queries in question require only the document structure and tag names to be satisfied, XPack can service the query without reading the entire document, which is an impossibility with XML.

Figures 46 and 47 show the results of a similar test performed over the DBLP data set. To enhance the clarity of the graphs, the results are split into two segments: the query performance over the documents with an original uncompressed size of less than 3MB is shown in Figure 46, while the performance for documents with an original uncompressed

Table 10: XPath Evaluation Time, SwissProt data set.

//Author

Time (ms)	DOM	gzDOM	XPack
≤ 750	0	0	26%
751–1000	0	0	74%
1001–4000	0	0	0
>4000	100%	100%	0

size of more than 3MB is shown in Figure 47. For both graphs, the test retrieved the author tags for each entry in the DBLP documents using the XPath query '//inproceedings/author'. These graphs demonstrate XPack's commanding performance advantage for executing XPath queries even for large XML documents.

Our final XPath experiment explores the query characteristics of XPack over the SwissProt data set; Table 10 presents the results. The query used in this experiment, '//Author', searches for the authors of the entries in the SwissProt data. XPack's compartmentalization of the compressed documents places all of the data relevant to this query at the beginning of the file where it can be accessed by the query processor or other applications quickly. XPack's binary encoding scheme creates another performance benefit for querying in that it avoids the expense of parsing text into memory objects, as with standard XML and DOM, by instead storing the document in a format that can be read directly into appropriate containers in memory. For this data set, the parsing overhead is significant as the files are all approximately 5MB, all of which must be read and parsed by the DOM implementation.

4.5 XPack Summary

The XPack document encoding and compression system achieves both good compression and strong query support for compressed documents. *Redundancy elimination* allows XPack to significantly reduce the size of Web documents, while *compartmentalization* separates the components of a document into logical containers. XPack's binary encoding scheme eliminates the expense of parsing XML text into memory objects by instead storing the document in a format that can be read directly into memory and immediately operated upon. XPack's compression performance compares favorably with other widely used XML

compression systems in testing with document sets having considerably different structural and content characteristics. XPack's document compartmentalization enables efficient document querying using XPack's aggregate document operators and the more general XPath query mechanism, which enjoys up to a 95% performance increase over queries using a standard XPath processor and text-based XML.

CHAPTER V

CONCLUSION

The dynamic Web represents a paradigm shift from an environment of static publication to one of dynamic data delivery that is efficient, timely, and well integrated. The ease with which data can be published and shared in the dynamic Web offers exciting opportunities for richer collaborations and discourse but requires a larger suite of integration and management technologies to be effective. While search engines became the "killer application" of the static Web, the techniques and assumptions underlying search engine technology no longer hold in the dynamic Web environment. Source discovery applications, service look up, and other core Web technologies must be adapted to this new environment using scalable techniques in order to tap the full potential of the dynamic Web.

5.1 Technical Contribution and Potential Impact

This dissertation research explored the problems associated with discovering and managing interesting services in the new environment of the dynamic Web. The DynaBot dynamic Web crawler addressed the new challenges involved with crawling the Deep Web using domain knowledge encoded in service class descriptions that specify the characteristics of an interesting source. The Page Digest Sentinel change monitoring system addressed the problem of data management in the context of a large-scale monitoring service. Finally, XPack expanded the ideas from the Page Digest system to provide better flexibility, data compression, and query support for applications using XML as their communication and archival format.

DynaBot. DynaBot is the first service-centric crawler for discovering and clustering Deep Web sources that uses a crawling approach rather than a centralized registry for discovering new services. Utilizing a crawling architecture allows DynaBot to discover

services that are not subscribed to a centralized registry along with sources that are mis-registered or inadequately described. DynaBot has three unique characteristics.

- DynaBot utilizes a service class model of the Web implemented through the construction of service class descriptions (SCDs).
- DynaBot employs a modular, self-tuning system architecture for focused crawling of the Deep Web. While many research and commercial crawlers utilize modular architectures, DynaBot applies modular crawling techniques to the Deep Web.
- DynaBot incorporates methods and algorithms for efficient probing of the Deep Web and for discovering and clustering Deep Web sources and services through SCD-based service matching analysis. This approach allows DynaBot to combine human knowledge contained in the service class description with source information gained from query probing to determine the relevance of a source to the service class.

DynaBot has been used to successfully recognize nucleotide BLAST Web services through probing analysis with a BLAST service class. The probing techniques and service class model should be applicable to other domains including medical abstract recognition, e-commerce, and Web news services. Further discussions of DynaBot, the service class model and approach, service class descriptions, and the automatic discovery problem can be found outside this dissertation in [78, 87, 67, 13, 86].

Page Digest Sentinels. The Page Digest Web document encoding is the first mechanism for representing Web documents that explicitly separates the individual characteristics of the document—such as content, structure, and attributes—into related containers. Separating the different facets of a document allows faster access to useful information, thereby enabling scalable and efficient access to information on the Web. The Page Digest document encoding format creates a foundation for an automatic Web change detection system, Page Digest Sentinels, that utilizes a scalable framework for monitoring Web information sources. This system also offers semantically rich data processing services that provide fine granularity

change detection with more expressive power than simple Boolean change tests. The design provides a framework for flexible and scalable Web change monitoring through the use of:

- *Efficient Data Management.* We encode Web documents in the Page Digest format, which encodes the structure and content of a document efficiently for fast load times and document operations.

- *Rich Processing Constructs.* Change monitoring requests operate over aspects of the document that the user has declared interesting, which provides more utility for users while supplying opportunities for processing optimizations.

- *Grouping.* Certain pages attract attention and large groups of interested users. Scalable systems must analyze and combine compatible monitoring requests to actively reduce computation, network usage, and local I/O.

The Page Digest document encoding and the Page Digest Sentinels Web change monitoring system are documented outside this dissertation in [16, 15, 85, 84].

XPack. XPack is the first queryable XML compression system that provides compression rates comparable to existing compression tools while retaining fast query support for compressed documents. XPack is general-purpose, does not rely on domain knowledge or particular document structural characteristics for compression, and achieves better query performance than standard query processors using text-based XML. XPack's main features are:

- *Redundancy elimination* allows XPack to significantly reduce the size of Web documents without requiring user-supplied semantic information or repetitive tree structures. XPack's careful management of a document's data eliminates much of the wasted space in XML documents and allows XPack to compress ordinary XML documents up to 20% better than other popular systems.

- *Compartmentalization* separates the components of a document into organized containers, untangling the XML document's intermixed structural and content features.

Compartmentalization allows XPack to access interesting components of compressed documents quickly and is a key contributor to XPack's query performance.

- *Binary Representation.* XPack's binary encoding scheme creates eliminates the expense of parsing XML text into memory objects by instead storing the document in a format that can be read directly into memory and immediately operated upon.

- *Compressed Data Access.* Unlike other widely used document compression systems, XPack provides general query facilities that operate over the compressed documents. Compressed data access allows applications to store data in a compact, space-saving format without sacrificing the ability to do ad hoc querying.

XPack's compression performance compares favorably with other widely used compression systems in testing with document sets having considerably different structural and content characteristics, while document compartmentalization and a binary file representation enable efficient document querying. XPack's characteristics make it ideal for use in general archival purposes, document dissemination and broadcast, and as a storage driver for storing XML efficiently in a DBMS.

5.2 Open Issues

DynaBot. The results of our experiments demonstrate the effectiveness of the service class model and the DynaBot discovery and matching agent. The results also suggest areas for further exploration to optimize the search and analysis process.

We are exploring the potential of dynamic learning techniques for reducing the resource consumption of the service analyzer by limiting the amount of effort it expends analyzing unlikely services. These techniques would perform one or more of the following functions: service filtering, maximum probe count adjustment, and query probe reordering.

- *Service filtering.* Using service filtering, the analyzer would evaluate the service based on its forms and eliminate it from consideration or reprioritize it if the service is unlikely to be a service class match.

- *Maximum probe count adjustment* would allow the service analyzer to dynamically adjust the number of queries attempted on a per-source basis using a comparison with previously encountered services.

- *Query probe reordering* would allow the service analyzer to dynamically reorder query probes like the static reordering described previously but using information gathered dynamically during the crawl.

The current DynaBot prototype includes one service filtering optimization, form filter, which eliminates any pages from consideration that either contain no form elements or whose form elements do not match the domain as defined by the service class description. For instance, a page containing a form with only list boxes and radio buttons would be eliminated from consideration if the service class description specified a free text input. This facility could be expanded to utilize information gathered from previous probing and matching operations. Using form similarity comparison, the service analyzer could measure the forms in a candidate service against previously matched sources and, based on this comparison, eliminate the service from consideration or adjust the probe count and order.

Another optimization that can be used to guide the analysis process is document text analysis. Many services contain technical jargon or other specialized vocabularies that distinguish these documents from those in other domains. Using techniques like the Levenshtein string edit distance [63] and term frequency analysis could help direct the crawler toward relevant hubs but might also be useful when performed on the start pages of services themselves. Much like form similarity comparison, these techniques could be used to adjust the service analysis properties based on the result of the comparison: services that are likely to match would be allocated more resources than those that are not.

There appears to be some synergy between DynaBot research and the Model Driven Architecture developed by the Object Management Group (OMG) [80]. In the Model Driven Architecture (MDA) [74], an abstract view of a system, called the *platform independent model* (PIM), can be used to construct a *platform specific model* (PSM) that combines the abstract PIM with specification details for interacting with a particular platform. The

goal of MDA is to provide a framework in which applications can be modeled in a platform-independent way and then converted automatically to a specific target platform. The PIM concept is similar to service classes and service class descriptions from the DynaBot architecture, while PSMs are analogous to DynaBot's service capability descriptions. Future DynaBot research will examine the MDA in more detail to determine potential benefits of applying MDA techniques to the service classification process.

Page Digest Sentinels. The Page Digest Sentinel system builds upon change detection and scalability research. There is ongoing work examining the problems of optimizing asynchronous change detection, computing change efficiently, and providing intuitive and productive user interfaces:

- *Asynchronous Optimization.* One open problem involves determining the optimal frequency with which to poll data sources for asynchronous change detection, i.e. automatically selecting a per-source poll frequency based upon programmatic evaluation of a source's change frequency. The Page Digest Sentinel concentrates on efficiently detecting changes in new versions of Web documents and leaves the choice of polling frequency to the user. There are several research efforts, most recently by Cho [26, 25] and Coffman [30], that address the problem of assigning limited sampling resources to minimize latency in asynchronous change detection. Intuitively, selection of an optimal poll frequency should minimize the resource consumption for a sentinel while still providing the user with timely updates.

- *Difference Generation Algorithms.* A minimal edit script between two versions of a document is a useful programmatic description of the changes to the documents. However, the typical primitive tree modification operators—insert, update, and delete of leaf nodes—provide highly detailed change descriptions that are an exact technical representation of the change but do not provide an intuitive feel for the changes that have occurred. When producing document comparisons for human consumption, an intuitive or high-level overview of the changes allows users to make better decisions about the data by minimizing the information overload often accompanied by highly

detailed change scripts. Grouping related changes together or generalizing change operations to the subtree level can produce more legible scripts, but there are other higher-level operations that may capture the meaning of the changes more accurately. For example, recent research has focused on copy, move, and glue operations [21] that attempt to make generated edit scripts more intuitive to the update operations that have been performed. Another interesting line of research opened by change detection systems involves improving how changes are represented to users. Although some systems provide a side-by-side presentation that highlights new and updated information [70], no formal study of change detection presentation techniques exists that we are aware of. While such innovations affect the user interface of a change detection system rather than the underlying algorithms, effective change presentation is an interesting problem in its own right that impacts the ability of Web users to manage information.

- *Extended Content Monitoring.* The Page Digest Sentinel system employs page-based change detection, operating over specific documents on the Web. A useful extension to this work involves creating sentinels that are content-based, monitoring interesting pieces of data without concern for precisely where the data is obtained. For example, a content-based system could monitor the stock price of a particular company without requiring a URI pointing to the data. Issues related to this type of monitoring include matching user requests with appropriate data providers, firing queries at multiple data source sites, and combining change detection results for presentation. Complications can also arise when resolving conflicts between information available from different sources or when triggering on aggregate information from multiple sources.

XPack. The XPack Web document compression system provides solid compression performance and excellent querying capabilities. Some open problems with respect to XPack include enhancing the system's compression performance to achieve even greater compression rates, possibly utilizing semantic information about the data being compressed. There are also opportunities in several application areas, including the DynaBot service analysis

and classification system, that can benefit from improve document handling performance and the ability to compare documents in rich and powerful ways.

- *Semantic Data Compression.* Many XML-specific compression systems rely on semantic information to enhance compression performance or as the basis for their compression algorithms. While XPack does not rely on semantic information, many XML technologies, including validating parsers, have been built with the expectation that semantic metadata will be available in the form of a DTD, XML Schema, or other semantic definition. There has also been work on inferring semantic type information from XML data files that do not have an associated DTD [76]. Incorporating semantic information into the XPack compression system could yield better compression and possibly superior performance as well.

- *Application Performance Optimization.* Present work on XPack demonstrates excellent query performance and data compression on documents converted from their "native" XML format. However, there are many applications for which generating XML is a significant concern, e.g. XML RPC using SOAP and dynamic XHTML content creation for Deep Web sources. XPack should be able to improve the performance for these and other applications that rely on efficient document creation and transmission.

- *Document Comparison and Similarity.* The Page Digest Sentinel change monitoring system employs techniques for comparing different versions of a document, and ongoing research with DynaBot includes query probing strategies that leverage document comparison for more effective source analysis. Future work for XPack includes migrating document comparison ideas found in our Page Digest research and incorporating these comparison techniques into the DynaBot query probing infrastructure. XPack's document compartmentalization provides a foundation for efficient and meaningful comparisons between documents: documents can be compared along many axes including structure, tag set, attributes, content, or combinations. Some of the research challenges include defining useful structural comparison tests and efficient, relevant

document summarization.

APPENDIX A

EXAMPLE SERVICE CLASS DESCRIPTION

The following service class description was used in our experimental evaluation and describes the class of nucleotide BLAST homolog search services.

```
<?xml version="1.0"?> <!-- -*- Mode: XML; -*- -->

<!-- $Id: alignment.scd,v 1.4 2003/10/09 19:07:52 rockdj Exp $ -->

<serviceclass>

  <!-- Types used by this service class description.  The types -->
  <!-- "string," "integer," and "whitespace" are predefined by the -->
  <!-- system.  Types can either be simple restrictions on the base -->
  <!-- types or compositions of previously defined types. -->
  <types>

  <!-- DNA Sequence -->
  <type name="DNASequence" type="string" pattern="[GCATgcat-]{10,}" />

  <!-- An AlignmentSequence is a string of the form: -->
  <!--  <name>: <m> <sequence> <n>  -->
    <type name="AlignmentSequence" >
      <element name="AlignmentName" type="string" pattern=".{1,100}:" />
      <element type="whitespace" />
      <element name="m" type="integer" />
      <element type="whitespace" />
      <element name="Sequence" type="DNASequence" />
      <element type="whitespace" />
      <element name="n" type="integer" />
    </type>

  <!-- The AlignmentString represents the separator characters -->
  <!-- between AlignmentSequences -->
    <type name="AlignmentString" type="string" pattern="\s+\|+[| ]*" />

  <!-- An AlignmentLine is two AlignmentSequences separated by an -->
  <!-- AlignmentString, which shows what nucleotides match in the -->
  <!-- two sequences. -->
    <type name="AlignmentLine">
      <element name="QueryString" type="AlignmentSequence" required="true" />
      <element type="string" pattern="[\n\r]{1,2}" />
      <element name="Alignments" type="AlignmentString" required="true" />
      <element type="string" pattern="[\n\r]{1,2}" />
```

```
    <element name="SequenceString" type="AlignmentSequence" required="true"
     />
  </type>

  <type name="BlockElement">
    <element type="AlignmentLine" required="true" />
    <element type="whitespace" />
  </type>

  <!-- The AlignmentBlock is a group of AlignmentLines that is  -->
  <!-- logically connected. -->
  <type name="AlignmentBlock">
    <element type="BlockElement" minOccurs="1" maxOccurs="unbounded" />
  </type>

  <type name="Hyperlink">
    <element type="string" pattern="(HREF|href)\s*=\s*[\x22\x27]" />
    <element name="Target" type="string" pattern="[^\x22\x27]+" />
    <element type="string" pattern="[\x22\x27]" />
  </type>

  <type name="Alignment">
    <element name="geneLink" type="Hyperlink" compile="false"/>
    <element name="score" type="string"
             pattern="[Ss]core\s*=\s*\d+[^,;\n \t]*[,;\n \t]"
             compile="false"/>
    <element name="Evalue" type="string"
             pattern="Expect\s*=\s*\d*(e-)?\d+\.?\d*[^,;\n \t]*[,;\n \t]"
             compile="false"/>

    <element type="AlignmentBlock" minOccurs="1" maxOccurs="unbounded" />
  </type>

  <type name="Alignments">
  <element name="Alignment" type="Alignment"
           minOccurs="1" maxOccurs="unbounded" />
  </type>

  <type name="EmptyDNABLAST">
    <element type="string" pattern=".*" />
    <element name="Message" type="string"
           pattern="\s+\*{3,}\s*(NONE|No\s+[Hh]its\s+.*|[Nn]one)\s*\*{3,}\s+"
           />
    <element type="string" pattern=".*" />
  </type>

  <type name="SummaryPage">
    <choice>
      <element type="Alignments" />
      <element type="EmptyDNABLAST" />
    </choice>
  </type>

</types>
```

```
  <!-- The control flow graph specifies the navigational portion -->
  <!-- of the site, describing how a service class memeber might -->
  <!-- proceed from one page to the next. -->
  <controlflow name="BLASTN">
    <vertices>
      <vertex name="start" type="HTMLform" origin="true" />

      <!-- type == match type, resultType==desired objects -->
      <vertex name="summary" type="SummaryPage"
              result="true" resultType="Alignment" />
    </vertices>
    <edges>
      <edge origin="start" destination="summary" />
    </edges>
  </controlflow>

  <!-- The example queries are used by the processor during evaluation -->
  <!-- of a site.  They include input parameters and expected results. -->
  <!-- Input parameters can also include hints that help find correct -->
  <!-- form parameters faster. -->
  <examples>
    <example>
      <arguments>

        <argument required="true">
        <name>sequence</name>
        <type>DNASequence</type>
        <hints>
          <hint>[Ss]equence|SEQUENCE</hint>
          <hint>query_data|QUERY_DATA</hint>
          <hint>[Qq]uery|QUERY</hint>
          <hint>q|Q</hint>

          <inputType>textarea|TEXTAREA|text</inputType>
        </hints>
        <value>CTGCATCAGTAATTTTAATAGAGACTCTTAGTGGTTATCAACAACTGG
TCAGTTGTTTTTTTGTTTTTTTTTTTTTCCACACTGCTCTCTGGATTCGAACCTAGTGAATTC
TGGGCTCAGTCCTCCTCCTCCATCTCTTTCTTGATCCAGTCCACGTAGTTGAGCACTTTG
GTGTAGAAGCCATAGCCCCTGCTGCACCCGATGCCCCAGGACACGATGCCCGTGGCCACC
CAGCGATCAGTGTTCGGGTCCCTTACTGGAAAAACGCCCCCACTATCCCCCTGGCAGGCG
TCCTGCTTTAAAGATGGGTGGCCCGCCACAAACATGTTTTGGGAAAACACAACCATTCTA</value>
      </argument>

      <argument required="false">
        <name>email</name>
        <type>EmailAddress</type>
        <hints>
          <hint>email</hint>
          <hint>e-mail</hint>
        </hints>
        <value>example@hotmail.com</value>
      </argument>
```

```
    <argument required="false" >
      <name>BlastProgram</name>
      <hints>
        <hint>[Pp]rogram|PROGRAM</hint>
        <hint>[Pp]rog|PROG</hint>
      </hints>
      <value>blastn</value>
    </argument>

    <argument required="false" iterator="true">
      <name>database</name>
      <type>string</type>
      <hints>
        <hint>[Dd]atabase|DATABASE</hint>
        <hint>[Dd]b|DB</hint>
      </hints>
      <value>.*[^\W]+.*</value>
    </argument>
  </arguments>
  <result type="SummaryPage" />

</example>

<example>
  <arguments>

    <argument required="true" user="true">
      <name>sequence</name>
      <type>DNASequence</type>
      <hints>
        <hint>[Ss]equence|SEQUENCE</hint>
        <hint>query_data|QUERY_DATA</hint>
        <hint>[Qq]uery|QUERY</hint>
        <hint>q|Q</hint>
        <inputType>textarea|text</inputType>
      </hints>

      <!-- part of the sequence on http://166.111.30.65/BlastQuery.html -->
      <value>AACTAATTGCCTCACATTGTCACTGCAAATCGACACCTA
             TTAATGGGTCTCACCTCCCAACTGCTTCCCCCTCTGTTCTTCCTGCTAGCATGTGCCGG
      </value>
    </argument>

    <argument required="false">
      <name>email</name>
      <type>EmailAddress</type>
        <hints>
          <hint>email</hint>
          <hint>e-mail</hint>
        </hints>
      <value>example@hotmail.com</value>
    </argument>

    <argument required="false">
```

```
        <name>BlastProgram</name>
        <hints>
          <hint>[Pp]rogram|PROGRAM</hint>
          <hint>[Pp]rog|PROG</hint>
        </hints>
        <value>.*[Bb]lastn.*</value>
      </argument>

      <argument required="false" iterator="true">
        <name>database</name>
        <type>string</type>
        <hints>
          <hint>[Dd]atabase|DATABASE</hint>
          <hint>[Dd]b|DB</hint>
        </hints>
        <value>.*[^\W]+.*</value>
      </argument>
    </arguments>

    <result type="SummaryPage" />
  </example>
 </examples>

</serviceclass>
```

APPENDIX B

CRAWLER SEED LISTS AND NUCLEOTIDE BLAST SERVICES

B.1 Manually Collected Nucleotide BLAST Services

http://arep.med.harvard.edu/cgi-bin/dpinteract/blastn
http://genopole.toulouse.inra.fr/blast/blast.html
http://www.genome.ou.edu/strep_blast.html
http://www.genome.ou.edu/nc_per_blast.html
http://genome3.cpmc.columbia.edu/~legion/int_blast.html
http://odin.mdacc.tmc.edu/RetinalExpress/est_search.htm
http://www.genome.ou.edu/flavus_blast.html
http://www.genome.ou.edu/gono_blast.html
http://bacillus.genome.ad.jp/BSORF_homology_genes.html
http://www.genome.ou.edu/nc_both_blast.html
http://hordeum.ipk-gatersleben.de/blast/
http://seasquirt.mbio.co.jp/icb/blast/blastsearch.php
http://phage.bioc.tulane.edu/blast/blast.html
http://phage.bioc.tulane.edu/blast/blast723.html
http://mammoth.bii.a-star.edu.sg/blast/blastn.html
http://www.bindingdb.org/bind/blast/BySequence.jsp
http://microarrays.ucsd.edu/blast/biogem/human_megablast.html
http://fugu.hgmp.mrc.ac.uk/blast/
http://www.genome.ad.jp/kegg-bin/SearchGenes?blast
http://poppel.fysbot.umu.se/blastsearch.html
http://www.genome.ad.jp/kegg-bin/SearchGenes?blast+genome
http://tools.neb.com/wolbachia/search.html
http://genopole.toulouse.inra.fr/blast/megablast.html
http://kinase.com/blast/blast.html
http://genome.gen-info.osaka-u.ac.jp/bacteria/o157/blast.html
http://danio.mgh.harvard.edu/blast/blast_grp.html
http://genome.gen-info.osaka-u.ac.jp/bacteria/vpara/blast.html
http://blast.ims.u-tokyo.ac.jp/blast_gb.html
http://blast.mpi-bremen.de/blast/
http://salmonella.utmem.edu/blast.html
http://www.microbial-pathogenesis.org/site/blast.php
http://www.chromdb.org/blast2.html
http://www.flu.lanl.gov/blast/
http://www.sanguis.mic.vcu.edu/blast.html
http://blast.ims.u-tokyo.ac.jp/blast.html
http://pgrc.ipk-gatersleben.de/blast/blast_server.php
http://cneo.genetics.duke.edu/blast.html
http://bighost.area.ba.cnr.it/BIG/Blast/BlastUTR.html
http://xenopus.nibb.ac.jp/blast/blast.html
http://plpa2linux.tamu.edu/blastIFAFS.html
http://tigrblast.tigr.org/euk-blast/index.cgi?project=osa1
http://nasc.nott.ac.uk/blast.html
http://blast.ims.u-tokyo.ac.jp/blast_ref.html
http://artedi.ebc.uu.se/Projects/Francisella/blast/
http://artedi.ebc.uu.se/Projects/Buchnera/blast/
http://wheat.pw.usda.gov/wEST/blast/
http://www.streppneumoniae.com/blast_search.asp
http://164.41.88.103/blast/blast.html
http://genome.wustl.edu/blast/client.pl
http://crobar.med.harvard.edu/blast/blast_cs.html
http://dicty.sdsc.edu/
http://129.237.147.182/blast/blast_cs.html
http://gib.genes.nig.ac.jp/cmprtv/blast2/main.php
http://woody.embl-heidelberg.de/gene2est/
http://www.bioinformatica.ucb.br/blast.html
http://138.23.191.152/blast/blast.html
http://blast.ym.edu.tw/blast/blast_cs.html
http://bioinfo.nhri.org.tw/blast/blast.html
http://www.plantgdb.org/cgi-bin/PlantGDBblast
http://www.rtc.riken.go.jp/jouhou/trsig/trsig_blast.html
http://www.ebi.ac.uk/blast2/parasites.html
http://jic-bioinfo.bbsrc.ac.uk/cgi-bin/ace/blast_ssr/millet
http://helix.genes.nig.ac.jp/camus/blast.shtml
http://mips.gsf.de/proj/thal/db/search/blast_arabi.html
http://search.usricegenome.org/
http://tigrblast.tigr.org/ufmg/

http://www.sanger.ac.uk/Projects/C_elegans/blast_server.shtml
http://www.sanger.ac.uk/Projects/S_coelicolor/blast_server.shtml
http://www.sanger.ac.uk/Projects/M_leprae/blast_server.shtml
http://www.sanger.ac.uk/Projects/P_falciparum/blast_server.shtml
http://www.sanger.ac.uk/Projects/S_pombe/blast_server.shtml
http://www.sanger.ac.uk/Projects/N_meningitidis/blast_server.shtml
http://www.sanger.ac.uk/Projects/M_tuberculosis/blast_server.shtml
http://www.sanger.ac.uk/Projects/S_cerevisiae/

B.2 Automatically Discovered and Classified Nucleotide BLAST Services

The following 12 URLs point to nucleotide BLAST services on the Web that were discovered and classified automatically during the DynaBot crawl on 6/2/2004. This crawl utilized the static LinkHint frontier that gave priority to the URLs containing the keyword "blast" in the Google 500 BLAST seed list. DynaBot examined 182 URLs and found 137 forms with 1038 parameters. The service analyzer attempted an average of 24.38 queries per form and took 63.22 minutes to complete the classification of the URLs.

http://genome-www2.stanford.edu/cgi-bin/SGD/nph-blast2sgd
http://sbcr.bii.a-star.edu.sg/BLAST/blastx_html
http://fugu.hgmp.mrc.ac.uk/blast/
http://sbcr.bii.a-star.edu.sg/BLAST/blastn_html
http://mouseblast.informatics.jax.org/prototype/
http://www.sanguis.mic.vcu.edu/blast.html
http://138.23.191.152/blast/blast.html
http://xylian.igh.cnrs.fr/blast/www_blast.html
http://nasc.nott.ac.uk/blast.html
http://www.ncbi.nlm.nih.gov/igblast/
http://crobar.med.harvard.edu/blast/megablast.html
http://sbcr.bii.a-star.edu.sg/BLAST/tblastn_html

The following 3 URLs are protein BLAST services that were misclassified by the DynaBot crawler during the 6/2/2004 crawl. These services were recognized due to the similarity between their "no result" page and that of the nucleotide BLAST servers. These sources could easily be handled in a wrapper mediator system since they would fail to produce matching results. Future work on dynamic learning of source characteristics will also help alleviate these few misclassifications.

http://sbcr.bii.a-star.edu.sg/BLAST/blastp_html
http://hits.isb-sib.ch/cgi-bin/hits_blast
http://www.binddb.org/blast/

B.3 Google BLAST Seed List

List available at *http://disl.cc.gatech.edu/DynaBot/google_seed.html*

APPENDIX C

CODE INFORMATION

C.1 DynaBot

The following statistics were collected from the 8/4/2004 snapshot of the DynaBot code base. More information about DynaBot and the Lawrence Livermore National Laboratory data integration project can be found at the DynaBot project page:

http://disl.cc.gatech.edu/DynaBot/

DynaBot **Driver**

		Source Code Lines		
Filename	Size (bytes)	File	Code	Comment
Driver.java	14756	524	245	184

Crawler Core

		Source Code Lines		
Filename	Size (bytes)	File	Code	Comment
Callback.java	2658	107	74	11
Crawler.java	10201	407	238	93
DNSResolver.java	329	20	7	9
DocumentFetcherInterface.java	344	23	7	11
DocumentProcessorInterface.java	352	21	6	11
DoubleFailLog.java	2368	98	42	32
FailInterface.java	682	39	7	25
FormInterface.java	764	45	8	28
FrontierInterface.java	1203	61	12	37
HeadTailPSF.java	6096	245	102	88
HTTPDocument.java	3112	173	72	71
LocalizedHTPSF.java	1516	65	36	12
LookupCachePSF.java	3597	129	52	53
PersistentStringFile.java	6009	265	136	69
PriorityUrlPSF.java	2956	116	49	40
UnsearchedUrls.java	10002	392	221	87
UnsearchedUrlsGZ.java	1376	53	34	9
URLInfo.java	5622	239	81	123
VisitedInterface.java	771	44	8	28
TOTAL	**59958**	**2542**	**1192**	**837**

Processing Modules

		Source Code Lines		
Filename	Size (bytes)	File	Code	Comment
ApacheDocumentFetcher.java	4327	163	98	35
DummyVisited.java	1178	61	19	30
HashVisited.java	1441	81	27	40
HTTPUnitDocumentFetcher.java	4354	161	99	35
JavaDNSResolver.java	589	31	10	14
LinkExtractor.java	1851	83	40	28
RandomWalkFrontier.java	2825	140	65	50
ServiceClassAnalyzer.java	11306	427	175	175
TraceFrontier.java	2651	132	62	50
TraceGenerator.java	2170	102	59	27
TOTAL	**32692**	**1381**	**654**	**484**

Service Form Handling

		Source Code Lines		
Filename	Size (bytes)	File	Code	Comment
FormParameter.java	9795	392	218	89
QueryArgument.java	10764	381	166	152
QueryEnumeration.java	13673	407	239	87
WebQuery.java	20669	645	328	190
TOTAL	**54901**	**1825**	**951**	**518**

Service Capability Script Generator

		Source Code Lines		
Filename	Size (bytes)	File	Code	Comment
HTTPFormQuery.java	6682	219	139	40
ScriptGenerator.java	8887	272	163	62
SourceProber.java	7387	191	129	27
XWrapElite.java	6426	180	116	28
TOTAL	**29382**	**862**	**547**	**157**

Service Class Description Handling

		Source Code Lines		
Filename	Size (bytes)	File	Code	Comment
ControlFlow.java	4143	156	68	62
Example.java	3603	120	56	38
MalformedServiceClassException.java	838	32	10	15
ServiceClass.java	9212	276	91	132
Vertex.java	2587	131	43	69
TOTAL	**20383**	**715**	**268**	**316**

Utilities

		Source Code Lines		
Filename	Size (bytes)	File	Code	Comment
CombinationEnumeration.java	4998	179	71	71
DBList.java	7653	346	209	55
DOMUtil.java	5501	206	135	24
EmptyNodeList.java	763	32	13	14
URLUtils.java	6802	260	138	77
Util.java	3585	152	91	30
TOTAL	**29302**	**1175**	**657**	**271**

Type Library

		Source Code Lines		
Filename	Size (bytes)	File	Code	Comment
Choice.java	6622	245	150	42
DatatypeException.java	239	13	3	7
DictionaryType.java	1988	76	47	20
Instance.java	4384	217	57	113
Namespace.java	5026	185	76	77
Type.java	26677	883	473	258
TypeMutationException.java	820	36	14	15
TypeRealizer.java	5024	165	66	57
TOTAL	**50780**	**1820**	**886**	**589**

C.2 XPack

The following statistics were collected from the 8/4/2004 snapshot of the XPack code base. More information about the XPack project can be found at the XPack project page:

http://disl.cc.gatech.edu/xpack/

XPack Core

		Source Code Lines		
Filename	Size (bytes)	File	Code	Comment
AttributeNameValueEncoding.java	1800	72	45	12
DeltaEncodedStringArray.java	4032	174	124	14
XMLPack.java	20999	770	481	155
XPack.java	11260	439	275	85
XPackIOConstants.java	955	42	21	9
XPackNode.java	6480	312	187	61
XPackNodeBase.java	656	44	10	26
XPackNodeBinary.java	4628	217	145	37
XPackNodes.java	4010	174	80	64
XPackReader.java	4427	195	144	34
XPackWriter.java	2337	103	69	23
XPathProcessing.java	4916	188	129	28
TOTAL	**66500**	**2730**	**1710**	**548**

XPack Test Suite

		Source Code Lines		
Filename	Size (bytes)	File	Code	Comment
Test.java	2364	71	21	40
TestTest.java	1096	45	13	25
XMLAttributeListDOM.java	3147	122	72	30
XMLAttributeListSAX.java	3103	108	66	26
XMLNodeCountDOM.java	3148	110	64	30
XMLNodeCountSAX.java	3490	112	60	37
XMLNonexistentTagSearch.java	1801	72	38	26
XMLTagListDOM.java	3200	111	64	30
XMLTagListSAX.java	3218	99	57	29
XMLXPathDOM.java	2723	104	60	26
XPackAttributeList.java	2167	87	46	26
XPackNodeCount.java	1472	57	23	25
XPackNonexistentTagSearch.java	1910	69	32	26
XPackTagList.java	1722	65	28	26
XPackXPath.java	1417	62	25	26
TOTAL	**35978**	**1294**	**669**	**428**

REFERENCES

[1] Altinel, M. and Franklin, M. J., "Efficient filtering of xml documents for selective dissemination of information," in *Proceedings of the 26th International Conference on Very Large Databases (VLDB '00)*, 2000.

[2] Ask Jeeves, Inc., "Ask jeeves search engine." http://www.ask.com/.

[3] Ask Jeeves, Inc., "Teoma search engine." http://www.teoma.com/.

[4] Barnard, D. T., Clarke, G., and Duncan, N., "Tree-to-tree correction for document trees," Tech. Rep. 95-372, Department of Computing and Information Science, Queen's University, Kingston, January 1995.

[5] Bergman, M. K., "The Deep Web: Surfacing Hidden Value." http://www.completeplanet.com/Tutorials/DeepWeb/, 2003.

[6] Boldt, A., "Cyclic redundancy check." http://www.wikipedia.org/CRC, 2002.

[7] Boyapati, V., Chevrier, K., Finkel, A., Glance, N., Pierce, T., Stokton, R., and Whitmer, C., "ChangeDetector^{TM}: A Site-Level Monitoring Tool for the WWW," in *WWW2002*, May 2002.

[8] "Extensible markup language (XML) 1.0," tech. rep., W3C, 1998.

[9] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S., "Web caching and Zipf-like distributions: Evidence and implications," *Proceedings of IEEE Infocom '99*, pp. 126–134, March 1999.

[10] Brin, S. and Page, L., "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1–7, pp. 107–117, 1998.

[11] Broder, A., Najork, M., and Weiner, J., "efficient url caching for worldwide web crawling," in *Proceedings of the International World Wide Web Conference*, 2003.

[12] Buneman, P., Grohe, M., and Koch, C., "Path queries on compressed xml," in *Proceedings of the 29th International Conference on Very Large Databases (VLDB '03)*, 2003.

[13] Buttler, D., Coleman, M., Critchlow, T., Fileto, R., Han, W., Pu, C., Rocco, D., and Xiong, L., "Querying multiple bioinformatics information sources: Can semantic web research help?," *SIGMOD Record*, vol. 31, no. 4, 2002.

[14] Buttler, D., Liu, L., and Pu, C., "A fully automated object extraction system for the world wide web," *Proceedings of IEEE International Conference on Distributed Computing Systems*, April 2001.

[15] Buttler, D., Liu, L., and Rocco, D., "Efficient processing of web page sentinels using page digest," tech. rep., Georgia Institute of Technology, May 2003.

[16] Buttler, D., Rocco, D., and Liu, L., "Efficient web change monitoring with page digest," *13th Annual International World Wide Web Conference WWW2004 (poster symposium)*, May 2004.

[17] Callan, J. P., Lu, Z., and Croft, W. B., "Searching Distributed Collections with Inference Networks ," in *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Fox, E. A., Ingwersen, P., and Fidel, R., eds.), (Seattle, Washington), pp. 21–28, ACM Press, 1995.

[18] Caverlee, J., Liu, L., and Rocco, D., "Discovering and ranking data intensive web services: A source-biased approach," Tech. Rep. GIT-CERCS-03-26, Georgia Institute of Technology CERCS, 2003.

[19] Chakrabari, S., van den Berg, M., and Dom, B., "Focused crawling: A new approach to topic-specific web resource discovery," in *Proceedings of the Eighth International World Wide Web Conference*, 1999.

[20] Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J. D., and Widom, J., "The TSIMMIS project: Integration of heterogeneous information sources," in *16th Meeting of the Information Processing Society of Japan*, (Tokyo, Japan), pp. 7–18, 1994.

[21] Chawathe, S. S. and Garcia-Molina, H., "Meaningful change detection in structured data," in *Proceedings of the 1997 ACM SIGMOD*, pp. 26–37, 1997.

[22] Chen, J., DeWitt, D., Tian, F., and Wang, Y., "NiagaraCQ: A scalable continuous query system for internet databases," in *Proceedings of the 2000 ACM SIGMOD*, 2000.

[23] Chen, Y.-F., Douglis, F., Huan, H., and Vo, K.-P., "TopBlend: An Efficient Implementation of HtmlDiff in Java," in *Proceedings of the WebNet2000 Conference*, (San Antonio, TX), November 2000.

[24] Cheney, J., "Compressing XML with multiplexed hierarchical PPM models," in *Data Compression Conference*, 2001.

[25] Cho, J. and Garcia-Molina, H., "Estimating frequency of change," tech. rep., DB Group, Stanford University, November 2001.

[26] Cho, J. and Ntoulas, A., "Effective change detection using sampling," in *Proceedings of 28th International Conference on Very Large Databases (VLDB)*, September 2002.

[27] Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S., "Web services description language (WSDL) 1.1," tech. rep., W3C, 2001.

[28] Cobena, G., Abiteboul, S., and Marian, A., "Detecting changes in xml documents," in *International Conference on Data Engineering*, pp. 41 –52, 2002.

[29] Cody, W. F., Haas, L. M., Niblack, W., Arya, M., Carey, M. J., Fagin, R., Flickner, M., Lee, D., Petkovic, D., Schwarz, P. M., II, J. T., Roth, M. T.,

Williams, J. H., and Wimmers, E. L., "Querying multimedia data from multiple repositories by content: the garlic project," in *VDB*, pp. 17–35, 1995.

[30] Coffman, E. G., Liu, Z., and Weber, R. R., "Optimal robot scheduling for web search engines," *Journal of Scheduling*, vol. 1, pp. 15–29, 1998.

[31] Consortium, W. W. W., "Wap binary xml content format," June 1999.

[32] Craswell, N., Bailey, P., and Hawking, D., "Server selection on the World Wide Web," in *Proceedings of the Fifth ACM Conference on Digital Libraries*, pp. 37–46, 2000.

[33] Davidson, S. B., Overton, G. C., Tannen, V., and Wong, L., "BioKleisli: A digital library for biomedical researchers," *Int. J. on Digital Libraries*, vol. 1, no. 1, pp. 36–53, 1997.

[34] Dobbertin, H., Bosselaers, A., and Preneel, B., *RIPEMD-160: A Strengthened Version of RIPEMD*. 1996.

[35] Doorenbos, R. B., Etzioni, O., and Weld, D. S., "A scalable comparison-shopping agent for the world-wide web," in *Proceedings of the First International Conference on Autonomous Agents (Agents'97)* (Johnson, W. L. and Hayes-Roth, B., eds.), (Marina del Rey, CA, USA), pp. 39–48, ACM Press, 1997.

[36] Douglis, F., Ball, T., Chen, Y.-F., and Koutsofios, E., "The AT&T internet difference engine: Tracking and viewing changes on the web," in *World Wide Web*, vol. 1, pp. 27–44, January 1998.

[37] Eckman, B., Lacroix, Z., and Raschid, L., "Optimized seamless integration of biomolecular data," in *IEEE International Conference on Bioinformatics and Biomedical Egineering*, pp. 23–32, 2001.

[38] Fallside, D. C., "XML Schema Part 0: Primer," tech. rep., World Wide Web Consortium, http://www.w3.org/TR/xmlschema-0/, 2001.

[39] FIPS 180-1, "Secure Hash Standard," tech. rep., U.S. Department of Commerce/N.I.S.T., National Technical Information Service, April 1995.

[40] Freitag, D. and Kushmerick, N., "Boosted wrapper induction," in *AAAI/IAAI*, pp. 577–583, 2000.

[41] Gathma, S. D., "Java GNU Diff." http://www.bmsi.com/java/#diff, 2002.

[42] Girardot, M. and Sundaresan, N., "Millau: an encoding format for efficient representation and exchange of xml over the web," in *Proceedings of the Ninth International World Wide Web Conference (WWW 2000)*, May 2000.

[43] Gold, R., "HttpUnit." http://httpunit.sourceforge.net, 2003.

[44] Google, Inc., "Google answers frequently asked questions." http://answers.google.com/answers/faq.html.

[45] Google, Inc., "Google directory frequently asked questions." http://www.google.com/dirhelp.html.

[46] Google, Inc., "Google groups frequently asked questions." http://www.google.com/googlegroups/help.html.

[47] Google, Inc., "Google local frequently asked questions." http://local.google.com/help/faq_local.html.

[48] Gottschalk, K., Graham, S., Kreger, H., and Snell, J., "Introduction to web services architecture," *IBM Systems Journal*, vol. 41, no. 2, pp. 170–177, 2002.

[49] Gravano, L., Ipeirotis, P., and Sahami, M., "QProber: A system for automatic classification of hidden-web databases," *ACM Transactions on Information Systems*, vol. 21, January 2003.

[50] Gravano, L., García-Molina, H., and Tomasic, A., "GlOSS: text-source discovery over the Internet," *ACM Transactions on Database Systems*, vol. 24, no. 2, pp. 229–264, 1999.

[51] Haas, L., Schwarz, P., Kodali, P., Kotlar, E., Rice, J., and Swope, W., "Discoverylink: A system for integrating life sciences data," *IBM Systems Journal*, vol. 40, no. 2, 2001.

[52] Haertel, M., Hayes, D., Tamllman, R., Tower, L., Eggert, P., and Davison, W., "The gnu diff program." Texinfo system documentation, Available by anonymous ftp at `prep.ai.mit.edu`.

[53] Hamming, R., "Error detecting and error correcting codes," *Bell System Tech. J.*, vol. 29, pp. 147–160, April 1950.

[54] Heydon, A. and Najork, M., "Mercator: A scalable, extensible web crawler," *World Wide Web*, vol. 2, no. 4, pp. 219–229, 1999.

[55] IBM Corporation, "IBM XML TreeDiff." www.alphaworks.ibm.com/formula/xmltreediff, 2002.

[56] Ipeirotis, P. and Gravano, L., "Distributed search over the hidden web: Hierarchical database sampling and selection," tech. rep., Columbia University, Computer Science Department, 2002.

[57] KartOO, Inc., "KartOO search engine." http://www.kartoo.com/.

[58] Kushmerick, N., "Wrapper induction: Efficiency and expressiveness," *Artificial Intelligence*, vol. 118, no. 1-2, pp. 15–68, 2000.

[59] Kushmerick, N., Weld, D. S., and Doorenbos, R. B., "Wrapper induction for information extraction," in *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pp. 729–737, 1997.

[60] Latifur Khan, Lei Wang, Y. R., "Change detection of xml documents using signatures," *WWW2002, Workshop on Real World RDF and Semantic Web Applications*, May 2002.

[61] Lawrence, S. and Giles, C. L., "Searching the world wide web," *Science*, vol. 280, no. 5360, p. 98, 1998.

[62] Lawrence, S. and Giles, C. L., "Accessibility of information on the web," *Nature*, vol. 400, pp. 107–109, 1999.

[63] Levenshtein, V. I., "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, pp. 707–710, 1966.

[64] Levy, A. Y., Rajaraman, A., and Ordille, J. J., "Querying heterogeneous information sources using source descriptions," in *Proceedings of the Twenty-second International Conference on Very Large Databases*, (Bombay, India), pp. 251–262, VLDB Endowment, Saratoga, Calif., 1996.

[65] Liefke, H. and Suciu, D., "XMill: an efficient compressor for XML data," in *ACM International Conference on Management of Data (SIGMOD)*, pp. 153–164, 2000.

[66] Liu, K., Yu, C., Meng, W., Wu, W., and Rishe, N., "A statistical method for estimating the usefulness of text databases," 1998.

[67] Liu, L., Buttler, D., Critchlow, T., Han, W., Paques, H., Pu, C., and Rocco, D., "Biozoom: Exploiting source-capability information for integrated access to multiple bioinformatics data sources," in *In Proc. of 3rd IEEE Symposium on Bioinfomatics and Bioengineering*, 2003.

[68] Liu, L., Pu, C., and Tang, W., "WebCQ: Detecting and Delivering Information Changes on the Web," *Proceedings of the International Conference on Information and Knowledge Management*, November 2000.

[69] Liu, L., Pu, C., Tang, W., and Han, W., "Conquer: A Continual Query System for Update Monitoring in the WWW," *International Journal of Computer Systems, Science, and Engineering. Special Issue on Web Semantics*, 1999.

[70] Liu, L., Tang, W., Buttler, D., and Pu, C., "Information Monitoring on the Web: A Scalable Solution," *World Wide Web Journal*, vol. 5, no. 2, 2002.

[71] loup Gailly, J. and Adler, M., "Gzip compression algorithm." http://www.gzip.org/algorithm.txt, 2004.

[72] Marian, A., Abiteboul, S., Cobena, G., and Mignet, L., "Change-centric management of versions in an XML warehouse," in *The VLDB Journal*, pp. 581–590, 2001.

[73] Meng, W., Wang, W., Sun, H., and Yu, C. T., "Concept hierarchy-based text database categorization," *Knowledge and Information Systems*, vol. 4, no. 2, pp. 132–150, 2002.

[74] Miller, J. and Mukerji, J., eds., *MDA Guide*. Object Management Group, 2003.

[75] Miller, R. and Bharat, K., "SPHINX: A framework for creating personal, site-specific web crawlers," in *Proceedings of the Seventh International World Wide Web Conference*, 1998.

[76] Min, J.-K., Park, M.-J., and Chung, C.-W., "Xpress: A queriable compression for xml data," in *Proceedings of the 2003 ACM Conference on Management of Data (SIGMOD '03)*, June 2003.

[77] Netscape, Inc., "Open Directory Project." http://www.dmoz.org/.

[78] Ngu, A. H. H., Rocco, D., Critchlow, T., and Buttler, D., "Automatic discovery and inferencing of complex bioinformatics web interfaces," Tech. Rep. UCRL-JRNL-201611, Lawrence Livermore National Laboratory, 2003.

[79] NLM/NIH, "National Center for Biotechnology Information." http://www.ncbi.nih.gov/, 2002.

[80] Object Management Group, "Model driven architecture." http://www.omg.org/mda/, 2004.

[81] "Pew Internet and American Life Project Survey. Search engines: a Pew Internet project data memo." http://www.pewinternet.org/reports/toc.asp?Report=64, July 2002.

[82] Quick, A., Lempinen, S., Tripp, A., Peskin, G. L., and Gold, R., "Jtidy." http://lempinen.net/sami/jtidy/, 2002.

[83] Rivest, R., "The MD5 message digest algorithm," tech. rep., Internet DRAFT, 1991.

[84] Rocco, D., Buttler, D., and Liu, L., "Sdiff," *Technical Report, Georgia Tech, College of Computing*, February 2002.

[85] Rocco, D., Buttler, D., and Liu, L., "Page digest for large-scale web services," in *Proceedings of the IEEE Conference on Electronic Commerce*, 2003.

[86] Rocco, D. and Critchlow, T., "Discovery and classification of bioinformatics web services," Tech. Rep. UCRL-JC-149963, Lawrence Livermore National Laboratory, 2002.

[87] Rocco, D. and Critchlow, T., "Automatic Discovery and Classification of Bioinformatics Web Sources," *Bioinformatics*, vol. 19, no. 15, pp. 1927–1933, 2003.

[88] Savourel, Y., *XML Internationalization and Localization*. SAMS, 2001.

[89] Sayood, K., *Introduction to Data Compression*. New York: Morgan Kaufmann, second ed., 2000.

[90] Shasha, D. and Zhang, K., "Approximate tree pattern matching," in *Pattern Matching Algorithms*, pp. 341–371, Oxford University Press, 1997.

[91] "Soap version 1.2 part 0: Primer," tech. rep., World Wide Web Consortium, 2003.

[92] Srinivasan, P., Mitchell, J., Bodenreider, O., Pant, G., and Menczer, F., "Web crawling agents for retrieving biomedical information," in *Proceedings of the International Workshop on Agents in Bioinformatics (NETTAB-02)*, 2002.

[93] Tanenbaum, A. S., *Computer Networks*. Prentice Hall, third ed., 1996.

[94] The Apache Foundation, "Apache XML Project." http://xml.apache.org/, 2002.

[95] Tolani, P. and Haritsa, J. R., "XGRIND: A query-friendly XML compressor," in *ICDE*, 2002.

[96] "TracerLock." http://www.tracerlock.com, 2001.

[97] UDDI.org, "UDDI executive white paper," tech. rep., UDDI.org, November 2000.

[98] UDDI.org, "UDDI technical white paper," tech. rep., UDDI.org, November 2000.

[99] Wagner, R. and Pescatore, J., "UDDI takes step forward but isn't ready for deployment," Tech. Rep. FT-18-0859, Gartner Research, 2002.

[100] Wang, Y., DeWitt, D., and Cai, J.-Y., "X-Diff: An Effective Change Detection Algorithm for XML Document," *International Conference on Data Engineering*, 2003.

[101] Witten, I. H., Moffat, A., and Bell, T. C., *Managing Gigabytes: Compressing and Indexing Documents and Images*. New York: Morgan Kaufmann, second ed., 1999.

[102] World Wide Web Consortium, "Well Formed XML Documents." http://www.w3.org/TR/REC-xml#sec-well-formed, 2000.

[103] Yahoo Inc., "Yahoo! homepage." http://www.yahoo.com/.

[104] Zadorozhny, V., Raschid, L., Vidal, M.-E., Urhan, T., and Bright, L., "Efficient evaluation of queries in a mediator for websources," in *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, 2002.

[105] Ziv, J. and Lempel, A., "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, pp. 337–343, May 1977.

VITA

Daniel Rocco was born in Bay Shore, New York and lived around the greater metro area until age 12 when his family moved to Atlanta, Georgia. His interest in computers began with his father's gift of a Commodore VIC-20 PC on which he learned computer operation and basic programming skills. This interest blossomed throughout Daniel's school years through the fostering of many friends who shared his enthusiasm for technology. He currently lives in Atlanta with his wife, Jennifer.