Building Blocks for Composable Web Services

A Thesis
Presented to
The Academic Faculty
by
David Buttler

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

College of Computing
Georgia Institute of Technology
September 2003
To my family:

Hyen-Kyung, Bruce, Geri, and Tim,

whose love and support made this possible.
Acknowledgements

This dissertation contains the results of my Ph.D. research carried out at the Georgia Institute of Technology. I am grateful to the many people who offered their advice and criticism to this dissertation in its early stages. Especially, I wish to thank my thesis supervisor, Professor Ling Liu, for giving me the opportunity to do this work, for the stimulating discussions which forced me to better explicate my ideas, for her constant support and consideration during my completion of the kind of research reported here, and for the freedom I experienced while working on this research project. I am greatly indebted to her for the guidance, encouragement and confidence which I felt privileged to receive during the past years.

I wish to thank professors Calvin Pu, Leo Mark, H. Venkateswaran, Ashok Goel, and Jiawei Han for taking part in my Ph.D. defense committee. Their critical reading and penetrating remarks make the final revisions both necessary and useful.

I have also benefited by the comments on various portions of the work reported here from Wei Han, Wei Ting, Henrique Paques, Daniel Rocco, James Coverette, Li Xiong, Lakshminarayanan Ramaswamy, and from the participants in the conferences and workshops in which I have participated.

I wish to take this opportunity to acknowledge the financial subsidies I received from the US Department of Energy, the Defense Advanced Research Projects Agency, the National Science Foundation, and the Georgia Institute of Technology for their support of my research work and for the assistance in travel to various conferences and workshops across the United States.

I would also like to take this opportunity to thank Dick and Sue Workman who gave me a great deal of help and support while I stayed in Atlanta.

Last, but not least, I wish to thank my family, especially my wife, for their understanding and love. I thank all my friends and colleagues who helped me in one way or another and who made my stay in Atlanta a pleasant one.

Since the manuscript went through a number of revisions, none of the people acknowledged should take responsibility for any remaining error.
# TABLE OF CONTENTS

DEDICATION iii  
ACKNOWLEDGEMENTS iv  
LIST OF TABLES viii  
LIST OF FIGURES ix  
SUMMARY xi  

I WEB SERVICE FUNDAMENTALS 1  
1.1 Motivation ........................................... 1  
1.2 Related Work ......................................... 2  
\hspace{1cm} 1.2.1 Web Service Standards ....................... 2  
\hspace{1cm} 1.2.2 Semantic Web .................................. 3  
\hspace{1cm} 1.2.3 Extensions to Web Services .................. 4  
1.3 Scope of Contribution ............................. 6  

II OBJECT EXTRACTION 9  
2.1 Omni Motivation ...................................... 10  
2.2 The Reference Document Object Model ............... 13  
2.3 Methodology: An Overview .......................... 17  
2.4 Algorithms for Object-rich Minimal Subtree Identification 21  
\hspace{1cm} 2.4.1 The Highest Fan-out Subtree Identification Algorithm (HF) ................. 22  
\hspace{1cm} 2.4.2 The Largest Tag Count Subtree Identification Algorithm (LTC) .................. 23  
\hspace{1cm} 2.4.3 The Largest Size Increase Subtree Identification Algorithm (LSI) ............... 26  
\hspace{1cm} 2.4.4 The Combined Algorithm and Its Performance ................................. 28  
2.5 Algorithms for Object Separator Discovery .......... 30  
\hspace{1cm} 2.5.1 The Sibling Tag Algorithm (SB) ................. 31  
\hspace{1cm} 2.5.2 The Partial Path Algorithm (PP) ................ 31  
\hspace{1cm} 2.5.3 The Standard Deviation Algorithm (SD) .......... 32  
\hspace{1cm} 2.5.4 The Repeating Pattern Algorithm (RP) ............ 33  
\hspace{1cm} 2.5.5 The Identifiable Path Algorithm (IPS) ............... 34  
\hspace{1cm} 2.5.6 The Combined Algorithm for Object Separator Discovery .................... 35  

v
LIST OF TABLES

1  Comparing H5, LSI, and LTC on canoe.com tag tree in Figure 4  ..................  22
2  Statistics used by the H5, LSI, and LTC algorithms on canoe.com tag tree in Figure 4  24
3  Comparing H5, LSI, and LTC on nbcio.com tag tree in Figure 6  ..................  27
4  Statistics used in comparing H5, LSI, and LTC on nbcio.com tag tree in Figure 6  27
5  Statistics used in calculating LSI on nbcio.com tag tree in Figure 6  ..................  28
6  Probability rankings for subtree identification algorithm  ..........................  29
7  Success rates for subtree identification combinations on test data  .................  30
8  Tag ranking for SB algorithm on Figures 4 and 1  .....................................  31
9  Partial paths and their count from the minimal subtree in Figure 4  ..............  32
10 Tag ranking from partial path rankings for Figure 4 and Figure 1  .................  32
11 Standard Deviation for tags from the minimal subtree in Figure 1  ...............  33
12 Repeated tag ranking for minimal subtree from Figure 4  ............................  33
13 A Table of Object Separator Tags  .........................................................  34
14 Object separator probability  .................................................................  35
15 Probability rankings for object separator algorithms  ...............................  36
16 Success rates for algorithm combinations on test data  .............................  36
17 Probability rankings for minimal subtree identification algorithms on validation data 37
18 Probability rankings for object separator tag discovery algorithms on validation data 38
19 Probability rankings for minimal subtree identification  .............................  39
20 Probability rankings for object separator algorithms  ...............................  39
21 Execution time in milliseconds  ..............................................................  43
22 Success rates for different heuristics and combinations on all 75 Web sites  ...  46
23 A fragment of the content summary for text fields from two search services  ...  81
24 Categories of XWrap Wrappers  ..............................................................  89
25 Web Sites Compared to MyYahoo'  .......................................................... 120

viii
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tree Representation for Library of Congress search results page</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>A sketch of the object extraction process</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Tag Tree for Cnose.com, Figure 3</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Tag Tree for ncbi.com, Figure 5</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>Precision and Recall of Object Construction; web sites sorted by precision</td>
<td>41</td>
</tr>
<tr>
<td>8</td>
<td>Execution times for Web sites; Web sites indexed by their parsing time</td>
<td>44</td>
</tr>
<tr>
<td>9</td>
<td>OminiSearch Architecture</td>
<td>48</td>
</tr>
<tr>
<td>10</td>
<td>Router Network</td>
<td>52</td>
</tr>
<tr>
<td>11</td>
<td>Service Selection Architecture Sketch</td>
<td>53</td>
</tr>
<tr>
<td>12</td>
<td>User-profile based query refinement</td>
<td>55</td>
</tr>
<tr>
<td>13</td>
<td>A sketch of source capability description of example data sources</td>
<td>63</td>
</tr>
<tr>
<td>14</td>
<td>An Example Run of Service Selection and Execution in BioSeek</td>
<td>74</td>
</tr>
<tr>
<td>15</td>
<td>Selection Accuracy for Varying Query Length</td>
<td>90</td>
</tr>
<tr>
<td>16</td>
<td>Selection Accuracy distribution</td>
<td>91</td>
</tr>
<tr>
<td>17</td>
<td>Selection Accuracy; Score plot</td>
<td>92</td>
</tr>
<tr>
<td>18</td>
<td>Metatext Selection Accuracy over the invisible Web</td>
<td>93</td>
</tr>
<tr>
<td>19</td>
<td>Performance Gains From Selection</td>
<td>94</td>
</tr>
<tr>
<td>20</td>
<td>Memory prices reported at Pricewatch.com</td>
<td>100</td>
</tr>
<tr>
<td>21</td>
<td>Sentinel Hierarchy</td>
<td>103</td>
</tr>
<tr>
<td>22</td>
<td>Tree fragment with corresponding Page Digest</td>
<td>106</td>
</tr>
<tr>
<td>23</td>
<td>Comparisons of various loading techniques</td>
<td>107</td>
</tr>
<tr>
<td>24</td>
<td>Transformation of Figure 22 into a structural shingle</td>
<td>118</td>
</tr>
<tr>
<td>25</td>
<td>Cross Site Comparison</td>
<td>119</td>
</tr>
<tr>
<td>26</td>
<td>Single Site Changes Over Time</td>
<td>121</td>
</tr>
<tr>
<td>27</td>
<td>Example of Grouping Sentinels</td>
<td>122</td>
</tr>
<tr>
<td>28</td>
<td>Index for Grouping Sentinels</td>
<td>123</td>
</tr>
<tr>
<td>29</td>
<td>Updated Tree fragment with corresponding Page Digest</td>
<td>125</td>
</tr>
</tbody>
</table>
SUMMARY

Web Services are emerging as a technology that will transform the Web from a reactive repository of data to a proactive service-oriented Semantic Web. To make the Semantic Web and the sea of Web services more efficient, scalable, and composable, we present three basic technologies that are building-blocks to a wide range of present Web services: automated extraction of dynamically generated data, automated selection of Web services, and efficient and scalable detection of meaningful changes in Web documents.

Automated object extraction is the essential component of many data intensive Web services that analyze, compare, and integrate dynamic content generated from deep Web databases behind the search interfaces. Web pages are designed to make data available for user browsing, and their human-centric design makes them difficult for machines to process the data and automatically integrate them into composite Web services. Automated techniques to recognize data from different Web sites, and enable it to be reliably extracted in the face of constantly updated page designs is crucial. We present the Omnit methodological for a fully automated object extraction system for Web pages, consisting of a layered approach to first identify data regions in pages, and then to extract individual objects. We evaluated Omnit using more than 3,200 pages over more than 100 diverse Web sites. Our algorithms for identifying the minimal object rich region achieves 96% success rate. Our algorithms for discovering object boundaries reach the success rate of 95%. Most significantly, our algorithms are fast, about 87 milliseconds per page with an average page size of 30KB. The overall system achieves precision between 95% and 96% (returns correct objects most of the time) and excellent recall — between 96% and 100% (few significant objects left out).

Tracking and detecting changes on the Web has become one of the fundamental services of enterprise computing and scientific computing. Search engines are popular tools that help users locate information on the Web, but they provide no support for tracking interesting changes. We use the concept of Web page sentinels and the Page Digest a structured Web page encoding scheme for
efficient detection of changes in Web pages. A key objective of our change detection research is focusing on scalability techniques for Web change monitoring systems which are designed to enable the handling of millions of information monitoring requests. The gains in scalability and performance stem from several key techniques. First, we introduce a new class of Page Digest based sentinels capable of monitoring changes in the content of a page, the structure and presentation of a page, as well as measuring the percentage of changes that have occurred. Second, we develop a set of mechanisms such as short-circuit evaluation, linear time algorithms for computing document and structure similarity, and data size reduction. Third, we develop a collection of sentinel grouping techniques based on the Page Digest encoding, effectively eliminating redundant processing and unnecessary network communication. We evaluate these techniques over a wide range of parameters, showing an order of magnitude speed up over previous efforts and provide mechanisms to scale the system beyond foreseeable bottlenecks in network bandwidth and storage requirements.

Web service selection is an important issue in composable Web service research. The increasing volume and diversity of Web services have led to a growing problem that conventional data management systems do not have, namely finding which Web services out of many candidate choices are the most relevant and most accessible to answer a given user query. We refer to this problem as the Web service selection problem. We introduce the notation and issues of Web service routing, and present a practical solution for designing a scalable Web service selection system based on multi-level progressive pruning strategies. The key idea is to create and maintain service capability profiles independently, and to provide algorithms that can dynamically discover relevant services for a given query based on the digital contents provided and through the smart use of source profiles and user profiles. Compared to keyword-based indexing techniques adopted in most of the search engines and software, our approach offers finer granularity of interest matching, thus it is more powerful and effective for handling queries with complex conditions.

The salient contributions of this dissertation research include a methodology for efficient and automatic object extraction that provide significant improvements over existing efforts, a comprehensive approach to service selection that combines highly accurate schema-based selection with generic content-based selection, and a scalable page digest-enhanced change detection framework and efficient sentinel processing algorithms that offer more than an order of magnitude improvement.
on the change detection efficiency.
CHAPTER I

WEB SERVICE FUNDAMENTALS

1.1 Motivation

Web Services are often touted as the next new technology that will enable wide spread data sharing and integration, creating new opportunities for collaboration between organizations and individuals as well as more capabilities. However, the fundamental revolution is really in the standards defined for communication and the use of widely accepted terms. Unfortunately, there are several stumbling blocks to making the current Web into a bazaar of interchangeable, composable, and accessible services.

First, there is a huge amount of legacy data available that is designed for human browsing. This data, by definition, uses HTTP for communication and HTML for the encoding of information, as opposed to newer standards such as Simple Object Access Protocol (SOAP) for communication and domain-specific, standardized XML as the data markup language. Thus, it cannot be easily incorporated into services. One typical approach is to create wrappers, but this usually fails to scale to the number of services available; also, wrappers turn out to be brittle, requiring them to be regenerated as often as the service changes any part of its interface.

Second, there is no centralized authority that allows users to find the services that provide the information that they are looking for. Search engines, a popular mechanism to find information on the static web, cannot assist in helping users locate appropriate services; they only index content that can be accessed by following links, and so miss the vast majority of the hidden web. UDDI registries are starting to appear to host WSDL-based services, but their search mechanisms rely on free-form descriptions of the business providing the service and a limited set of non-standard schema descriptions. Schemas are typically as descriptive as method interfaces in programming languages.

Manually converting these sources to full members of the Web Service community is unlikely to succeed for several reasons. First, it would require an overwhelming investment of effort, requiring
the entire infrastructure that delivers the information to be changed to support the new standards. Second, the schemas of hundreds of domains would need to become standardized. This by itself is a huge challenge because many organizations that produce the data on the Internet are fiercely independent, and instinctively resist standardization efforts. Some domains cannot even create a standardized terminology because of the rapid evolution of the field. For example, in bioinformatics, only a small set of terms have been standardized. Since the underlying science is still evolving at a rapid pace, efforts to create a formal ontology that describes all information in the field would be premature.

1.2 Related Work

1.2.1 Web Service Standards

The Web Service Description Language, or WSDL [33], is an XML-based standard that defines a uniform mechanism to describe how to invoke services, including the invocation mechanism (such as an HTTP request to a specific URI), and the arguments for the invocation. The services operate on messages containing either document-oriented or procedure-oriented information. All operations and messages are described abstractly, allowing multiple bindings to different protocols and message formats (known as endpoints). While WSDL is extensible enough to describe a large variety of endpoints, it currently only defines standard HTTP GET, POST, and MIME protocols to transport information in text, HTML, XML, or SOAP.

Universal Description, Discovery, and Integration of Web services, or UDDI [125, 122], is a standardization effort for describing Web services so that they are easier to locate and to integrate into Web service based programs. UDDI defines a registry that allows authorized users to add descriptions of existing services and to retrieve the description of previously stored services. Currently, the UDDI definition is focused on a business service naming registry; it provides information on who is publishing a service, a description of what the service does, and where to get the specification, typically a WSDL definition, of how to interact with the service. The UDDI definition contains five main ideas: Entity — the entity component refers to the organization that publishes a set of services. Service — the service component describes what a particular service does. Descriptions are short, non-technical sections of text, defined for human browsers to understand the intent
of the service. Binding — the binding component describes what protocol is used in the service and where the document is that precisely describes how to invoke the service. For example, a service can be specified to use WSDL with a pointer to the URI that specifies the WSDL definition. 

`tModel` — the `tModel` specifies the taxonomy of a particular service, the terms used and the technical specifications. Publisher assertion — the assertion indicates a relation between two services. The publishers of each service must both issue the same assertion for the relation to be visible, or the same publisher must publish both services named in the assertion.

Resource Description Framework, or RDF [73, 92], is a simple graph-based data model. It consists of three concepts: subject, property, and object. The property defines a relationship between the subject and the object. Each of the subject, property, and object may be a node in the graph or a (typed or untyped) string. RDF was originally designed to represent metadata of Web resources, such as the author, creation date, and version. It has been extended to represent information about any resource that can be identified on the Web. One goal of RDF is to provide a common framework for representing information so that it may be exchanged between different applications. While RDF does provide this capability, it is also dependent on the development of a common vocabulary that is agreed upon among all parties who wish to share a particular set of RDF data.

### 1.2.2 Semantic Web

The Semantic Web was introduced by Tim Berners-Lee, the creator of the original World Wide Web.

The Semantic Web is the representation of data on the World Wide Web. It is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners. It is based on the Resource Description Framework (RDF), which integrates a variety of applications using XML for syntax and URIs for naming.

The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation [15].

The Semantic Web is also commonly used as the generic name given to efforts in making Web-based information and services more automated and capable of semantically correct processing by machines. Semantic Web efforts encompass efforts in developing a repository of common knowledge [79], annotation of data [62], ontology creation and ontology conflict resolution [106], and
semantic transformations of data and metadata [68].

One prominent approach to creating the infrastructure for the Semantic Web is the DAML+OIL [43] ontology language, and the DAML-S [6, 7] Web service description language. DAML+OIL (DARPA Agent Markup Language combined with the Ontology Interchange Language) is a language that allows concepts to be defined in relation to each other (equivalence, part-of, generalization-of, and so on), and is built on the RDF definition.

DAML-S uses this language to describe services in three parts: the service profile, the service model, and the service grounding. The service profile describes what the service does, describing the information required to invoke the service and the type of information that it provides, and actions that it may perform. The service model describes how the service works, including all of the steps that must be performed, the order, and the information required at each step to invoke the service. The service grounding defines how a service is invoked (specific protocols to be used, specific address to communicate with), and is often a mapping to a WSDL description of a service.

1.2.3 Extensions to Web Services

Web Services provide a basis for extending the quantity and quality of distributed computing. However, there are several difficult challenges introduced by the loose-coupling inherent in such a distributed architecture. Here we examine some of difficulties that researchers have addressed. Our intent is to provide a flavor of the issues, and not to provide a complete list of possible research topics.

One of the most obvious issues is the composition of Web Services. Currently, programmers piece services together one by one, much as they would construct a software program from components and library calls. However, due to the increased capabilities for attaching semantics to services, the distributed nature of services, and the variety and overlap of services, automated techniques for automatically composing services into programs have been proposed. One approach is to allow users to declaratively state what a program needs to accomplish, and have a system stitch together components that will fulfill the requests. This has strong analogies to functional programming. Indeed, some projects have adapted Golog [96], a prolog-like language, to create a system to automatically compose services using theorem-provers to match services to each other and provide
results for user queries

Other researchers have taken a database style approach [124] to services, offering a high-level language to express user constraints and an optimizer to create optimal query plans that satisfy these constraints. This may be coupled with a transactional system that will monitor the progress of a particular query plan, rolling back or compensating actions that become obsolete due to failures in later parts of a query plan.

Composition of services can also be thought of as a workflow, where the entire workflow may represent a new service. While DAMLS provides basic workflow description in the service model, many other efforts have defined their own workflow specification, designed to solve problems in a specific domain. Workflows typically include elements to order services into groups that are executed in sequence or in parallel, as well as providing control flow constructs such as repeated execution, conditional execution, and exception handling. More sophisticated constructs provide transactional capabilities to a hierarchy of services, or provide alternate execution paths if one component of the workflow fails.

Workflow standards for business use include XLANG from Microsoft [120], Web Services Flow Language (WSFL) from IBM [81], Business Process Execution Language for Web Services (BPEL4WS) [5] from a consortium including BEA, IBM, Microsoft, SAP, and Siebel, and Of-bz [69], an open-source workflow execution engine. Scientific workflow systems include efforts to provide a high-level, abstract specification of a scientific process that can be transformed into an executable workflow process [57, 56], and to specify how services can be composed with semantic consistency to manage scientific processes [48].

One common assumption in Web service systems is the availability of a repository of services, defined in a consistent way. Despite efforts in UDDI and WSDL, standardization, such a registry is not available for the majority of services. Several research projects have focused on the discovery and classification of Web services [25, 83, 54, 52, 98, 66], often with the goal of selecting document databases most appropriate for a given query. This work typically focuses on servers that do not have a rich content description given by the service provider, and the categorization is typically created based on an independent analysis of the data content of the services.
1.3 Scope of Contribution

The goal of the work described in this thesis is to make current (legacy) services more capable, so that they can be accessed by a larger variety of users, and composed into larger and more functional services. There are a large number of components required to make services accessible, reliable, scalable, active, and composable. This thesis focuses on aspects of services that increase access, extend the services to be active, and to provide a basis for compositability. The components we describe are required to access services that are designed for human consumption (HTML forms) and make them available as highly functional and efficient Web Services; other components are required in selecting appropriate services for each user request; a final, higher-level component, activates services so they proactively send important fresh information to users as opposed to only responding to individual requests for information.

The research challenges we address are efficiency, scalability, and robustness at each level of a service, including methodologies in extraction of objects from existing services, selection of appropriate services for user requests, and activation of services. Our approach to these challenges is to apply fast algorithms and novel data structures to provide efficiency and scalability while preserving their capabilities as the services, and the data they produce, evolve. There are other critical components that we have not addressed here, including reliability, and transactional support.

The first component we discuss extracts data from HTML pages presented for human browsing. Since the vast majority of accessible on-line data is not designed for processing, it must be extracted from its description in HTML, a presentation markup language, and converted into chunks of information suitable for machine processing. Manually creating programs (wrappers) to convert the presentation text into processable data is labor-intensive and brittle, whether it is done by information providers, information gatherers, or third parties. Most data sources on the Internet tend to evolve quite rapidly, changing the layout of the site, the presentation of the data, and, in many cases, changing the types of data they present. The difficulty here is to develop automatic techniques to discover what parts of a dynamically generated Web page contain data, and to extract the data from the page, without requiring human input. This process is much more flexible to the inevitable changes a Web site undergoes as there are no hard-coded rules about how the information should be
extracted.

The second component selects Web sites that provide useful information. Before a service may be accessed or integrated into a workflow or data fusion system, it must be known, and there must be sufficient meta-data describing the service and the data it produces to make an informed decision as to whether or not it is worthwhile in a particular context. Service selection makes use of all the information that is known about services — including provider published, user submitted, and automatically gathered — to choose the subset of known services that are most likely to provide high-quality relevant information quickly to the user.

Once a service has been located, and its data is made accessible via data extraction, the quality of the data becomes a vital component. One useful metric of data quality is the freshness. This is the difference between the time that the data is presented to a user and the last time the data has changed or is valid. While some data is insensitive to freshness, many types of information are time critical and must be within a certain freshness interval to be valid. Only providing access to legacy services is insufficient. To ensure that the data provided by the integrated services is as valuable and reliable as possible, data access must be accompanied by the ability to push important changes in the data to users. This changes a service from being purely reactive to user requests to being proactive in satisfying user needs with respect to the data.

Each component is evaluated in terms of efficiency, scalability, and robustness. Object extraction is evaluated not only in relation to manual methods, but on an absolute scale, determining how long is required to extract data from a particular service. In addition, object extraction is evaluated based on how robust it is in relation to changing format. However, rather than artificially manipulating the output of a source, or waiting for sources to change to see what the effect would be, we analyze the results of the object extraction algorithms over a large variety of sources, measuring the success rate on many sources as a proxy for a single source changing over time. Service selection is also measured in multiple ways: first, the scalability improvements that selecting appropriate sources brings to the execution of a distributed query plan are modeled and evaluated. Second, the effectiveness of different techniques are evaluated, and the information required of each method is discussed. Finally, the methodology of change detection and service activation is extensively examined for scalability issues, comparing it to related efforts, micro-benchmarking essential
components, and providing an overall evaluation of the efficiency of the system.

The rest of the thesis proceeds as follows. Chapter 2 outlines the Omini system, explaining the automatic data extraction methods. This allows services to aggregate data from other sources without the expensive manual creation of brittle data extraction components. Chapter 3 describes problems in service selection that have prevented automated composition of relevant services. We propose a technique that will allow components to be automatically identified and inserted into a registry so that they can be accessed by service selection and composition systems. Chapter 4 describes techniques to provide efficient change detection over Web pages. We show more than one order of magnitude improvement over the original WebCQ system by improvements in algorithms, data structures, and implementation choices. Chapter 5 concludes with a summary of the contributions of the thesis and describes future work to extend these building blocks and create other components of fully composable Web services.
CHAPTER II

OBJECT EXTRACTION

Data extraction is a basic building block in developing composable services as it addresses the enormous number of existing services that provide an HTML interface to their data. Without a mechanism to access these services a significant amount of available information could not be accessed. Automated techniques are essential in addressing this issue due to the sheer quantity of available information, and because services evolve. Automation allows us to address a much larger number of available services, and it also provides techniques to maintain the capability to extract information as services change over time. And perhaps more importantly, automatic extraction systems allow services to maintain their current human-centered interfaces, while still being available as composable web services.

Even for services that present their data as XML, data extraction may still be required. Having data described in an XML document, does not guarantee that it is intelligible or simple to integrate with other services, it only eases the parsing of the data. Discovering where information is located is still important. While this was not an obvious problem with early XML files that consisted of the smallest possible number of tags wrapping the information, as XML tools have progressed, and users put it to more and more complex uses, locating where information is located in an XML document has become more difficult. While the techniques presented here are obviously not sufficient to transform any type of XML into more manageable data, it does assist in recognizing typical information patterns where they are present.

This chapter presents a methodology and a set of experiments for a fully automated object extraction system for Web pages. Our methodology consists of a layered framework and a suite of algorithms. A distinct feature of our approach is the full automation of both the identification of data object regions from dynamic Web pages and the discovery of object boundary separators. We implemented the methodology in the Omni object extraction system and evaluated the system using more than 3,200 pages over 75 diverse Web sites. Our experiments show three important and
interesting results: First, our algorithms for identifying the minimal object rich subtree achieve 96% success rate over all Web pages we have tested. Second, our algorithms for discovering object separator tags reach the success rate of 95%. Most significantly, the overall system achieves precision between 96% and 100% (returns only correct objects) and excellent recall (between 95% and 96%, with few significant objects left out). The minimal subtree identification algorithms and the object boundary identification algorithms are fast, about 87 milliseconds per page with an average page size of 30KB.

In addition, we have extended the techniques described here to apply the PageDigest format (described in Section 4.3). This has resulted in a 40% increase in the overall extraction times, as opposed to using the typical DOM format. The efficiency of the algorithms reduces the cost of object extraction to a fraction of the cost of parsing the page for known sites.

2.1 Omini Motivation

The exponential growth of information accessible through the World Wide Web makes the Web an increasingly important source of information. Search engines have become ubiquitous tools for accessing and finding information on the Web. Not surprisingly, the explosive growth of the Web has also made information search and extraction a harder problem than ever. As of February 2000, the publicly indexable Web (static pages) contains more than 800 million pages. No search engine indexes more than one sixth of the indexable Web [78]. To make the matter even worse, not only the number of static Web pages increases approximately 15% per month, the number of dynamic pages generated by programs (i.e., the Web pages behind the forms) has been growing exponentially. The huge and rapidly growing number of dynamic pages forms an invisible Web, out of the reach of search engines.

To address the search problem over dynamic pages, several domain-specific information integration portal services have emerged, such as Excite's Jango and cnet.com. These integration services offer an uniform access to heterogeneous collections of dynamic pages using wrappers [41], programs that encode semantic knowledge about specific content structures of Web sites, and use such knowledge to aid object extraction from those sites. The current generation of wrappers have been developed and maintained by semi-automatic wrapper generation systems [1, 8, 41, 74, 59, 87, 113].
Wrappers generated semi-automatically often encode programmers’ understanding of specific content structure of the Web pages to guide the object extraction process. Manually encoding programmer’s understanding of the Web pages as extraction markers has two detrimental drawbacks. First, it makes the extraction program extremely vulnerable to any changes on the specific text and content structure used as extraction markers. Second, semi-automated extraction methods are inherently difficult to scale up to the rapid growth of the Web; the amount of manpower required for encoding semantics of every existing or new Web site to perform extraction can be huge and unmanageable. In addition, the maintenance of semi-automatically generated wrappers for frequently changing Web sites turns out to be labor intensive and error-prone. Consequently, most of the information integration services based on wrappers (e.g., Excite Jango and cnet.com) have serious limitations on the breadth and depth of their coverage (i.e., number of sites and information accessible from each site). They have a hard time to effectively incorporate additional or new content providers into their existing integration framework. In addition, they need significant manpower to make sure that the working wrappers can keep up with the evolutionary changes of the corresponding Web sites.

In analogy to search engines that fully automate the crawling of static Web pages, we argue that one way to achieve robustness and scalability required for Internet-scale information integration service is to employ a fully automated approach to extracting objects from dynamic Web pages. By full automation, we mean that the object extraction algorithms should be designed independently of the presentation features of the Web pages. Thus objects of interest can be extracted from dynamic Web pages without embedding any programmers’ semantic knowledge of the pages, such as the specific ways in which objects are laid out or the specific locations where the navigational links and advertisement information are placed in the Web pages.

Fully automated object extraction is possible because of the following empirically observed regular patterns in dynamic pages. Dynamic Web pages typically consist of three different types of presentation regions:

1. The data object region, which presents the primary content provided by the content provider.

   We sometimes also refer to such object region as the primary content region. For instance, Amazon.com’s on-line bookstore Web site will display the book objects returned from a search request in such a primary content region.
2. The advertisement region, which presents the information about other products offered by the content provider or about related products offered by other companies.

3. The navigational region, which presents a collection of navigational links, offering convenience for surfers to jump to other Web sites provided by the same content provider (such as Amazon’s music site, Amazon’s software site).

Therefore, the problem of automatically extracting objects from a Web page should be addressed in two phases. First, one needs to develop methods that can automatically locate the primary content region (the data object region) in any given Web page. Then, one needs to have effective mechanisms that can correctly identify the object boundaries within the primary content region.

With these baselines in mind, we propose a fully automated approach to extracting objects from dynamic Web pages. In this paper we present a methodology for automating the object extraction process and a set of experiments that show the viability of the proposed methodology. Our methodology has two distinct features. First, it models Web pages by tag trees and uses a set of subtree identification algorithms to locate the smallest subtree in a page which contains all the objects of interest (e.g., by ignoring advertisements). Second, it employs a suite of object separator identification algorithms to find the correct object separator tags that can effectively separate objects. Both steps are fully automated. The subtree identification stage considerably reduces the number of possibilities considered in the object separator discovery stage. We implemented the methodology in a prototype system called Omni. The Omni system has been tested over 500 Web sites by both end users and a wrapper generation system, XWrapElite [61]. Our algorithms for minimal subtree identification and correct object separator discovery are fast.

The entire process is \( O(n) \), where \( n \) is the size (length in characters) of an input Web page. Our approach for extracting objects from dynamic Web pages is effective. We conducted a series of experiments over more than 1,200 Web pages from 75 popular Web sites, the results were consistent and satisfying, and attaining recall ratio of 95% and precision ratios of 96% on all the sites we examined.
Before explaining the details of our approach, we would like to note that fully automated approach to information extraction from Web pages is just one of the big challenges in building a scalable and reliable information search and aggregation services for the Web. Other important problems include resolving semantic heterogeneity among different content providers, efficient query planning and fusion for gathering and integrating the requested information from different Web sites, and intelligent caching of retrieved data. The focus of this chapter is solely on automated information extraction from dynamically generated Web pages.

The rest of the chapter proceeds as follows. We introduce the basic terminology in Section 2.2. Section 2.3 presents an overview of our object extraction methodology. Then we describe the algorithms for object-rich subtree identification in Sections 2.4. Section 2.5 describes the object separator discovery algorithms. Section 2.6 reports the experiments and demonstrates the validity and effectiveness of our object extraction approach through an analysis of our experimental results. Section 2.7 discusses the related work and shows the quantitative analysis of our approach. We conclude the chapter with a discussion on related work in Section 2.7 and a summary in Section 2.9.

2.2 The Reference Document Object Model

The reference document object model defines the logical structure of well-formed Web pages and the way such a page is accessed and manipulated. We use the W3C Document Object Model (DOM) [64] as the basis of our reference Document Object Model. In addition to the set of core interfaces defined in W3C DOM for creating and manipulating Web pages, we provide a formal definition of a number of concepts that are used in formulating our object extraction methodology. The Web pages considered in the rest of the chapter are HTML pages. XML pages [20] can be seen as a special case by our object extraction algorithms [23].

Well-Formed Web Page

A Web page consists of text and tags. A tag is marked by a tag name and an optional list of tag attributes enclosed in a pair of opening and closing brackets "<" and ">". Text is a sequence of characters in between two tags. By HTML [105] specification standard, tags in a well-formed Web page appear in pairs. A tag whose name does not start with a forward slash (i.e., "/") is called a start tag; otherwise it is called an end tag and the name of an end tag is the name of its corresponding
start tag proceeded by "\"", Pages that are not well formed can be converted to well-formed pages. We refer to such a transformation as page normalization.

Tree Representation of Web Pages
A well-formed Web page can be modeled as a tag tree. All the internal nodes of a tag tree are tag nodes and all leaf nodes are content nodes (numbers, strings, or other data types such as encoded MIME types). A tag node denotes the part of the Web page identified by a start tag and its corresponding end tag and all characters in-between. A tag node is labeled by the name of the start tag. A leaf node denotes the content data (text) between a start tag and its corresponding end tag or between an end tag and the next start tag in a Web page. A leaf node is labeled by its content. An example tag node in an HTML page is \texttt{<title> Home Page \textit{<title>}, where \texttt{<title>} is the same of the tag node and the text string Home Page is a leaf node.

\textbf{Definition 1 (Tag Tree)}
A tag tree of a page \( D \) is defined as a directed tree \( T = (V, E) \) where \( V = V_I \cup V_C \), \( V_I \) is a finite set of tag nodes and \( V_C \) is a finite set of content nodes: \( E \subseteq (V \times V) \), representing the directed edges. \( T \) satisfies the following conditions: \( \forall (u, v) \in E, (v, u) \notin E \); \( \forall u \in V, (u, u) \notin E \); and \( \forall u \in V_C, \exists v \in V \) such that \( (u, v) \in E \).

For any node \( u \in V \), we use the predicate \texttt{node\_Name(u)} to refer to the tag name of the tag node \( u \) or the content data of the content node. We use the predicate \texttt{parent(u)} to refer to the parent node of \( u \), \texttt{parent(u)} = \{w|w \in V, (u, w) \in E \}. The root node of a tree \( T \) is the only node which does not have a parent node. Similarly, for any node \( u \in V \), we use \texttt{children(u)} to refer to the set of child nodes of \( u \), \texttt{children(u)} = \{w|w \in V, (u, w) \in E \}. A node \( w \) is a child node of \( u \) if and only if there exists an edge \( (u, w) \in E \). When two nodes \( u \) and \( v \) have the same parent node, we say they are siblings. We use \texttt{siblings(u)} to denote the set of sibling nodes of \( u \), \texttt{siblings(u)} = \{w|w \in V, \texttt{parent}(w) = \texttt{parent}(u) \}.

\textbf{Definition 2 (path: \rightarrow*)}
Let \( T = (V, E) \) be the tag tree for a Web page \( D \). There is a path from node \( u \in V \) to node \( v \in V \), denoted by \( u \rightarrow^* v \), if and only if one of the following conditions it satisfied:

\begin{enumerate}
\item \( u = v \)
\end{enumerate}
(ii) \((u, v) \in E\)

(iii) \(\exists u' \in V, u' \neq u \text{ and } u' \neq v, s.t. u \rightarrow^* u' \text{ and } u' \rightarrow^* v.\)

If \(u \rightarrow^* v\), then \(u\) is called an ancestor of \(v\) and we say that node \(v\) is reachable from node \(u\).

There is a path from the root node to every other node in the tree. For a given node, the path expression from the root of the tree to the node can uniquely identify the node. Therefore, in subsequent sections we sometimes use such a path expression to refer to the node. Consider the tag tree in Figure 1. The root node has the node name \(html\). \(body\) is the node name of a child node of \(html\). The path from the root node \(html\) to the \(title\) node goes through the \(head\) node. It can be expressed as \(html \rightarrow^* title\). An alternative method to represent a path is to use XPath notation [34]. For example, the expression \(html/head/title\) can also be used to describe the path from the \(html\) node to the \(title\) node in Figure 1. Whenever a node has siblings of the same tag name (such as the \(br\) node in Figure 1), we use the XPath convention to avoid ambiguity. For instance, the path expression \(html/body/\text{br}[1]\) denotes the path from the \(html\) node to the first \(br\) child node of \(body\).

The numerical number in the square bracket immediately following the node name denotes the appearance order of a node in the tag tree if it has siblings of the same node name.

**Figure 1:** Tree Representation for Library of Congress search results page.

**Definition 3 (Subtree)**

Let \(T = (V, E)\) be the tag tree for a Web page \(D\), and \(T' = (V', E')\) is called a subtree of \(T\) anchored at node \(u\), denoted as subtree\((u)\) \((u \in V')\), if and only if the following conditions hold:

- \(V' \subseteq V, \text{ and } \forall v \in V, v \neq u, \text{ if } u \rightarrow^* v \text{ then } v \in V'\);

- \(E' \subseteq E, \text{ and } \forall v \in V', v \neq u, v \notin V, \exists u \in V', w \neq v, \text{ and } (v, w) \in E'\).
For a tag tree $T = (V, E)$, the total number of subtrees is $|V|$. We call a subtree anchored at node $u$ a minimal subtree with property $P$, if it is the smallest subtree that has the property $P$, namely there is no other subtree, say subtree$(w), w \in V$, which satisfies both the property $P$ and the condition $u \Rightarrow^\ast w$ ($u$ is an ancestor of $w$).

**Definition 4** (Minimal Subtree with Property $P$)

Let $T = (V, E)$ be the tag tree for a Web page, and subtree$(u) = (V', E')$ be a subtree of $T$ anchored at node $u$. We call subtree$(u)$ a minimal subtree with property $P$, denoted as subtree$(u, P)$, if and only if $\forall e \in V, v \neq u$, if subtree$(v)$ has the property $P$, then $v \Rightarrow^\ast u$ holds.

Consider Figure 1, there are two subtrees that contain all of the hr nodes, the subtree anchored at html and the subtree anchored at body. The subtree anchored at body is the minimal subtree that contains all of the hr nodes.

In addition to the notion of subtree and minimal subtree, the following concepts are used frequently in the subsequent sections to describe our object extraction algorithms.

- **fanout$(u)$**: For any node $u \in V$, we use fanout$(u)$ to denote the cardinality of the set of children of $u$. fanout$(u) = |\text{children}(u)|$ if $u \in V_F$ and fanout$(u) = 0$ if $u \in V_C$.

- **nodeSize$(u)$**: For any node $u \in V$, if $u \in V_C$, i.e., $u$ is a leaf node, then nodeSize$(u)$ denotes the content size in bytes of node $u$. Otherwise, $u$ is a tag node, i.e., $u \in V_T$ and fanout$(u) > 0$. We define nodeSize$(u)$ to be the sum of the node sizes of all the leaf nodes reachable from node $u$, i.e., nodeSize$(u) = \sum_{v \in \text{children}(u)}(\text{nodeSize}(v))$.

- **subtreeSize$(u)$**: For any node $u \in V$, we define the size of the subtree anchored at node $u$, denoted by subtreeSize$(u)$, to be the node size of $u$, i.e., subtreeSize$(u) = \text{nodeSize}(u)$.

- **tagCount$(u)$**: For any node $u \in V$, if $u \in V_T$ is a leaf node, then tagCount$(u) = 0$. Otherwise, $u \in V_T$ is a tag node and tagCount$(u) = 1 + \sum_{v \in \text{children}(u)}(\text{tagCount}(v))$. tagCount$(u)$ refers to the total number of tag nodes of which $u$ is an ancestor.

- **largestChildNode$(u)$**: For any node $u \in V_T$, we define the largest child node of $u$ to be the child node that has the largest size. More precisely, largestChildNode$(u) = \{w | \forall v \in \text{children}(u), \forall e \in \text{children}(u), v \neq w, \text{nodeSize}(v) \leq \text{nodeSize}(w)\}$.
appearanceCount\((u, v)\): For any node \(u, v \in V, v \in children(u)\), we define the appearance count of the child node \(v\) to be the number of times that a node named \(\text{nodeName}(v)\) appears as the child node of \(u\), i.e.,
\[
\text{appearanceCount}(u, v) = \left| \{ w \mid w \in children(u), \text{nodeName}(w) = \text{nodeName}(v) \} \right|.
\]

\(HACC(u)\): For any node \(u \in V\), \(HACC(u)\) denotes the highest appearance count of the child node of \(u\). We define \(HACC(u)\) to be the highest number of times a single tag name appears as a child of \(u\). That is, \(HACC(u) = \text{appearanceCount}(u, w)\), where the node \(w \in children(u)\) satisfies that \(\forall v \in children(u), v \neq w, \text{appearanceCount}(u, v) \leq \text{appearanceCount}(u, w)\).

Consider the tag tree in Figure 1. \(\text{fancy}(\text{html}) = 2, \text{fancy}(\text{body}) = 67, \text{tagCount}(\text{head}) = 2, \text{tagCount}(\text{body}) = 77, \text{AppearanceCount}(\text{body, p}r) = 20, \text{AppearanceCount}(\text{body, hr}) = 21\), and \(HACC(\text{body}) = 21\). Both tag \(a\) and tag \(hr\) appear twenty-one times, and are considered the highest appearance child node of \(\text{body}\).

2.3 Methodology: An Overview

In Omini, the problem of automatically extracting objects from Web pages is addressed in two phases: identifying the region and boundaries of objects in a Web page and extracting those objects that are of interest.

The first phase focuses on identifying the primary content region in an arbitrary Web page and then identifying the object boundaries within the primary content region. We develop a selection of methods that can automatically locate the primary content region (the data object region) in any given Web page. We also devise a set of mechanisms that can correctly identify the object boundaries within the primary content region by discovering the right object separator for each primary content region. For HTML pages, such separator can be a single HTML tag or an ordered list of HTML tags. The second phase is dedicated to actual extraction of objects, which involves the use of the phase one result to locate the primary content region, the use of object separator to separate the content region into textual objects (object construction), and the refinement that cleans up some extraneous information in between the extracted objects or surrounding around the extracted objects.
Concretely, we model the two-phase process of extracting objects from Web pages as a four-step automated process. Figure 2 shows a sketch of the four steps and the tasks to be performed in each step. A user or an application may submit a URI to the system to initiate the object extraction process. The result returned is a list of objects extracted from the given Web page.

![Diagram](image)

**Figure 2:** A sketch of the object extraction process

**Step 1: Preparing a Web page for extraction**

This step prepares the Web page for extraction. It takes a URI from an end-user or an application, and performs three tasks: First, the Web page specified by the URI will be fetched from a remote site. Second, the fetched page will be cleaned using a syntactic normalization algorithm, which transforms the given Web page into a well-formed Web page. Third, the well-formed Web page will be converted into a tag tree representation based on the nested structure of start and end tags. The tag tree construction algorithm is omitted here due to space restriction.

**Step 2: Object-rich subtree identification**

After locating a Web page \( P \), the next step is to identify which part of the page is the primary content region. Let \( T \) be the tag tree of the page \( P \). The task of locating the primary content region of \( P \) can then be reduced to the problem of locating the minimal subtree of \( T \) which contains all the objects of primary interest. We call this task the **Object-rich minimal subtree identification**. By examining various dynamic Web pages and their tag trees, one can indisputably observe that the minimal subtree
of a page that contains all the data objects of interest is often the subtree that has either the largest content size, or the largest tag count, or the highest fanout. Therefore, an effective way to choose the correct subtree of a given Web page is to compare the fanout, the content size, and the tag count of all subtrees in a given Web page. Based on these observations, we develop three individual algorithms in terms of the content size, the tag count, or the fanout of subtrees respectively (see Section 2.4 for further detail). Each of the three algorithms can successfully identify the correct subtree in some Web pages but fails in other Web pages, depending on the types of presentation design of the Web pages. Put differently, some types of Web pages have their primary content regions embedded in the highest fanout subtree, whereas others may have their primary content regions embedded in the largest size subtree or largest tag-count subtree. As a result, each algorithm may identify a different subtree as the minimal subtree of a given Web page. To resolve the disagreement, we let each algorithm produce a ranked list of subtrees and then provide a method to combine them. The combined algorithm compares the three ranked lists and chooses the subtree that has not only higher fanout but also larger content size and larger tag count.

**Step 3: Object separator discovery**

Once the primary content region (i.e., the minimal subtree that contains the data objects of interest) is found, the next step is to decide how to separate data objects from each other, and from any other information in the primary content region. We refer to this task as the Object separator discovery. A main challenge of object separator discovery is to develop practical methods that can fully automate the process of identifying the correct object separator tags. Such object separator tags will then be used in step four to effectively separate objects in the primary content region and extract the objects of interest. By examining a large number of Web pages, we observe that object separator tags often have a number of interesting properties. First, the tag that serves as the object boundary separator in a Web page appears at least the same number of times as the number of objects the page contains. Thus, such object separators can be derived using tag count, sibling tag counts, repeating tag pattern counts, and partial path counts. For example, multiple occurrences of an object separator tag often share the same partial path; thus the appearance count of the partial path can be relatively high. We also observed that the number of times that the object separator tag and its immediate siblings co-appear is relatively high comparing with the appearance count of other sibling pairs.
(see Section 2.5 for more detail). Second, there are a number of tags that are frequently used to identify the object boundaries in various types of content structure of HTML pages, for example, the paragraph separator tag \texttt{p} for paragraph structure, the table row separator \texttt{tr} for table structure, and the list item separator \texttt{li}. To capture these observations, we have designed a set of individual algorithms, each of which independently produces a ranked list of object separators based on tag appearance counts, standard deviation, identifiable tags, partial path count, or sibling count. We also develop a mechanism to combine these independent algorithms into a methodical approach to object separator discovery. We have measured the performance of all the five algorithms over more than 3,200 pages from 75 Web sites. The results show that each of the algorithms performs well for some types of Web pages but fails on others. However the combined algorithm outperforms all individual algorithms for all the Web sites we examined (see Section 2.6 for more detail).

Step 4: Extracting objects of interest from a page

This step consists of two tasks: Candidate Object Construction and Object Extraction Refinement. Candidate object construction is the process of extracting objects from the raw text data of the Web page using the minimal subtree identified in Step 2 and the object separator identified in Step 3. A candidate object is the entire fragment between two adjacent object separator tags. The reason that we use the term candidate object is because there are times when objects extracted using the object separator tag may not be the correct objects. For example, when the Web pages have data object region and advertisement region intertwined, an object separator tag may be used inside an actual object, causing the actual object to be split into multiple candidate objects. The object construction algorithm analyzes the common hypertext structure of the list of candidate objects extracted and determines if any adjacent fragments should be combined into a single object accordingly.

Object Extraction Refinement is the process of eliminating candidate objects that do not conform to the minimum set of structural properties satisfied by the majority of extracted objects. Concretely, in the process of constructing the objects, extraneous objects such as list headers or footers may occasionally be extracted. The first task of object extraction refinement is to derive a set of structural properties that are common for a majority of candidate objects. Then this set of properties will be used as the refinement criteria to prune the extraneous candidate objects. Those objects that are structurally not of the same type as the majority of extracted objects will be removed. Examples
of extraneous objects are the objects that do not have a minimal set of tags which have appeared in many candidate objects, or objects that contain a large portion of unique tags, or objects that deviate significantly from the normal size common to most of candidate objects.

In addition to the functional requirements, there are two non-functional requirements for designing object extraction algorithms for the Web. First, we need to ensure that the algorithms developed for minimal subtree identification are robust with respect to a variety of Web page changes both in content structure and in presentation layout. Second, the algorithms should be able to perform extraction at the Internet-scale. Put differently, all extraction algorithms should scale to the vertical (more sophisticated) or horizontal (geographic coverage) growth of the Web.

2.4 Algorithms for Object-rich Minimal Subtree Identification

The main task of the object-rich minimal subtree identification is to locate the minimal subtree from an arbitrary Web page, which contains all the objects of similar structure in that page. The success rate of the minimal subtree identification algorithms is critical to the accuracy of the entire object extraction process. We have discussed the main ideas and motivation for object-rich minimal subtree identification algorithms in the previous section. In this section we first describe in detail the three individual minimal subtree identification algorithms: the highest fanout (HF) algorithm, the largest tag count algorithm (LTC), and the largest size increase (LSI) algorithm. We also show that, for a given Web page, the highest ranked subtree by one algorithm may not agree with the highest ranked subtree by another algorithm. In other words, each of the three algorithms may successfully identify the correct subtree in some Web pages but fails in other Web pages. Such disagreement happens not only on different Web sites but also on different Web pages of the same Web site. The latter is primarily due to the fact that different search requests to the same Web site often result in different numbers of objects returned in a page. When the number of objects in the data object region is relatively small compared to the rest of the page, it is likely that one of the algorithms may fail in terms of either fanout or tag count or size difference between the subtree and each individual object (also referred to as size increase in short). To resolve this type of disagreement, in this section we also introduce a combined algorithm. It compares fanout, tag count, and size increase of all subtrees and chooses the minimal subtree that has higher fanout, larger tag count, as well as larger content.
size and size increase. Examples are provided throughout the discussion.

2.4.1 The Highest Fan-out Subtree Identification algorithm (HF)

The HF identification algorithm is the simplest of the three. It ranks all nodes of a tag tree by their fan-out and chooses the highest fan-out node as the minimal subtree. This algorithm works well for Web pages that have almost no advertisement or navigational regions. However, many dynamic Web sites today provide more than the search result objects. For example, most e-commerce sites want to provide brand-recognition as well as a consistent and highly evolved look-and-feel for their Web sites. These Web pages are likely to contain several navigational aids and other page elements that may not be directly related to the content of the query results. In such cases, the highest count algorithm does poorly.

Consider the Web page and its tag tree in Figures 3 and 4. The objects that we are interested in extracting from this example Web page are obviously the search results, namely the twelve news items marked by the twelve tables at the right side of the tree. Obviously, the correct minimal subtree is the subtree anchored at the tag node html/body/form[2]. However, by HF the highest ranked subtree is the one anchored at the tag node html/body/form[2]/table[1]/tr/td[2]/font, even though it does not contain any of the news items. The list of top four ranked subtrees by HF and their fanout values are given in Table 1 and Table 2 respectively. Another situation where the HF algorithm may fail is when the number of navigational links is larger than the maximum number of query results displayed on a single page.

<table>
<thead>
<tr>
<th>Rank</th>
<th>HF</th>
<th>LSI</th>
<th>LTC</th>
</tr>
</thead>
</table>

Table 1: Comparing HF, LSI, and LTC on canoec.com tag tree in Figure 4

22
2.4.2 The Largest Tag Count Subtree Identification Algorithm (LTC)

The LTC algorithm is motivated by the observations that data objects typically contain several mark-
up tags and that a subtree of the highest fan-out may not necessarily have the largest tag counts. 
Given a Web page and its tag tree, the LTC algorithm produces a ranked list of object-rich subtrees 
in two steps. First, we rank all subtrees in ascending order by the total number of tags they have. 
Obviously, when comparing a subtree anchored at node u with another subtree anchored at an 
ancestor of u, the ancestor will always have more tags. Hence, in the second step we walk down 
the ranked list and re-examine those subtrees that have ancestor relationship. The subtree that
has the highest appearance count of a child node tag will be ranked higher than the other subtree. Concretely, for each subtree, say $T_i$ in the ranked list, we compute it with every other subtree, say $T_j$, in the list. If $T_i \Rightarrow^* T_j$, i.e., they have an ancestor relationship, then we compute the highest appearance count of the child node for both $T_i$ and $T_j$, i.e., $H\text{ACC}(T_i)$ and $H\text{ACC}(T_j)$. If $H\text{ACC}(T_j) > H\text{ACC}(T_i)$, then $T_i$ and $T_j$ will exchange their ranking positions in the ranked list. Otherwise $T_i$ will be compared with the next subtree after $T_j$ in the ranked list. The process continues until all the subtrees are re-examined.

Recall the Web page in Figure 3. Compare the two subtrees html/body and html/body/form[2] in Figure 4. The child tag form in the subtree html/body has the highest appearance count of 2,
whereas the child tag `table` in the subtree `html/body/form[2]` has the highest appearance count of 13. Therefore, LTC ranks the subtree `html/body/form[2]` higher than the subtree `html/body`. Table 1 lists the top five ranked subtrees obtained by HF and LTC separately. The LTC algorithm ranks the correct minimal subtree as its number one choice, whereas the HF algorithm fails.

Figure 5: nbc.com Search Result http://www.nbc.com – on October 15, 2000

The LTC algorithm implies that the more tags that are in a particular subtree, the more likely it will contain the data objects. However, there are cases in which the LTC algorithm fails. Typically, LTC fails when there is a node that has the largest tag count but it is not on the same tree branch as the correct minimal subtree. LTC also fails when there is a node on the same branch as the minimal subtree, which has a child node that has a higher appearance count than any of the children.
nodes of the correct minimal subtree. Consider the Web page from the nbeil Web site shown in Figure 5 and its tag tree in Figure 6. html/body/blockquote is the correct minimal subtree which contains only the ten search results. However, by LTC the node html/body is the highest ranked subtree. LTC fails in this case because there exists a child node in html/body, which has a higher appearance count of 13 than any of the children nodes of the correct subtree html/body/blockquote (see Table 3 and Table 4).

2.4.3 The Largest Size Increase Subtree Identification Algorithm (LSI)

The LSI algorithm takes a tag tree of an arbitrary Web page and ranks all of its subtrees in two consecutive steps. First, it ranks all subtrees by their content size in byte. Since an ancestor of a
Table 3: Comparing HF, LSI, and LTC on nbci.com tag tree in Figure 6

<table>
<thead>
<tr>
<th>Rank</th>
<th>HF</th>
<th>LSI</th>
<th>LTC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>body/br</td>
<td>body/br</td>
<td>body/br</td>
</tr>
<tr>
<td>3</td>
<td>body/br/table/[7]/[n/td]</td>
<td>body/br/table/[7]/[n/td]</td>
<td>body/br/table/[7]/[n/td]</td>
</tr>
<tr>
<td>4</td>
<td>body/br/table/[7]/[n/td]</td>
<td>body/br/table/[7]/[n/td]</td>
<td>body/br/table/[7]/[n/td]</td>
</tr>
</tbody>
</table>

Table 4: Statistics used in comparing HF, LSI, and LTC on nbci.com tag tree in Figure 6

<table>
<thead>
<tr>
<th>Rank</th>
<th>Format (HF)</th>
<th>Size Increase (LSI)</th>
<th>Tag Count / LSA / LTC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>70</td>
<td>554T</td>
<td>33 / 10</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>1310</td>
<td>71 / 10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>192</td>
<td>23 / 9</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>129</td>
<td>12 / 11</td>
</tr>
</tbody>
</table>

node will always have a larger content size, in the second step we re-examine the ranking order between ancestors and their descendents, and re-rank ancestor/descendant pairs by the difference between their subtree size and the size of their largest child node. LSI promotes the fact that the greater the difference, the more likely the subtree is the minimal subtree. Recall the Web page in Figure 5 and its tag tree in Figure 6. By LSI the highest ranked subtree is `html/body/blockquote`. It is the minimal subtree that contains all the news objects in the page. However, both LTC and HF fail on this example. Table 3 shows the top four ranked subtrees by HF, LTC, and LSI. The statistics we use to determine the LSI rankings are given in Table 4 and Table 5.

The LSI algorithm is motivated by the following observations. First, when the highest fan-out algorithm fails, it usually fails on navigation menus in Web pages, which typically contain only links and descriptive link names. In contrast, the minimal subtree containing the data objects returned from a search is much larger in terms of the number of bytes. Second, the minimal subtree that contains the set of data objects of interest may not have the highest fanout or the largest tag count, but in most cases it will have a much larger size than any subtree of which it is not a descendant. Our experience shows that LSI has the highest success rate comparing to HF and LTC. However, there are cases where HF or LTC succeeds but LSI fails. Typically, when the data objects in a page are small in size and there are relatively too few search results in the page, the size difference between the object-rich minimal subtree and its largest child could be smaller than the size difference of a
Table 5: Statistics used in calculating LSI on nbc.com tag tree in Figure 6

<table>
<thead>
<tr>
<th>Rank</th>
<th>Subtree?</th>
<th>Subtree Size</th>
<th>Largest Child Size</th>
<th>Largest Child Node</th>
<th>Size Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
<td>4</td>
<td>true</td>
<td>8</td>
<td>5512</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
<td>5712</td>
<td>true</td>
<td>4</td>
<td>3518</td>
</tr>
<tr>
<td>3</td>
<td>true</td>
<td>278</td>
<td>true</td>
<td>4</td>
<td>282</td>
</tr>
<tr>
<td>4</td>
<td>true</td>
<td>114</td>
<td>true</td>
<td>15</td>
<td>129</td>
</tr>
</tbody>
</table>

non-minimal subtree node and its largest child.

2.4.4 The Combined Algorithm and Its Performance

We have discussed three individual subtree identification algorithms. They all produce a ranked list of subtrees and then choose the highest ranked subtree as the “correct” minimal subtree. All three algorithms work independently toward the same goal — finding the minimal object-rich subtree that contains all of the data objects. However, each of the three algorithms uses a completely different criterion for subtree identification (such as fanout, tag count, or content size). Consequently, they do not always agree on their highest ranked choice and each of the algorithms is successful for only a subset of the Web pages. One way to improve the accuracy of object-rich minimal subtree identification algorithms is to develop a combined algorithm that finds the best way to combine the three independent algorithms. The goal is to make the combined algorithm successful for a much larger set of Web pages.

A well-known approach for combining evidences from two or more independent observations is to use the basic laws of probability [63]: Let \( P(A) \) be the probability associated with the result of applying algorithm \( A \) over a Web page, and \( P(B) \) be the probability associated with the result of applying algorithm \( B \) over the same Web page. The formula \( P(A \cup B) = P(A) + P(B) - P(A \cap B) \) will produce the compound probability \( P(A \cup B) \) for locating the correct minimal subtree in this Web page. For example, if the probability factors that a subtree is the minimal subtree in a Web page are 78%, 63%, and 85%, then the compound probability for that subtree is 89% (78% + 63% + 85% – 78% × 63% – 78% × 85% – 63% × 85% + 78% × 63% × 85% = 89%). We refer to this compound probability the theoretical maximum success rate.

To combine the three individual algorithms, there are four possible combinations \( \sum_{i=1}^{3} C(3,i) - 28 \).
4 - 4) in addition to the trivial case of none and the three individual algorithms. To determine which of the combinations produces the best overall result for most of Web pages, we need to measure how well a combination algorithm performs. Concretely, we need to know the probability distribution of each of the three algorithms. This can be obtained by measuring the performance of each algorithm over a set of test Web pages. In addition, we need to know both the theoretical maximum success rate and the observed success rate. Then we determine the success rate for each of the four combinations.

Performance Measures of Individual Algorithms

To understand the performance of the three individual algorithms on different types of Web pages, we conducted a series of experiments on over 1,000 Web pages from 25 different Web sites. An empirical probability distribution for the success rate of each individual algorithm is listed in Table 6. Whenever an algorithm is applied to a Web page, it produces a ranking of subtrees in the Web page. For each rank there is an associated probability (success rate). This probability is empirically determined by applying the individual algorithm over the set of test Web pages and calculating the percentage of times the correct subtree appears at each rank for each algorithm.

Table 6: Probability rankings for subtree identification algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Rank 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>0.63</td>
<td>0.32</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>ETC</td>
<td>0.42</td>
<td>0.37</td>
<td>0.17</td>
<td>0.00</td>
</tr>
<tr>
<td>LSI</td>
<td>0.91</td>
<td>0.19</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Performance Measure of the Combined Algorithms

For each of the four possible combinations, a combined algorithm for subtree identification is implemented. To determine which combination is the best in general, we provide both the theoretical maximum success rate and the observed success rate. The former is obtained by using the probability formula, which computes the compound probability for each subtree in every Web page from our test set, based on the observed probability at each rank of each algorithm (given in Table 6). The later results from measuring the success rate of our implementation of the four combined algorithms. Concretely, each of the combination algorithms uses probabilities from the individual algorithms to rank each subtree in the Web page. The probability that a particular subtree is the minimal subtree
is computed by combining the probability assigned to the subtree from each individual algorithm using the probability formula. For a particular Web site, we determine the percentage of times a combination algorithm correctly chooses the minimal subtree from all test pages of that site. Finally, the observed success rate is calculated by averaging the normalized percentage numbers from each Web site in our test list.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Success</th>
<th>Combo</th>
<th>Experimental Success</th>
<th>Theoretical Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>HF</td>
<td>0.63</td>
<td>HT</td>
<td>0.63</td>
<td>0.87</td>
</tr>
<tr>
<td>LTC</td>
<td>0.06</td>
<td>HS</td>
<td>0.31</td>
<td>0.97</td>
</tr>
<tr>
<td>LSI</td>
<td>0.19</td>
<td>VS</td>
<td>0.34</td>
<td>0.97</td>
</tr>
<tr>
<td>HTS</td>
<td>0.36</td>
<td></td>
<td>0.99</td>
<td></td>
</tr>
</tbody>
</table>

Table 7 shows the success rates of all combination algorithms over the test data. To conveniently represent a combination, each algorithm is abbreviated by a one-letter acronym: HF by H, LTC by T, and LSI by S. Thus, HTS stands for the combination of the HF, LTC, and LSI algorithms. While some of the combinations performed equally well on the 25 test Web sites, the combination of all three algorithms is the best choice. This is because each of the algorithms may fail in different circumstances, and using a combination of two algorithms is usually insufficient when the algorithms disagree with one another.

2.5 Algorithms for Object Separator Discovery

After the object-rich subtree identification process, the problem of discovering the object separator tag in a Web page is reduced to the problem of finding the right object separator tag in the chosen minimal subtree. We address this problem in two steps. First, we need to decide which tags in the chosen minimal subtree should be considered as candidate object separator tags. There are several ways of choosing the object separator tags. One may consider every node in the chosen subtree as a candidate tag or just the child nodes of the chosen subtree as the candidate tags. Based on the semantics of the minimal object-rich subtree, it is sufficient to consider only the child nodes in the chosen subtree as the candidate separator tags. Second, we need a method to identify the right object separator tag from the set of candidate tags, which will effectively separate all the objects.

In this section we describe five object separator discovery algorithms, each independently ranks the
candidate tags. Similar to the subtree identification algorithms, these five object separator discovery algorithms may not agree with each other for all types of Web pages. Thus, we also discuss the method to best combine the rankings of these five algorithms such that the combined algorithm will have higher accuracy and succeed in a larger set of Web pages.

2.5.1 The Sibling Tag Algorithm (SB)

The SB algorithm counts pairs of tags that are immediate siblings in the minimal subtree, and ranks all pairs of tags in descending order by the number of occurrences of the pair. For those pairs of tags that have equal occurrence, the ranking follows the order of their physical appearances in the Web page. Table 8 ranks sibling pairs from the Library of Congress tag tree in Figure 1 and Canoe.com tag tree in Figure 4. The first tag of the highest ranked pair is chosen as the object separator. In Figure 1 the (te,pre) tag pair appears before the (pre,a), and thus it ranks higher. The sibling tag algorithm is motivated by the observation that the objects identified by highest count sibling pairs are more likely to be of the same object type than the highest count single tags.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Canoe.com</th>
<th>Library of Congress</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>table cabe</td>
<td>pre,a</td>
</tr>
<tr>
<td>2</td>
<td>tag,bal</td>
<td>pre,a</td>
</tr>
<tr>
<td>3</td>
<td>b,ing</td>
<td>sub, a</td>
</tr>
<tr>
<td>4</td>
<td>sub,table</td>
<td>b, jp</td>
</tr>
<tr>
<td>5</td>
<td>table,mp</td>
<td>sub, a</td>
</tr>
<tr>
<td>6</td>
<td>map,table</td>
<td>a, x</td>
</tr>
<tr>
<td>7</td>
<td>table,form</td>
<td>b, jp</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>b, jpe</td>
</tr>
</tbody>
</table>

2.5.2 The Partial Path Algorithm (PP)

The PP algorithm is motivated by the observation that the multiple instances of the same object type often have the same tag structure. It lists all paths from a candidate node to any other node which is reachable from this candidate node in the chosen subtree, and counts the number of occurrences of each identical path. The list of candidate tags is ranked in descending order first by the count of all the identified paths and then the length of the paths. If two paths have an equal count, then the longer path will rank higher than the shorter one because it indicates more structure. If there are no
paths with a length more than one, such as in Figure 1, this algorithm reduces to choosing the tag with the highest count. Table 9 lists all of the partial paths for the example Web page in Figure 4. The PP rankings for this example Web page and the Library of Congress page in Figure 1 are given in Table 10.

Table 9: Partial paths and their count from the minimal subtree in Figure 4

<table>
<thead>
<tr>
<th>Path</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>table/col</td>
<td>24</td>
</tr>
<tr>
<td>table/col/subtable/col/first/first</td>
<td>24</td>
</tr>
<tr>
<td>table/col/subtable/col/first/second</td>
<td>24</td>
</tr>
<tr>
<td>table/col/subtable/col/first/first</td>
<td>24</td>
</tr>
<tr>
<td>table/col/subtable/col/first/first/last</td>
<td>28</td>
</tr>
<tr>
<td>table/col/first</td>
<td>13</td>
</tr>
<tr>
<td>table/col/first/second</td>
<td>13</td>
</tr>
<tr>
<td>table/col/second</td>
<td>13</td>
</tr>
<tr>
<td>table/col/second/second/first</td>
<td>13</td>
</tr>
<tr>
<td>table/col/second/second/first/first</td>
<td>13</td>
</tr>
<tr>
<td>table/col/second/second/first/last</td>
<td>13</td>
</tr>
<tr>
<td>table/col/second/first/last</td>
<td>12</td>
</tr>
<tr>
<td>table/col/second/first/last/first</td>
<td>12</td>
</tr>
<tr>
<td>table/col/second/first/last/last</td>
<td>12</td>
</tr>
<tr>
<td>table/col/second/first/last/last/last</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 10: Tag ranking from partial path rankings for Figure 4 and Figure 1

<table>
<thead>
<tr>
<th>Rank</th>
<th>tag</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>hr</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>per</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>reg</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>form</td>
<td>12</td>
</tr>
</tbody>
</table>

2.5.3 The Standard Deviation Algorithm (SD)

The SD algorithm measures the standard deviation in the distance (in terms of the number of characters) between two consecutive occurrences of a candidate tag, and then ranks the list of candidate tags in ascending order by their standard deviation. It is motivated by the observation that the multiple instances of the same object type in a Web page are typically about the same size.

Consider the tag tree for the Library of Congress Web page shown in Figure 1. From the subtree identification step, the subtree anchored at the node HTML/body is the chosen minimal subtree. Among the set of child node tags, some tags have much higher counts than the others do. For example, the tag hr appears twenty-one times, the tag a appears twenty-one times, and the tag per occurs twenty times. We refer to these tags as the highest count tags. The standard deviation in distance is calculated between two consecutive occurrences of hr tag, between two consecutive occurrences of

32
pre tag, between two consecutive occurrences of a tag, and so on. Table 11 shows the top three candidate tags by applying the SD algorithm to the library of congress example. It ranks the candidate tags in ascending order by the standard deviation in distance, with the smallest standard deviation first. When an object separator tag is also used for other purpose in the chosen minimal subtree, the effectiveness of the SD algorithm may be reduced.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Tag</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yr</td>
<td>1.18</td>
</tr>
<tr>
<td>2</td>
<td>pre</td>
<td>1.12</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>1.22</td>
</tr>
</tbody>
</table>

2.5.4 The Repeating Pattern Algorithm (RP)

The RP algorithm chooses the object separators by counting the number of occurrences of all pairs of candidate tags that have no text in between. It computes the absolute value of the difference between the count for a pair of tags and the count for each of the two paired tags alone and then ranks the candidate tags in ascending order by this absolute value. The intuition behind this algorithm is that a single tag may be used to mean many things, but a pattern of two or more tags is more likely to mean just one thing. When there are no such pairs of tags in the chosen subtree, the RP algorithm produces an empty list. Y means that the RP algorithm has no answer about which of the candidate tags is the object separator tag. Consider the subtree html/body/form[2] in Figure 4. Table 12 shows a ranked list of all tag pairs in ascending order by the difference between the pair count and the tag count.

<table>
<thead>
<tr>
<th>Tag Pair</th>
<th>Pair Count</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>table, u</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>ang, br</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>img, table</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>form, table</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>h4, tag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>br, table</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
2.5.5 The Identifiable Path Algorithm (IPS)

The IPS algorithm ranks the candidate tags of the chosen subtree according to the list of system-supplied IPS tags. The IPS tags are those tags that are identified by the system as the most commonly used object separator tags for different types of subtrees in Web pages. The idea behind this algorithm is the following. First, we observe that Web pages, generated either by hand or by authoring tools or by server programs, often consist of multiple presentation layouts within a single page, each is defined by some specific type of HTML tags. For example, a Web page may contain a table marked by table tag \texttt{table}, a list marked by the list tag \texttt{ul} or \texttt{ol}, and a paragraph marked by the tag \texttt{p}. Second, each such a presentation layout tends to use regular structure. For example, a table tends to use the row tag \texttt{tr} and the column tag \texttt{td} to define rows and columns of the table; and a list tends to use the list item tag \texttt{li} to define the list structure. Therefore, for each presentation layout (i.e., a subtree type), there are a few tags that are used consistently for separating objects within the subtree.

Based on these observations and the Web pages we have tested, we create a list of object separator tags for each type of subtrees as shown in Table 13. The full list of object separators is composed of all the identified tags for each type of subtree listed in Table 13, with duplicates removed.

<table>
<thead>
<tr>
<th>Subtree</th>
<th>Tag List</th>
</tr>
</thead>
<tbody>
<tr>
<td>body</td>
<td>table, tr, td, blockquote, div, pre, br, a</td>
</tr>
<tr>
<td>header</td>
<td>h1, h2, h3, h4, h5, h6</td>
</tr>
<tr>
<td>footer</td>
<td>dl, dt</td>
</tr>
<tr>
<td>dl</td>
<td>dt, dd</td>
</tr>
<tr>
<td>font</td>
<td>table, h1, h2, h3, h4, h5, a, p</td>
</tr>
<tr>
<td>ol</td>
<td>h1, h2, h3, h4, h5, h6</td>
</tr>
</tbody>
</table>

The next step is to determine the rankings of these commonly used object separator tags. Table 14 lists the distribution of all object separator tags we observed in our tests of Web sites. Based on these experimental results, we produced a ranking of the full list of IPS tags. We refer to such an ordered list as \texttt{IPSList: \{tr, dl, table, p, li, dt, font, ul, br, div, br, h, pre, dd, blockquote, a, span, td, br, h4, h3, h2, h1, strong, em, i\}}. For tags of the same rank, the order is arbitrary.

34
<table>
<thead>
<tr>
<th>Tag</th>
<th>Number of time used as object separator</th>
</tr>
</thead>
<tbody>
<tr>
<td>tr</td>
<td>41</td>
</tr>
<tr>
<td>all</td>
<td>20</td>
</tr>
<tr>
<td>table</td>
<td>19</td>
</tr>
<tr>
<td>pr</td>
<td>10</td>
</tr>
<tr>
<td>fr</td>
<td>8</td>
</tr>
<tr>
<td>fe</td>
<td>6</td>
</tr>
<tr>
<td>f</td>
<td>5</td>
</tr>
<tr>
<td>be</td>
<td>4</td>
</tr>
<tr>
<td>at</td>
<td>4</td>
</tr>
<tr>
<td>dis</td>
<td>4</td>
</tr>
<tr>
<td>br</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>poss</td>
<td>1</td>
</tr>
<tr>
<td>eng</td>
<td>1</td>
</tr>
<tr>
<td>background</td>
<td>1</td>
</tr>
<tr>
<td>s</td>
<td>1</td>
</tr>
</tbody>
</table>

2.5.6 The Combined Algorithm for Object Separator Discovery

We have discussed the five individual algorithms for object separator discovery. Similar to minimal subtree identification algorithms, they may disagree with each other on the highest ranked object separator tags. The basic law of probability discussed in Section 2.4.4 is used to combine them such that a higher success rate can be obtained. We refer to the success rate computed using the probability formula in Section 2.4.4 the theoretical maximum. There are 26 possible combinations ($\sum_{i=0}^{5} C(5, i) = 26$) in addition to the trivial case of none and the five individual algorithms. To see how well each combination does and then determine which of the combinations produces the best overall result, we need to have the success rate (probability) distribution for the five algorithms and to know both the theoretical maximum success rate and the observed success rate of each combination. Table 15 lists the success probability distribution for the five object separator discovery algorithms. The probability distribution for object separator discovery algorithms was obtained by running each of the five algorithms over 1,000 Web pages from 25 Web sites.

Table 16 lists the theoretical maximum and observed success rate of all combinations. To conveniently represent a combination, each algorithm is abbreviated by a one letter acronym: PP by P, SB by B, SD by S, RP by R, and IPS by I. Thus, RSIP stands for the combination of the RP, SD, IPS, PP, and SB algorithms. The theoretical maximum for each combination was calculated by combining the success rates of the individual algorithms involved (i.e., the numbers in Table 15) using the
Table 15: Probability rankings for object separator algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP</td>
<td>0.78</td>
<td>0.18</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>SIB</td>
<td>0.48</td>
<td>0.06</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>SS</td>
<td>0.86</td>
<td>0.25</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>RFF</td>
<td>0.73</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>IPS</td>
<td>0.68</td>
<td>0.20</td>
<td>0.04</td>
<td>0.04</td>
</tr>
</tbody>
</table>

The observed success rate for a combination was determined by applying the combination algorithm to the entire test Web pages. Similar to individual algorithms, object separator discovery, each of the combination algorithms also produces a ranked list of tags with the tag of highest success rate ranked first. The probability for a given combination at a given rank is the average of the percentages of success from all 25 Web sites. The percentage of success for a given Web site was determined by the percentage of times the combination algorithm correctly chose a minimal subtree from all test pages of the site. Based on the numbers in Table 16 and the fact that each of the five algorithms may fail on some Web pages, the combination of all five algorithms is naturally the best choice.

Table 16: Success rates for algorithm combinations on test data

<table>
<thead>
<tr>
<th>Combine</th>
<th>Success</th>
<th>Combine</th>
<th>Success</th>
<th>Combine</th>
<th>Success</th>
<th>Combine</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS</td>
<td>0.80</td>
<td>SSJ</td>
<td>0.84</td>
<td>RSP</td>
<td>0.85</td>
<td>RSP</td>
<td>0.84</td>
</tr>
<tr>
<td>RSB</td>
<td>0.70</td>
<td>RSP</td>
<td>0.86</td>
<td>RSP</td>
<td>0.83</td>
<td>RSP</td>
<td>0.85</td>
</tr>
<tr>
<td>RSB</td>
<td>0.68</td>
<td>RSP</td>
<td>0.88</td>
<td>RSP</td>
<td>0.83</td>
<td>RSP</td>
<td>0.85</td>
</tr>
<tr>
<td>SIB</td>
<td>0.92</td>
<td>RSP</td>
<td>0.95</td>
<td>RSP</td>
<td>0.95</td>
<td>RSP</td>
<td>0.95</td>
</tr>
<tr>
<td>RSP</td>
<td>0.95</td>
<td>RSP</td>
<td>0.95</td>
<td>RSP</td>
<td>0.95</td>
<td>RSP</td>
<td>0.95</td>
</tr>
</tbody>
</table>

2.6 Validation of the Methodology: Experimental Results

2.6.1 Experimental Setup

To run our experiments, we downloaded and cached Web pages from 75 different Web sites. To automatically retrieve the pages we first generated a random list of 100 words from the standard Unix dictionary. Then we fed each word into a search form at each of the 75 Web sites. After retrieving the pages we discarded those pages which returned no results. All experiments were
carried out on the local version of the pages so as not to overload Web sites and to be able to obtain consistent results over time.

For each Web site, example pages were manually examined to determine the path of the minimal subtree as well as all possible object separator tags. The results of the algorithms were compared with the actual minimal subtree and separator tags; the rank that the algorithms choose for a particular subtree or separator tag is recorded for each Web page.

The success rate of an algorithm is calculated in two steps. First, for each Web site we calculate the percentage of the downloaded pages in which the highest ranked subtree for the minimal subtree identification algorithms is the correct minimal subtree. We also calculate the percentage of pages in which each object separator algorithm chooses the correct separator tag. These percentages are then averaged over the collection of Web sites to determine the success rate for individual algorithms and their combination.

2.6.2 Validation Tests

We have discussed the combined algorithm for minimal subtree identification in Section 2.4.4 and the combined algorithm for object separator tag discovery in Section 2.5.6. To validate the effectiveness of our methodology, we ran the algorithms over another 2,200 Web pages from 50 different Web sites. For each of the 2,200 pages, we applied the three algorithms and the combination of the three HTS. Table 17 lists the experimental results of the minimal subtree identification algorithms. The validation experiment indicates that, while the individual algorithms are not extremely stable, it is beneficial to combine them. We also performed validation tests on the object separator discovery algorithms. Table 18 shows the probabilities for the object separator algorithms over the validation Web sites. Our validation tests show that the combined algorithm for object separator tag discovery is at least as good, if not superior, for all Web sites.

Table 17: Probability rankings for minimal subtree identification algorithms on validation data

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Rank 1</th>
<th>Rank 2</th>
<th>Rank 3</th>
<th>Rank 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTF</td>
<td>0.66</td>
<td>0.22</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>LSC</td>
<td>0.61</td>
<td>0.24</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>JSH</td>
<td>0.94</td>
<td>0.06</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>HTS</td>
<td>0.95</td>
<td>0.04</td>
<td>0.01</td>
<td>0.00</td>
</tr>
</tbody>
</table>

37
Table 18: Probability rankings for object separator tag discovery algorithms on validation data

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Rank 1</th>
<th>Rank 2</th>
<th>Rank 3</th>
<th>Rank 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD</td>
<td>0.66</td>
<td>0.63</td>
<td>0.65</td>
<td>0.50</td>
</tr>
<tr>
<td>RP</td>
<td>0.63</td>
<td>0.64</td>
<td>0.60</td>
<td>0.65</td>
</tr>
<tr>
<td>PP</td>
<td>0.53</td>
<td>0.48</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>SB</td>
<td>0.55</td>
<td>0.62</td>
<td>0.60</td>
<td>0.60</td>
</tr>
<tr>
<td>LSPW</td>
<td>0.53</td>
<td>0.50</td>
<td>0.60</td>
<td>0.60</td>
</tr>
</tbody>
</table>

2.6.3 Recall and Precision

In addition to success rate, we also use recall and precision to evaluate how well our algorithms (minimal subtree extraction algorithms and object separator discovery algorithms) perform and the end-to-end performance of the Omni object extraction process, namely the fraction of objects extracted by Omni that are real objects (precision) and the fraction of real objects in a page which are extracted by Omni (recall).

In other words, recall is the percentage of positive instances of the target concept (e.g., the minimal subtree, or the object separator tag, or the real objects) that are correctly identified. Precision is the percentage of extractions made that are correct. Both of these numbers can be defined in terms of false positives (FP), false negatives (FN), and true positives (TP). \[
\text{Precision} = \frac{TP}{TP + FP} \quad \text{and} \quad \text{Recall} = \frac{TP}{TP + FN}.
\]

The set of experiments reported in this section was conducted over 3,200 Web pages from 75 Web sites.

Precision and Recall of Minimal Subtree Identification Algorithms

For the minimal subtree, a true positive is an instance where a minimal subtree exists and it is correctly identified by the algorithms. A false positive is an instance where a subtree is mistakenly identified as a minimal subtree. A false negative is an instance where the minimal subtree exists but it is missed by the algorithms. We evaluated the precision and recall of our subtree extraction algorithms using more than 3,200 dynamic Web pages over 75 popular Web sites. Table 19 shows the precision and recall measures obtained from the experiments. A recall value of 0.99 for the LTC minimal subtree algorithm gives the fraction of the actual minimal subtrees which are identified by LTC with respect to the set of Web pages used for this experiment. Concretely, it means that 99% of the minimal subtrees was identified by the LTC algorithm over the entire set of Web pages used in the experiment. Similarly, the precision value of 0.68 gives the fraction of the identified
minimal subtrees that are correct. Concretely, it says that among the set of Web pages used in this experiment, 68% of the subtrees identified by this algorithm are correct minimal subtrees. There is 32% of cases where the LTC fails to find the correct minimal subtree in a Web page.

Table 19: Probability rankings for minimal subtree identification

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Success</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>HF</td>
<td>0.64</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>LTC</td>
<td>0.64</td>
<td>0.99</td>
<td>0.68</td>
</tr>
<tr>
<td>L3G</td>
<td>0.61</td>
<td>1.00</td>
<td>0.93</td>
</tr>
<tr>
<td>RITS</td>
<td>0.96</td>
<td>1.00</td>
<td>0.96</td>
</tr>
</tbody>
</table>

Precision and Recall of Object Separator Discovery Algorithms

Similar experiments were done for object separator discovery algorithms. Table 20 shows the recall and precision measures obtained. For object separators, a true positive is an instance where an object separator exists and it is correctly identified by the algorithm. A false negative is an instance where the object separator exists but it is missed by the algorithm. A false positive is an instance where a tag is mistakenly identified as an object separator. The recall value of 0.96 for the Sibling tag algorithm (SB) gives the fraction of the actual objects, in the entire set of Web pages used in the experiment, which are extracted by the SB algorithm. The precision value of 0.92 of SB gives the fraction of the extracted objects, over the entire set of Web pages used in the experiment, which are correct. Precision of object separator extraction is bounded by the success rate of minimal subtree extraction algorithms.

Table 20: Probability rankings for object separator algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Success</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D</td>
<td>0.68</td>
<td>0.96</td>
<td>0.68</td>
</tr>
<tr>
<td>BP</td>
<td>0.62</td>
<td>0.96</td>
<td>0.67</td>
</tr>
<tr>
<td>SP</td>
<td>0.69</td>
<td>0.96</td>
<td>0.59</td>
</tr>
<tr>
<td>PP</td>
<td>0.95</td>
<td>0.96</td>
<td>0.85</td>
</tr>
<tr>
<td>SB</td>
<td>0.92</td>
<td>0.96</td>
<td>0.92</td>
</tr>
<tr>
<td>RS/SPE</td>
<td>0.93</td>
<td>0.96</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Precision and Recall of the entire Omni Object Extraction Process

An Omni object extraction process proceeds in two phases: identifying the region and boundaries of objects in a Web page and extracting those objects that are of interest. We have reported the experimental results on the precision and recall measures of the minimal subtree algorithms for
identifying the primary content region in an arbitrary Web page and the object separator algorithms for identifying the object boundaries within a primary content region. Now we report our experimental results on the precision and recall measures of the phase two, where a selection of methods is used to automatically locate the primary content region (the data object region) in any given Web page, separate a primary content region into a list of candidate objects (object construction), and extract the correct objects by cleaning up some extraneous information in between the candidate objects or surrounding around the candidate objects.

The precision and recall measures were conducted over the set of 75 Web sites. We first collect all the data needed for conducting the experiment. For example, we use the following process to generate the reference list of actual objects of interest in a page: For each web site, a representative page was randomly chosen, and the number of actual data objects in the page is manually counted and saved to a file. This file serves as a reference of the total number of real objects in the page, which is then used to compute precision and recall of object extraction process. Now we run the object extraction algorithms over the randomly selected Web page and compare the objects extracted by Omim object extraction with the reference file to determine the number of correctly extracted objects, the number of non-data objects, and the number of data objects containing extraneous information, such as advertisements. From these numbers obtained the precision and recall rates for the page are calculated. This process runs about five times for each Web site and the average for each Web site is plotted in Figure 7. The precision gives the percentage of extractions made that are correct. In other words, it says the fraction of the extracted objects by Omim which are correct (with respect to the reference file). The recall gives the percentage of the actual objects that are correctly extracted by Omim. As observed from Figure 7, Omim obtains an extremely good average recall at 96.7%, meaning that few real objects are missed, and a fair average precision at 84.3% in the experiment over the 75 Web sites. This means that the Omim approach has a very low miss rate, and almost all actual objects are extracted by Omim. However, about 16% objects extracted by Omim are not real data objects of interest, namely the fallout rate, i.e., the fraction of non-data objects which are extracted is still somewhat high.

To highlight different aspects of the experimental results, we plotted the precision and recall values obtained by the Omim object extraction system in two different graphs. The left graph in
Figure 7: Precision and Recall of Object Construction; web sites sorted by precision

Figure 7 plots the precision and recall of each site as a line graph, where the web sites are sorted according to their precision measures. This graph allows us to easily compare the precision and recall of each individual web site.

The right graph in Figure 7 is derived from the first graph. It displays the distribution of precision and recall over all of the web sites. For example, from the second graph we can see that there are about 30 web sites that have precision between 70% and 90%, while only one site has recall between 70% and 90%. Almost all web sites have recall between 90% and 100%, while precision varies over different web sites.

It is interesting to note that in the left graph of Figure 7, other than four cases, where recalls are lower or zero, the recall for the rest of 75 web sites is nearly a perfect 100%. But the precision is not nearly as good as recall. In the 75 web sites tested, Omini often returns (extracts) one or two objects that do not contain any content data. The web pages used in this experiment usually contain four to a dozen of real data objects. Therefore for those pages that have only four or five objects, one or two incorrect objects would reduce the precision to 75% or 50%.

Now let us take a closer look at the four web sites that have lower or zero recall. We see that the two web sites, which have zero precision and recall, are www.edusearch.de and www.quote.com. A main reason that Omini failed in these two web sites is due to the lack of a clean syntactic division between objects in the sense that the tag used to separate real data objects was also used
inside the objects. As a result, all the objects extracted by Omini from Web pages in these two sites contain pieces of the real data objects. The other two Web sites that show the lower recalls are www.powells.com and www.searchking.com. A lower recall usually shows that the Omini extraction system misses some real data objects of interest when extracting objects from the Web pages. By analyzing the hyperlinks structure and presentation of these two Web sites, we observe that their search results are not organized in a uniform manner on a page. Often one data object is displayed in a syntactically nested form, inside of another data object, even though both objects should be viewed conceptually as sibling objects.

These observations show that Omini approach will perform not too well or break for the pages that are not organized in presentation layout according to conceptual semantics of the data objects. Such problems may be due to incorrect formatting or ambiguities in the programs that are used to generate the dynamic the original HTML pages.

In the first prototype implementation of Omini, we made the design choice. We considered returning all correct objects to be more important than filtering out objects that may not contain data or that are false. Therefore, we did not implement any candidate object refinement algorithms that would improve precision and recall. We believe that this type of pruning can also be done in application software that uses the Omini system to provide data objects in Web pages, such as a wrapper system [87, 61] or a search engine for searching dynamic Web content like OminiSearch [24]. Possible object extraction refinement techniques include removing objects that do not have a minimal common set of tags, or removing objects that have too many unique tags, or removing objects that contain no content, or removing objects that deviate significantly from the average object size. Applying object refinement techniques would substantially improve the precision without significant impact on the recall. This is also one of our ongoing research areas within the OminiSearch project.

2.6.4 Performance

We have measured the execution time of both minimal subtree identification and object separator discovery algorithms over 75 popular Web sites. The measurements were taken on a Sun Enterprise 450 server, with four 400-MHz UltraSPARC-II processors and 1 GB of RAM running Solaris 7. The software was implemented in Java and run on the Java HotSpot Client virtual machine (build
1.3.0, mixed mode). The times are reported in milliseconds. We obtain the measurement for each algorithm in three steps. First, the algorithms were run ten times over each page and the execution time per page was calculated by averaging the numbers obtained in the ten runs. In each of the ten runs, the order of evaluation of Web sites and Web pages was randomized to reduce the impact of start-up costs (for example, the JIT compilation of methods in the JVM). Then the execution times from all of the pages from each Web site were averaged to produce an average time per Web site. Finally, the execution time for each algorithm, was determined by an average of its execution times per Web site. Table 21 shows the numbers obtained by running the experiments over the 75 Web sites.

<table>
<thead>
<tr>
<th>Table 21: Execution time in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extraction Style</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Extraction by-page</td>
</tr>
<tr>
<td>Extraction by-site</td>
</tr>
</tbody>
</table>

To understand the performance of extraction algorithms, we measure their execution time from two different perspectives. One perspective assumes that the subtree identification and the separator tag discovery are performed every time when data objects are extracted from a page. Thus, we measure the execution time of the extraction algorithms by considering the time spent for identifying the minimal subtree and the object separator tag plus the time spent for locating the chosen subtree and extracting the actual objects using the chosen separator tag. We call this approach the extraction - by - page method. The other perspective assumes that the minimal subtree and the correct object separator tag, once identified for a given Web site, are remembered for subsequent use.

The execution time in this case only considers the time to locate the chosen subtree in a page using the minimal subtree path chosen previously for the same Web site plus the time spent to actually extract the objects using the separator tag recorded previously. We refer to this method as the extraction - by - site method.

Figure 8 provides a performance comparison of the time for parsing page, the time spent for running extraction algorithms by the extraction - by - page method, and the time spent for running extraction algorithms by the extraction - by - site method, over 75 different sets of Web sites. It
has one anomaly. The pages we cached from the Web site Enzyme.org have a much larger number of data objects compared to the page size than the rest of the sites, some pages contain over 5000 objects. As a result, pages from this site also take much longer to process than pages from other sites.

Table 21 and Figure 8 both show that the execution time of the extraction - by - site method is approximately a half of the time used for the extraction - by - page method, an order of magnitude faster. Another interesting observations obtained from the experiments is the fact that, for the minimal subtree identification algorithms, the correlation between total time and the number of subtree nodes is much stronger than the correlation of time and page size. Readers who are interested in more detailed performance comparisons may refer to [23].

Figure 8: Execution times for Web sites: Web sites indexed by their parsing time

2.7 Comparison with Related Work

The object - extraction methodology described in this paper was motivated by several research results reported in the literature [37, 46, 45].

The WHIRL system [36] uses tag patterns and textual similarity of items stored in a deductive database to extract simple lists or a list of lists (lists of hyperlinks). They present three main methods for identifying interesting structures in a Web page: fruitful, which corresponds with highest fanout; anchorlike, which extracts hot lists based on the similarity of the anchor text to the rest of the object; and R-like, extracting the text in an object that is similar to the type of the text that a user is interested in extracting. They analyzed the success rate of their technique as the percentage of times
the number one ranked structure found by their extraction algorithm corresponded to a structure extracted by a hand-written wrapper from the same page. In batch use, where there is no user interaction to choose the correct page, they achieve between 19% and 70% success rate (for fruitful and anchor-like extractions respectively). Since the Web sites used for experiments reported in [36] were not listed in the paper or the Web site, a direct comparison of the success rate is unavoidably imprecise. It is important to note that [36] reports only the success rate of locating a structured region used in a wrapper, not the precision and recall of extracting actual objects.

Among related research in data or object extraction area, our work was mostly inspired by the initial work on object boundary extraction by Embrey and his colleagues at BYU [45] and the research project on wrapper generation system XWrap at Georgia Tech. However, our approach differs from the BYU approach in a number of significant ways, which play critical roles to make our algorithms to outperform the BYU system.

First, their full system relies on an ontology system to extract specific pieces of information into a predefined schema. In contrast we consider algorithms that transform a dynamically generated page into a list of text objects without the aid of an ontology or an existing schema.

Second, their techniques assume the problem of locating the relevant part of the page. We have identified a suite of characteristics to identify where the data content is in a wide variety of dynamically generated pages, whereas the BYU system assumes that their system can only work with the Web pages whose data object region is the highest fanout subtree.

Third, we have improved the operation of our object separator extraction algorithms to handle a much larger variety of dynamic Web pages with more sophisticated presentation layouts (such as the Web pages in Figure 3 and Figure 5). More concretely, we adopt two of the five object separator heuristics from [45], namely SD and RP, with minor changes. Our IPS algorithm is an evolution of their identifiable tag (IT) heuristic. It chooses tags based on a predefined list of common object separators. We found this to be inflexible when a larger variety of Web sites are considered. Therefore, instead of using the same list of candidate separators for all kinds of Web sites, we use different lists depending on the type of the tag node at which the minimal subtree is anchored. This allows us, for example, to list the tag tr first for tables, it first for lists, table for body tags, and so on.

Our IPS algorithm has much higher extensibility and scalability with respect to the fast evolution of
the Web. We did not include the highest count (HC) heuristic, which ranks tags based on the number of times they appear. The main reason is that our Partial Path (PP) algorithm is much more powerful in identifying correct object separator tags from complex Web pages. Furthermore, for simpler Web sites (such as the Library of Congress pages shown in Figure 1), our PP algorithm is reduced to HC when there are no paths with a length of more than one. We also rejected their ontology match algorithm because it heavily relies on knowing about the domain of a Web site and having a detailed ontology for that domain, whereas the main goal of our object extraction methodology is to develop algorithms that can fully automate the object extraction process. We believe that a fully automated approach is the best way to develop a scalable object extraction system for dynamic Web pages.

For the sake of performance comparison, we have implemented the object separator heuristics proposed in [45]. The hand-written ontology algorithm was not considered since there is no documentation in [45] about how to extract ontology schema about Web pages by hand. Also our objective is to compare the two approaches in terms of the quality of the automated extraction algorithms proposed and in terms of how well these extraction algorithms perform when there is no programmer or user intervention.

We conducted a set of experiments over about 3,200 Web pages from 75 Web sites. Table 22 shows a comparison between our approach and the BYU approach. As observed from the table, for the cases where the BYU approach performs well, our algorithms perform equally well or better. For those cases where the BYU approach performs poorly, our algorithms still perform well, with an average of 95% success rate.

Table 22: Success rates for different heuristics and combinations on all 75 Web sites

<table>
<thead>
<tr>
<th>BYU</th>
<th>Opinion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>HC</td>
<td>0.38</td>
</tr>
<tr>
<td>PP</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SDF</td>
<td>0.11</td>
</tr>
<tr>
<td>SF</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>0.11</td>
</tr>
<tr>
<td>IF</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>0.44</td>
</tr>
<tr>
<td>IF</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SDF</td>
<td>0.32</td>
</tr>
<tr>
<td>SF</td>
<td>0.33</td>
</tr>
</tbody>
</table>

-46
2.8 Discussion

Once the basic object has been extracted from the page, more refined techniques can be applied to identify the internal structure of the objects. This may be done by identifying simple types contained within the text string using regular expressions (to identify numbers, currency, dates, etc.), type dictionaries (days of the week, months of the year, or other well-known enumerated types), or domain dictionaries (proper nouns, product names, etc.). Once simple types have been identified, a partial structure of the object may be determined by the order relationship of the basic types, and any remaining HTML structure.

We differ from the similar work done by Embley and others at BYU in several ways: (1) We are interested in a wider variety of pages than their narrow definition of "data-rich" pages. Thus we have developed minimal subtree identification algorithms to help us discover where data is residing in a page. (2) To extract data from a web page (as opposed to only identifying where object boundaries are), they require a manually developed schema/ontology that describes the domain of data that they are extracting. This allows them to insert data directly into a database, but it severely limits the types of web sites where their techniques will work.

Our work has several contributions. It is a complete system that takes a dynamically generated Web page and extracts data objects from it. This includes locating the data region, separating the objects, and constructing the resulting textual objects from the system. Other systems have concentrated on incorporating domain specific ontologies to assist in extracting objects or data. We have consistently applied techniques that do not require human intervention. This allows the system to be applied to a wide range of web sites without any modification.

We have also released the source code for our system\(^1\) to promote further research into automated object extraction. While we are sure that many companies (such as ExciteJango, or Whizbang! Labs) have developed some type of wrapper system, they keep their code and techniques proprietary. Other research projects, while they have described their techniques in various conferences and journals, have not released the source code for their data extraction systems, at this time.

\(^1\)http://orini.sourceforge.net
Currently we use a simple and efficient technique to combine our basic heuristics. We are looking into incorporating more sophisticated techniques, including machine learning techniques that can improve the reliability of the system. Our main concern at this stage has been to produce a working system that provides a balance between speed and reliability.

**Applications** The Omni system has been an excellent platform on which to build solutions for Web data gathering and integration challenges. It is currently being used in two projects at Georgia Tech, the OmniSearch dynamic search engine, and the XWrap Elite wrapper generation toolkit.

**OmniSearch System** We are developing a search engine for dynamic Web pages based on the Omni object extraction system. The OmniSearch architecture consists of four main components: a search execution engine, the Omni object extraction engine, an autonomous Web crawler to discover and categorize new sites, and a context catalog to store discovered Web sites, their relevant contexts, and associated information extraction rules.

![Diagram of OmniSearch Architecture]

*Figure 9: OmniSearch Architecture*

Users enter a search request, and choose from a list of contexts for their search (e.g., general Web search, book search, news search, etc.). The search execution engine first requests a list of Web sites that match the chosen context for the search from the context catalog. Then, for each Web site it constructs the appropriate URL based on the user search request and the Web site search interface
description. The URIs are passed to the Omini object extraction engine, which retrieves the Web pages and returns extracted objects to the search execution engine for formatting. Results from each Web site are grouped together and sent back to the user. If a page format or query interface change was detected and the object extraction process failed, new rules are automatically discovered and an update is sent to the context catalog.

The final component of OminiSearch system is the autonomous Web crawler. It constantly searches the Web, looking for new sources that have searchable interfaces and are related to the contexts stored in the catalog. In addition, users can suggest static pages to add to the catalog or even define new contexts to incorporate into the meta-search.

Our current research in this area is focused on developing a crawler capable of identifying what type of information is available at a Web site (its domain), such as bioinformatics, general Web search, book store, or e-commerce. This system can immediately take advantage of the service selection techniques described in Chapter 3. The OminiSearch system combines all of these efforts in a Web query system that complements traditional search engines that focus on static pages, such as Google or Alta Vista.

**XWrap Elite** Another application using Omini object extraction is XWrap Elite. XWrap Elite takes a sample page as input, and generates a wrapper to convert the HTML into meaningful XML data through four steps. As a side effect, tagged XML data for the sample page is produced to help validate the process.

First, XWrap Elite uses Omini to transform an HTML page into a list of textual objects. Next, XWrap Elite studies the extracted objects to obtain a group of element separators and then decomposes the objects into elements. Element separators include both HTML tags and plain-text strings. The group of element separators determines an Element Extraction component. Third, XWrap Elite analyzes the elements from the remaining objects to learn element patterns based on regular expressions and element orders, and generates alignment rules to group similar elements into the same location across objects. A wrapper developer inputs tagging rules by assigning an element name to each group. XWrap Elite generates an Element Tagging component according to the alignment rules and the tagging rules. Finally, XWrap Elite packages a wrapper by integrating Omini object...
2.9 Object Extraction Summary

We have presented a methodical approach to fully automating the object extraction for dynamic Web pages. Our methodology has two important features. First, it provides a suite of algorithms that can automatically identify the smallest subtree that contains the primary content regions of Web pages. Second, it employs a set of object separator discovery algorithms that can correctly identify the object boundaries in a page. Our methodology was implemented in the Omini object extraction system. We tested and evaluated Omini in a series of experiments (Section 2.6) using more than 3,200 Web pages from 75 Web sites (primarily electronic commerce sites). The experimental evaluation consists of three parts. First, the result of minimal subtree identification algorithms is examined to determine the number of Web pages where the minimal subtree was successfully chosen. Omini averaged a 98% success rate for this stage. Second, the result of the object separator discovery algorithms was measured to determine their success rate. This process yielded a 96% success rate for Omini. Third, the overall system achieves 100% precision (returns only correct object separators) and 96% recall (with very few significant objects left out). We also replicated BYU’s system [45] without the ontology heuristic (the human-dependent component). Omini compares favorably to the BYU information extraction system, and to the WHIRL structure identification system.

Fully automated object extraction is an important and necessary component in the construction of scalable and robust next-generation information search and aggregation services on the Web. Not only is it useful in accessing data, but it is also useful as a subcomponent of service selection: since many services do not export a complete and accurate summary of their contents, in order to develop a useful summary of a service, data must be extracted and digested from strategic probing queries that will reveal the content of a source. Also, object extraction can be extremely useful in monitoring for changes. Since many services wrap their results in content-rich HTML, including advertisements, navigation elements, and design elements, a data page may change much more frequently than the actual data it contains. By applying extraction techniques to only monitor the data portions, there can be significantly fewer false change notifications.
CHAPTER III

SERVICE SELECTION

Locating and accessing information in the rapidly growing, heterogeneous, distributed collection of data sources available in the Internet is a difficult problem of growing importance [16, 51, 53, 80, 102]. Current estimates place the number of dynamic sources at over 200,000 [14]. This scale requires mediators that can act as a single point of access to help users manage both the enormous number of sources and the variety and quantity of content available.

Given a large number of basic component services, effective composition requires that relevant ones can be efficiently selected. There are two basic approaches to selecting appropriate services for user requests: static and dynamic selection. In static selection, the services that will be accessed for a given group of requests is created offline by a combination of human and computer processing. Users may be able to select the correct category for their search by, for example, navigating a tree of choices to find the most specific and accurate set of services for their request.

Dynamic selection takes an arbitrary query and selects from the currently available services the most appropriate to respond to the request. Selection is based on the context of the query, the query, and each service's capabilities and content. This allows services to be seamlessly added into the selection process without explicit knowledge of each the provider or the consumer of the services; if the service is relevant.

3.1 Introduction

Service selection is a process of directing user queries to appropriate services by constraining the search space through query refinement and pruning irrelevant services. Concretely, effective service selection not only reduces the query response time and the overall processing cost, but also eliminates unnecessary communication overhead over the global networks and over the individual information sources.

We divide the problem into two cooperating parts: false-positive reduction and false-negative
reduction. The former is aimed at reducing useless answers that fail to fulfill user's needs. The latter is targeted at reducing the amount of useful answers that the system fails to deliver to the user. Since a broadly defined query inevitably produces many false positives, the main mechanism to reduce them is by query refinement, which helps a user narrow the query definition to focus on the useful answers. To reduce false negatives we use source selection to help users identify and locate the current set of relevant information providers in a fast changing environment. At the same time, source selection also prunes irrelevant information sources for a user query, thus reducing the overhead of contacting the information servers that do not contribute to the answer of the query.

In the rest of the chapter we first outline the main research issues in service selection for large-scale distributed information systems such as digital libraries. Then we present two complementary techniques and a system design that combines query refinement with service selection.

3.2 Problem Definition

Service selection consists of a hierarchical network (a directed acyclic graph) of nodes, with external information providers at the leaves and service selection components acting as mediating nodes (see Figure 10). Different service selection components are specialized in different domains, such as a bioinformatics component that specializes in genomics, proteomics, and drug design processing.

![Figure 10: Router Network](image)

Before describing different techniques in service selection in detail, we first briefly overview an
Example system architecture from which our prototype is based, and describe how source-capability information can be collected, and how our prototype supports integrated access to multiple heterogeneous data sources. Figure 11 presents a sketch of the architecture.

**Figure 11:** Service Selection Architecture Sketch

A walk through of the architecture in Figure 11 with an example of a simple genomic analysis task follows. To examine a DNA sequence in alignment with similar sequences from BLAST data sources, a scientist must use different tools with three different interfaces and convert the output from each one to a format acceptable to the next. More concretely, the scientist may start with a DNA sequence in a text file, then cut and paste the DNA sequence text into the search interface of a BLAST data source, say NCBI BLAST, to perform a search for similar sequences. The scientist would need to save results and extracts sequence identifiers of best matches manually and feed the sequence identifiers into another Web-based data source, say PDB BLAST, to retrieve full-length sequence text of best matches. Again the scientist needs to save results and convert formats to use a command-line tool in creating a multiple sequence alignment [118].
Using the service selection framework, the scientist first needs to enter the DNA sequence as the search keyword and select the output options such as generate a multiple sequence alignment using the best matches of similar sequences. Once the client manager parses the query, level-one service selection will identify the types of candidate data sources needed to answer this query. The level-two selection will prune the set of candidate data sources based on their query capabilities. The adaptive query scheduler generates corresponding subqueries to data sources selected by the first two levels of selection, and defines the ordering (schedule) of executing subqueries. The level-three selection collects the types of dynamic information needed for relevance reasoning and perform further irrelevance pruning at runtime. The results returned from selected data sources by the runtime will feed into the result filtering and packaging module to perform the final stage of query fusion. The fused query results will be returned to the scientist on the screen or delivered in a file. Other components, such as the crawlers, the source capability profile recognizer, and the wrapper generator are described in Chapter 5.

3.3 Query Refinement

In a large, rapidly evolving network of information servers, there are no expert users because any user's knowledge is quickly out of date. Users will inevitably submit poorly defined queries that produce enormous result sets with many false positives (useless answers that fail to fulfill the user's needs). Such enormous result sets are likely to adversely impact system performance and overwhelm the user with unwanted material. Query refinement refers to any query modification mechanisms that explore and utilize query semantics to reduce false positives [117]. Typical query refinement algorithms rely on collocation of terms. A commonly used approach consists of two basic steps: computing the set of documents that contain one or more terms from the user's query and then suggesting to the user the terms with the highest cumulative frequency over the computed set of documents [126]. A variation is to use the conditional probability of term collocation [117] to compute and recommend terms that are related to a given query. Each refinement iteration offers the user the top $n$ terms with the highest conditional probabilities.

Here we describe an approach to query refinement which uses the user query profiles as a means
to assist a user in formulating well-focused queries (see Figure 12). The main idea is to derive recommended terms based on the semantic context and scope of what the user wants in a particular query, and replace the terms that are too broad in the original query definition with the recommended terms. For instance, in response to a query on "genes" the query refinement step in service selection will derive the following recommended terms: "BLAST" (basic local alignment and search tool), "PDB" (protein database), "genbank" (a well-known repository of genes and unique published keys for genes — GeneIDs). These recommended terms can be obtained either directly from the user's feedback on the query context or derived from domain-specific knowledge (for example, bioinformatics domain ontology).

A significant difference between the user-query profile approach to query refinement and the conditional term collocation approach lies in the ways by which collocated terms are derived. In contrast to the term collocation proposal used in [117, 126], which relies on collocation of terms in the source documents, the query refinement approach, driven by user-query profiles, computes and recommends terms to focus a query primarily based on the domain knowledge of the terms used in the original user query, and thus is independent of the collection of source documents over which the query is posed. An obvious advantage of such an approach is its ability to reduce false positives before the actual run of the query, thus enhancing the efficiency and accuracy of the query refinement algorithms.

![Diagram](image)

**Figure 12:** User-profile based query refinement

Query expansion may also be restricted to terms that are already known, either by terms that
already appear in the service description catalog, or second order terms which have been derived from terms in the catalog via precomputed glossaries, or a general thesaurus, such as WordNet [2]. This works well for ad hoc queries where restricting query expansion eliminates terms which cannot be matched to known sources (and thus do not provide additional value for service selection algorithms). However, for long-term standing queries (such as Continual Queries), restricting the query terms to only those known at a particular point in time limits the value of subsequent executions, as new services may be discovered by the system and known sources may mutate or disappear. To manage these types of problems several different strategies may be employed: promiscuous expansion, where query term expansion and service selection is redone at every execution, conservative expansion, where service selection is done once, and subsequent executions use only services selected in the initial selection process, and hybrid expansion, where query expansion is only done on initial execution, but service matching is done on subsequent executions.

3.4 Schema-based Service Selection

3.4.1 Capability Based Selection

We begin by introducing a motivating example and a predicate metadata model in which user queries, user query profiles, and source profiles are captured. Then we present the design of our service selection algorithms.

3.4.2 Motivating Example

Bioinformatics sources available over the Internet have diverse and yet limited query processing capabilities. Most information servers where data resides (such as PDB, NCBI, or EMBL), only support limited types of selection or similarity queries. This introduces some interesting query processing challenges as illustrated below.

**Example 1** Consider a pharmaceutical researcher who wants to research drugs to combat HIV. To understand the approach the researcher may take to combat this virus, it is important to understand how the virus works.

The HIV virus itself is composed of two RNA strands encased in a protein envelope. The viral envelope has 2 proteins, named gp120 and gpt1. The gp120 binds to CD4, a receptor protein on
a type of white blood cell, called CD4+ T cells. The gp41 then causes the fusion of the HIV with the T cell. After the virus has merged with a cell, the viral RNA is inserted into the cytoplasm of the cell. Each virus particle has 2 copies of its RNA genome, which are transcribed into DNA in the infected cell and integrated into the host cell chromosome with the help of an enzyme called reverse transcriptase. The viral RNA copies itself into the DNA of the cell, causing the cell to produce more of the viral RNA. The RNA transcripts produced from the integrated viral DNA serve both as mRNA to direct the synthesis of the viral proteins and later as the RNA genomes of new viral particles, which escape from the cell by budding from the plasma membrane, each in its own membrane envelope.

One possible solution is to use drugs to prevent the virus from attaching to receptors on cells so that other white blood cells, called killer T cells, can recognize, ingest, and destroy the viral package before it has a chance to infect a new cell. There are many ways of developing such drugs. One common method is experimental: the process is to physically test a compound against a sample of the virus or of a protein that the virus binds to. This process may be labor intensive. In addition, choosing a compound to test is quite difficult because most pharmaceutical companies have a catalogue of several million compounds making an exhaustive search extremely slow, tedious, and error-prone. Another way to develop drugs is search properties of known compounds for promising candidates. There are several techniques that are relevant here. First, a researcher may find all related proteins to a protein known to be involved in a disease process, such as gp120, or CD4. Second, a researcher may want to find all drug compounds that are similar to a drug that affects the protein they are interested in (for example, a reverse transcriptase inhibitor or a drug that inhibits the binding of gp120 to CD4). A third technique is to analytically determine the effect of a drug compound on the protein and the related proteins. Some efforts have been made in modeling how chemical compounds affect a protein based on a computational model of the chemical and protein as well as with comparisons with known interactions between the protein and a similar chemical, or between the chemical and a similar protein.

For more details on how HIV operates, see [http://www.niaid.nih.gov/factsheets/howhiv.htm](http://www.niaid.nih.gov/factsheets/howhiv.htm)
Consider the example query. First, a researcher may use PDB to find the structure for CD4, gp120, or gp41. Then the researcher may wish to find similar proteins to compare structure, function, or related research. To do this, he needs to take the sequence encoding of the proteins discussed above, such as gp120, translate the sequence given by PDB into a sequence suitable for searching other data sources. Such a translation is often done by replacing amino acid names with their single letter encoding. Then he takes this sequence and submits it to multiple similarity matching sources. Examples include NCBI’s BLAST tool—blastp, or one of its many mirror sites and other BLAST sites that are not strict mirrors of NCBI, such as EMBL, DBJ, or KEGG. The BLAST searches will list proteins similar to the one submitted, as well as how the amino acid sequence aligns with each of the similar proteins. Finally, all of the relevant publications for the similar proteins will be gathered, from literature databases such as PubMed. Now the researcher enters the drug design step. First, he needs to understand how chemical compounds can affect the proteins identified. The goal here is to find a chemical that will inhibit gp120 from binding with the CD4 receptor, while minimizing the interference with regular cellular function.

Pharmaceutical companies usually maintain a list of chemical compounds with on the order of a million entries. Through mathematical modeling, how each compound interacts with the physical structure of each protein identified with the function of HIV can be predicted (with varying degrees of success). Examining studies of how the chemical has affected the function of similar proteins is another way of predicting how the chemical will interact with a protein. We can express a query searching for the proteins identified above as well as all similar proteins as $\text{similar}\{1, \{\text{keyword} = \text{"HIV"} \land (\text{protein} = \text{"gp120"} \lor \text{protein} = \text{"gp41"} \lor \text{protein} = \text{"CD4"})\}\}$. This search is fairly complex and cannot be processed by any known source in one-step. Thus, to process this query, our system needs to break down the end-user query into source-specific queries that are executable at individual sites, such as NCBI, PDB, or EMBL. One possible plan is to break the query into the following series of queries. (1) Query PDB: $(\text{keyword} = \text{"HIV"} \land (\text{protein} = \text{"gp120"} \lor \text{protein} = \text{"gp41"} \lor \text{protein} = \text{"CD4"}))$, obtaining the structure of these known proteins; (2) For any protein $r$ from the result of (1), convert $r$ into a protein sequence $r_s$, (3) at NCBI execute a BLAST query for each $r_s$; (4) filter out all results that are not with-in a similarity of 1, as defined by the BLAST similarity

Through this example, we observe two interesting facts. First, the extraction and use of the PDBs and NCBI source profiles plays a critical role in selecting the relevant data sources for the query. Second, even simple selection queries against a single data source across the Internet may have more complications due to the source-specific content and its limited query capability. The situation becomes more sophisticated when we have queries over multiple distributed data sources that are heterogeneous in both information content and their query capabilities.

3.4.3 Metadata Description Model

The metadata description model [86, 85] is designed to be an object-relational model. Typical components of the metadata model are classes, a set of (simple or composite) attributes associated with each class, a class hierarchy described by a subclass-superclass partial order. We use a unary relation to describe each class and a binary relation to describe each attribute.

We model queries with select, project, join, and union operations and the built-in comparison predicates such as \( \leq, <, = \) and \( \neq \). We assume set semantics for queries. For convenience of our analysis, we consider only conjunctive queries. A conjunctive query \( Q \) consists of a head predicate with arguments, denoting the result template, and a body, representing a binding pattern [109] of \( Q \). The arguments of the predicate that are provided as input parameters of the query are expected to be bound. The arguments of the predicate that are produced as output of the query are free variables.

We use lower case letters for variable names and uppercase letters with bars to denote tuples of variables and constants. We describe a conjunctive query \( Q \) by a quadruple \((Q_{\text{body}}, Q_{\text{in}}, Q_{\text{out}}, Q_{\text{cond}})\) where \( Q_{\text{body}} \) is the set of virtual types used in \( Q \), \( Q_{\text{in}} \) is the set of input arguments, \( Q_{\text{out}} \) is the set of output arguments, and \( Q_{\text{cond}} \) is the conjunction of comparison atoms.

A user may pose queries on the fly (without using any pre-defined views or classes). For each user query and the result patterns, we create a set of virtual object types as its result place holder, which describes all the arguments used in the query, including the classes or relations, the data types, the domain constraints, and the usage (as input or output parameter) of the arguments.

**Example 2** Consider online BLAST search sites (sources) for protein sequence similarity, such as NCBI. Suppose we want to search for a similar protein sequence, protein structure, and related
research to the protein gp120, published in 2001 to better gauge the effects of a new drug.

We may express the query \( Q \): find similar protein sequence, protein structure, and related research, for gp120 where publication year = 2001 as a conjunctive query of the following form:

\[
\text{query}(p, a, t, j) = \text{Protein}(p), \text{Literature}(m), \\
\text{sequence}(p, a, t), \text{structure}(p, j), \text{author}(m, a), \\
\text{year}(m, t), \text{title}(m, a), \text{journal}(m, j), \\
g = 2001 \land \text{SIMILAR}(p, \text{'gp120'}). 
\]

\( \text{query}(p, a, t, j) \) is the head of the query, and its arguments protein sequence \( p \), structure \( a \), author \( t \), title \( j \), and journal \( j \) are its distinguished variables. In terms of relational SQL, the distinguished variables of the query correspond to attributes appearing in the \textit{SELECT} clause. The rest are atoms of the body of the query, and are the bounding pattern of the query. Note that the equality predicates in the \textit{WHERE} clause are represented by equating variables in different atoms of a conjunctive query.

The following is the internal representation of this conjunctive query:

\[
Q_{\text{from}} = \{ \text{Protein}(p), \text{Literature}(m) \}, \\
Q_{\text{in}} = \{ \text{sequence}(p, \text{'gp120'}), \text{year}(m, 2001) \}, \\
Q_{\text{out}} = \{ \text{sequence}(p, a), \text{structure}(p, j), \text{author}(m, a), \\
\text{title}(m, t), \text{journal}(m, j) \}, \\
Q_{\text{cond}} = \{ \text{BLAST}(p, \text{'gp120'}) \land g = 2001 \}
\]

The researcher who poses the query does not need to be aware of what information sources are currently available and which data schemas or pre-defined views should be used to access them.

The data independence as such allows the service selection to incorporate newly added information sources seamlessly into the system without affecting the way how queries are posed and how answers are delivered, thus higher scalability is achieved, especially when the collection of information sources available is large and frequently changing.

Before we show how the query is routed to the most relevant data sources, we first introduce the concept of source-capability profiles, which play a critical role in pruning irrelevant data sources.
3.4.4 Source Capability Profile

A source capability profile tells what is in an information source (content description) and what types of services (capability description) are provided about its content. It contains not only the content and query capability description, but also statistics on the local data (for example, the size of relations), availability of the source with respect to the access cost and access authorization, as well as update frequency and capabilities of the source. In addition, each source may export information about itself by giving values to a list of meta-attributes such as FieldSupported (the list of optional fields), Linkage (the URI where the source should be queried), ContentSummaryLinkage (the URI of the content summary of the source). In this section we will focus only on the source category, content, and query capability descriptions, since they are the essential components of the source profile and are used extensively in each step of the service selection process.

The category and content description of an information source describes what is in the information source. The content description of an information source tells us what types of objects are in the source. The category description tells us what type of domain the source data are used for and the IsA categorization of the source. The source category description often contains information that can be used to verify an input (selection) condition or fill an output parameter of a query.

We model the contents of an information source in terms of the object types and the object access constraints that the source objects must satisfy. Each source object type is described by a unary relation. Each access constraint is described using a conjunction of built-in comparison atoms of the form $a \sigma b$ where $a$ is an attribute of a source type and $b$ is a constant drawn from a domain compatible to the domain of $a$. We may view a source content description as a collection of views defined over the source.

The query capability description of an information source tells which types of queries the source can answer about its content. We model the query capabilities of an information source $S$ using capability records, each is denoted by $(S_{in}, S_{out}, S_{cond})$. $S_{in}$ denotes the set of permissible input arguments. $S_{out}$ denotes the set of permissible output arguments. $S_{cond}$ denotes the logical constraint ($\land$ or $\lor$) on the mandatory input arguments.
In summary, we denote each information source by a triplet \((S_{\text{det}}, S_{\text{rel}}, S_{\text{cap}})\) where \(S_{\text{det}}\) denotes the textual description of the category of the source, \(S_{\text{rel}}\) is a set of source relations, each may be associated with some access constraints. \(S_{\text{cap}}\) denotes a set of query capability descriptions, each is of the form \((S_{\text{det}}, S_{\text{rel}}, S_{\text{comd}})\) (see Figure 13).

Consider the query in Example 2. Suppose we have extracted and collected the source profiles of the information servers as shown in Figure 13, among many others. Using the user query profile and the source profiles, we may conclude, without running the query, that some of the sources are obviously not contributing to the answer of \(Q\). For instance, we can immediately determine that Sources 7, 8, and 9 are not relevant to this query, because they are focused on Motif searches and not protein sequence similarity. We can also conclude that Sources 4, 6, 10 are not able to contribute to the answer of \(Q\). However, the reasoning here is more subtle. We are interested only in articles that are published in 2001. However, Source 6 only has articles published before 1965. Source 4 requires a file input format not available from the known output of any of the sources, and Source 10 only provides book information. Thus, we are left with sources 1, 2, 3, and 5. The mechanism used in making such routing decision will be described in Section 3.4.5. Readers may refer to [85, 86] for further detail on how to obtain source profile and service selection techniques.
Figure 13: A sketch of source capability description of example data sources
3.4.5 Service Selection: The Main Steps

The ultimate goal of service selection is to constrain the search space for a query over a large collection of available information sources by reducing the overhead of contacting the information sources that do not contribute to the query answer.

Given a user query $Q$, a user query profile of $Q$, and a set of source content and capability descriptions, we design service selection as a two-phase process. At the query refinement phase, mechanisms are applied to refine the original query into a well-focused query, aiming at reducing the false positives in the query result set and enhancing the quality and the degree of accuracy of the results produced from source selection. In this section, we concentrate on the second-phase of the service selection task – source selection and its two-step selection process. Readers may refer to [85] for the detailed algorithms and further discussions.

**Step 1: Level-one relevance pruning.**

This step serves as the first-round selection which discovers the candidate information sources whose content descriptions are in some ways related to the scope of a query $Q$ (for example, in terms of substring matching or in concept similarity). Other factors such as unavailability of the sources or affordability of the sources should be considered at this step too. For level-one relevance pruning we use the user query scope description of $Q$ and the content and category description of the sources. Source profiles that are redundant (covering the same information from the same source) are also removed at this stage. We call the set of sources selected by this step as target information sources of level-one relevance.

Consider Example 2 query, level-one relevance pruning will prune the information sources whose contents are not relevant to biomedical literature, protein sequence structure, or protein sequence similarity, based on the list of source profile descriptions in Figure 13, among many others. It will find that sources 7, 8, and 9 are not relevant to the query answer.

**Step 2: Level-two relevance pruning.**

This step prunes the information sources that have level-one relevance but do not offer enough query capability to contribute to the answer of $Q$. The decision is made based on the input and output arguments of $Q$, the user query profile of $Q$, and the query capability descriptions of the
sources. The user query capacity description of Q and the source profiles are used in the level-two relevance pruning. We call the set of sources selected by Step 2 as target information sources of level two relevance.

The process for level-two relevance pruning has two phases. In the first phase we prune the information sources of the following three cases:

- (1) data sources that have no input or output arguments, which are relevant to the arguments used in the user query, or
- (2) data sources that have conflict with the interest of the user query (such as the query selection conditions do not match the access constraints of the sources), or
- (3) data sources that have arguments corresponding to the mandatory input parameters of the user query but these arguments can only be used as input and are not included in the list of output arguments of the sources.

The information sources selected in the first phase will be passed to the second phase where more sophisticated pruning is conducted in the process of generating an executable plan of the query. For example, the following two additional cases are pruned accordingly:

- (4) data sources (say S_j) whose output capability are not enough to satisfy the input requirement of the other sources (say S_i) when an inter-site join from S_i to S_j is required,
- (5) Data sources whose mandatory input requirement is higher than the input arguments that the user query provider, and there are no other information sources executed earlier which would have enough output capability to complement such requirement.

Consider Example 2 query, level-two relevance pruning will further prune away Sources 4, 6, and 10 because they are incapable of contributing to the query answer due to the restriction on the scope of query interest (for example, Source 10 is a bookstore not an technical article source or protein database), or the constraints on the list of mandatory input or output arguments of the sources (for example, Source 4 requires a file input), or the conflict of query interest with the access constraints associated with the sources (for example, Source 6 only provides citations for articles published before 1965, whereas the query is interested in only articles published in 2001).
In addition, there is a need to identify and prune mirrored or replicated sources. This step is usually delayed until last to allow for the greatest flexibility in choosing the most effective data source for a particular query. Mirroring and replication pruning are an orthogonal type of service selection. This type of service selection is based on characteristics other than schema, such as performance, trust, or geographic location.

3.5 Metatext Service Selection

Metatext is the text used to describe a service. Used loosely, this includes the service schema, the service description, and third-party descriptions of a service. However, as schemas are already explicitly used in other methods of service selection, our use of metatext excludes them. Metatext searches are particularly useful for locating sources that contain data about a specific topic, but where the general schema and content is unknown.

Metatext for a service comes from two primary sources. The first source is the service provider's through registries (such as UDDI) that describe a service from the providers point-of-view. The second source is external descriptions found on Web pages which link to services. Specialized crawlers can traverse known repositories and extract the metatext for each individual service. Another type of crawler may use search engines to discover sites that point to a given service, and extract the link and descriptive text around the link to add to the list of descriptions for each service.

Using this technique to match searches to services may be difficult because service descriptions may come from many sources: an explicit registry such as UDDI, a Web page containing the invocation interface for the service, or third-party catalogs of services. Other factors also complicate the accuracy of this technique, include the completeness and accuracy of the service description, conflicting descriptions, and mismatch between a user conceptualization of a service and the service providers conceptualization. As trivial example of this conceptual mismatch, consider a user who is looking for car recommendations, while a service provider lists automobile ratings.

There are two distinct phases in using metatext for service selection: discovery and matching. The first phase is indexing descriptions from web pages, and WSDL and UDDI description tags. Crawler gathers Web service descriptions from UDDI registries and from Web pages that host HTML services (query forms). All terms used in the descriptions are indexed using standard
indexing techniques, including removal of trivial words (stop words), and stemming, to conflate similar terms. Note that the list of stop words may be different between WSDL services and HTML services.

The second phase is matching queries to services based on the metatext. One effective technique is to treat the index terms as a Vector space and to map a query into the vector space, based on its terms. Queries may then be matched to services based either on distance within the space, or based on other similarity metrics, such as the well-known cosine similarity metric. In such a scheme, queries may be annotated by users or semi-automated query expansion. This allows more flexibility in finding matches when terms are ambiguous, or non-standard.

3.6 Secondary Service Selection Criteria

While functionality and relevance are the primary types of information used in selecting appropriate services to answer a given request, there are several other types of information that may be considered to further refine the selected services, either in scheduling the requests sent to different systems, or in further pruning potential services. Here we describe non-functional considerations, as well as external service annotations.

3.6.1 External Service Annotations

Service descriptions may be annotated by third parties in a variety of different ways. Service annotations are kept in an extended registry where they may be used to select services based on user preferences, or feedback on the quality of each annotation. Annotations may supplement each part of the standard service description. There are many types of information regarding a service which cannot be provided by the same source as the service. Examples include functional descriptions (such as automatically derived or user supplied schemas for schema-less services), non-functional user-assigned metrics such as trustworthiness or end-to-end response time, as well as third-party descriptions of content, categorizations, and coverage of a service.

Schema and constraints  Schemas may be explicit, as in the case of WSDL, or they may be derived from the service interface and data response, as in the case of HTML services. Derived
schemas are object types that are ascribed to a set of data by a third-party or by an automated process. For example, an automated service probing component may detect a Web service embedded in an HTML page. By randomly generating queries based on the exposed form interface and a library of possible service types, the prober may be able to generate meaningful responses [52, 112]. These responses may be categorized by the library entry used to create the response, and by type recognition components that can identify basic types (numbers, prices, company names, etc.).

Derived schemas are inherently less meaningful than schema’s designated by a service provider, as they cannot recognize all of the semantics inherent in the extracted data. However, in many cases, such as HTML form-based Web services, the service schema may not be published, or may be published in a non-standard format or in an unknown location. In such situations a derived schema may be the only schema available, or it may provide the best available approximation of the actual schema.

Field Cardinality  Field Cardinality expresses the coverage of a service. Cardinality for any particular invocation response is unlikely to represent the total cardinality of the underlying data site. For example, services such as Google and Amazon.com list a subset of results for any one invocation of the service. Additional results, if available, must be obtained by subsequent service invocations. An approximation of the cardinality may be estimated by other means: A service such as Google may return a number (“10 out of approximately 100,000 results shown”) mixed in with the data results. The results from an HTML service invocation may include a navigation bar that indicates a number of pages; the semantics associated with the number of pages of results are rarely more fine-grained than meaning “one” “few” or “many” pages of results.

Anonymous Access  Many services require registration before they can be accessed. To access these sources through a centralized query system, users must entrust their authentication information with a mediator. This allows selective access to restricted sources, while general users cannot have access to those particular sources. In certain cases, the mediator itself may be able to acquire independent authority to access a source and relay query results to users, but this is rarely the designed-for interaction pattern for a service.
3.6.2 Non-functional Service Characteristics

Non-functional service characteristics are those aspects of a service that do not directly relate to the capabilities or content of the service. Examples include response time, volatility, or freshness.

Each non-functional characteristic reported by a service provider may be supplemented by third-party information. Some types of non-functional characteristics may have different values for a number of reasons. Providers and users have different perspective on some directly measurable aspects, such as response time and freshness. For example, users consider end-to-end delay while providers can only measure local response time. Providers may also use a different methodology to arrive at an answer that provides certain benefits to them. In extreme cases, characteristics may be misreported, either maliciously or because the data has not been updated for long periods of time.

**Response Time** Response time is easy to measure for simple request-response services, however, there tends to be a great deal of variance between best and worst case response times. This is due to several factors, some of which are local to the service (server load and contention of local resources), and others are caused by intermediaries or client network and processing load.

One use of response time is to allow adaptation to occur based on the responsiveness of a chosen server. Common adaptation mechanisms, such as the Ginga framework [104], monitor execution to detect when the throughput or response time falls below a given threshold, and then proactively switch to an equivalent service provider. Statistics may be maintained on each service provider and the response time updated as the performance changes over time.

**Volatility** Volatility is a measure of how much a response varies over time given the same input. High volatility is expected from many services where data is presented from underlying dynamic sources, such as stock prices during trading sessions, current weather conditions, current time, etc. Low volatility should be expected from services that present data that changes slowly, such as listing heads of state, or matches for BLAST queries in bioinformatics. This type of volatility models the volatility of real life data. Volatility measurements of a service can either assist in categorizing the type of data it is presenting, or identify when a service is no longer matching the model of the data that underlies the service.

69
Freseness  Multiple services may provide information from the same domain, but at different update rates. Freseness is a measure of how recent, and therefore how accurate and relevant, data from a service is. For example, typical free services that list stock price information provide information at least twenty minutes out of date. Other services will provide less than a minute delay of information, but will charge a fee (either use based, or periodically) for access.

3.6.3 User Annotations

Several service characteristics can only be provided by its users. Service quality can only be measured with respect to the users; one example of service quality is individual and aggregate trust levels. Other characteristics may be reasonably provided by a service, but a user may disagree with either the measurements, or their perceptions of these values. These types of observations can directly annotate services so that other users may choose to accept either the service provided values, or user reported values. Here we focus on those characteristics where users are the sole arbitrator.

Trustworthiness  One critical discriminator between equivalent services is the trustworthiness associated with each service. Well-known, "branded" services such as Amazon\(^1\) or Barnes and Noble\(^4\) have a much higher trust value than relatively unknown equivalent services such as Cody’s Books\(^5\). This type of evaluation of sources is currently under heavy research in peer-to-peer systems. The value of such systems depends on wide-spread trust among many autonomous units. Similarly, the value of any service depends on the level it is trusted to produce relevant, accurate, and non-malicious results.

User and Group ranking  Users grow familiar and comfortable with specific services. This is the essential value of branding, and is closely related to trustworthiness. However, it is not always practical for each individual to develop their own rankings of services. One powerful technique to provide the value of individual rankings without explicit input from each user is recommendation systems. These systems gather feedback from many different users and provide consensus rankings.

\(^{1}\text{http://www.amazon.com}\)
\(^{4}\text{http://www.bn.com}\)
\(^{5}\text{http://www.codybooks.com}\)
3.7 Schema-based Service Selection in Bioinformatics

Modern bioinformatics data sources are widely used by molecular biologists for homology searching and new drug discovery. User-friendly and yet responsive access is one of the most desirable properties for integrated access to the rapidly growing, heterogeneous, and distributed collection of data sources. The increasing volume and diversity of digital information related to bioinformatics (such as genomes, protein sequences, protein structures, etc.) have led to a growing problem that conventional data management systems do not have, namely finding which information sources out of many candidate choices are the most relevant and most accessible to answer a given user query.

There is a huge and growing amount of bioinformatics data residing in specialized databases today, accessible over the Internet. Most of the databases present a public interface with some limited query processing capabilities. The Molecular Biology Database Collection [13, 50], for example, currently holds over 500 data sources, not even including many tools that analyze the information contained therein. The most popular resources include those concerned with protein sequences (such as SWISS-PROT, an annotated protein sequence database, and PIR, the Protein Information Resource), protein structure (such as PDB, the Protein Data Bank), genome data (such as AceDB, a Caenorhabditis elegans database), DNA (deoxyribonucleic acid) sequences (such as EMBL, the European Molecular Biology Laboratory and Gen Bank), motifs (such as PROSITE, a database of protein families and domains, and PRINTS, a compendium of protein fingerprints), and sequence matching (such as BLAST (Basic Local Alignment Search Tool) searches, available at several sites such as NCBIσ , EMBLσ , KEGGσ , DDHσ , and so forth).

It is widely recognized that bioinformatics data sources are extremely helpful in assisting molecular biologists, geneticists, and biochemists to understand the biochemical function, chemical structure, and evolutionary history of organisms. More importantly, they assist researchers to use information collected or generated about the human genome, such as protein sequences, DNA sequences, protein structure, and chemical compounds, to design drugs to prevent and cure disease.

Bioinformatics data sources over the Internet have a wide range of query processing capabilities.

http://www.embl-heidelberg.de
http://www.prosite.ad.jp
http://www.ddh.qimr.ac.jp

71
Most Web-based sources allow only limited types of selection queries. Data from one source often must be combined with data from other sources to give scientists the information they need. Several data integration systems [58, 50, 95, 118, 40] have been created to provide users with integrated access and a single point of contact to multiple, heterogeneous bioinformatics data sources. One of the critical challenges for providing integrated access to bioinformatics data sources is the problem of effectively locating the right information from the right data sources and incorporating newly added capabilities or data sources in answering queries. More concretely, it is widely observed that not all the bioinformatics data sources can contribute to a query at any given time. Thus, it is important to route a query to only those data sources that are capable of answering the query. We refer to this problem as the service selection problem [86].

Service selection is a process of directing user queries to appropriate servers by constraining the search space through query refinement and source selection. Concretely, effective service selection not only reduces the query response time and the overall processing cost, but also eliminates unnecessary communication overhead over the global networks and over the individual information sources.

Service selection is of particular importance to large-scale bioinformatics query systems for a number of reasons. First, popular online systems for searching life sciences data (such as genomic data sources) match queries to answers by comparing a query to each of the sequences in the data source. Efficiency in such exhaustive systems is crucial since some servers process over 40,000 queries per day [95]. Furthermore, resolution of each query often requires comparison to over one gigabyte of genomic sequence data. While exhaustive systems are practical at present, they are becoming prohibitively expensive, even with database indexing techniques. Second, different bioinformatics data sources in differing formats have been set up to support different aspects of genomics, proteomics, and the drug design process. Some of these sources are huge and growing rapidly. Statistics show that bioinformatics data sources are now doubling in size every 15-16 months, and the number of users and the query rates are growing as well [65]. Third but not last, there are growing demands for answering simple key-word or string matching based queries with comprehensive categories of information. For instance, cancer researchers may expect to use an
integrated bioinformatics query system to help identify genes that respond to low-doses of radiation. This problem is difficult because the information required by the scientists is spread across many independent, Web-based data sources, each using their own query interfaces with their own data formats and limited query processing capabilities. How to locate the relevant data sources that are capable of answering a query is critical to the performance of any integrated query system for transparent access to multiple bioinformatics data sources.

Surprisingly, most existing bioinformatics data integration systems [39, 40, 58, 94, 50, 118] do not provide the support for service selection even though some of them offer sophisticated query optimizations. Queries may be routed to data sources that are irrelevant or cannot contribute to the answers. As a result, not only is the response of queries delayed but also the throughput of the data servers is affected, not mentioning the additional network traffics incurred. In this chapter we first present an overview of BioSeek and show how it can be used to integrate access to bioinformatics data from heterogeneous data sources. Then we introduce the BioSeek source-profile based service selection scheme, including the use of source-capability profiles to capture diverse and limited source content and query capabilities, and the multi-level progressive pruning algorithm for locating relevant data sources in answering queries from a large and growing collection of sources. To illuminate our discussion, we sketch several research scenarios that substantiate the need for cross-source queries and service selection based optimization. The main contribution of the chapter is the concept of source-capability based service selection and its multi-level progressive pruning strategy for selecting the most relevant data sources in answering a bioinformatics query. We also report the first prototype development effort and our initial experimental result for the service selection algorithm.

Prototype A prototype of the BioSeek service selection subsystem is currently under testing. Figure 14 shows an example run of the first and second level selection, the query scheduling, and query execution in the first prototype of BioSeek. The source capability profiles used in this version is generated manually. We are working on building the first version of the Bio Crawlers by extending XWRAP Elite toolkit [87]. The source capability profiles are inferred based on the source information collected by the Bio crawlers.

73
3.8 Extended Techniques in Service Selection

Due to the tremendous number of services that are already online, and rapidly increasing number of new services coming online, finding the set of services to satisfy each user needs is becoming harder and harder. There are many proposals for systems to help alleviate the problem, from the generic UDDI registries to logic-based matchmaking to metadata sharing protocols such as STARTS. Each mechanism must solve three basic problems: gathering data about each of the services, creating an efficient internal representation of the data and user queries, and finally selecting data providers that are relevant to each user query.

Data Gathering There are several methods that mediators use to gather data: Provider Sponsored Registries such as UDDI rely upon service providers to list all of the metadata describing the service as well as how to access the service. UDDI is based on standards such as WSDL to provide precise descriptions of how to interact with a service, but standards on describing and categorizing services
have not yet emerged. User Annotated Users or third-party experts can provide significant amount of information about a service, including user-perceived categorization, service schema, service trustworthiness or quality, and responsiveness. This information may either be peer-reviewed using a type of peer-trust mechanism [129], or provided by an authoritative source, such as an e-business ranking service. Automatically Gathered Perhaps the most attractive mechanism for gathering information is to use automated techniques to pull categorization and descriptions of services, in a manner similar to which Google discovers pages which are relevant to particular keywords [103]. This avoids the problems of inaccurate or incomplete data that may be given by either providers or users, but it is difficult to create rich, semantically meaningful descriptions of services without human involvement. At times human intelligence may be co-opted from related efforts of volunteers; perhaps the largest example of this approach is the Google PageRank algorithm which relies on the human-created hyperlink structure of Web sites to imbue enough meaning into the search of important and relevant pages to make it effective.

Data Representation Before data can even be gathered for different services, a mediator must decide on a format that will facilitate the type of algorithms it intends to employ in matching queries to services. Mediators may use several different forms of data representation for different tasks, including: Logical Descriptions In logic-based matchmaking services, such as [82], services are described using concepts from an ontology, described in a language like DAML+OIL (an ontology markup language) [43]. Schema Information Schemas may be described as relational representations, or by hierarchically using the XML Schema language [121]. Data Range/Constraints Arguably part of the schema definition, constraints define the ranges of permissible data in a schema. This type of information is inherently semantic, although loose approximations may be made Word index frequencies For text databases there have been several efforts [54, 52, 98] to describe the documents contained in databases. The most popular method is to list word frequencies for each database, extending documents similarity mechanisms to database selection. Hierarchical Categorization Services may be classified according to a given ontology, similar to the Yahoo!

10http://www.yahoo.com/
Google directories \textsuperscript{11} which provide a hierarchical classification of Web pages.

One promising model, described in Section 3.8.1, is based on the vector space model from classical information retrieval work. This model provides considerable flexibility, and has been shown to be effective in a wide range of environments. This provides some advantages over logic-based descriptions, as it is much easier to create from standard schemas and text descriptions. Due to the inherent fuzzy representation of information, it cannot be as precise in matching queries to sources as well-described logical descriptions and properly formed queries; this is a mixed blessing as it caters to much looser requirements from general users who must create the queries.

Selection Techniques There are a huge number of different database selection techniques based on different types of data pertaining to a source: Logical Subsumption "This database contains information described by the query" Schema "The schema and constraints of the database match the query schema" Synopsis "The database description is similar to the query description" Content Summary "The content of this database is similar to the query" / users may browse content samples to see if the database is relevant. We have already presented techniques based on service schemas and service metatext (which includes synopsis). We present another schema-based technique based on the search space efficient, and can model both legacy HTML form-based services and new services described by Web Services Description Language (WSDL) \textsuperscript{33}, and other formal description languages.

We begin with a review of the traditional vector space model, and then we describe the adaptations required to apply this model to selecting appropriate services for user queries.

\textsuperscript{11}http://www.google.com/dirhp
Vector Space Model

Definition 5 Let $K = \{ k_1, \ldots, k_l \}$ be the set of all index terms, where $l = |K|$ is the total number of index terms in the system. A weight $w_{i,j} \geq 0$ is associated with each index term $k_i$ of a document $d_j$. For index terms which do not appear in the document, $w_{i,j} = 0$. Each document $d_j$ can be represented as an index term vector: $\vec{d}_j = \{ w_{1,j}, \ldots, w_{k,j} \}$.

The vector space model treats documents as sets of terms where each unique term is an orthogonal axis in the vector space. Documents can then be viewed as points in the vector space. This model provides a mechanism to loosely capture the semantics of individual documents.

Each term from a document has an associated weight. Documents can then be measured for similarity to each other or to queries based on the term weights. There are many mechanisms to determine how to assign weights to terms. The simplest mechanism is a binary Boolean weighting scheme, where a term gets a weight of 1 if it occurs in the document, and a weight of 0 if it does not. This allows for fast algorithms that can gather documents related to the terms. The drawback to using Boolean weights is that the relative importance of each term is not considered. This means that if a single term in a query does not occur in a document, it is completely irrelevant (over-aggressive pruning). Likewise, if a document include the query terms, it is relevant, regardless of the importance of the terms in the document (lex prunimg).

A more useful mechanism to determine term weights is the popular $tf \cdot idf$ factor. Term frequency, $tf$, is the ratio of the occurrences of a term $i$ in a document relative to the most frequent term in the document: $tf_{i,j} = \frac{f_{i,j}}{\text{max}_{k=1}^l f_{k,j}}$, where $f_{i,j}$ is the frequency of term $i$ in document $j$. This attempts to capture the relative importance of an index term in the document.

Inverse document frequency, $idf$, provides a factor for how discriminating an index term is for a particular document. A term that appears in all documents is useless for distinguishing between documents in the set, but may be useful for distinguishing between different sets of documents. Inverse document frequency $idf_i = \log\frac{N}{n_i}$ where $N$ is the total number of documents, and $n_i$ is the number of documents containing term $i$. 

77
Document Similarity The similarity of documents is important in many applications. In traditional information retrieval, similarity is used to find documents that are relevant to a query. Documents that are more similar to a query are then judged more relevant. Similarity may also be used to find Web services that are relevant to a particular request, either by matching the similarity of a service schema to the query schema, matching the query content to content indexed from a service, or by matching the description of a service to a query.

There are many ways to measure the similarity of two documents. One example technique is to use cosine similarity. This measures the cosine of the angle between two document vectors $d_j$ and $d_k$, and is computed as

$$\text{cosSim}(d_j, d_k) = \frac{d_j \cdot d_k}{\|d_j\| \times \|d_k\|} = \frac{\sum_{i=1}^{n} w_{j,i} \cdot w_{k,i}}{\sqrt{\sum_{i=1}^{n} w_{j,i}^2} \cdot \sqrt{\sum_{i=1}^{n} w_{k,i}^2}}$$

Service Information Space Since we intend to model services, there are several details that need to be adapted from a text-document based information model. First, there are at least three different classes of information associated with each service: (1) Schema information, both formally described by the service provider and informally obtained by analyzing the results of example queries to the service. Schema information describes input and output messages separately. (2) Metadef description of the service, both from the service owner and from external sources that link to the service. (3) Data content of a service. Each of these classes can be represented as a separate information space that can be modeled by distinct vectors.

Second, mapping in each information space may follow different rules. As a simple example, stop words may be completely different between spaces. In addition, schema index terms may need to be adjusted or expanded to convert strings that represent objects and attributes into semantically meaningful English phrases. For example, schema’s often use contractions such as the fields ToAddress, FromAddress, Subject, and MessageBody from an email service. Index terms in a schema may also be expanded to include more information, allowing the semantics of particular terms to be disambiguated, distinguishing between different senses of a term.

\[\text{Field names taken from a Web service described on the http://www.xmethods.com website, May 2003}\]
Service Description  We describe Web services with seven components: service categorization, input schema, output schema, schema constraints, non-functional parameters, content summary, and a synopsis. This information may either come directly from the service provider, it may be submitted by users and other third parties, or it may be gathered by the mediator. First we describe the full set of information that a service provider may release, then we discuss supplemental information that users may provide beyond what may come from a provider.

Service Type Categorization  Services may be categorized into a hierarchy based on the type of information they provide. Services that target users in a particular domain, such as BLAST search services in the bioinformatics domain, may increase the visibility of their service by providing standardized categorization information consistent with other sources in a domain. Alternatively,

Schema Description  Input and output schemas are logically represented as an object-attribute matrix \( S \). Each row of the matrix represents an object from set \( O = \{o_1, \ldots, o_n\} \) — the set of all object types; \( m = |O| \) is the total number of distinct object types in the system.

Objects are represented as a list of attributes. Let \( A = \{a_1, \ldots, a_m\} \) be the set of all attributes, where \( n = |A| \) is the total number of attributes. A weight \( w_{i,j} \geq 0 \) is associated with attribute \( a_j \) of an object \( o_i \). For attributes which are not a part of a particular object, \( w_{i,j} = 0 \). From these definitions, the schema matrix \( S \) can be represented as

\[
\begin{bmatrix}
  w_{1,1} & \cdots & w_{1,n} \\
  \vdots & \ddots & \vdots \\
  w_{m,1} & \cdots & w_{m,n}
\end{bmatrix}
\]

Each object description for a particular service, \( o_i \), can be represented as an index term vector: \( \vec{o}_i = (w_{i,1}, \ldots, w_{i,n}) \). We represent a particular object using the notation \( \text{object}(a_{r1}, a_{r2}, \ldots, a_{rn}) \) or \( \text{object}(w_1, w_2, \ldots, w_m) \). In addition, two summary vectors, \( \vec{S} \) and \( \vec{S}^3 \), can be derived from the schema matrix \( S \), summarizing the object types and attribute types respectively: \( \vec{S} = (o_1, o_2, \ldots, o_n = \sum_{j=1}^{n} w_{i,j}) \) and \( \vec{S}^3 = (a_1, a_2, \ldots, a_m = \sum_{i=1}^{m} w_{i,j}) \)
Service Constraints  In addition to a description of the schema of the input and output messages to a service, there are constraints on the type of data that a service is willing to provide. For example, phone listing services would only provide information about subscribers in a certain region, area code, or country code. Bookstores may only provide information about books that have been published after a certain date. Genome sequence databases may only provide information about the genomes of a limited set of species.

Constraints may be provided over two types of data: range data (numbers, dates, prices), and categorical data, where data elements come from a relatively small finite set (the days of the week). Constraints are much more difficult to create, process, and maintain for free-form text fields. These types of constraints are equivalent to logical statements that describe the data content, and have the associated problems of consistency, correctness, and understandability.

Data Content  Content descriptions for services are necessarily complex, as content is not directly comparable across all services. The schema for the data output of a service is extremely useful in the description. Not only does it provide some context for the information, but it allows multiple summaries of the content based on the types of different fields.

The simplest content summaries are for numerical types (prices, temperature, sizes, etc.). Standard statistical summaries can be estimated based on known results, such as minimum, maximum, and average.

Text fields are more difficult to summarize as there is no guarantee that the text follows any particular format, such as an article in a newspaper database. Text content can be indexed using common document indexing techniques. This usually includes stemming words and removing stop words ('and', 'or', 'the', etc.). Once the text content has been digested in this manner, an index may be constructed that lists the occurrence of words in the field, such as the one shown in Table 23[1].

3.8.2 Query Description

As our view of services is a message-based interface, a query schema includes both the structure of the request message, and the expected structure of the response. A full query definition includes both

Table 23: A fragment of the content summary for text fields from two search services

<table>
<thead>
<tr>
<th>Word</th>
<th>PubMed Occurrence</th>
<th>Sports Illustrated Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancer</td>
<td>1,487,644</td>
<td>basketball 6183</td>
</tr>
<tr>
<td>prostate</td>
<td>52,130</td>
<td>hockey 6148</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>basketball</td>
<td>993</td>
<td>cancer 55</td>
</tr>
<tr>
<td>hockey</td>
<td>812</td>
<td>prostate 3</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

a query schema (names and types of objects and attributes for input and output messages), as well as input constants and possibly join variables. For schema matching, object vectors representing only the schema of the query are generated. Similarly, the service schema consists of the objects and attributes of the input and output messages of the service.

Query Widening Query expansion mechanisms use a glossary to expand terms to lists of similar terms. Query widening mechanisms, on the other hand, provide a mechanism for extending the number of relevant sources based on the output of previously selected sources. Query widening is a relevant technique when the original service selection produces too few matches, or if the user does not have enough information to create a query that accesses the services that have the information required.

The basic algorithm for query widening is to recursively select services: Given a query, select all services that respond with additional fields that can be used to augment the query; for each such service, create a new query that adds all of the additional fields as optional inputs to the original query; repeat until one branch of the query tree contains all of the desired query outputs, a collection of n leaves contains all desired outputs, or no additional services can be selected.

The query tree thus created becomes a template for a query plan accesses all of the required sources to bring in the required information. This query plan may become quite complex, leading to successive executions of different queries, joins between the results from several different services, and unions of equivalent services.

There are several problems with the algorithm as stated here. First, a straight-forward execution
of the widening algorithm may produce a non-optimal path for gathering all the required services. Second, there is potential for exponential growth of additional services in each round of widening. Third, there will be a significant number of paths in the query tree that will lead to services which are not relevant to the original query, complicating the results. While serious, these problems do not present a fundamental roadblock in developing a useful query widening extension to basic service selection.

3.8.3 Alternative Service Selection Mechanisms

3.8.3.1 Schema Matching

A relaxation of exact schema matching is to select services that are similar to the schema being queried by the user. There are two distinct problems here: defining a useful notion of similarity that can be automatically computed, and rewriting the given query into a form that will extract useful information from the similar Service. Here we focus on the first problem of defining a computable similarity metric.

Applying this technique to selecting services has certain features which may or may not be desirable depending on the type of query/search being performed. Typically, a random query defined by a user will not exactly match any existing service. However, users may be satisfied if their query can be automatically matched with a service similar to their request. For example, Google will sometimes automatically resubmit a query that returns no results, substituting terms that it deems are misspelled.

To accomplish a relaxation of exact schema matching, services are described by object vectors. An object vector is defined by an object name and a list of attributes. For example, consider a book object with attributes title, author, publisher, and price: Book (title, author, publisher, price).

When dealing with multiple sources defining the same object, it is often the case that there are incompatibilities between definitions of objects. To overcome these incompatibilities we introduce the notion of a universal object vector. A universal object vector is essentially a union of all attributes from all compatible objects. For example, a book description from an online bookstore may include attributes such as price or discount, while a library’s book description may contain

82
attributes such as on reserve or checked out. The universal object vector that includes both of these objects would look like `Book (title, author, publisher, price, dis-
count, onReserve, checkedOut).

A universal object vector describes sources by a weighting vector. The bookstore would be
described as `Book (1, 1, 1, 1, 0, 0, 0)`, while the library would be described as `Book
(1, 1, 1, 0, 0, 1, 0)`. As new sources are recognized by the system, attributes that do not
exist in the universal object vector are appended. This allows existing sources to keep their current
weighting by padding the weight vector with zeros.

This weighting from the universal object vector manages the addition of different attributes
between different sources, but there are at least three semantic differences that are not handled:
attributes with different names that mean the same thing; attributes with the same name which mean
different things; single attributes in one service are multiple or nested attributes in another service.

While these are real and important problems, they are addressed by mechanisms external to the
universal object vector. For example, attributes in a local service that have a different meaning than
attributes in the universal object vector are mapped into a virtual object vector compatible with the
universal object vector.

In general, objects may contain attributes that are also complex objects, termed nested objects.
In our description, nested objects are flattened; each attribute name is replaced by the path from
the root to the attribute. For example, author may be a complex object consisting of first name and
last name: `Book (title, author/firstName, author/lastName, publisher, price)`. Objects are assumed to be acyclic and finite (nesting is limited).

In this scheme, queries are also described by vectors. There are two major types of query
interfaces: typed, message-based interfaces; and keyword-based interfaces. Common Web queries
are typically simple keyword-based queries, possibly with a number of topic or category selections.
For example, the Google query interface allows simple keyword queries that can be modified in the
generalization page to only select pages of a particular language. Amazon's default query interface is a
simple keyword query and a selection of which "store" to do the search in (all stores, books, DVD's,
electronics, and so on). Keyword-based interfaces will typically match any object field.
Typed message-based interfaces are common in Web services described in Web Services Description Language (WSDL [33]). Queries are interpreted as an input message for a particular binding, and the query results is the returned output message. Legacy HTML-based form queries may also be typed. For example, the NCBI BLAST query page \(^4\) has an interface that provides a field to enter a protein sequence, and another field to select the database to search against. The results of the search are homologs of the entered protein sequence (similar sequences), showing the similar section, the similarity score, and several other fields describing the match.

To include requirements for mandatory input and output attributes, weights are assigned to attributes. The default weight for an attribute is 1. Weights for required attributes are \(10^{|\text{size of vector}|}\), where \(|V|\) is the size of the vector. This allows all matches with a similarity of less than \(k \times 10^{|\text{size of vector}|}\), where \(k\) is the number of required attributes, to be pruned, since they do not include the required attributes. When dealing with universal object vectors, weights may also be 0 if the source does not contain an attribute from the universal vector.

### 3.8.3.2 Content-based Service Selection

A second technique for selecting sources is based on the content of the sources as opposed to their explicit or derived schema. Content modeling is especially useful in unstructured sources, such as document databases, or Web page search engines. Several research efforts [25, 83, 54, 37, 52, 66, 98] have explored the idea of selecting specific document databases to forward user queries to based on the relevance of the query to the entire collection. These methods are based on collecting document frequencies for a list of terms, and term weights for each document. Most of 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 [52], estimate the contents of text databases through probing queries. For our purposes, we will assume this information is known, and ignore the issue of whether it was volunteered by the source or estimated.

Each database can be represented by \(N\), the total number of documents in the database, and a set of triples \([f_i, w_i, \sigma_i]\) where \(f_i\) is the number of documents in the database that contain term \(i\), \(\sigma_i\) is the average weight of term \(i\) for the set of documents containing the term, and \(\sigma\) is the standard

\(^4\text{http://www.ncbi.nih.gov/BLAST/}\)

84
deviation of the weight of term $i$.

Individual databases may be considered relevant to a query based on the similarity of the query terms to the terms in the databases. For example, consider a query containing a single term $t_i$ corresponding to term $i$ in a database $D$. The relative frequency of the term in the database can be modeled by the value $\frac{f_i}{n}$. This value can be used to rank each database, but is unlikely to be accurate because it does not consider the relative importance of the term in the database. By using the term weight, databases may be filtered by the importance of the term to documents in the database. This may either be done by a threshold [83], or by selecting a given number of databases [66]. Each mechanism has certain advantages depending on whether the databases are organized into a hierarchical structure and whether the values are exact or estimated.

The mechanism for content modeling and service selection based on content models is complementary to schema-based selection. Some sources do not have schemas, or the schemas are not provided and are difficult to automatically infer; for these sources schema-based selection is unavailable and another mechanism, such as content-based selection, must be used. Other sources may have a schema, but schemas alone may not provide enough discrimination to reduce the number of services relevant to a particular query. In this case, content selection can help reduce and rank services by the probability that they contain data relevant to the query.

Content models are typically constructed by sampling instances from a service and building up an index of data returned from a set of searches. There are several methods to choose from when determining what queries to send as a sample set: First, query terms may be selected from a particular domain. For a general search interface, the set may be a random selection of words from a dictionary. For a domain specific search, terms may be drawn from descriptions of a domain, or sample data values in the domain. For example, search terms may be drawn from a list of author names for a book-related site. A second method is to randomly select query terms submitted by users. This has the advantage of capturing what users are querying about. Assuming that query patterns have a power law distribution, the technique should eventually capture the majority of queries for non-volatile Services. This has the disadvantage that there will be a large number of relatively obscure search terms which will not be indexed.

85
3.8.3.3 Data type matching

Services that do not present an explicit schema, such as legacy HTML services, may nevertheless have an implicit schema that can be derived from its output. These are specific content models that typically encode type information as well as the range of the types. Types may be simple constructs, such as strings, integers, or dates, or they may be more specific, such as a sequence alignment in a BLAST search result, or a publisher in a book search result.

Example 1: Rapid advance with little standardization has occurred with bioinformatics data in the past, though there are significant standardization efforts via centralization of most new data in places such as NCHI and EMBL. However, there are still several tools that have not been following the standardization. Typically these tools are generated by short-term research projects at universities or research labs, and made public. Since there is no long term support for these tools they inevitably fall behind standardization efforts. Since the data types output by these tools remains constant, it is easier to identify tools that produce the correct type of output as opposed to a uniformly named output.

Example 2: In publishing the lack of standardized terms is typically not a problem, while there is significant variety of terms between Services, there is the standard Dublin Core15, and variations usually take only a limited number of forms. Books are recognizable as a combination of “Author”, “Title”, “Publisher”, and “Price”. These translate into loosely defined data types of Proper Name, String, Company Name, and Price. In this example, Proper Name is not a strictly defined type (such as Number, or DNA sequence), but a type drawn from a list of samples (a name dictionary) and a probability. The probability must be attached to instances of these types as they may not be primarily used in the same context across different examples. For example, the book “Nixon by Iwan W. Morgan contains a title with a valid entry from a proper name dictionary. In other cases, a book may not contain any author information, either because the work is unattributed or because the data is missing from the description. Matches to the book type may range from exact: containing exactly the fields that are required for a book with no ambiguity, to possible: containing some information required for a book, and possibly in a book context.

15http://www.dublincore.org
The advantage of relying on data type matching is selecting services is that there is no conflict in field names, however, field layout and meaning are still not solved. This mechanism should select a large number of services that loosely fit requirements. The disadvantage, in addition to not considering the semantics of the recognized data types, is that the technique is not specific enough for generic query interfaces as it does not remove many candidate sources for simple queries.

3.8.4 Discussion

All of the service selection mechanisms may be done interactively, allowing users to describe their queries until they are specific enough to select sufficient services to answer the query, but not so general as to return more results than are useful.

Another technique that can be applied from traditional document searches is the concept of ranking queries. As discussed in Section 3.6, there are several characteristics that may make one service more desirable than another service with equivalent functionality. By ranking services according to these characteristics, initial query results may be retrieved from the highest ranking services, providing results until the user is satisfied by successively sending the query to lower ranked services.

3.9 Experiments

The analysis of service selection techniques poses several interesting challenges. This type of system is nontrivial to model analytically or through traditional simulation based methods. Accordingly, our experiments focus on real data sets and the inherent complexity involved in analyzing the results. To illustrate this idea, we observe that service selection works well when services export data with schemas from the same namespace, and use the same naming conventions for similar concepts. However, in the heterogeneous Web environment, different service providers create services with a multitude of schemas, naming conventions, languages, and concepts. Some differences are obvious (for example, the salary difference between an American and a European job), while other differences are much more subtle (for example, the salary difference between a job in a rural town and New York city). Each type of difference brings challenges for detecting which services are relevant.
In this section, we present three selection experiments over three different data sets. The selection experiments are designed to measure the efficacy of schema-based selection and metatext selection in the most probable scenarios. We also demonstrate the performance of the prototype system.

**Experimental Data** For experiments in service selection, we have gathered data from three different sources: the XWrap wrapper repository at Georgia Tech\textsuperscript{16} ; WSDL services advertised at the XMethods UDDI registry\textsuperscript{17} in June of 2003; and a list of HTML forms collected by BrightPlanet\textsuperscript{18}.

The XWrap wrapper repository contains wrappers that were generated using the XWrap Elite semi-automated wrapper generator toolkit. These wrappers were generated by a wide variety of people, from casual Web users to developers creating wrappers for specific applications. This gives the data set a wide variety of subjects, but it also means that many of the wrappers lack any semantically meaningful information. From a data set of over 1200 wrappers, we eliminated all of the wrappers that did not have any human-added schema information (presumably those developed by casual Web surfers), reducing the data set to just over 500 wrappers. Of these, we identified approximately fifty categories, shown in Table 24. This data set displays several interesting features. First, schemas are not consistent between wrappers. In any category, it is unusual to have more than three or four wrappers with the same schema. Second, even though some wrappers are over the same data source, their wrappers are rarely identical. Third, there are numerous misspellings (for example, “title” for “title”) and many wrappers use schema terms from a non-English vocabulary. Finally, the categorization displays a Zipf-like distribution; a few categories are extremely popular, but most categories have fewer than four examples.

The WSDL data set comes from a public UDDI registry from a Web site promoting the development of Web services. These wrappers were generated by a wide range of developers to demonstrate and learn this particular technology. While the services have a much more formal schema than the XWrap wrappers, they portray a heavy bias toward programming language style naming

\textsuperscript{16}\url{http://www.cs.gatech.edu/projects/dsid/wrapperrepository/index.html}
\textsuperscript{17}\url{http://www.xmethods.com}
\textsuperscript{18}\url{http://www.complextaplanet.com/}
Table 24: Categories of XWrap Wrappers

<table>
<thead>
<tr>
<th>book 1</th>
<th>x-counter 6</th>
<th>with search 46</th>
<th>action 27</th>
<th>person 25</th>
</tr>
</thead>
<tbody>
<tr>
<td>conference 21</td>
<td>rdf 15</td>
<td>job 14</td>
<td>rest url 13</td>
<td>stock 17</td>
</tr>
<tr>
<td>weather 10</td>
<td>stats 3</td>
<td>news 8</td>
<td>school 7</td>
<td>fitness 7</td>
</tr>
<tr>
<td>toy 9</td>
<td>result 6</td>
<td>Tank 9</td>
<td>vue 4</td>
<td>hardware 9</td>
</tr>
<tr>
<td>german 5</td>
<td>canvas 4</td>
<td>video 4</td>
<td>software 4</td>
<td>quote 4</td>
</tr>
<tr>
<td>foundry 8</td>
<td>sports 3</td>
<td>food 3</td>
<td>car 3</td>
<td>smart 3</td>
</tr>
<tr>
<td>author 3</td>
<td>harms 3</td>
<td>text 2</td>
<td>schedule 2</td>
<td>vegetarian 2</td>
</tr>
<tr>
<td>cs 2</td>
<td>time 2</td>
<td>stock 1</td>
<td>stock 1</td>
<td>spend 1</td>
</tr>
<tr>
<td>picture 1</td>
<td>race 1</td>
<td>race 1</td>
<td>magazine 1</td>
<td>location 1</td>
</tr>
<tr>
<td>case 1</td>
<td>company 1</td>
<td>business 1</td>
<td>benefit 1</td>
<td>terms 1</td>
</tr>
<tr>
<td>author 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

conventions. For example, a currency exchange rate service may have an input object called “get-
tRateRequest” consisting of two fields, “country1” and “country2” and an output object called “get-
tRateResponse” consisting of the field “result”. There are also a large number of login/registration
style services that form part of a larger interaction pattern for retrieving data from a particular ser-
vice. Metatext for these services are derived from the description tags in the UDDI registries and
the WSDL descriptions.

The HTML forms from BrightPlanet consist of a set of URIs and a categorization for each URI. The
BrightPlanet website claims to index over 100,000 different forms. We retrieved a fraction of
these sources: just the first page listing for each category, collecting approximately 7000 different
URIs. Of these, approximately half were inaccessible (for example, because of a 404 HTTP error
indicating that the page no longer exists). This data set has no externally supplied schema, meaning
service selection must be done using metatext or content based mechanisms rather than schema
selection.

3.9.1 XWrap Data set

The first experiment compares the accuracy of schema-based selection with metatext selection over
the XWrap Elite wrapper data set. For this experiment, the schemas for each wrapper were collected,
and all semantically meaningless parts of the schema were discarded. For example, a schema may
have ten fields, but seven of those fields may have a default name, so the extracted schema would
have three fields. The metatext is the context of each wrapper, in this case the HTML page con-
taining the form for the service. The metatext selection algorithm used Boolean matching where a
service was considered relevant if all of the query words appeared in the wrapper context.

Queries for schema selection were created by selecting a service description and converting it into a query profile. Queries against the metatext of services were created by selecting random words from the context of one service. The query words were restricted to be non-unique (so they occurred in more than one service context) and at least three letters in length.

Figure 15: Selection Accuracy for Varying Query Length

Figure 15 (left) shows selection accuracy for queries over the XWrap data set using schema selection with a varying number of terms per query. Accuracy dramatically improves as the number of terms per query increases. The graph shows a leveling off effect after seven terms as most source profiles had fewer than seven meaningful terms in the data set. This result points to a possible optimization, where services are indexed by the field names in their input and output objects, and services are chosen based on the most selective mandatory query field first, quickly pruning...

90
the search space. Compared with simple metatext selection, this technique offers an average 10% improvement, in addition to guaranteeing that the selected services are capable of responding to the query.

Figure 15 (right) displays the average accuracy of metatext selection queries for different query lengths. Small queries using fewer than four words have extremely poor accuracy as they lack enough terms to sufficiently distinguish the meaning of the query. While this may at first seem alarming, as typical search engines report that the majority of the queries received fall in this size range, there is good reason to believe that users interested in a more sophisticated query response will provide more context than a simple keyword, either through more sophisticated queries or through user profiles that provide a context for a search.

![Figure 16: Selection Accuracy distribution](image)

Figure 16 displays the accuracy range for schema and metatext selection queries using 10 query terms. Ten terms was chosen for this experiment as it provides the maximum accuracy, as seen in Figure 15, while striking a balance between simplicity and complexity. The majority of cases have a perfect accuracy rating in this experiment. However, given the data set, this result is not as surprising as it first appears. First, a large number of services are members of a category containing only a few instances. Second, there are a few services in a non-English language. Often, these services are coincidentally in the same category (likely because the same developer created the services in a single area of interest). Finally, the accuracy is measured at the category level, meaning that while the selected services may match the category, in the case of metatext selection techniques, they may

91
be incapable of responding to all parts of the query.

3.9.2 Invisible Web

The third experiment measures the effectiveness of metatext selection for general HTML form-based services. This experiment is run over the invisible web data set. The experiment measured the accuracy of metatext selection by comparing the services selected by the metatext selection methods with the labels for the services provided by the CompletePlanet service from BrightPlanet\(^{19}\). The metatext for the services is taken from the HTML context in which each form is contained. Queries for this experiment were generated by selecting a number of terms from each form context. Terms were restricted to be non-unique (appearing in multiple contexts), to consist only of letters (A-Z), and to be at least three characters long. A correct match is determined if the classification of the page from which the query is derived substantially matches the classification of the matching page. The graphs in Figure 18 show the accuracy for selection using three different thresholds to determine whether a match is correct. The results indicate that metatext selection is broadly accurate, but somewhat imprecise.

\(^{19}\)http://www.completeplanet.com
3.9.3 Performance

We also report on two performance experiments. Performance measurements were taken on a Sun E450 server, with 4 400-MHz UltraSPARC-II processors, and 1 GB of RAM running Solaris 7. The software was implemented in Java and run on the Java HotSpot Client virtual machine (build 1.3.0, mixed mode). Each experiment was run 20 times and the results were averaged to mitigate the impact of start-up costs such as just-in-time compilation of methods, and other execution anomalies such as garbage collection.

In executing queries over external web sources we have not noticed a strong correlation between the time taken to retrieve results from a web site and the number of results retrieved. Even sites that return no results for a particular query may take as long as a site that returns the most results. There are two reasons for this observation. First, the current prototype only retrieves the first page of results from any source, and does not try to exhaustively retrieve all results for a query. Second, external network costs dominate query execution time for most applications. Execution costs for a single example source in the current implementation averages between two and four seconds; as noted in Chapter 2, we have found that extracting data from a page after it has been retrieved averages only 100 milliseconds. The execution times for queries to individual web sources were randomly sampled from a set of actual execution times taken from the working prototype.

In the first experiment, we vary the percentage of sources that are categorically relevant and
completely relevant at the schema level. The first figure on the left side of Figure 19 shows the execution time required for querying between 20-1000 sources, where 30% to 40% of sources are irrelevant and ten threads are used to retrieve data in parallel. Query fusion for integrated access without selection performs the worst. Query fusion with level-two selection performs better than query fusion with only level-one selection. Note that in our selection scheme, level-two selection is built on top of level one selection results.

In the second experiment shown on the right side of Figure 19, it demonstrates that arbitrarily increasing the number of threads servicing a single query suffers from the law of diminishing returns. Using 20 threads rather than 10, execution time without selection is nearly halved. However, using 70 threads is only 14% faster than using 60 threads. This is even more pronounced when selection is used, because the number of selected sources is fewer, and they are covered by available threads more quickly; additional threads do not speed up the process.

5.10 Related Work

Recent standardization efforts have been aiming to making service selection and automated service composition easier. The most basic service standard that may assist in this endeavor is WSDL [33], which provides a uniform mechanism to describe how to invoke services, including the invocation mechanism (such as an HTTP request to a specific URI), and the arguments for the invocation. While this is an important component, there are two key shortcomings of this standard: it provides
no mechanism to discover the WSDL descriptions, and there is no standard to define how each
service is defined, so individual developers are free to choose to name methods and parameters of a
WSDL described service in any manner that is convenient for them.

UDDI [123, 122] attempts to address the problem of discovering services by providing a cen-
tralized registry for different services to be listed in. The UDDI definition is focused on a business
service naming registry; it only tells who is providing the service and where to get the (WSDL)
specification of how to interact with the service. While the UDDI registry provides a good starting
point for gathering information about various services, it has many shortcomings which prevent it
from solving the service selection problem in all but the most trivial of cases. Descriptions embed-
ded in the registry are typically short and do not provide adequate information with which to invoke
the service, rather they are non-technical sections of text. The tModel, which specifies the taxonomy
of a particular service, is the most promising portion of a UDDI description as it allows for a UDDI
entry to contain a limited taxonomy of a service. However, this portion is the most ignored portion
in current UDDI registries (for example, the XMethods20 , Microsoft 21 and IBM 22 registries
have entries which typically ignore this concept).

To overcome limitations as described above, several people have proposed an ontology-based
approach to service description, forming the technical basis for what is commonly known as the
"Semantic Web". The most prominent of the approaches so far is the DAML+OIL [43] ontology
system, and the DAML-S [6, 7] Web service description language. DAML+OIL is a language that
allows concepts to be defined in relation to each other (equivalence, part-of, generalization-of, and
so on). DAML-S uses this language to describe services in three parts: the service profile, the
service model, and the service grounding. The service profile describes "what the service does"
including the information required to invoke the service and the type of information that it provides,
and actions that it may perform. The service model describes how a service works, including all
of the steps that must be performed, and in what order, in order to invoke the service. The ser-
vice grounding defines how a service is invoked (specific protocols to be used, specific address to
communicate with), and is often a mapping to a WSDL description of a service.

20http://www.xmethods.org
21http://uddi.microsoft.com

95
One drawback to the semantic web approach is that it assumes that all information will eventually be described in the complex language that is defined by the proponents, and that the ontologies that each service definition will reference will be compatible. This ignores two trends that are common in the current generation of Web services, including the HTML-based services and those that use WSDL and UDDI registries. First, people will use the minimum necessary amount of technology and description to get a service functioning. While some may argue that functioning requires a full semantic description, this definition does not account for the huge number of existing services that work well enough to accomplish the goals that their authors have. Second, any organization, or group of people, tends to create semantic descriptions that fit their own organization and goals. Standardization in schemas has not been a goal of most organizations. This is evidenced most strikingly by examining database schemas from different organization, and the tremendous research efforts (such as [77, 100, 108]) in database integration.

3.11 Service Selection Summary

The nature of scientific research and discovery leads to the continuous creation of information new in content or representation or both. Despite the efforts to fit molecular biology information into standard formats and repositories such as the PDB (Protein Data Bank) and NCBI, the number of databases and their content has been growing, pushing the envelope of standardization efforts such as mmCIF [127]. Providing integrated and uniform access to these databases has been a serious research challenge. Several efforts [39, 58, 50, 95, 118, 46] have sought to alleviate the interoperability issue, by translating queries from a uniform query language into the native query capabilities supported by the individual data sources. Typically, these previous efforts address the interoperability problem from a digital library point of view, treating individual databases as well-known sources of existing information. While they provide a valuable service, due to the growing rate of scientific discovery, an increasing amount of new information (the hot-off-the-bench information that scientists would be most interested in) falls outside the capability of these previous interoperability systems or services.

In this thesis, we address the problem of providing automated or semi-automated access to new information that has recently become available, sometimes by changing the representation format
of an existing database. The lag between the discovery and making the information available is primarily due to the human intervention needed to translate the new information into an existing database format, or to augment a database with new formats or fields. We believe that the increasing rate of scientific discovery and publication will make this problem increasingly serious, since more databases will be augmented more frequently, and new databases will be created to publish the new information.

We have described the BioSeek service selection scheme for providing fast access to the growing new information that remains elusive with the current technology. The main contribution of this part of the thesis is the application of the concepts and techniques called service selection to increase the degree of automation in new information access and to reduce the amount of unnecessary delay due to contacting sources that cannot contribute to given queries. Service selection uses metadata, called source-capability profile, to support dynamic matching of each query with the information sources that are relevant to, and capable of, responding to that query. By updating a source profile, new information or capabilities of the source are immediately accessible by queries processed through service selection. Furthermore, addition of new bioinformatics data sources and capabilities can be dynamically incorporated into the subsequent execution of queries. The main thrust of our service selection scheme is its intelligent source selection powered by the source-profile based multi-level progressive pruning strategy. Our experiments show the increased benefits of service selection as the number of data sources available to a query increases.

Another contribution of service selection is in combination with change detection. Chapter 4 focuses on detecting changes in individual pages; while this is an essential building block for generic change monitoring, often it is too specific for users. Rather than specify a single page to monitor, users may be more interested in monitoring a specific focused topic. In such a situation, they require a system that can automatically locate the services that provide that type of information, and monitor them for new information. In addition, service selection can monitor the set of relevant services, adding new services to be monitored as they are discovered. This allows monitoring requests to be much more powerful, and scale to a much larger set of concepts.
CHAPTER IV

CHANGE DETECTION

A composite service that has been created by composing several other component services can be viewed as a distributed query plan or a workflow. Each time information from the composition is required, the entire workflow is re-executed. If an individual user needs updated information from the service, then the entire workflow must be re-executed. One way to reduce the overall cost is to cache the data at each component service, and watch for local updates, propagating interesting changes up the query plan. This allows users to be updated with important changes, and allows each service to optimize locally for efficient notifications.

The Internet and the World Wide Web has made a huge amount of useful information available online. The freedom of publishing and sharing fresh information online is phenomenal. It is increasingly difficult for individuals to keep abreast of fresh information. Search engines are popular tools that help users locate information on the Web, but they provide no support for tracking changes and delivering interesting changes to the right users at right time.

In this chapter we describe the concept of Web page sentinels and how to build a sentinel system for efficient monitoring of changes to Web pages. This chapter has two main contributions. First we present a coherent framework that captures different characteristics of Web pages to provide a comprehensive monitoring system for content, structure, and several other properties of Web pages. Second, we develop a suite of system-level facilities for efficient processing of Web page sentinels, including (1) a new class of Web page sentinels capable of monitoring changes to the content of a page or the structure and presentation of the page, and delivering changes with a measure of the percentage of changes that have occurred, (2) a Page Digest encoding scheme to improve the performance of individual page sentinels through mechanisms such as short-circuit evaluation, linear time algorithms for document and structure similarity, and data size reduction, and (3) we develop a collection of sentinel grouping techniques based on the Page Digest encoding, aiming at reducing and eliminating redundant processing in large scale monitoring systems by grouping sentinels of similar
monitoring interests together and by introducing sentinel group based processing mechanism. We demonstrate the effectiveness of these techniques over a wide range of parameters, showing an order of magnitude speed up over existing Web-based information monitoring systems.

4.1 Overview

The World Wide Web offers a unique publishing medium that allows people to broadcast information with few of the traditional barriers to widespread communication. This freedom has fostered widespread growth of online communities that publish information on a near limitless array of topics. Commercial and altruistic ventures provide information and services that online users rely on for decision making in all facets of life, from entertainment to real estate, news, weather reports, stocks, merchandise prices, and myriad others. Web context 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 also highly variable: some pages—e.g., stock quote services—are updated frequently, requiring people 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.

Many users are becoming increasingly interested in automated tools to help them deal with the volume of information on the Internet and the rate of change of that information. Consider the Web site prizewatch.com, which provides a large repository of street prices for various computer components supplied by hundreds of online vendors. Figure 20 shows an example page from their site that lists prices for computer memory. Users trying to build or upgrade a computer will use prizewatch.com to monitor the prices of desired components. However, manually monitoring such Web sources for changes is a tedious process that does not scale well to more than a handful of pages of interest. Infrequently updated pages risk being forgotten, while monitoring pages that update frequently can quickly deluge the user with information. While the value of information on the Web does not decrease, the cost of retrieving the fresh information becomes too high to do manually.
Automatic Web change monitoring provides several compelling advantages even for simple scenarios. First, automatic systems remove the burden of monitoring from the user, allowing them to concentrate on other efforts while being assured of receiving a timely notification when an interesting change occurs. Second, Web change monitors can track many different sources simultaneously; users can handle more data effectively, making them more productive and increasing the quality of their decisions.

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. We describe 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:

- **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.
• Efficient Data Management. We encode Web pages in the Page Digest format, described in
Section 4.3, rather than HTML. The Page Digest format encodes the structure and contents of
a document efficiently for fast load times and efficient evaluation.

The rest of the chapter proceeds as follows. Section 4.2 defines Web monitoring sentinels and
introduces the language we use to specify them. Section 4.3 describes the Page Digest encoding,
which is used to process Web documents efficiently. Section 4.4 describes new classes of change
detection techniques for Web pages and algorithms for processing sentinels individually. We also
demonstrate how the Page Digest encoding can be leveraged to group sentinels for highly efficient
processing. The experiments in Section 4.5 show the performance characteristics of the system and
the improvements of the current system over previous efforts. We conclude by discussing related
work and future research directions.

4.2 Web Page Sentinels

The heart of any Web page monitoring system is the sentinel, a description of changes that are of
interest to the user. Sentinels provide the description of what sites are interesting to users and how
to access those sites. We have enhanced this essential description of a sentinel to support richer
processing semantics and optimizations as the basis for constructing a scalable Web monitoring
system.

A sentinel must describe which page to monitor, the type of information to monitor, when
and how often to notify a user of changes, and the time periods over which to look for changes.
To capture each of these characteristics, we have defined a sentinel as a tuple: (ID, Target Page,
Trigger Condition, Notification). The ID is a unique sentinel identifier that includes information
about the user. The target page specifies what Web page to monitor and includes all of the necessary
information to access it, such as authentication or cookies. The trigger condition defines how and
when to access a page to check for changes, while notification defines how the user receives updates
for detected changes.
4.2.1 Sentinel Types

There are several types of interesting changes of interest to users, which we have generalized into two basic categories: content change and structural change. Content changes include any change to text visible on a Web page as well as changes to links to other Web pages or images. Structural changes modify the tag structure of the page, which affects the relationships between content elements. These two basic categories are refined in the following four derived categories:

- **Any change** — Monitors a page for changes in content or structure. This may either look for a single change or every change between two documents.

- **Content change** — Monitors for changes to the content of the document. Examples include word and phrase changes as well as complex regular expression matching. It also includes changes to hyperlinks and images.

- **Structural change** — Monitors for changes to the structure of the document. Examples of structural changes include changes to the presentation of a document (the tag names and attribute values), page layout (for example, when a navigation bar has moved from one location to another, or the items in a list have been reordered), as well as annotations of the document (such as comments and meta tag values). Presentation changes can be particularly useful for Web designers who like to monitor popular sites for changes in design.

- **Combined change** — Monitors for changes in both structure and content. Examples of this type of change are new rows added to tables, items removed from a list, and fragments moved to a different part of the page. By explicitly defining a combined change over structure and content, sentinels can accurately target specific types of changes that would be difficult to capture using content or structure alone.

Figure 21 displays the organization of sentinel types into a sentinel hierarchy. A sentinel is a specialization of the sentinels that it appears under, and this hierarchical generalization enables a class of evaluation optimizations that we explore in Section 4.4. Intuitively, the hierarchy states that no sentinel type below *any change* can be triggered unless *any change* itself has also been triggered.
Any of the sentinels may be modified to check for changes in only specified portions of a page to focus change detection on the interesting parts of the page. This allows a user to ignore changes in advertisements, timestamps, or other uninteresting parts of a page. Modified sentinels are specializations of their unmodified counterparts for optimization purposes.

4.2.2 Sentinel Triggers

Sentinels are defined in a rich language that specifies exactly what kinds of changes will be detected, which pages or page fragments to monitor, and how often to check for changes. An important component of any sentinel is the trigger, which triggers sentinel evaluation. The trigger specifies a trigger condition, firing interval, and change detection reference. More complex triggers can be defined as Boolean combinations of triggers.

Condition. The condition defines the type of monitor installed over the target; a condition can be composed from a Boolean combination of other conditions. Conditions are always stored in conjunctive normal form, allowing the evaluation of a sentinel to be short-circuited if any one of its conjuncts evaluate to false.

Firing Interval. The firing interval specifies how often the page is to be monitored. This is expressed in multiples of the system-wide minimum time interval. Sentinels are a content-based change detection system, implying that query evaluation occurs after an interesting event. However, for passive sources—including most Web servers—sentinels must be evaluated periodically to determine if there has been an interesting change to the page. By default, the firing interval is set for the minimum time interval allowed by the system, although users may specify any other interval.
which is a positive integer multiple of the minimum. Restricting the allowable time intervals greatly increases the ability of the system to optimize execution strategies.

Change Detection Reference. The final part of a trigger is the change detection reference. This defines which historical version of a page is to be compared with the current version to determine if a relevant change has occurred. By default it is the previous version, although longer time intervals are supported, as are comparisons to a fixed version of a document.

Sentinel Modifiers Parameters In addition to the required components specified above, a sentinel may be modified with parameters that focus change detection on particular aspects of a page. Parameters provide a powerful mechanism for detecting meaningful changes to a document. For example, a parameterized attribute sentinel can be used to consider only link attributes; a further restriction could consider only relative links or links not pointing to a particular domain such as doubleclick.com or other well-known advertisers. Sentinel parameters are specified using syntax like that of an SQL WHERE clause and can include regular expression operators.

Location. The type of the sentinel is optionally modified by a location expression that restricts the scope of the sentinel. Sentinels provide two generic approaches to capture the location within the monitored page: bounding boxes or path expressions. Path expressions define a location by identifying an interesting subtree to be monitored using XPath [34]. Bounding boxes can be specified in terms of either content text or tags. A tag bounding box defines a location by a list of prefix and suffix tags that enclose the location to be monitored. Likewise, a textual bounding box uses prefix and suffix strings to enclose the region to be monitored; the text considered for the bounding box does not contain tags or comments. A more detailed description of bounding box location techniques may be found in [89]. Failure to locate the specified object triggers a change notification. In addition, an error report notifies the user that the sentinel's location definition is no longer valid.

4.3 Page Digest Encoding

An important problem affecting the scalability of any system that interacts with the Web is the processing of the markup languages, such as HTML and XML, that are used on the Web. The organization of Web data is not tuned for efficiency and is highly redundant, leading us to consider
techniques to leverage an alternate data encoding that focuses on efficiency. In particular, some
sentinels, such as those dealing with structure, type, or size, do not operate over all aspects of
a document when evaluating what changes have taken place. The Page Digest Web document
encoding [111] can increase the efficiency of sentinel processing by providing efficient access to
the different characteristics of a page required by advanced sentinels. All documents in the sentinel
processor’s local data cache are stored in the Page Digest format; we demonstrate that using this
encoding significantly reduces the time required to transfer the page from disk to memory. The
encoding has the following characteristics:

- **Redundancy elimination.** The Page Digest encoding reduces much of the redundancy present
  in Web documents by altering the representation of the data while preserving structure and
  semantics of the document. This reduced size translates into I/O savings when interacting
  with a document on disk or over a network. However, the Page Digest encoding is not a
  compression scheme, so it does not require decompression to be useful and can be processed
  in memory in the same format as is used for storage.

- **Exposed document characteristics.** The Page Digest encoding reorganizes a document to
  expose many attributes of a tree that would otherwise have to be computed. For example, the
  number of nodes in a document tree is an interesting attribute for some sentinels that is not
  immediately available given a DOM tree of the document. Another example is a comprehensive
  list of types and the frequency that each type is used. These characteristics are stored in
  the header of the Page Digest for efficient access.

- **Superior processing layout.** Most basic operations over document trees take considerably
  less time using a Page Digest encoding of the document rather than a DOM representation.
  There are several reasons for this: (1) Page Digest operations like traversal and node search-
  ing operate over a arrays, as opposed to traversing a tree by chasing pointers from node to
  node. (2) Comparison operations require less work, for example, comparing integer values is
  much more efficient than comparing strings in determining whether two types are similar. (3)
  Loading a compacted tree into memory is more efficient because parsing is much simpler.
Figure 22: Tree fragment with corresponding Page Digest

Figure 22 shows a tree-structured document fragment with its matching Page Digest encoding. Node types appear to the right of the node in the tree. 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 with the label "TEXT" followed by the content of the node (represented as a single-letter variable). The document fragment represented by this tree is a table that contains one row with two columns. The first column of the table contains an image while the second column contains a nested table with an image and some text.

To the right of the tree is its Page Digest encoding. In general, the Page Digest encodes three separate characteristics of a tree: structure, node labels, and content. Structure is encoded in ST by a depth first ordering of children of nodes in the tree representation of the document. Node labels are encoded as a mapping from the tree nodes, TM to a set of labels, TL. Content is encoded as a mapping from leaf nodes in the tree, CM, to a set of strings, unique for each tree, CL. The Page Digests shown in Figure 22 have been specialized to handle HTML and XML pages by including attributes as another encoded characteristic. Attributes are encoded as a mapping from nodes, AM, to sets of name-value pairs, AL. For a formal definition of the Page Digest encoding, refer to [111].
Figure 23: Comparison of various loading techniques

The Page Digest format splits structure from content and provides the opportunity to only load the structure of a page for structural change sentinels. Similar optimizations can be used for size, attribute and type sentinels to reduce the I/O cost of evaluating these sentinels. Initial performance results, shown in Figure 23, indicate that creating a Page Digest from HTML is roughly equivalent in cost to using standard HTML or XML parsers, while loading a serialized version of the Page Digest is orders of magnitude faster than loading serialized XML. To give some perspective to the graph, the vertical line indicates the approximate size (1250 nodes) of the CNN home-page2. The data used to generate this graph was a randomly generated XML document resembling a bushy tree. Further experiments have shown that both the Neko HTML parser and the Xerces XML parser perform worse on more deeply nested trees, while the tree shape does not affect the performance of the Page Digest parser.

It may be worth noting that the current crop of XML serializers and parsers perform poorly on large documents with deeply nested structure. Because the algorithms rely on recursion, parsing fails when the depth of the document exceeds the stack space of the implementation language. Our

2http://www.cnn.com Average size as measured from daily snapshots taken during 2002
implementation does not suffer from this drawback as we eliminate the use of recursion in our algorithms in favor of internally managed data structures. While these structures may also run out of memory, it is on trees that are several orders of magnitude larger.

4.4 Sentinel Processing Techniques

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.

4.4.1 Problem Categorization

The primary costs in processing sentinels include acquiring a new version of the monitored data (network costs), maintaining the previous versions of the data (storage and retrieval costs), and comparing versions of data to determine whether or not any possible changes are relevant to interested parties (processing costs).

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. As we assume that all sources are autonomous, their cost is 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 overloaded.

Storage costs are also a major consideration in designing a sentinel system. Previous versions of Web pages cannot be maintained in primary memory due to the size and growth rate of the data. Average page size for many sites is 30 KB; any system which attempts to handle a large number of sentinels quickly runs out of available memory to keep even a minimal history of each page in memory. This implies that previous versions must be kept in secondary storage and only brought into memory when needed for comparison with new versions. This local data transfer is a significant fraction of the time required for evaluating sentinels over each page. While using the Page Digest encoding allows us to dramatically reduce local I/O cost, storage cost is not the focus of this study as it was not found to be the primary bottleneck in processing large numbers of sentinels.
Here we focus 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 page and to optimize processing of large groups of sentinels. We employ several methods to achieve more efficient operation. First, the algorithms use incremental processing to reduce the amount of computation used for a single sentinel based on a cost-benefit analysis of low cost change indicators such as page signatures. Second, Web pages are encoded in the Page Digest format, described in Section 4.3, rather than HTML. The Page Digest encoding 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 pages attract many users that are interested in changes to the page. Many sentinels over such a popular page can be expected to be similar. By combining sentinels we can reduce processing costs, network transmission time, and local I/O.

4.4.2 Single Sentinel Processing

Individual sentinels are executed in stages: preprocessing, location masking, and finally sentinel-type specific processing.

Preprocessing All sentinels are first checked to see if applying a page signature to the new page will reduce overall computation. This is determined by checking the probability that a page has changed (based on the change history of the page), compared 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 may be 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 a binary string where each bit corresponds to a node in ST, the depth-first traversal of the page. Bits are set for each entry that matches the XPath location expression, or the bounding box defining the portion of the page to watch for changes. See Figure 30 for an example location mask.
4.4.2.1 Specific Sentinel Processing

Each sentinel type processes different components of the Page Digest. Content sentinels operate mainly over the content list, as well as the link and image attributes. Structure sentinels monitor the structure array, the type list, and the attribute lists, as well as comments. Combination sentinels may monitor any part of the document encoding.

Content Sentinels Content sentinels cover text, links, and images. There are four main types of text-based content sentinels: any change, keyword/phrase change, regular expression change, and data type changes. Any change monitors for any type of change to the content. Keyword and phrase changes monitor for the addition and deletion of specific strings to a document. This can be used for while list or black list monitoring, to ensure pages only contain words from an approved list, or do not contain words from a banned list. This type of monitoring is useful to watch sites for specific topics (for example, monitoring an information technology news site for mention of 'IBM').

Regular expression monitoring scans the content looking for patterns of text. This type of monitoring is useful to look for changes in strictly formatted data (for example, monitoring for price changes).

Data type changes watch for changes to parts of the text that conform to a specific data type. Examples of data types include generic numbers, prices, percentages, and dates. Data types may support advanced comparison operations (such as greater than), depending on the type definition. Most data types are based on complex, predefined regular expressions, coupled with code for a comparison operation. However, extension mechanisms exist to allow an arbitrary parser to recognize other types which do not conform to a regular expression (for example, data types that match entries in a dictionary). This allows additional data types to be dynamically added to the system without any architectural changes.

All text change detection algorithms operate over the content list, and can ignore other parts of the Page Digest encoding. The algorithms are based on straightforward traversals of the text to match the user-defined regular expressions or phrases. This eliminates the complex logic and computational expense of locating content in either a flat text representation or a conventional tree representation.
Link and image change sentinels share several characteristics. First, the actual link itself is embedded in a tag attribute (of the a and img tags, respectively), making data location a matter of traversing the tag and attribute lists—an efficient operation. Second, users may specify that interesting changes to links or images are restricted to particular containers, relative links (compared to the page base), or intra-page links.

Finally, changes could be measured in the value of the link or the content pointed to by the link. Without loss of generality, we assume that the only interesting changes are in the value of the link itself, and not the content of what is being pointed to. To monitor the content pointed to by the links, a synthetic page may be generated containing signatures (such as an MD5 hash) of the content of each link. This reduces the problem to monitoring the content of the synthetic page, although it does induce order dependency—sentinels over synthetic pages must be processed after the derived sentinels over the actual content pages. It also adds another action to be performed if there are any link changes on the original base page: derived sentinels may be added or dropped as the links change, and the synthetic page must be updated to point to the updated derived sentinels.

Structure Sentinels Structure sentinels monitor a page for changes in layout, changes in the types of tags used, and changes in the order of the tree. In addition, structure sentinels monitor parts of Web pages that are "invisible" when rendered in a Web browser. Because structure sentinels do not monitor the content of a Web page, it can avoid loading an entire page into memory. 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. This requires less than 4 KB to store the structure, tag map, and tag names in memory. As can be seen in Figure 33, 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 computation is done.

Combination Sentinels Combination sentinels monitor changes to both content and structure. This includes adding items to lists, rows or columns to tables, or moving navigation bars from one location to another. Because there must be a structure change before it checks for associated content changes, a combination sentinel can apply the same optimization as a structure sentinel if there is a
reasonable probability that there would be no change. Note that this type of optimization would be inappropriate if multiple sentinels are monitoring the same page, and some of them need to access the content.

**Structure-based optimization** Structure information can be used to dramatically speed up the execution of generic change detection algorithms. Algorithm 1 demonstrates how a tree change detection algorithm can be optimized to simple leaf comparisons when the structure has not changed.

**Algorithm 1 Smart Difference Compare**

- Let $O$ = DecodeToPage($O$) of page at time $t_i$ (indexed from disk)
- Let $W$ = Page at time $t_j$
- Let $S$ = SmartIndex
- Let $S$.size = 0
- Let $D$.size = 0
- for $n$ < $S$.size do
  - if $S$.size > $O$.size then
    - if $S$.size > $O$.size + $O$.size AND $S$.size > $D$.size then
      - $D$.size = $n$ (adjust size of subtree at index)
    - else
      - if $S$.size > $D$.size then
        - $D$.size = $n$ (adjust size of subtree at index)
  - end if
- end for

#### 4.4.2.2 Page reordering

Occasionally two pages will be identical except for the order of its elements. This type of change may occur because advertisements are reordered, search results are reordered, or items in a purchase order are reordered in an XML document. Consequently, users may wish to ignore this type of change.

To eliminate false positives from changes in reordering, sentinels must be able to efficiently detect trees that are identical except for the order of some elements. Traditional tree change detection algorithms are not suited to this task as they search for additions, deletions and updates, making them too costly: $O(n^2)$ where $n$ is the number of nodes in the tree [28]. To address this issue we have developed a linear time algorithm that will determine if two trees are structurally invariant except for the order of the branches.
Theorem 1 Two trees with the same number of subtrees of each size are structurally invariant.

Proof sketch
Let $T_1$ and $T_2$ be two trees, and let $T_j[i]$ be the set of subtrees in $T_j$ of size $i$.
If $T_1$ and $T_2$ are structurally invariant, then there exists a number $n$ equal to the number of subtrees in both $T_1$ and $T_2$.
If $|T_1[i]| \neq |T_2[i]|$ for $i = 1, \ldots, n$, the trees cannot be structurally invariant.
There exists a one-to-one mapping from each subtree in $T_1$ to $T_2$, where the matched subtrees are structurally invariant (by induction). Since each child of the root of tree $T_1$ is mapped to a structurally invariant subtree of $T_2$, and $T_1$ and $T_2$ are of the same size, $T_1$ and $T_2$ are structurally invariant.

A linear time algorithm to calculate whether or not two trees are structurally invariant is shown in Algorithm 2.

Algorithm 2 Linear time algorithm to detect structural invariance
Let $DFN = (\nu_1, \nu_2, \ldots, \nu_m)$ be the nodes from the
tree of page UI in depth-first order.
Let $\nu_j = (\nu_j, \nu_1, \nu_2, \ldots, \nu_{n_j})$ such that $n_j$ is the number of children of $\nu_j$, for $j = 1, \ldots, n$.
Let $\nu_{j,k} = (\nu_{j,k}, \nu_{j,k,1}, \ldots, \nu_{j,k,n_j})$ such that $\nu_{j,k}$ is a list to the index of all children of $\nu_{j,k}$ in $DFN$, for $j = 1, \ldots, n$.
Let $S_n$ be a vector of size $n$, where each entry, $s_{n,j}$, is initialized to 1; $j = 1, \ldots, n$.
* calculate size of subtree
for $k = n$ to 1, step -1 do
  for $j = 1$ to $k$ do
    $s_{kj} = s_{kj} + s_{kj-1}$;
  end for
end if
for $k = n$ to 1, step -1 do
  end for
* event subtrees of each size
Let Count be a vector of size $n$, where each entry, $c_{kj}$, is initialized to 0; $j = 1, \ldots, n$;
for $j = 1$ to $n$ do
  $c_{kj} = c_{kj} + s_{kj}$;
end for
* check whether $T_1$ and $T_2$ are structurally invariant
for $k = n$ to 1, step -1 do
  if $c_{kj} \neq m_{kj}$ then
    trees are not structurally invariant.
  end if
end for

4.4.2.3 Threshold Change
On the users are interested in a general measure of change between two versions of a document, rather than looking for a specific change. For example, page designers may be interested in monitoring how the layout and presentation of a site changes. They are not interested in individual font,
image, or layout changes, but when large portions of the page exhibit these types of changes, they want to be alerted so that they can review how the style of the page has changed. There is a similar type of interest in the content of a page. Updates may not be interesting individually, but large changes in the content are important events.

For these types of sentinels we introduce a similarity metric that scores the changes between documents on a scale of 0 and 1. For either content or structural similarity, we provide algorithms that provide approximate algorithms that can measure the similarity of two documents in linear time. For the textual content changes, we use shingles [22], which provides fast approximations of the similarity of two documents. For structure changes, we have several approaches: measuring tag similarity, path similarity, edit distance, and an adaptation of shingles to apply over the structural characteristics of a page. Here we provide some experimental results that illustrate the usefulness of the similarity metric over a large number of similar documents.

Content Similarity

Content consists of three distinct parts of a Web page: text, links, and images. Each type of content must be dealt with individually to compute the similarity. An overall similarity can be obtained by a weighted combination of each of the different metrics.

Link Similarity Link similarity is the percentage of the hyperlinks that have changed between versions of a page. Let \( H_i \) be the set of hyperlinks contained in page \( D_i \), the \( i \)th version of page \( D \). Similarly, let \( H_j \) be the set of hyperlinks contained in page \( D_j \). Then we define the link similarity of two pages as:

\[
\text{LinkSimilarity}(D_i, D_j) = \frac{|H_i \cap H_j|}{|H_i \cup H_j|}
\]

Note that this definition does not take into account repeated links.

Image Similarity As with link similarity, image similarity is the proportion of images that change between two versions of a document. However, there is a caveat with image similarity: the content of an image may change over time, changing the content of the page, without changing the pointer to the image. Let \( I_{pi} \) be the set of image pointers in the \( i \)th version of a Web page. Let \( Id_i \) be the set of signatures of the images in \( I_{pi} \). A signature of an image may be computed, for example, by applying an MD5 hash of the image file. This implies two measures of image similarity.
between versions \(i\) and \(j\) of a page \(D_i\):

\[
Image\_Similarity(D_i, D_j) = \frac{|I_{p_i} \cap I_{p_j}|}{|I_{p_i} \cup I_{p_j}|}
\]

OR

\[
Conservative\_Image\_Similarity(D_i, D_j) = \frac{|I_{d_i} \cap I_{d_j}|}{|I_{d_i} \cup I_{d_j}|}
\]

Conservative\_Image\_Similarity defines a strict similarity which takes into account the possibility of image changes over time. However, it is much more expensive to compute as it requires the retrieval of each image and the computation of a signature of the image.

Text Similarity The final form of similarity is of the visible text in the document. This could be accomplished by tokenizing the text and computing the same form of similarity metric as for links and images. However, this does not take into account the order of words, and significantly simplifies the document. In addition, it is difficult to accurately determine appropriate tokenizing characters. In [22], Broder presents a technique to efficiently compute the similarity two text documents using shingles.

Intuitively, shingles are a random sample of the text in a page. The key is that because the random mapping is constant across all pages, and the results are sorted, the samples are directly comparable across different pages. The overlap in the sample between pages is an indicator of the overlap between the entire pages.

Structure Similarity
Structure consists of every part of a Web page that is not content: tags, attributes, tree structure, comments, and header values. To check for changes to layout, presentation, or other structural aspects between versions of a page, one or more of structural page facets must be monitored.

Tag Similarity Tag similarity measures how closely the set of tags match between two pages. In XML documents, tags are one component of schema; pages that use a similar set of tags will likely have a similar schema. A simple measurement compares the overlap between the set of tags used in each page. Let \(T_i\) be the set of tags contained in page \(D_i\), and \(T_j\) be the set of tags contained in page \(D_j\). Simple tag similarity of two pages is:

\[
Tag\_Similarity(D_i, D_j) = \frac{|T_i \cap T_j|}{|T_i \cup T_j|}
\]
However, this is not satisfactory for several reasons. One critical problem is that pages conforming to the same schema, such as HTML, have only a limited number of different tags; one page may contain a large number of a particular tag, while the comparison page may contain relatively few occurrences of the tag. To compensate for tag frequency, we can introduce a weighted similarity measure. Let \( t_{ik} \) be a member of \( T_i \), and \( w_{ik} \) be the number of times tag \( t_{ik} \) appears in \( D_i \). Also, let \( t_{jk} \) be the corresponding tag in \( T_j \), and \( v_{jk} \) be the number of times tag \( t_{jk} \) appears in \( D_j \). If there are \( n \) unique tags that occur in pages \( D_i \) and \( D_j \), then

\[
\text{WeightedTagSimilarity}(D_i, D_j) = \frac{\sum_{i=1}^{n} 2 \cdot \min(w_{ik}, v_{jk})}{\sum_{i=1}^{n} (w_{ik} + v_{jk})}
\]

**Path Similarity** Path similarity measures the similarity of paths between two different documents. A path is defined as a list of connected nodes starting at the root and terminating in a leaf node. Path similarity can be measured in several different ways: binary, where a path is either equivalent or not; partial, where the number of comparable nodes in each path are discovered; or weighted, where the nodes are weighted according to their distance from the root.

Binary similarity is the simplest to implement, as each unique path in one version can be matched with its equivalent in the second version of the tree using database join techniques, such as hash joins. Similarity is then the ratio of the matched paths to the total number of paths in the larger tree.

Partial and weighted path similarity measures are much more difficult to compute. Since there are \( n! \) possible mappings between the paths between two trees, exhaustive algorithms that produce the optimal similarity score are infeasible. However, since most trees representing Web pages are not balanced, optimizations can be made so that only paths of equal length are compared. While this does not affect the worst case performance of the algorithm, typical performance is significantly better.

**Edit Distance Similarity** Edit distance measures the minimum number of node insertions, deletions, and updates required to convert one tree into another. This can be converted into a similarity metric by normalizing the number of edit operations with the number of nodes in the tree representing the larger document. Let \( N_i \) be the set of nodes in the tree representation of document \( D_i \).
Then,

\[ \text{EditDistanceSimilarity}(D_i, D_j) = \frac{\text{editDistance}(D_i, D_j)}{\max(|N_i|, |N_j|)} \]

**Structural Shingle similarity** Monitoring a set of tags, even when weighted by frequency is seldom enough to capture all of the interesting structural changes to a web page. And the similarity in tags does not correspond to what users intuitively feel constitutes a “similar” document. In addition, algorithms based on path similarity or tree edit distance may be too expensive to compute for either large documents or for a high volume of documents. To address these concerns we have adapted the shingle technique [22] for measuring the similarity of the structure of two documents. This provides a constant time comparison algorithm, and requires a linear time pre-processing step to extract shingles from a document’s structure. Note that pre-processing only needs to be done once for each version of the document, and the resulting shingles can be reused for comparisons with subsequent versions.

Documents can be viewed as a sequence of tokens. Tokens can be letters, words, lines, or another grouping. For our purposes, we consider a token to be derived from the Page Digest encoding of a Web document. More specifically, it is a combination of the structure and tag components of the encoding. We map a combination of the tag name and number of children of a tree node to a single token. A tree can be re-encoded in terms of these tokens. For example, the path HTML/HEAD/TEXT[1] is mapped to the sequence of tokens [HTML, HEAD, #TEXT0]. In this example, the node labeled HTML has two children, HEAD has one child, and #TEXT has no children. Applying this transformation to an entire document results in a long sequence of tokens. For example, applying the transformation to the example tree in Figure 22, results in the sequence shown in Figure 24.

A shingle is a contiguous subsequence of tokens taken from a document. Figure 24 shows the list of shingles of length four in a table. Given a list of shingles from two documents, it is possible to identify both resemblance and containment. Resemblance between documents \( D_i \) and \( D_j \) is defined as

\[ r(D_i, D_j) = \frac{|S(D_i, w) \cap S(D_j, w)|}{|S(D_i, w) \cup S(D_j, w)|} \]

where \( S(D_i, w) \) is the operator that creates shingles of length \( w \) from document \( D_i \). Similarly,
containment of document $D_i$ in $D_j$ is defined as

$$r(D_i, D_j) = \frac{S(D_i, w) \cap S(D_j, w)}{S(D_j, w)}$$

For convenience and faster processing, shingles may be converted into numbers with a hashing function. This hashing function should provide a high degree of confidence that there will be little or no hash collisions where two shingles map to the same value. Constructing an appropriate hash is made considerably easier by ensuring that the hash space is significantly larger than the shingle space. Depending on the number of tokens in a shingle (or window length), this may be trivial.

Another technique that can greatly assist in the efficiency of resemblance is to only keep a relatively small sketch of each document. It has been shown [22] that a sampling from a permutation of the set of shingles, chosen uniformly and at random, can be used in an unbiased estimator of resemblance between two documents. One efficient way to achieve this is by applying a pseudo-random number generation algorithm to the hashed values, sorting the results, and choosing only the smallest $k$ of the resulting values. Figure 24 shows an example of this using a value of $k = 3$.

![Tree Structure Sequence](image)

<table>
<thead>
<tr>
<th>Sequence fragment; window size = 4</th>
<th>Hash values</th>
<th>Random Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_3$</td>
<td>$r_{13}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4</td>
<td>$b_4$</td>
<td>$r_{14}$</td>
</tr>
<tr>
<td>tablet t2 t3</td>
<td>$b_5$</td>
<td>$r_{15}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_6$</td>
<td>$r_{16}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_7$</td>
<td>$r_{17}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_8$</td>
<td>$r_{18}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_9$</td>
<td>$r_{19}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{10}$</td>
<td>$r_{20}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{11}$</td>
<td>$r_{21}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{12}$</td>
<td>$r_{22}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{13}$</td>
<td>$r_{23}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{14}$</td>
<td>$r_{24}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{15}$</td>
<td>$r_{25}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{16}$</td>
<td>$r_{26}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{17}$</td>
<td>$r_{27}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{18}$</td>
<td>$r_{28}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{19}$</td>
<td>$r_{29}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{20}$</td>
<td>$r_{30}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{21}$</td>
<td>$r_{31}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{22}$</td>
<td>$r_{32}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{23}$</td>
<td>$r_{33}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{24}$</td>
<td>$r_{34}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{25}$</td>
<td>$r_{35}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{26}$</td>
<td>$r_{36}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{27}$</td>
<td>$r_{37}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{28}$</td>
<td>$r_{38}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{29}$</td>
<td>$r_{39}$</td>
</tr>
<tr>
<td>tablet t2 t3 t4 t5</td>
<td>$b_{30}$</td>
<td>$r_{40}$</td>
</tr>
</tbody>
</table>

**Figure 24**: Transformation of Figure 22 into a structural shingle

**Example Results**

118
To compare how accurate the different methods of structural similarity are, we sampled documents from ten Web sites and compared them to a snapshot taken from MyYahoo\textsuperscript{8} on February, 2002. The bar chart in Figure 25 shows how four of the algorithms rank the similarity of the sites listed in Table 25. As can be seen, the simplistic tag comparison algorithms and the relatively sophisticated edit distance similarity algorithm give relatively high scores to many sites that are completely dissimilar, while the structural shingle similarity algorithm gives low scores to diverse Web sites and a high score to similar Web pages. In addition to providing a much more accurate view of the similarities of different documents, the structural shingle similarity metric executes in constant time, depending only on the per-set sample size (here a sample size of 100 shingles was used), and not the size of the document.

Figure 25: Cross Site Comparison

\textsuperscript{8}http://my.yahoo.com. MyYahoo! is a customizable news portal that presents a list of the top headlines of the day, updated on an hourly basis.
A second demonstration of the relative accuracy of the different structural similarity metrics is given in Figure 26. This figure measures the similarity of pages sampled daily from MyYahoo! from February, 2002 through February, 2003. As time progresses each of the similarity metrics detects some change from the original documents, however the sensitivity to change varies dramatically; the structural shingle similarity metric is sensitive to the small changes that occur to the MyYahoo! Web pages overtime, while the tag similarity metrics show almost no change over time. In this experiment the edit distance metric shows a more realistic reflection of the actual amount of change, however at a much higher price algorithmically.

Note that the path similarity metric is not shown in these graphs due to the expense of computing optimal and approximate similarity scores between even relatively small documents, such as the pages from MyYahoo! Simple algorithms that implement that path similarity metric do not add additional accuracy over an edit distance algorithm and they are much more inefficient.

4.4.3 Multiple Sentinel Processing

For popular pages, many people will be interested in monitoring for changes. This overlap in interest presents opportunities for dramatic increases in efficiency by semantically grouping related change requests. For example, consider the three sentinels depicted in Figure 27. Two of the sentinels are image sentinels, while a third is monitoring content. Grouping on the type allows the image sentinels to be combined so that both sentinels can be satisfied by a single pass over the list of images in the page (a traversal of the src attributes of the img tag). A second type of grouping matches the location expression of different sentinels. This allows both sentinels to share the cost.
of identifying the portion of a page on which they operate. Since this cost is typically much higher than the cost to evaluate most sentinels, and since location identification must be done in any case, we always group by location before grouping by sentinel type.

Grouping begins by combining all sentinels that are installed on the same Web page. The new version of the Web page is retrieved from its source once and maintained in memory. Sentinels that are not active during the current execution round (for example, because they have a longer trigger period) are skipped. This grouping minimizes the network bandwidth required to evaluate sentinels and reduces local I/O. Grouping also multiplies the effect of incremental processing. For example, a single page signature comparison can short-circuit processing for the entire group of pages when there has been no change. This adjusts the cost/benefit analysis, increasing the probability that these techniques will provide a net benefit.

Figure 26: Single Src Changes Over Time
Next, sentinels are grouped together based on the locations within the page that they are monitoring. This effectively reduces the size of the document that each sentinel needs to process, a critical improvement for super-linear algorithms. Cheaper algorithms, such as detecting changes in the tags used in a document can also significantly benefit from grouping with other sentinels defined over the same location—finding the location is the dominant cost for these sentinels.

Sentinels may also be grouped based on the sentinel hierarchy described in Figure 21, executing more generic sentinels before less generic sentinels, as shown in Figure 28. This index of sentinels demonstrates each type of grouping: short circuit evaluation for pages which have not changed, location based grouping, and hierarchical evaluation.
A particular execution of sentinels installed over an individual page proceeds as follows. The new version of the page is retrieved from the network, and a page signature, such as an MD5 hash, is computed to determine if there has been any change from the previous version. If the signatures do not match, the generic any change sentinel is run first if one is defined on the page. If it cannot find a change, no further sentinels are executed.

After the any change sentinel, each unique location mask is computed. If a location mask is a subset of another location mask, the sentinels from the smaller location mask are executed only after sentinels in the larger mask have finished. This allows the sentinels over the smaller parts of the document to be skipped if similar sentinels from the larger mask cannot find any changes.

For each location, installed generic types of sentinels are executed: a sentinel over the content (text, links, or images), executes before more specific sentinels over specific keywords, phrases, regular expressions, or links to particular Internet domains. Generic structural sentinels, such as monitoring the structure for any change, are processed before similarity and other and structure based sentinels (layout, attributes, comments, and type).

Figure 28: Index for Grouping Sentinels

A second optimization is to merge sentinels that can be computed in one pass. For example, all attribute sentinels with the same location expression can be combined. For keyword sentinels,
a single pass is required to retrieve a list of each word in a page; every keyword query can then operate over the keyword lists for the new and old page. Regular expressions can be combined as well. Rather than traversing the entire document for each match, the document is traversed once to retrieve all of the content; regular expressions can then operate over a single string rather than traversing the tree structure of a page for each content string. This last optimization is a byproduct of the characteristics of the Page Digest described in Section 4.3, although the technique may be adapted for any page encoding.

**Grouping Performance Discussion** Sentinel grouping provides a significant savings in many cases with a cost of increased processing. Implementing grouping imposes overhead costs on all installed sentinels but provides no cost savings for groups consisting of a single user. However, we believe that the costs of grouping are outweighed by its benefits, especially for large scale systems. We include experimental results that support this belief and show the relative benefits of grouped over independent evaluation.

### 4.4.4 Page Digest and Sentinel Grouping

Using the Page Digest format, page modifications can be easily marked and reused. Page changes are processed using an annotation of both versions of the Page Digest being considered with a bit-map array describing where and how nodes have changed. The annotation array has one entry for each node in the tree that describes the Web page; this array corresponds directly to the structural array, $ST$, in the Page Digest for the page. Each bit-map entry has one bit for each type of change that can happen to an individual node: insert, delete, update, move, and copy. Also, the bit-map contains entries for changes that happen to descendant nodes. This marking scheme allows multiple annotations to be made on each node and eliminates redundant processing for multiple sentinels in a group. As an example, Figure 29 shows an update of the tree shown in Figure 22, with the changes highlighted on the Page Digest encoding. The array marking the new version after change detection would be as follows:

$$\{160, 160, 32, 0, 1, 128, 128, 128, 128, 4, 0, 128, 4, 0, 0, 0, 0, 0, 0, 4, 0\}.$$  

The number 1 represents an insertion, 4 represents an update, 32 signifies that a descendant was
inserted, 128 signifies that a descendant was updated, and 160 signifies both an insert and an update in one or more descendants.

Using Masks to Limit Change Computation
To isolate changes that are appropriate for a particular type of sentinel, a simple mask can be applied to the annotation arrays. For example, if annotation arrays have already been computed for the any change sentinel, then the mask for the generic attribute sentinel will consist of only those nodes that have attributes. This mask will be further refined by the image sentinel to contain only those nodes that contain links to images. Masks for the content sentinels can be more complex. For example, a table sentinel may have a multi-layered mask. The first layer of the mask includes only content nodes that are descendants of a table node. This layer is shared between all table-based sentinels regardless of their specific parameters. The second layer in the mask marks the specific content nodes relevant to the parameters of this table sentinel. Each marker in the list of table sentinels can refine the mask to include only those nodes that are relevant to that particular sentinel. Masks can be applied to the annotation array in linear time and can be kept in memory for reuse the next time the sentinel group is triggered for execution.
4.5 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. 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.

4.5.1 Sentinel Performance

4.5.1.1 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 [88] to the monitoring 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 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 31 shows the performance of the WebCQ implementation compared to the Page Digest enhanced sentinel processing system with and without grouping.

4.5.1.2 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 $\Omega_i / \Omega^p$ describing the popularity of page $i$, where $\Omega$ is a normalizing
constant, $i$ is the $i^{th}$ most popular page, and the exponent $\alpha$ describes the slope of the curve. Low values of $\alpha$ indicate that many of the sites have similar popularity, while higher values of $\alpha$ 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 $\alpha$ values between 0.6 and 0.9 [21]; the distribution of sentinels over Web pages should also follow this pattern.

The left graph in Figure 32 shows the execution time for groups distributed in a Zipf-like fashion with four different $\alpha$ 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 showing a fixed 10 sentinel group size shows the system in a simplified environment. The four Zipf-like lines reflect the observed popularity distributions in Web proxy caches.

The right graph in Figure 32 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. Algorithms over conventional encodings require significantly more work (as shown in Figure 34).

4.5.1.3 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 33 demonstrates the relative costs of grouping sentinels depending on the group size. The pages used

Figure 32: Zipf-like distribution of sentinels and execution time
for this experiment were based on the data collected from Yahoo!

Figure 33: Grouping cost per sentinel

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 MDS computations. Putting these same sentinels into groups of size 1 means creating 50,000 groups.

There are five lines in Figure 33: 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 MDS 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. This behavior is required of any system that must scale to millions of sentinels.

4.5.2 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 4.3. Here we compare the time required to perform equivalent change detection algorithms over the two formats.

Figure 34: Sentinel Cost

Figure 34 compares execution times for the two basic sentinel types of content and structure change. This experiment used 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.

4.6 Related Work in Change Detection

Change detection is a rich field with a long history. Early efforts in tree-based change detection come from [119, 90]. More recent results on tree to tree changes come from the work on creating a minimal edit scripts to convert one tree into another done by Shaista [115, 130, 116] and Chawathe [27, 28]. There have been significant efforts to adapt change detection to specific semi-structured formats, including NiagraCQ [29], Xdiff [125], and Xyleme [93, 35] for XML documents, as well as AIDE [42, 30] and the ChangeDetector™ [18] system for HTML documents.

The main distinction of our capabilities with other document change detection systems is that we base our algorithms off of the Page Digest encoding. We also include extended capabilities
to monitor page structure and layout, as well as threshold changes to content, links, or structure. To allow the system to scale up to millions of concurrent sentinels, we have introduced efficient algorithms and data structures that target bottlenecks we discovered in previous implementations of this type of system.

Typical change-detection algorithms, such as AT&T Labs HTML diff algorithm [30], 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).

XDiff [125], presents a document-to-document change detection algorithm for unordered XML documents. While this is preferable for database-derived documents where order is incidental, HTML documents and some XML documents rely on the implicit order in the structure. Generic change detection systems should handle both ordered and unordered documents. Our system includes algorithms to handle fast order-agnostic monitoring of documents, although this was not presented here due to space considerations. XyDiff [35] 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.

Previous work on document similarity has generally focused on content similarity, as with page shingles [22]. Current structural similarity has focused on XML schema similarity, such as [101] which focuses on pair-wise similarity between documents with unknown, but similar, DTD's. This requires $O(n^2)$ for each document comparison. Other work has transformed the problem of structural similarity between documents into time-series similarity solved by a Discrete Fourier Transform [49]. This improves running time to $O(n \cdot \log n)$ for comparisons between two documents, and demonstrates reasonable results over a small set of synthetic data.

In contrast, our measure of structural similarity applies the shingle technique to the structure.
of a document. This requires $O(k)$ to create the representation of a single document (where $k$ is the number of nodes), and provides constant time comparison of two documents. The savings in computation comes at a price of a small trade-off in accuracy, which can be arbitrarily reduced to tune the technique for different requirements. Our experiments have shown excellent results for popular Web sites. We plan on further exploring the use of structural shingles for similarity queries to document repositories.

Other systems that monitor multiple Web pages for multiple users include the NiagaraCQ project [29] at the University of Wisconsin-Madison, and the Change Detector™ system from WhizBang! Labs³. 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. Page Digest based sentinels offer a different type of monitoring solution. 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 textual information, structural information, and threshold changes (changes to a given percentage of the content or structure of a document).

Change Detector™ 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. Domain specific changes, such as changes to company executives, are implemented as external modules that generate synthetic Web pages that can be monitored by the base system.

While a lot of this work has proven optimal algorithms for change detection, relatively little work has gone into defining more efficient formats to process the data. Due to small document sizes (typically around 30 KB for common HTML documents), fast CPU processing, and relatively slow I/O operations, data format conversion and loading data from disk are the most expensive operations. By being aware of trends in computer architecture and bottlenecks [55], significant savings can be realized [47]. We have investigated a particular format that has proved efficient for loading and storage of XML and HTML, however, there is still considerable room to develop more efficient

³http://www.changedetector.com
encodings optimized for particular domains or tree shapes (bus/ly, deep, or mixed). In addition, algorithms that can take advantage of the more efficient encodings may be able to significantly improve on generic algorithms developed for generalized tree structures.

4.7 Change Detection Summary

We have presented a new mechanism for detecting changes to HTML and XML documents based on the Page Digest encoding. Our approach has three major areas: First, our change detection system provides standard change detection mechanisms, including insertion, update, and deletion of different types of content. Second, we provide operators for more advanced change detection operators, such as threshold based changes for different types of content and structure; in addition we provide techniques to monitor different aspects of a page, including content, tag types, attributes, and structure. Finally we provided a set of unique optimization techniques based on the Page Digest encoding that dramatically improves the performance of the system.

The unique set of features provided by the Page Digest sentinels form a powerful base to build extended monitoring capabilities. 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.

Change detection is an essential component to creating active and composable Web services. By adding change detection to a service, it can cease being a passive service that only responds to users when they request information and become an active service that pushes user-defined important information to where it is needed, while it is still fresh. The fact that it is user-defined is extremely important, as users typically have a vastly different view of important information than providers do – relying on providers to push information has be widely regarded as a failure, partially because users had to little control over what information was pushed to them, how they were sent information, and when they received it. By creating an independent service that can be layered on top of existing services (either those created by providers or those dynamically composed from other services), users have at opportunity to get the fresh information they want, when they want it,
and through the mechanisms they want it delivered.
CHAPTER V

CONCLUSIONS AND FUTURE WORK

Web Services have become an important component of the Web, powering popular sites like Google and Amazon, as well as providing a new level of accessibility to programming distributed, loosely coupled systems. As the Web evolves, services will become ubiquitous, enabling common computing applications and devices to have instant access to the global net of information services that people have come to rely on, including news, weather, shopping, and distributed access to entertainment and scientific data. These services will move into every area of life, from traffic information and automated route planning services for drivers, to an integral part of cutting-edge scientific experiments in genomics and nanotechnology.

5.1 Technical Contribution and Potential Impact

This dissertation has covered three components for building and deploying efficient, scalable, and robust services. The components covered techniques in information extraction to address information contained in legacy services, change detection to make services proactive, and service selection for locating services that are relevant for individual requests.

5.1.1 Omini and Information Extraction

The Omini information extraction system addresses the need to retrieve data from services that contain nonstandard information interfaces. As the Web transforms from a human-oriented content distribution medium to a bazaar of services, there is a significant amount of valuable information that remains locked in a legacy world of proprietary databases and idiosyncratic interfaces. To unlock the potential of this information, it needs to be adapted to more modern service architecture, allowing seamless access from any service or device, without the need to involve end-users at every stage of information gathering. Omini provides automatic mechanisms to automatically recognize data content in dynamically generated pages from a large variety of sites.
The information extraction algorithms have formed the basis for the XWrapElite [60] wrapper
generation toolkit. The toolkit is available as a service, and users have generated over one thousand
wrappers for a wide variety of Web sites. In addition, the wrappers have been used as the basis
for advanced data extraction workflows as part of a Department of Energy initiative in applying
computing technology to assist in scientific research [114].

5.1.2 PageDigest Sentinels and Change Detection

WebCQ is an active Web service that allows users to monitor arbitrary Web pages with a rich set of
content-based sentinels. It provides a complete change detection service, from monitoring, to noti-
fication, to intuitive displays of exactly the information that has changed in the context of the page
where it changed. The current system is used by well over one hundred different users who have
installed more than 1400 sentinels over a wide variety of Web pages, from corporate, to scientific
to entertainment domains. The conceptual contribution of the system applies to traffic control, drug
studies, online-shopping, and nearly any information domain where the value of the information
depends on how fresh it is.

This thesis addresses weaknesses in WebCQ system in terms of scalability and efficiency. The
main improvements come in the area of data encodings, extended sentinel capabilities, and highly
efficient processing algorithms. The PageDigest encoding provides a highly efficient and compact
storage mechanism for Web pages. The encoding significantly reduces I/O time by eliminating
redundancy in HTML, and simplifying the parsing required to translate a document from disk to
memory. Sentinel capabilities were extended by adding capabilities for monitoring degree of change
and changes in structure or presentation, to the rich content monitoring that is already available.
The efficient algorithms introduced here leverage the page digest encoding and the sentinel format
to reduce redundant computation. In addition, through careful observation of existing systems, we
identified bottlenecks and targeted optimizations to reduce critical path delays. These optimizations
scale up the system, allowing from ten to one hundred times more sentinels on equivalent hardware.

5.1.3 AQR and Service Selection

The initial prototype of our service selection mechanism is the Adaptive Query Routing system.
This is a Java-based Web service that transforms simple keyword queries into precise domain
queries, selects current and relevant services, sends out the subqueries, and performs result fusion and packaging. The prototype is backed by a library of wrappers to extract information from a wide variety of HTML-based services. This enables a wide variety of mediated applications, such as information integration, online shopping, directed topic-specific search, and meta-searches (searches that combine the results of several search engines), all from a single convenient interface. The impact of service selection extends far beyond single applications such as online shopping. Combined with change detection sentinels and information extraction, service selection provides powerful topic monitoring. In addition, there is a significant portion of current research directed towards scientific data management incorporating workflows; an essential component of an adaptive workflow execution is the ability to automatically select from current available sources based on a number of criteria. We anticipate that service selection will play an increasingly important role in future workflow and integration systems.

5.2 Open Issues

In Sections 2.7, 3.10 and 4.6 we discussed and compared work that is directly related to the content of this dissertation. Here we discuss a broader range of open issues that are relevant to the context of this work, but not the precise issues addressed.

Information Extraction Omini was designed to extract a specific type of information from pages that are the result of a query. It leverages the structure information that is consistent across different queries to the same dynamic generation template. The Omini design focuses on extracting the result rich component of a page that is returned upon answering a query. Thus, when there are more than one component in an answer page which is object rich, Omini can only identify one, resulting in a high error rate. We are involved in ongoing work to relax this assumption so that all portions of a query result page may be correctly identified for information extraction using a technique called THOR [26].

There are several other categories of information that may be extracted from Web pages that are not represented by this particular extraction technique:

- **Non-query Results** There are a wide variety of pages that contain useful information but are
not generated by a dynamic template in response to a query. Examples include the CIA World Factbook,\(^1\) the data presented in reviews at various review sites, or detailed article information pages stored in the CiteSeer\(^2\) database. These data pages share a similar format, but have no list of “records” stored in the page itself. Other sites present useful information in uniquely formatted pages that do not even share structural similarity with other pages.

- **Question Answering** Information extraction software that attempts to provide direct answers to questions (for example, “Who is the president of the United States?”) would necessarily need to use different techniques to locate the specific piece or pieces of information that they are looking for [75, 97, 76]. Techniques include identifying trusted sources, such as the CIA World Factbook, extracting specific single facts from Web pages, and ranking conflicting results to provide a confidence level in the retrieved answer.

- **Ontology mapping** Other promising techniques include mapping pages into a domain-specific ontology [44]. This technique requires a fully specified ontology that recognizes a specific type of data description. For example, many newspapers publish obituaries. An ontology that captures the typical information that is present in an obituary, and how each piece of information relates, allows parsers to map individual obituary entries into a uniform relational or object format.

- **Multi-page results** Omini focuses on the first result page. When the answer to a query consists of multiple pages, Omini treats each page independently. The prototype discussed here is limited to handle only the first page.

- **Inter-page analysis** As mentioned above, some sites present data in individual pages. Such data sources lend themselves to site-wide inter-page analysis to discover regular expressions that define the structure of the data [38]. This may be effective for simple query results as well, as long as there is very little non-query related information on the page as well. Inter-page analysis must detect advertisements that change relative to pages or queries to reduce the amount of “garbage” information that is extracted along with relevant query related

\(^1\)http://www.cia.gov/cia/publications/factbook/
\(^2\)http://citeseer.ist.psu.edu/
information.

- **Partial Results** Omini focuses on extracting the entire set of query results from a page. However, users may not be interested in each object. XWrap Elite [60] provides a filter for the resulting objects so that more complex and precise queries may be made to services than are natively provided.

- **Interactive Extraction** Users may need precise control over what data is extracted from a page. Domain-specific languages, such as WeBL [72] or W4F [113] provide a simplified mechanism for programmers to specify the data to extract from a page, but are too technical for the average users. Other attempts have addressed an interactive form of data extraction [87, 12, 10, 11] that allows common users to graphically select information to extract.

- **Query Mapping** Although Omini explores automated techniques to extract data from query results, it does not focus on the problem of transforming a query to fit arbitrary query interfaces. Various efforts have been made to standardize query interfaces, such as STARTS [51] for Web search engines, but due to competitive interests, these efforts have not produced much cooperation. Although new technologies such as SOAP [17] and WSDL provide a consistent syntax and protocol for query interfaces, they do not address the fundamental semantic problem. Other work has attempted to provide techniques to automatically match the ontologies for the input fields of query interfaces [99]. This mechanism can be used to identify how a query to one interface may be translated to query another service.

**Change Detection** There are two basic types of change detection: synchronous change detection and asynchronous change detection. Synchronous change detection has been well-studied for centralized systems, and is described in detail in database textbooks. Decentralized or distributed synchronous change detection systems for most types of changes are technically feasible but require standardization. For example, work on standardizing content updates for web logs (blogs) has led to the development of RSS (Really Simple Syndication) [128], although even this is a highly controversial standard.
Current research focuses on optimizing asynchronous change detection, computing change efficiently, and providing intuitive and productive user interfaces:

- **Asynchronous Optimization** For asynchronous change detection systems, research into determining the optimal frequency with which to poll the data sources is still in the early stages. In our implementation, we focus on efficiently detecting changes in new versions of web documents rather than choosing optimal times to request new versions. There are several research efforts, most recently by Cho [32, 31] and Coffman [70], which tackle the problem of assigning limited sampling resources to minimize latency in asynchronous change detection.

- **Difference Generation Algorithms** Computing a minimal distance between two versions of a document is not necessarily appropriate when conveying the meaning of the changes to a human. The typical primitive tree modification operators (insert, update, and delete of leaf nodes) are not a useful representation of change. Generalizing to subtrees as opposed to simply leaf nodes is an improvement, but there are other higher-level operations that may capture the meaning of the changes more accurately. Recent research has focused on copy, move, and glue operations [28] that attempt to make generated edit scripts more intuitive to the update operations that have been performed. Our continuing work on Sdiff [110] addresses difference generation on the PageDigest encoding.

- **Change Representation** Another interesting line of research opened by change detection systems is improving how changes are represented to users. Many systems on the Web simply present the new data without indicating what has been removed, updated, or added. Most systems lack mechanisms to capture and display information that has been deleted or modified over time, though some systems provide a side-by-side presentation that highlights new and updated information [89]. While such innovations are mostly to the user interface of a change detection system, it is no less essential for effective management of information.

- **Extended Content Monitoring** Our research has focused on page-based change detection. One useful extension is to create sentinels that are content-based, for example, monitoring the stock price of a particular company without having to specify a URI. Issues related to this type of monitoring include firing triggers at multiple data source sites, and combining change
detection results. Complications can arise in resolving conflicts between information available from different sources or in triggering on aggregate information from multiple sources.

Note that PageDigest-based sentinels are inappropriate for Web pages which contain insignificant markup. The change algorithms leverage the structural information to locate the changes — since text nodes are typically treated as a single unit, extended difference techniques must be applied to have fine-grained change detection for large text nodes that have no structure. This drawback makes the techniques discussed here unsuitable for text-based document retrieval and text document monitoring.

Service Selection We have explored several techniques in service selection, under the assumption that the bulk of services will be described in a fairly simple format — either a simple schema, a bare text description, or a content summarization. We have assumed that the set of services that we are choosing between are in similar formats. As standards begin to see wide use, there will be other techniques that promise higher rates of accuracy and completeness, including work done in declarative languages [124], and applications of functional programming techniques [96, 91].

We presented a look at several areas of related work in Section 3.10, however there are many more issues in selecting services than we have explored here.

- **Discovery** The service discovery problem lies in finding ways to describe and categorize services and data resources as they are discovered, either through a submission-based process or through automated crawling techniques. Automating the service discovery process provides opportunity for a consistent level of quality and complexity in the descriptions of various services. The current way in which services are described relies on human input. This is inherently inconsistent process with varying levels of thoroughness, specificity, and quantity of meaningful terms. More sophisticated descriptions provided by automated discovery engines or strict standards may allow service selection to apply techniques to ensure more complete and trustworthy results.

- **Optimization of Service Selection Algorithms** Our selection techniques operate at a basic level — selecting services based on their capability, and only considering factors such as performance in a secondary role. Choosing the optimal set of services to answer any particular
request requires considering a larger number of factors than we have examined, including overall quality of service, trustworthiness of services, provenance of information in different services, and other factors.

- **Service Selection in Mobile Environments** Intermittent, mobile, or low-bandwidth connections pose other challenges. For example, some services may require transoding before they can be used in a particular context or on a particular device. Optimizing service selection to find services that are relevant in the context of a mobile device and that do not exceed the capabilities of the device in terms of power, bandwidth, resolution, or processing power are all open issues.

- **Efficiency** While service selection greatly improves efficiency over naive query strategies, we have not explored techniques to increase the effectiveness of the service selection process. This is especially important when service selection operates over millions of potentially relevant services. Some work has been done to cluster services, as in the case of the BrightPlanet directory [14]. Clustering provides an eager classification of services according to their description. There are also opportunities to cache selection results from previous queries, allowing selection for re-issued queries to only focus on the set of services that have changed since the last invocation.

Efficiency can also be examined from the standpoint of services that are intermittent or evolving over time. How accurately the selection metadata reflects the actual state of the services can have a huge impact on efficiency and accuracy. This can be somewhat ameliorated by applying change detection capabilities to watch for evolving services, and by using low-level network protocols to ensure servers are still available.

- **Adaptation** Dynamic adaptation is an important aspect of service selection in heterogeneous environments where the quality, responsiveness, and availability of services can vary dramatically during a single invocation. When combining services into complex workflows or query plans, adapting to these changes can have a dramatic effect on the efficiency of the system [4, 71, 84, 3, 67, 104].

143
Experimentation and Validation  We have attempted to present the performance characteristics and effectiveness of each of the techniques described in this paper. In particular the Omini work on information extraction provides a validation framework based on a test bed of results measuring two distinct phases of the information extraction process. However, the evaluation techniques are still in early stages of refinement. In particular there are no known standardized benchmarks by which to compare such topics as service selection and change detection. For further attempts to provide efficient, scalable, and robust implementations of these and other basic building blocks, a set of benchmarks need to be developed. We anticipate that there will be several aspects of the benchmarks that would measure different aspects of change detection, service selection, or information extraction concentrating on basic, common tasks, and providing a set of domain-specific tasks that capture common tasks in a specific domain (such as information extraction in a BLAST search in bioinformatics, selecting appropriate bioinformatics sources for a BLAST query, or monitoring a set of pages from a specific Web site). Successful standardization efforts include the TPC benchmarks for database management system performance [107] and TREC, the text retrieval conference organized by NIST (the National Institute of Standards and Technology)\(^3\).

While we have listed several important research challenges associated with building Web services, there are many important issues not mentioned. Here we have opened the door to further research that will provide increased robustness and a richer source of content and information management as the Web fades from an artificial constructed reality to part of the fabric of everyday life.

5.3 Summarization

This thesis has presented three building blocks for building composable Web services: object extraction, service selection, and change detection. While not a comprehensive list of all types of components for composing services, these three components demonstrate a range of different levels of abstraction in the construction of services: data extraction and interfacing with legacy information, integration of information from multiple service peers, and aggregation and extension of services to provide additional value based on the content of existing services (service aggregation

\(^3\)http://trec.nist.gov/
and monitoring).

These components are particularly useful in the context of legacy data already available online. This data does not conform to newer standards for automated information interchange, or it was primarily designed for human browsing rather than automated processing. It is loosely defined, inconsistent, rife with syntactic errors that make it unsuitable for merging with the more rigidly defined standards with precise semantics and a more unforgiving⁴ data representation language.

The easiest solution, from a technical point of view, is to have each organization or person responsible for the existing services convert them into resources that comply with current standards so that they can integrate with the rest of the Web Service community. For purely social reasons this is unlikely to succeed, as many people are unwilling to change a service (often provided for free) that continues to fulfill the need for which it was originally designed. Resources are much more likely to be invested in new services or in augmenting the current value, rather than rewriting existing applications.

What we have described in this thesis is mechanisms useful to make current services more capable, without a reinvestment by the current owners to change how they interact with the rest of the world. This type of effort was originally devoted to creating wrappers that would make services appear to behave in a manner that consistent with applications that want to integrate the information provided. However, wrappers were typically labor-intensive, brittle creations that were only useful to their creators, and did not make the services available to the rest of the Internet in a standard way. Instead of continuing in making more advanced wrappers for services, we have concentrated on creating pieces of an infrastructure that will obviate the need for wrappers as they were originally implemented.

We have argued that successful building blocks must exhibit three characteristics: efficiency, scalability, and robustness. These characteristics are essential for any fundamental component of

⁴ Draco (c.659-c.601 B.C.E.) introduced the first written legislation to Athens. His code was consistent in that it decreed the death penalty for crimes both low and high. Similarly, a conforming XML processor must "not continue normal processing" once it detects a fatal error. Phrases used to amplify this warning have included "halt and catch fire"; "bust"; "flush the document down the toilet"; and "penalize innocent end-users" [19].
large-scale systems. As the Web continues to grow, services that display these characteristics will become an integral part of the network fabric, while services that are fragile, fail to scale, or are inefficient will be discarded in favor of new technology.
REFERENCES


149


VITA

Background  David Buttler was born in Loma Linda, California. When he was four, he convinced his parents to move to Alberta, Canada. In search of new challenges and life experience, he left the prairie to teach English for a year in Korea during his undergraduate program. He received a Bachelor of Science in Mathematics from Andrews University, Michigan in 1995, and a second Bachelor of Science in Computer Science from the University of Alberta in 1998. He spent two years as a research programmer at the Oregon Graduate Institute before beginning his Ph.D. studies at the Georgia Institute of Technology in 1999.

Projects  He has had the good fortune to work on the following research projects: At Georgia Institute of Technology, he created the Omni project, a fundamental component of XWrap Elite, and has contributed to the WebCQ project. He is currently working on service selection, participating in the Department of Energy SciDMC (Scientific Discovery through Advanced Computing) project for scientific workflows, and another DOE project for automated discovery and wrapping of bioinformatics information sources. At Oregon Graduate Institute he joined the database and systems group, and participated in the Continual Query project, the Cooperative Agent-based systems project, the XWrap wrapper generator, and the Query Routing projects.