Nanyang Technological University

Information Concierge for the World Wide Web

Ph.D Thesis

Submitted to the School of Computer Engineering

of the Nanyang Technological University

by

Li Zhao

For the degree of

Doctor of Philosophy

2005

Abstract

With the rapid growth of information on the web, a means to combat information overload is critical. Web data extraction systems has been developed to transform, evaluate, manage and present Web documents on behalf of requirements of various applications. Earlier work described in literature usually focuses on single phase in whole data extraction process, especially generation of rules to transform Web documents; i.e., wrapper induction. They appear ad-hoc and are difficult to integrate when a holistic view is taken. Each phase in the data extraction process is disconnected and does not share a common foundation to make the building of a complete system straightforward.

In this thesis, we focus on conceptual studying of current concierge systems in literature, and demonstrate a holistic framework to Web information concierge systems. The principle component of our proposal is the notion of a document schema. Document *schemata* are patterns of structures embedded in documents. Queries based on schemata to Web documents are defined. Unlike traditional database schemata, we define schemata that represent fragments of Web documents rather than a whole document. Based on the layout of Web documents and appearance of these schemata in Web documents, some attributes of schemata can be exploited to evaluate how sensitive to user requirements the fragments are, how similar multiple documents are, etc. Once the document schemata are obtained, the various phases (e.g., training set preparation, wrapper induction, and document classification) can be easily integrated.

The feasibility of the framework depends on two aspects: (1) how consolidate the framework is; i.e., whether there is a theoretical basis to help analysis and integration of related techniques. (2) How efficient is the schema generation. In the first direction, we study the relationship among tree language theory, logical program and Web documents, and implement the framework using various techniques to show its flexibility. In another direction, it is proved that there is no practical algorithm to detect schema in general. The thesis studies a special class of schemata (k-schemata), and proposes some efficient methods to detect k-schemata. By parsing documents to trees, an $O(n \log n)$ algorithm is introduced to detect frequent structure patterns (k-schemata) from these trees. By studying the relation between these structure patterns and flatten substrings in original documents, we devise a data structure, so-called DocItem trees to store schemata detected. Linear DocItem tree construction algorithms are presented. Schemata also can be located in DocItem tree in linear time.

The framework and related algorithms in this thesis imply improved efficiency and

better control over the extraction procedure. Our experimental results confirmed this. We also demonstrate an application of the framework — a visual query language similar with QBE query language. Importantly, since a document can be represented as a *vector* of schema, it can be easily incorporated into existing systems based on vector model of documents as the fabric for integration.

Acknowledgments

This thesis is the summary of the work in the past three years and the future research direction. The effort of writing it cannot be separated from support of other people. I would like to express my gratitude to these people for their kindness.

Associate Professor Ng Wee Keong, my supervisor, for his constructive suggestion and patience to show me future direction. As an advisor, he demonstrates knowledge and patience that I need as a research student. He goes through my work seriously word by word, and provides necessary research training to us. As the active director for Centre for Advanced Information Systems, he provide a conductive environment to my research. His efforts in organizing weekly presentation in centre enrich my knowledge.

Liu Zehua, **Li Wenyuan**, **Sun Aixin** three creative postgraduates in the our research group, for their cooperation in development and research. Also thanks for they review the paper I wrote.

Chen Ling, **Chen Qi**, **Khin Myo Win**, **Erwin Leonadi**, **Myo Myo Naing**, **Ong Kok Leong**, **Sun Aixin**, **Wang Yi Da**, **Xu Yi**, **Zhao Qian Kun**, **Yin Ming**, **Li Zhuowei**, who are postgraduates in our lab. We seat together almost every week to share our ideas, their knowledge shows the wide research areas to me.

Jiang Chang'an for all the care, concern and encouragement given by her during the working and writing of the first year report. Bless you.

Last but maybe the most important, **my parents**, for all support and sacrifice that they have given me to pursue my interest in research. Without them all these work is impossible.

Thank you all.

Contents

List of Figures

List of Tables

List of Algorithms

List of Notations

$\alpha, \beta, \alpha_i, \beta_i$ ..... Symbols
Σ ..... Alphabet
$\mathcal{T}(\Sigma)$ ..... Terms over Alphabet Σ
$\mapsto$ ..... Conforma to
t, t_i ..... Terms
n, n_i ..... Nodes in Trees
A, B, A_i, B_i ..... Atoms
$\mathcal{T}(\Sigma)$ ..... Terms
G, G_i ..... Grammars
$L(G)$ ..... the language generated from G
2^Q ..... Regular Set of Set Q
$dom(t)$ ..... Domain of Term t
DI ..... Data Instance
FO ..... First Order
MSO ..... Second Order
x, y, x_i, y_i ..... Variables Ranging over Elements
X, Y, X_i, Y_i ..... Variables Ranging over Set of Elements
τ ..... Built-in predicates for unranked trees
NTT ..... Non-deterministic Tree Transducer
TA ..... Tree Automata
SQA ..... strong query automata
Q ..... State set in Tree Automata
q, q_i ..... states in Tree Automata
F ..... Final state in Tree Automata
δ ..... Move Relation in Tree Automata
ω ..... Weight of Schema in a Document

ξ . . . . CharData (Chapter 5)

Chapter 1

Introduction

1.1 Background

The advances of information and communication technologies make it easier than any time to deliver data according to our desire. The exponential growth of data online produces a huge variety of information resources. Today we can find airline schedules, weather forecasts, up-to-date news or even the nearest restaurant on the Web. Unfortunately, the information sea surrounding us is overloading receptivity of human and nobody can absorb all the water in the sea. A question was rising — what is the meaning for us to produce a stunning quantity of information enthusiastically if we cannot find knowledge or wisdom in it.

"Without tools and methodologies for gathering, evaluating, managing, and presenting information, the Web's potential as a universe of knowledge could be lost." — *John December [1994].*

From the beginning of last decade, standards such as HTTP, HTML were widely adopted and people distributed data based on the infrastructure of the Web. Users can directly reach these data sources using browsers such as Mozilla and Internet Explorer. HTML Forms are the tools most frequently used by users to interact with Web service providers and gather interesting information. While the quantity of information becomes too huge to gather and process manually, we need alternative Web data process approaches — automatically information collection, collaging, managing and presenting.

In the past decades, database management systems (DBMS) are well-studied to manage large volume electronic structural data, and provide efficient accessing to the data.

Success of DBMS in relational data management is based on its solid relational model, convenient query interfaces, etc. As the substantial problems of Web data process is similar to DBMS — managing large volume data, relevant concepts of DBMS to Web data are extensively studied in recent years. Although DBMS techniques cannot be expected to solve all problems on Web, they are valuable complements of other related techniques; e.g., Web data mining , information retrieval (IR), natural language processing (NLP). This thesis will focus on studying those DBMS concepts relevant to information process on Web: (1) Modeling and querying Web documents. (2) Data Extraction. (3) Data Integration.

1.2 Information Concierge for Web

In this section, the task of the thesis is briefly introduced — to build an information concierge for Web.

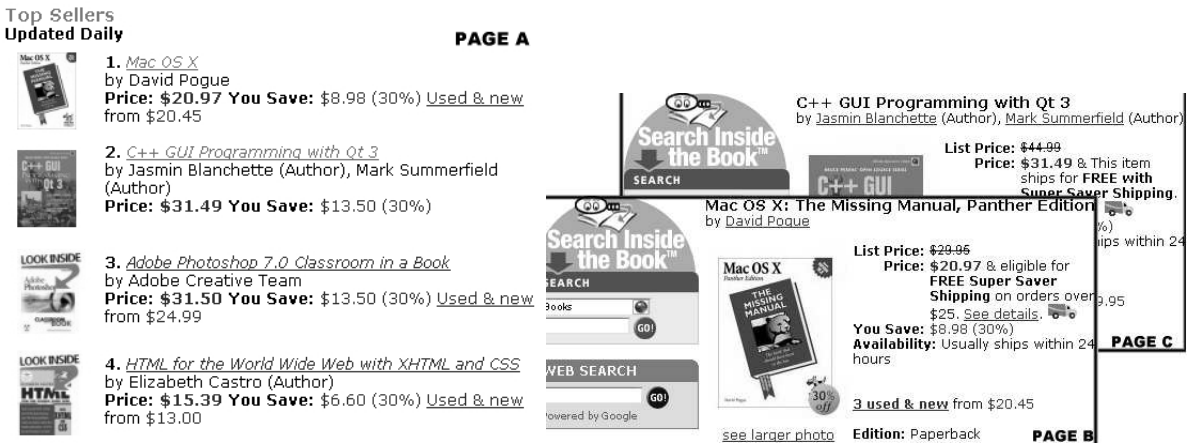
1.2.1 Semantic Web versus Concierge for Human-centered Web

Web is not a human-to-human communication media only, Tim Berners-Lee, the designer of Web argues that the Web should be designed to allow machines to participate and help Web to be an information and knowledge repository. W3C's effort on Semantic Web [Berners-Lee et al., 2001] is a step in this direction. Semantic Web can be viewed as a global database, and Web sites are suggested to be re-engineered to follow Semantic Web specifications; e.g., RDF [W3C, 2000], which is successful in some areas.

However, not all enterprises and organizations want to disclose their information in an open infrastructure for various reasons. In these situations, only human-friendly Web documents are presented. Thus, we need concierge systems that provide machine-sensitive data accessing over these human-friendly documents. In the following parts of this thesis, if there is no confusion, concierge, information concierge and Web information concierge have the same meaning.

1.2.2 Challenges and Objectives

A concierge system should extract structural data from semistructural Web documents. The challenges for a concierge system mostly come from the following characteristics of semistructural Web documents:



(a) Top Seller Books (b) Book Details

Figure 1.1: Sample Pages

- Semistructural documents have irregular structures only. As the examples in Figure 1.1, some books have second hand price while others have none; the same book's information is presented with different formats.
- Document schemata are hidden from users. In this thesis, a schema is referred as a structure pattern in Web documents. Even when some documents are generated by regular rules and share common schemata, these schemata usually are hidden to users.
- It is a hard problem to induce hidden schemata from sample documents.

Without restriction on the structure of schemata, we notice that general schema detection problem is hard. Similarly, putting proper restriction on document structures is important for efficiently extraction. The problem of how to choose models for Web documents includes:

- There is a tradeoff between expressive capability and operation efficiency of Web document models.
- The model chosen should be convenient to build relation with other formalism and be theoretically analyzed.

Data extracted that shares common schemata should be integrated, such that a user can query integrated data. Data integration needs to know the similarity among data, and

it also needs to choose data model carefully to guarantee efficiency. Summarizing the problems of building a concierge system, we briefly state the objective of this thesis as below:

We need to build a **concierge system** *between human-centered Web data resources and end-users, which is based on a consistent* **data model** *over* **irregular documents** *with* **implicit schema**, *and provide* **efficient** *and* **extensible** *query interface to other applications.*

1.3 Framework for Web Information Concierge

A Web information concierge system must have the capability to efficiently extract data from Web. To extract data from a given Web document efficiently, a solution is to build a program, so-called a wrapper [Kushmerick, 2000a], reading the document and outputting fragments of the documents. The extracted document fragments are named data instances.

At first, wrappers are built using generic programming languages, like Java and C, and it is still a feasible method when the wrapper only needs to extract data from several Web documents. With the coming of Web information concierge systems, more specific programming languages [Gupta et al., 1997] are invented, to analyze Web data structures and output special parts in a procedural way. However, the wrappers written using these procedural languages are rather ad-hoc, and difficult to be compared with each other. Just like the relationship between procedural database query languages and declarative database query languages, people is inclined to use declarative wrapper generation languages in a unique framework, such that it is easier to compare these wrappers and share knowledge to generate wrappers.

Fortunately, much conceptual modeling [Liu et al., 2002b, da Silva et al., 2002] studying on Web data opens the door to define declarative wrapper generation languages. Usually, Web data is modeled in a two-layer framework. The physical layer contains data instances to be extracted, and the logic layer contains concepts describing syntax structure of data instances. A declarative wrapper generation language describes abstract concepts of a class of data instances, and an extraction engine then can map these concepts to physical data and do real extraction.

Conceptual modeling of Web data and its relationship with wrapper generation lan-

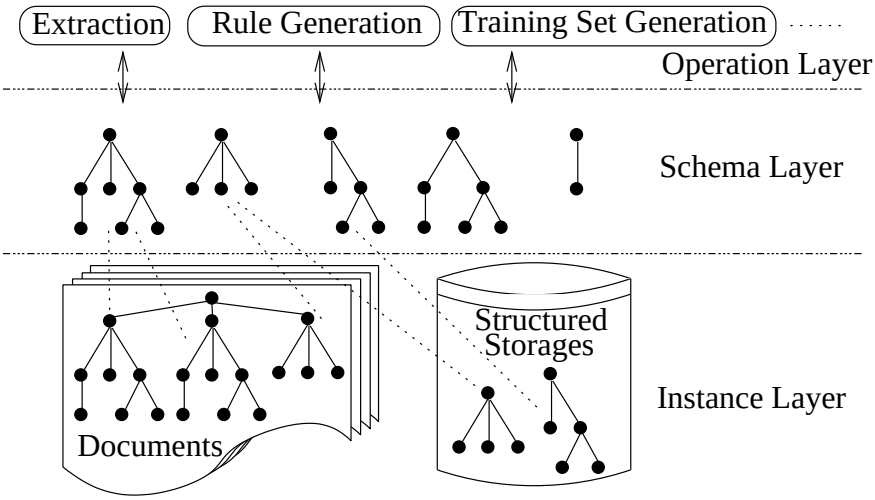


Figure 1.2: Framework for Web Data Extraction

guages are main points of this thesis, which will be detailed discussed in Chapter 4. In this thesis, concepts corresponding to data instances are denoted as schemata. This thesis defines schemata that can describe both original Web data and extracted data, thus, providing a common basis to an declarative language that can access both Web data and extracted data.

Another problem for concierge system is to provide convenient query interface for users. Compared with research on how to model Web data, the studying is weak to provide complicated query operations, e.g., how to obtain the similarity among extracted data. This thesis introduces some initial work on this direction.

We organizes our research work of conceptual modeling of Web data and how to query Web data in a framework that draws ideas from relational databases. A relational database contains at least a relational data model and relational operations. A relational data model includes *data instances* (2-dimensional tables consisting of sets of tuples) and *schemata* (descriptions of those instances). Relational operations are based on sets; i.e., the inputs and returned results of them are tables, instead of individual tuples. Tuples in a relational database may have different structures and can be changed by operations. However, the schema of a table is static. Thus, it is possible to provide a consistent way for all those operations to handle various tuples in the system.

We adapt the relational framework in our Web data extraction framework (Figure 1.2). As our framework works on semi-structured data, it has substantial differences

from the relational framework. Relational schemata only describe linear tuples, while in our framework, schemata should describe tree structures. A relational instance is a table with a unique schema. In our framework, a data instance may be a document, a fragment of a document or a piece of extracted structured data. A document may correspond to multiple schemata, and a schema may correspond to multiple data instances. In Figure 1.2, these dash lines among data instances and schemata represent corresponding relationships among them.

The core of our framework is the *schema layer*. Schemata describe data instances and provide information to the operation layer. To put our framework into practical systems, there are some interesting problems to address; e.g., how to detect these schemata. In the following chapters, these problems will be discussed in detail.

1.4 Related Techniques

In this thesis, we shall introduce the techniques that enable construction of Web information concierge systems based on the proposed framework. This section gives an overview over related areas that lead to our work. More details will be covered in later chapters.

As mentioned in Section 1.3, how to build the conceptual modeling is one of our main consideration, i.e., how to describe syntax structure of data instances. As most Web data are HTML or XML documents, and usually there are corresponding grammars to generate these documents, it is intuitive to use these grammars to describe syntax structure of data instances. Grammar induction thus is an enabling technique.

1.4.1 Machine Learning and Grammar Induction

To discover hidden schemata from a class of documents is an important task of a concierge system. Usually, the schemata can only be guessed based on evidence observed in downloaded documents. The schema detection problem is strongly related to machine learning, defined by Mitchell [1997] as:

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Machine learning techniques, such as NLP and grammar induction is the heart of many information extraction applications. For example, word segmentation and part

of speech tagging [Appelt and Israel, 1999] are important to extract data from free text documents. Similarly, to extract Web documents, grammar induction is introduced in literature [Kushmerick, 2000a]. Based on materials from Gold [1967], grammar induction can formulated as a machine learning problem as:

- Task: To give a hyperthesis grammar of an target language.
- Experience: A set of examples about the target language. When an example belongs to the language, it is positive, otherwise negative.
- Measurement: The conformity of the generated hyperthesis grammar with the grammar of target language.

Based on the difference of experience given, grammar induction is classified to several sub-problems. For instance, when examples are labeled with positive and negative, it is a supervised grammar induction. If only positive examples are given, it is unsupervised grammar induction. More classes of grammar induction will be introduced in later sections. Generally, unsupervised grammar induction is much more difficult than the supervised one. To detect hidden schemata from Web documents can be formulated as grammar induction problem. However, normally, there are only positive examples on Web, and schema detection is not an easy task.

As the limitation of sample documents available on Web, only very trivial languages can be induced. Gold [1967] argues that only finite languages can be induced from positive examples; i.e., even regular languages cannot be induced. From more complete survey of recent research [Angluin and Smith, 1983, Knuutila, 1994], we see the languages can be induced from positive is not such trivial type, Knuutila [1994] proves that no general algorithm is available to induce a language in polynomial time in the size of the machine accept the language. Chapter 5 will discuss more about grammar induction.

1.4.2 Structural Data Mining

Research on grammar induction is quite mature, especially for these theoretical problems such as, what type of languages can exactly induced from what type of information in hand. However, considering the performance requirements of practical concierge systems, it is almost impossible to induce exact grammar of given Web data. One solution to this problem is to exploit structural data mining techniques. The idea is that we can first detect some structures that may contain most important contents from Web data,

compared with the original Web data, the syntax of the selected structures should be much simpler; then it may be easier to induce grammar for these structures.

There are so many data available in databases that to discover much knowledge from databases is almost an impossible mission for human without the aid of computers. The motivation of data mining is to provide methods to find knowledge in large volume data in databases. *Classification, Clustering* and *Association Rule mining* are the common tasks of data mining. Structural data mining is a sub-area in data mining that detect structural patterns from data. Tree pattern mining from semi-structured documents is studied in these two years [Asai et al., 2003, Chi et al., 2003, Termier et al., 2002, Nijssen and Kok, 2003, Zaki, 2002]. Clustering and pattern detection techniques are exploited to induce grammar by some heuristic approaches [Geertzen, 2003].

1.4.3 Semistructural Data Query Processing

Operation layer is a user will directly handle, and query is the basic component in this layer. Most documents on Web are HTML or XML documents, and we concentrate our discussion on query over XML and HTML documents. Thus, it is intuitive to represent Web documents using XML model, which is a labeled graph data model. In this model, nodes represent elements, and nodes are connected by parent-child relationships, XLinks, etc. Web documents have some special characters [Florescu et al., 1998] that make query over Web documents much different from database query. XQuery and XSLT are specially designed XML query languages. As the existing of recursive functions in these languages, they are both turing-complete [Kepser, 2004]. Optimization techniques for them are hot research area in these two years. To guarantee the efficiency of query processing, some relative restricted query languages are defined [Florescu et al., 1998]. In this thesis, we focus on devising query languages that can be processed in polynomial time.

1.5 Organization of Thesis

This thesis is organized as bellow. In this chapter, we give a brief introduction of the background of study about information concierge systems, which build an interface that can be understood by machines to efficiently access data on Web. An overview of our framework is given to connect DBMS concepts to concierge systems. The problems encountered in the study are briefly presented.

Chapter 2 surveys related literature that influences our work to build information concierges. It discusses problems in various components of concierge systems, including the kernel component to extract data and some auxiliary components that help extraction and query over extracted data. Before going on to discuss a unified view over all components in a concierge system, this chapter provides an introduction to related techniques, and discusses the common and different points among them. The state-of-the-art related systems are compared with our work.

Chapter 3 does initial study of the framework introduced in this chapter. Based on the framework, a Web document model, the extraction and query over Web documents are introduced. Building this framework is the first step to analyze existing approaches and integrate them.

Chapter 4 introduces the details inside the framework. Different characterizations of the framework are demonstrated from various perspectives; e.g., logical program and tree automata theory. The cross-fertilization relationships among these characterizations can help the further study. Based on these views, a Web data extraction language (Wdel) with expressive power roughly coinciding with monadic second-order (MSO) logic is introduced. Wdel is used to define rule of extraction by defining the mapping from schemata of Web documents and schemata of extracted data.

Chapter 5 demonstrates algorithm details. Firstly, a constrained version of schema detection is introduced. By parsing Web documents into trees and schema detection is deduced to a problem of common subtree detection problem. A $O(n \log n)$ algorithm is introduced to solve the problem. By connecting the problem with common substring detection, we devise a data structure, named DocItem tree, which can be constructed in time linear in the document size. DocItem tree can act as a dictionary of schemata that allow schema search in linear time in the schema size.

Chapter 6 demonstrates the flexibility of our framework by introducing how to provide auxiliary operations in the framework. Chapter 7 introduces the implementation details of discussed techniques and presents experiment data to show the feasibility of them.

Chapter 8 makes conclusion of the thesis. This thesis introduces the results and contribution of my research work in last three years. The last chapter will summarize them and propose potential research problems.

Chapter 2

Literature Survey

In the concierge framework of Figure 1.2, *extraction* is the kernel component in the operation layer. In this chapter, we first briefly introduce basic constituents of extraction operation by demonstrating an initial Web data extraction system. Some improvement to the initial system is discussed to investigate research directions. Related techniques supporting the extraction component are also discussed. The state-of-the-art of the information concierge related techniques are the foundation of work in following chapters.

2.1 Basic Model for Web Data Extraction

In this section, we focus on discussing extracting data from Web documents. Most data available on Web are represented using semi-structured data model, for example, HTML, XML and LaTeX documents. These documents are not like free texts written in natural languages without obvious structures. In semi-structured documents we can find delimiters such as tags in HTML pages, which construct the structure of documents. On the other hand, these structure information is not perfect like those in structured data. Data schemata describing the rules of structure organizing usually are hidden from users or do not exist at all. Much unrelated information or noise such as commercial advertisement or wrong handwritten codes may appear in Web documents. Sometimes the delimiters in these documents may be used to define the visual layout displayed in Web browsers, and have no relation with semantic information of the documents.

For above reasons, automatically extract information from semi-structured Web documents is not a trivial task. To explain the problems in extraction in detail, we will begin with some notions of various aspects of a Web data extraction component, including how

to represent Web documents, schemata, extraction tasks, etc. A class of representations of these aspects is called a Web data extraction model. Such a model should at least consist of five aspects:

- the data model of target Web documents to be extracted,
- the output data model delivered to users.
- the extraction rule model of how to process these target documents and extract data from them,
- the methods to generate extraction rules,
- the methods to execute extraction rules,

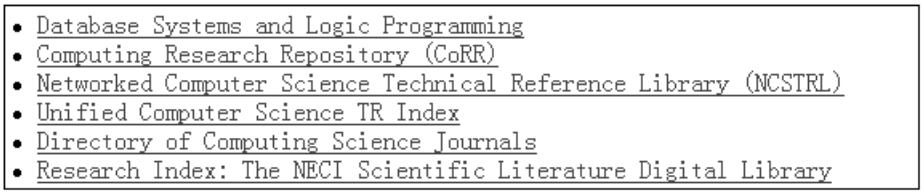
2.1.1 Target Document Model

Most websites are built based on backend databases [Florescu et al., 1998]. They generate HTML or XML documents given a set of rules of how to map data in structured databases into semi-structured documents. These rules are invisible to us. It is essential for us to find the rules from the evidence inferred from the structures and the contents of documents we can access.

Generally, Web documents can be modeled as directed graphs [Florescu et al., 1998]. However, Kushmerick [2000a] argues that there is a trade-off between the complexity of target document models and efficiency of extraction rule induction, and the rule generation based on graph model is relative difficult to be implemented efficiently. Sometimes, Web page layout is quite simple and regular, such as Figure 2.1(a) that is rendered from document in Figure 2.1(b). Kushmerick [2000a] suggests a relational target document model to describe these simple Web documents, described as below:

$$\left\{ \begin{array}{c} \langle b_{1,1}, f_{1,1}, e_{1,1}, \ldots, b_{1,j}, f_{1,j}, e_{1,j}, \ldots, b_{1,n}, f_{1,n}, e_{1,n} \rangle \\ \langle b_{i,1}, f_{i,1}, e_{i,1}, \ldots, b_{i,j}, f_{i,j}, e_{i,j}, \ldots, b_{i,n}, f_{i,n}, e_{i,n} \rangle \\ \langle b_{k,1}, f_{k,1}, e_{k,1}, \ldots, b_{k,j}, f_{k,j}, e_{k,j}, \ldots, b_{k,n}, f_{k,n}, e_{k,n} \rangle \end{array} \right\}$$

In this model, each row is a record in Web documents. Each $f_{x,y}$ is a string in documents to be extracted, and $b_{x,y}$ and $e_{x,y}$ are delimiters around these fragments.



(a)An HTML page listing several search websites

```
01  <LI><A HREF="¨http://www.informatik.uni-trier.de/~ley/db/index.-
      html¨">Database Systems and Logic Programming</A>
02  <LI><A HREF="¨http://xxx.lanl.gov/archive/cs/intro.html¨">Comput-
      ing Research Repository (CoRR)</A>
03  <LI><A HREF="¨http://www.ncstrl.org/">Networked Computer Science
      Technical Reference Library (NCSTRL)</A>
04  <LI><A HREF="¨http://www.cs.indiana.edu:800/cstr/search¨">Unified
      Computer Science TR Index</A>
05  <LI><A HREF="¨http://fas.sfu.ca/project...ons/CMPT/cs-journals/¨">
      Directory of Computing Science Journals</A>
06  <LI><A HREF="¨http://www.researchindex.com/¨">Research Index: The
      NECI Scientific Literature Digital Library</A>
```

(b) an HTML text example

Figure 2.1: Example of Web Data

Example 2.1. In the relational model, lines 1 through 6 in Figure 2.1(b) are six records; each record contains information of a search Web search engine including its name and URL. These lines in this simple HTML document are easy to be represented using a relational table. For example, the first line is represented as below:

⟨<LI><A HREF=", http://www.informatik.uni-trier.de/~ley/db/index.html, ", >, Database Systems and Logic Programming,</A>⟩

In this tuple, $b_{1,1}$ is <LI><A HREF=", $f_{1,1}$ is http://www.informatik.uni-trier.de/~ley/db/index.html, $e_{1,1}$ is ", etc.

2.1.2 Output Data Model

The HTML document in Figure 2.1(b) is a string s over a finite alphabet Σ; i.e., $s \in \Sigma*$, where $\Sigma*$ is the application of Kleene closure to Σ. Data extraction components extract

sub-strings from it. Suppose a user want to know all engines' name and URL from the document, these interesting parts in s can be modeled as standard relational data model. In the relational model, each relation (or table) is a set of records. Each record is a tuple with fixed arity; i.e., the number of atomic fields in the tuple. A tuple can be denoted as $\langle f_1, f_2, \ldots, f_n \rangle$, where f_1 through f_n are fields in the tuple and n is arity of the tuple. The domain of these fields is the set of sub-strings in s; i.e., each field can be a sub-string in the HTML document.

Based on this model, interesting contents in Figure 2.1(b) can be stored in a table shown in Example 2.2, where each row is a tuple and each column corresponds to a field.

Example 2.2. The relational table extracted from the HTML document in Figure 2.1(b) is drawn as below:

$$\left\{ \begin{array}{cc} \langle URL_1 & SiteName_1 \rangle \\ \langle URL_2 & SiteName_2 \rangle \\ \langle URL_3 & SiteName_3 \rangle \\ \vdots & \vdots \end{array} \right\}$$

This relational data model is not enough to describe various kinds of data extracted. For example, a query submitted to *Google* will return an HTML page that is very simple to read by human. However, it is not easy to resort to a unique relational table to store all information in it.

If we submit a query "Wrapper generation" to *Google*, it will return an HTML page like the one in Figure 2.2. This page contains a list of records about online resources related to "wrapper generation". We can see that some records maybe have fields that are different from others. This page also uses different layouts to organize records describing resources from the same Website. These information is difficult to be modeled as a standard relational table. The problem is more complex if we represent Web documents as data graphs. These more complicated situations will be discussed later.

2.1.3 Rule Model

In above sections, relational model is exploited to describe both target Web documents and extracted data. Kushmerick [1997] defines *wrapper* that fills the gap between target Web documents and extraction data, as below:

Definition 2.1. Given:

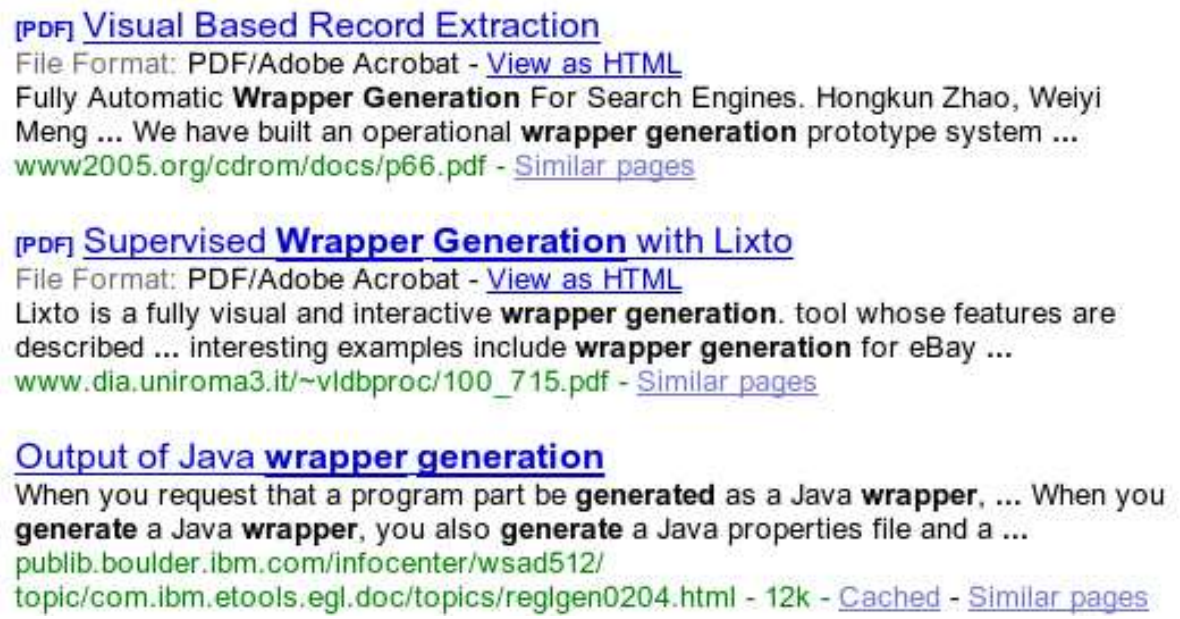


Figure 2.2: Fragment of HTML Page Returned by *Google*

- a set of example Web pages $\mathcal{P} = \{\ldots, P, \ldots\}$,
- a set of *label* $\mathcal{L} = \{\ldots, L, \ldots\}$,

A wrapper W is a function from $\mathcal{P}$ to $\mathcal{L}$.

This is why sometimes we also call Web data extraction components as wrappers. However, we believe it is more flexible to divide a wrapper into an extraction rule model and an extraction algorithmic framework that interprets the rule model. The extraction rules represent the mapping relationship between target Web documents and extracted data. Extraction Algorithm execute these rules to do real extraction. Defining an independent rule model can highlight the mapping knowledge representation, which is important to compare various methods, as shown in later chapters.

Using the relational model in Example 2.2, Kushmerick's [1997] basic extraction algorithm can process a Web documents that can be represented as a list of tuples: $\langle \ell_1, f_{1,1}, r_1, \ell_2, f_{1,2}, r_2, \ldots, \ell_n, f_{1,n}, \ell_n \rangle \ldots \langle \ell_1, f_{k,1}, r_1, \ell_2, f_{k,2}, r_2, \ldots, \ell_n, f_{k,n}, r_n \rangle$. That is, the delimiter pairs around the same fields of all tuples are the same. Thus, only the delimiter pairs for the first tuple are needed to do extraction. Kushmerick [1997] defines a rule model that represents a rule as a tuple of delimiter pairs $\langle \ell_1, r_1, \ell_2, r_2, \ldots \ell_n, r_n \rangle$. For the

target document in Example 2.1 and extracted data in Example 2.2, the extraction rule is:

⟨<LI><A HREF=",",>,</A>⟩

2.1.4 Extraction Procedure Model

Algorithm 1 Extraction

Require: Web Document $P = \langle \ell_1 f_{1,1} r_1 \ell_2 f_{1.2} r_2 \ldots \rangle$
Require: Rules $R = \langle \ell_1, r_1, \ell_2, r_2, \ldots, \ell_n, r_n \rangle$
Ensure: $O = \langle f_{1,1}, f_{1,2}, \ldots f_{1,n} \rangle \ldots$
Ensure:
  while !EOF(P) **do**
    for $i = 1; i \leqslant n;$i++ **do**
      $f_i \leftarrow$ substring between ℓ_i and r_i in P;
    end for
  end while

Extraction rules describe how to map from target documents to extracted data. A tuple $\langle \ell_1, r_1, \ell_2, r_2, \ldots, \ell_n, r_n \rangle$ is an extraction rule of the basic extraction rule model ℓ_i is a text string marks the left side of ith field f_i and r_i marking the right part. The extraction procedure reads target Web document P, and map P to output tuples O. Algorithm 1 is the pseudo-codes of an extraction procedure.

In the extraction rule, each pair of ℓ_i, r_i can be used to locate a text string to be extracted. The extraction procedure starts from the beginning of document P and then locates interesting fragments one by one.

2.1.5 Rule Generation Model

There are two basic approaches to build rules, so-called *Knowledge Engineering Approach* and *Machine Learning Approach* [Appelt and Israel, 1999]. The first method needs a person called "knowledge engineer" who is familiar with the extraction component to write rules. Quite different from that, machine learning approach does not need people to deal with the details of the working mechanism of an extraction component and develop rules manually.

Obviously, knowledge engineering approaches is easy to implement if there exist experts of Web data extraction. This is the reason why those concierge systems based on

these methods [Gupta et al., 1997] can appear years before systems based on machine learning methods. However, to generate rules manually is time-consuming and error-prone. Kushmerick [1997] introduces some initial study about inducing extraction rules based on the models introduced above.

Kushmerick [1997] defines rules generation as a grammar induction problem; i.e., suppose target Web documents are generated from a target grammar; rule generation is to make some hypothesis of the target grammar based on some training documents selected from target documents.

Algorithm 2 RuleInduction

Require: $\mathcal{P}$ — Training Documents
Require: F — the set of fields in $\mathcal{P}$
Ensure: R — Rule Set
  $R \leftarrow emptyset$
  $E \leftarrow$ all possible rules
  for $e \in E$ **do**
    if $Extraction(\mathcal{P}, e) \neq \emptyset$ **then**
      $R \leftarrow R \cup e$
    end if
  end for

Algorithm 2 is the extraction rule generation procedure based on extraction model introduced above. It first exploits an oracle to label all fields to be extracted. This oracle can be a person or an intelligent program [Rajaraman and Ullman, 2001]. Then, it generates all possible extraction rules; i.e., a list of string pairs enclosing each field. Finally, the algorithm tests all these rules using Algorithm 1. If a rule can produce the same output fields as those labeled by the oracle, the rule is inserted to the final rule set.

2.1.6 Improvements of Basic Model

The basic data extraction model can handle relational structural documents only. Using the simple rule ⟨`<LI><A HREF=",",>,</A>`⟩, Algorithm 1 can extract from the document in Figure 2.1(b), and produces a table like that in Example 2.2. As stated in [Kushmerick, 1997], about half of Web documents can be extracted based on this model. However, it will fail in more complicated documents.

Example 2.3. Suppose we insert the following HTML code into the Web documents in Figure 2.1(b) before line 1:

```
<LI><A HREF="http://home/"><B>Information</B></A>
```

The basic extraction model cannot work any more. If the extraction procedure still uses the same rule, it will extract wrong information, i.e., there will be unwanted tags in the name extracted. Moreover, no rule can be generated based on the basic model. More enhanced versions of the basic model are introduced [Kushmerick and Thomas, 2002, Sakamoto et al., 2001, Thomas, 1999a]. In this section, we focus on the improvement on the target document model and corresponding changing of rule model.

Tabular Models Target Documents The basic wrapper model can extract data from Web documents with relational structures. Relational structural data appear in numerous Web documents, however, the contents surrounding these relational structures make situation more complex. Kushmerick [2000a] enhances the basic extraction model to extract more complicated documents, so-called tabular documents. The first extension is the Head-Left-Right-Tail (HLRT) model in which extraction rules have two additional delimiters: *Head* and *Tail*. These delimiters are used to locate relational fragments in documents. For example, to solve the problem in Example 2.3, we can define a Head delimiter – `</A>` to skip the new line, and then process the left lines like the procedure in the basic extraction model. Now we should change the extraction rule to

⟨ `</A>,<LI><A HREF=",",>,</A>,</HTML>`⟩

Example 2.4. If we add another line before each record in the Web document in Figure 2.1(b), the HTML document is changed to below:

```
        ⋮
01   <LI><A HREF="http://xx.xx">icon1</A>
02   <LI><A HREF="http://www.informatik.uni-trier.de/~ley/db/index.
       html">Database Systems and Logic Programming</A>
03   <LI><A HREF="http://xx.xx">icon2</A>
04   <LI><A HREF="http://xxx.lanl.gov/archive/cs/intro.html">Comput-
       ing Research Repository (CoRR)</A>
05   <LI><A HREF="http://xx.xx">icon3</A>
06   <LI><A HREF="http://www.ncstrl.org/">Networked Computer Science
       Technical Reference Library (NCSTRL)</A>
        ⋮
```

Neither basic model nor HLRT model can correctly extract those websites' information from the above documents. Kushmerick [2000a] suggests the second extension Open-Close-Left-Right (OCLR). OCLR extraction rule exploits two strings o and c to mark the beginning and end of each record. To extract the above documents, extraction rule is revised as below:

⟨ `</A>,<LI><A HREF=",",>,</A>,</LI>`⟩

In this rule, `</A>` is used to located the beginning of each record, while `</LI>` is used to located the end and each record. Head-Open-Close-Left-Right-Tail (HOCLRT) is another enhancement of the basic model that combines OCLR and HLRT model to provide more flexibility.

These extraction models introduced in this section extend the basic extraction model by adding some delimiter to locate a regular relational fragment in Web documents. They can be used in more situations on the Web, but still many Web documents are not tabular or the tabular structure cannot be represented using these extraction models introduced.

Hierarchical Models for Target Documents Much information is not easy to be described using tabular structured documents, for example, the contents of a book, the file folders information of a hard disk, etc. Although most relational databases integrated XML storage features, which exploits nested relational tables to store transformed hierarchical XML documents, their performance is not enough for storing and querying large volume Web data. We prefer hierarchical structured documents to describe such kind of information and define hierarchical extraction models to extract data from these documents.

Example 2.5. Figure 2.3 is a part of the contents from a book. Unlike the HTML document in Example 2.2, we cannot use the relational data model to represent this kind of information. The contents have a hierarchical structure shown in Figure 2.4.

Nested-Left-Right (NLR) is an extension of the basic model to process the above documents. To extract this documents, we can use an extraction rule as below:

$$\left\{ \begin{array}{ll} \langle \ell_{1,1}, r_{1,1} \rangle & \left\{ \begin{array}{ll} \langle \ell_{1.1,2}, r_{1.1,2} \rangle & \left\{ \begin{array}{l} \langle \ell_{1.1.1,3}, r_{1.1.1,3} \rangle \\ \langle \ell_{1.1.2,3}, r_{1.1.2,3} \rangle \end{array} \right\} \\ \langle \ell_{1.2,2}, r_{1.2,2} \rangle & \langle \ell_{1.2.1,3}, r_{1.2.1,3} \rangle \end{array} \right\} \\ \langle \ell_{2,1}, r_{2,1} \rangle & \{\ldots\} \end{array} \right\}$$

```
01 Part I Introduction
02   1 The concept of the C++ Standard Template Library
03     1.1 Genericity of components
04     1.2 Abstract and implicit data types
05     ⋮
06   2 Iterators
07     2.1 Iterator properties
08     ⋮
09 Part II Algorithms
10   1 ···
```

Figure 2.3: Contents of Book

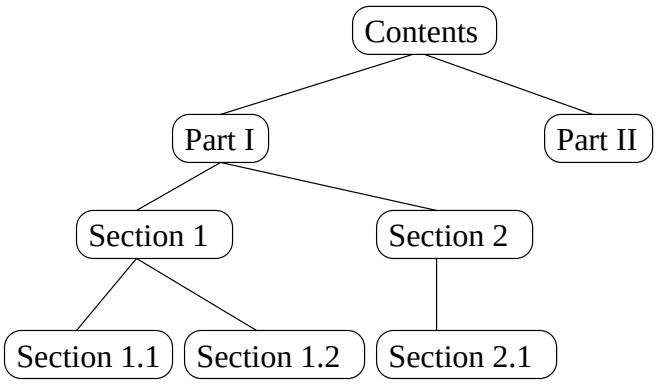


Figure 2.4: Hierarchical Structure of the Contents of a Book

In this rule, each column is a set of delimiter pairs to locate fields corresponding to nodes in the same layer of the hierarchical structure. For example, $\langle \ell_{1,1}, r_{1,1} \rangle$ and $\langle \ell_{2,1}, r_{2,1} \rangle$ in the first column can locate Part I and Part II.

All these extraction models introduced so far are based on an assumption that there is a common surrounding delimiter pair for the same fields in each records. This assumption is too strong, although for some simple Web documents it's true. More expressive models will be introduced in later parts of this thesis.

We must notice that there exists a tradeoff between extraction rule models' expressive power and the automatic rule generation efficiency. For example, the NLR model takes time growing up exponentially in the number of fields, and HOCLRT model requires much larger training set than that needed by OCLR and HLRT models to achieve the same precision. It is important to define compromise model that is "good" enough to represent Web documents and easy to generate extraction rules.

2.2 Variant Formalisms for Target Documents and Rules

In Section 2.1, we have introduced an execution procedure that extracts documents with relational formats. Extraction procedures of this model are quite simple and are mostly based on relational data model. Relational model is very weak to describe complicated documents. In Chapter 4, we shall compare expressive capability of relational document model and other document models in detail. We have presented documents that cannot be extracted by the basic extraction model in Example 2.5. Although some enhanced models are introduced, we can easily find documents fail them. We want the extraction model can be more expressive and flexible, otherwise the extraction procedure may be fragile and cannot get any information in many situations.

There is a tradeoff between expressive power and learning efficiency. The rule learning procedure introduced in the basic extraction model uses inductive learning method to generate extraction rules. Kushmerick [2000a] compares the rule learning efficiency of the basic wrapper model, NLR, HTLR, OCLR, etc. We can see the time cost of learning procedure is very expensive. One reason for the high cost is Kushmerick [2000a] builds rule generation procedure on PAC-learning model [Valiant, 1984]. We shall introduce some other learning models and heuristic methods that can induce acceptable extraction

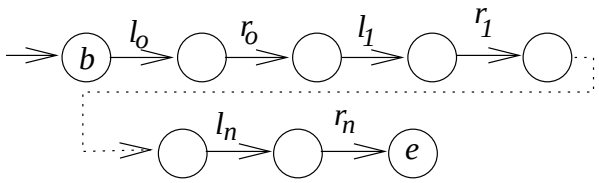


Figure 2.5: Finite State Automata

rules later. In this section, we concentrate on extraction models that exploit variant target document models and rule models.

2.2.1 Word Language

These extraction rules models we introduced can be analyzed using formal language theory, while documents can be treated as some instances of certain languages. Analyzing them using formal language theory is important to understand them and facilitate the implementation. First, we introduce how to analyze the documents to be extracted using word language theory, and then tree language and stochastic language are discussed in this section.

Regular Language Many extraction models use grammatical induction techniques to learn extraction rules that roughly coincide with regular languages; i.e., the extraction rules are some regular expressions and documents are treated as regular languages generated by the regular expressions [Chang and Lui, 2001, Crescenzi et al., 2001].

For example, the basic extraction rule $\langle \ell_0, r_0, \ell_1, r_1, \ldots, \ell_n, r_n \rangle$ can be represented as a regular expression $*\ell_0 \perp_0 r_0 * \ell_1 \perp_1 r_1 \ldots \ell_n \perp_n r_n$, where $\perp_i$ and * are symbols that does not appear in target documents. Given a target document and the regular expression, the system will try to match them, where $\perp_i$ can match any strings in documents. If the document and the regular expression are match, then those strings matching with $\perp_i$ are extracted.

For sake of the strong connection between word language and finite state automata (FSA), it is straightforward to build finite state automata that can match a regular expression to target documents. The regular expression above can be transformed to the finite state automata in Figure 2.5, starting from initial state b, read input ℓ_0, and then search forward for r_0, if r_0 is found, then output strings between ℓ_0 and r_0, and so on.

Some systems [Chidlovskii et al., 2000, Hsu and Chang, 1999, Muslea et al., 2001]

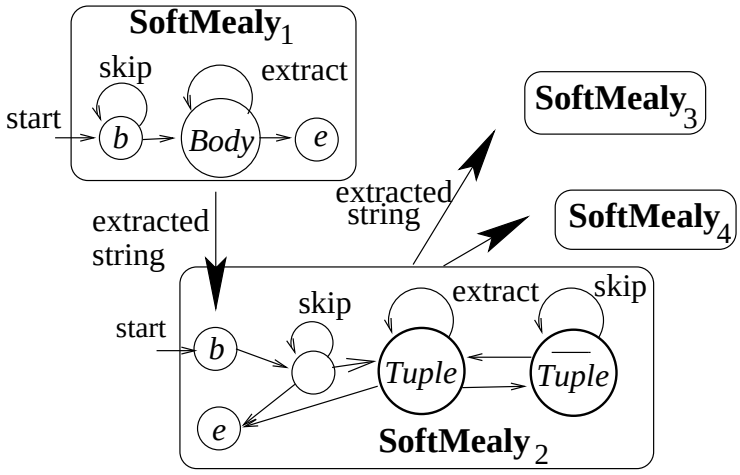


Figure 2.6: Multi-pass SoftMealy Automata

exploit finite automata as extraction machine. As the flexibility of regular expression; e.g., wildcard character '*' can match any string, they can deal with some complex situations. Consider an HTML document of a bookstore Website, we may meet the following problems during data extraction.

- Missing attributes: A book entry may include author, publisher, publish date, comments, etc. Sometimes, in a document listing a series books, one book has comment information and another has no.
- Multi-valued attributes: A book may have not only one author.
- Multiple attribute orderings: Some book entries put author name before title, and some put the name after title. Consider another situation, a Web server may send XML documents and corresponding XSL to a user's Web browser, and the Web browser can decide the ordering of sibling elements according to the XSL. An extraction procedure maybe does not want to interpret the XSL but it still needs to deal with the possible multiple ordering of these sibling elements.
- Disjunctive delimiters: The document uses different delimiters to mark the same fields of different book entries. For example, the titles of hot sales appear in bold format, and other titles do not.

Hsu and Chang [1999] devise a class of FSA named SoftMealy Automata to formalize the extraction machine in their extraction model. Figure 2.6 is an example that contains

four SoftMealy automata where each circle is a state of SoftMealy state and each edge is a transition. The automata reads Web documents as input and the grammar of the automata describes the condition of transition. In the extraction transition, the automata will write extracted data to output. In this model the extractor can deal with those complex situations perfectly. For example, the tuples in an HTML document have two kinds of delimiters, SoftMealy_2 will output texts containing tuples and feed other two automata, SoftMealy_3 and SoftMealy_4. SoftMealy_3 and SoftMealy_4 can deal with these two disjunctive delimiters respectively and locate correct tuples.

Context-free Grammar Finite automata have enough expressive capability to process most Web documents. If we still need more powerful extraction models, build a data extraction procedure that can interpret context-free grammar is a choice. For example, we cannot use these finite-state grammars to extract the *middle column* of a HTML table. Research [Levy and Joshi, 1978, Bianchi, 1996] has been done to discuss how to learn context-free grammar efficiently and use it in data extraction. MINERVA [Crescenzi and Mecca, 1998] is an example of using context-free grammar as its extraction rules, MINERVA especially considers how to deal with errors and exceptions in Web documents. Extraction rules of MINERVA can declare what to do when exceptions occur and recover to normal state.

2.2.2 Tree Language

In Chapter 4, we shall see that the basic extraction rules have similar expressive capability as schemata in relational databases. Comparing to various XML schema languages suggested [Lee and Chu, 2000], it is very naive. As Web documents are mainly HTML and XML documents, and HTML specification today is a XML application, these XML schema languages can be the basis of extraction rule definition. Extraction rules with expressive power less than these XML schema languages are impossible to describe all extraction tasks on Web documents conforming to these schema languages. Formally studying these XML schema languages is a guide to design extraction rules.

Murata et al. [2001] analyze some XML schema languages' expression capability in a framework based on a kind of formal language theory — tree language theory [Comon et al., 1997]. They use tree language theory instead of classical formal language theory; it is because tree languages are invented to process trees and intuitive to analyze XML schemata that have hierarchical structure. Murata et al. [2001] define three subsets of regular tree

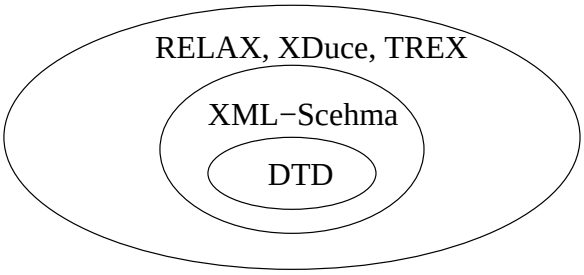


Figure 2.7: Expressive Power of Different Schema Language

grammar: *local* tree grammar, *single-type* tree grammar, and *restrained-competition* tree grammar. Regular Tree languages can represent all documents represented by context-free word languages.

In their paper, Murata et al. [2001] prove that these existing XML schema languages can be categorized into the four classes of language grammar. DTD belongs to local tree grammars, XML-Schema belongs to restrained-competition tree grammars and RELAX, XDuce, TREX belong to regular tree grammar. The relationship of their expression capability is shown in Figure 2.7.

For a website, documents usually are generated from some grammars; i.e., the target documents conform to some schemata. Thus, the rule generation process is like a reverse-engineering task to find the schemata conformed by target documents. To extract accurate information from target documents, defining an extraction rule model that has the same expressive capability with these XML schema languages used by these documents is an intuitive way. Some XML parsers provide validators to test whether an XML document conforms to a given schema. An extraction procedure needs more features than these validators. In Chapter 4, we shall define an extraction rule model that have expressive capability higher than regular tree grammar.

2.2.3 Stochastic Language

We have introduced how to improve extraction models' flexibility to deal with complex situations such as missing attributes, multiple attribute orderings, etc. Some extraction models [Crescenzi and Mecca, 1998] can process errors and exceptions. However, these methods can only make binary decisions and not easy to deal with uncertain situations. Sometimes we need deal with uncertain evidence to extract data.

For example, Bootstrapping extraction system [Grumbach and Mecca, 1999] matches

texts in documents with those field values classified and stored in a repository, then judge which class the parts in documents are, and which parts need to be extracted. For example, suppose there are values `"52#, Graduate Hall, NTU"`, `"48#, Nanyang Valley, NTU"` in the repository that are classified to *mail addresses*, then, if they appear in a target document, Bootstrapping system will extract them. If there is a string "12#, Graduate Hall, NTU" that appears in the document but does not appear in the repository, it is difficult for the system to decide whether to extract or not. To deal with this situation, DataPro [Lerman and Minton, 2000] devises a stochastic extraction model; its rules can describe the probability of whether a text string should to be extracted if it has special patterns.

Hidden Markov Model (HMM) is a more general stochastic finite state automata model. In this model, automata states are associated with probability tables with respect to a certain distribution, transitions also occur according to a fixed distribution. To exploit HMM in Web data extraction, each state is associated with a text pattern that may match with strings in documents to be extracted with a probability. For example, `"12#, Graduate Hall, NTU"` has the same pattern `"#num, #name, NTU"` with the other two address strings; a HMM model can be constructed that accepts string with this pattern as a mail address with high probability. Much research [Seymore et al., 1999, Leek, 1997, Bikel et al., 1997] has been done to employ HMM in data extraction, there exist efficient algorithms learning HMM. Stochastic context-free grammars (SCFGs) are another kind of stochastic automata grammars. Hiemstra and de Vries [2000] discuss how to use SCFG learning techniques to find knowledge from large trees. Church and Mercer [1993] discuss the approaches of applying SCFG to information extraction.

2.3 Auxiliary Components in Concierge Systems

Extraction components are kernel parts in information concierges. In previous sections, we introduce how to define extraction models for these components. This section will introduce some auxiliary components aiding extraction.

2.3.1 Pre-processing of Extraction Components

To enhance an information concierge, we have so far introduced the direction of improving the extraction models, e.g.; we can extend the extraction rules and make them describe

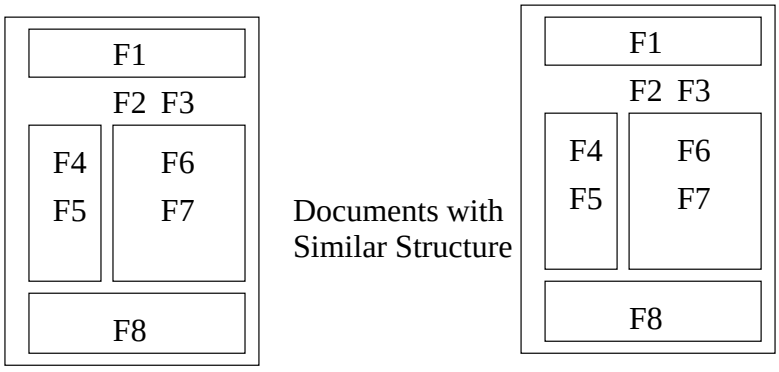


Figure 2.8: Web Page Layout and Feature Distribution

more complicated situations. Another direction of extensions is simplifying these situations by pre-processing Web documents before passing them to extraction components.

Detecting Informative Parts in Documents There are many redundant contents on the Web, such as mirror sites and identical documents available from different URL. For a single document, there may exist intra-page redundancy; i.e., the data appears in multiple pages and are not valuable to a user. If such redundancy are removed from Web pages, it will be easier for a user to annotate which parts in these Web pages are important, which is useful to prepare training set for automatically Web data extraction methods.

A typical website includes much intra-page redundancy information such as navigator panels, advertisement banners and company logo picture, etc. For example, Figure 2.8 gives a possible layout of Web pages. The top rectangle contains an advertisement banner, the left panel is a navigator area, the right panel contains important information and the bottom panel is an area list several hyperlinks pointing to related Web documents. Except the right panel, most of the visible areas in this layout contain overlay information that repeats in many Web documents and a user usually do not want to extract them multiple times from various documents, although these areas may help a user to browse websites and deliver some information.

Usually, Web documents are divided into a set of disjunctive content fragments using some delimiters. For example, websites builders used to employ `"<TABLE>"` to split a Web page to several blocks. It is possible to judge which fragments contain useful information by looking into distribution of contents in various fragments.

Lin and Ho [2002] introduce a method to detect informative parts in Web documents that are likely interesting to users. It assumes HTML documents to be extracted have multiple "tables", and there exist some similar documents with the same sets of tables. Then, if there is a table appearing in various documents, it compares the contents of this table in documents. If contents in this table change little in almost all these documents and contain similar information, it is reasonable to guess this table is a redundant part and do not contain much valuable information.

Lin and Ho's [2002] method contains five steps. The first step is extracting content blocks (CBs, HTML fragments in HTML documents). In this step HTML documents are divided into several blocks, and each block is a text string inside a table. For example, each of the four rectangles is rendered from a table in Web documents. After extract CBs, the method extracts the most important features of each CB in step 2. There are many algorithms to extract keywords from a fragment of texts and the keywords of CBs can be used as a typical class of features. For example, in Figure 2.8, there are six features distributed in four tables and two features outside tables. In step 3, the method calculates entropy values of all the features extracted using the below formula.

$$H(F_i) = -\sum_{j=1}^{n} w_{ij} \log_2 w_{ij}$$

This formula calculates entropy $H(F_i)$ of each feature F_i. Value w_{ij} is the weight of F_i in document D_j. Here the authors use keywords as features. The weight of a feature is a value used to measure the importance of the keyword in the documents. Step 4 obtains the CB's entropy by summing up all embedded features' entropy values: $H(CB_i) = \sum_{j=1}^{k} H(F_j))$, where CB_i contains k Features: F_1 to F_k. The entropy of CB represents the information embedded. High entropy value is evidence that shows the CB contains large proportion of redundant or useless data. Then we can easily categorize these CBs based on their entropy values. If the entropy value of a CB is higher than a threshold we can treat this CB is a trivial fragment in documents, otherwise, it is an informative fragment.

Lin and Ho's [2002] approach only introduces how to use keywords as features. Sometimes a user may have more choices of features. For example, a bookstore website usually categorizes books with respect to their contents and provides a category list in most pages in it. If we only use keywords to classify, the category table is obviously redundant and will be treated as non-informative fragment in Lin's approach, but sometimes this cate-

gory list is interesting to a user when he want to know what kinds of books are provided by this website. In another paper, Wang et al. [2002] introduce how to use more kinds of features to classify tables, including layout, content type and word group. Layout features describe the numbers of rows and columns in a table. Content type features mean what type of information is embedded in the tables; e.g., hyperlinks, pictures, numerical string, etc. Word group features are the same as keyword features in Lin and Ho's [2002] method. Based on these features, Wang et al. [2002] exploit decision tree and SVM to classify all these tables and judge which tables are informative, and can classify tables into finer granularity. In the example above, most category tables use a column or a row to show descriptions of categories, and provide hyperlinks to the documents belonging to this category. By combining the features of layout and content type of a table, we can guess which table contains the category list.

These two methods both use `<TABLE>` tags to divide Web documents to smaller granularity, although many other tags such as *hr, tr, td, a, p, br, h1, h2, strong, b* and *li* can be used to control the page layout of Web documents. In another literature, Embley et al. [1999a] introduce a method that uses these tags as features and devise a heuristic algorithm to combine these features to detect which parts in Web documents may contain interesting information.

Automatic Document Annotation Like the basic data extraction model introduced, many systems need users to annotate Web documents as samples to train rule generation components. Although we can select informative fragments first, and ask the users to annotate these fragments only, it is still time-consuming to annotate Web documents.

For example, if a company wants to compare all the online information about production items provided by its suppliers, it need generate wrappers corresponding to each website of suppliers. To generate wrappers, a user needs to annotate the available documents to tell rule generation components which parts are item names and which parts are prices, etc. If the websites change their layout, the user needs to annotate these documents again. A big library that wants to collect information about recent books published will also meet these problems. In these two examples, we notice that library or company may maintain a database of books' information and production items' information. Using these information we can generate ontology knowledge base and use these knowledge to recognize information fields in documents and annotate them automatically.

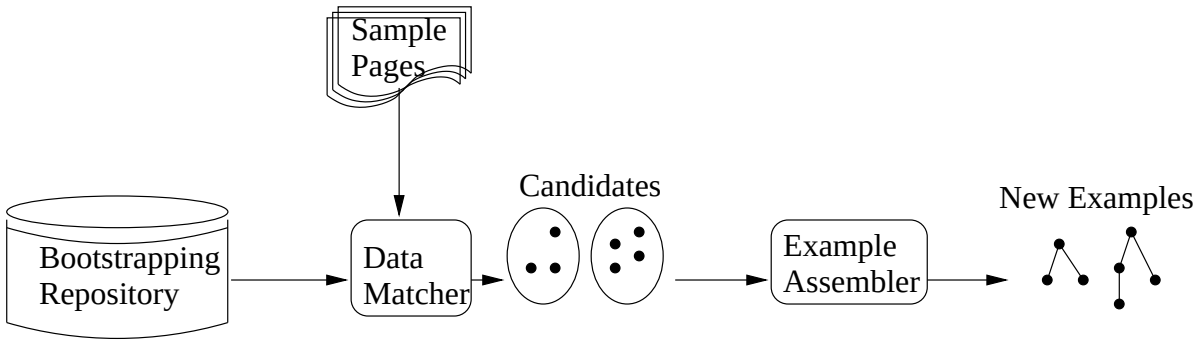


Figure 2.9: Annotate Examples Automatically

Grumbach and Mecca [1999] devise a method so-called Bootstrapping to exploit existing knowledge to annotate documents and extract data. Figure 2.9 shows the steps of this method. Firstly, this method needs a bootstrapping repository containing a set of objects that appear in the websites to be extracted; i.e., the domain. For example, for a company that wants to extract production items' information, the repository should contain some objects of item names. Then *Data Matcher* matches the objects with the text appearing in target documents. Data matcher does not require the objects and text strings in documents match exactly, Grumbach and Mecca [1999] introduce some heuristic methods to judge if they are match when they are similar. For example, "Machado de Assis" and "Jose Maria Machado de Assis" are both the names of the same people. When there are text strings that match objects in repository, the method should extract the text strings.

This approach concentrates on extract interesting data automatically, we can also use it annotate example documents to train rule generation procedure. This method can work very well in the situation that various type of objects have seldom overlay parts and the repository is large enough. In this situation, the Data Matcher can generate high precision results. It is also an interesting research direction to utilize more features of Web documents to improve precision of Data Matcher. The bootstrapping repository can be generated from legacy database or previously existing wrappers.

2.3.2 Post-Process for Extraction Components

So far the techniques discussed are used to build extraction components or aid extraction component construction. After extraction components are built, whether the extracted

data is the very information we want to extract? Whether the extraction rules are still validated when the target website changed its layout? Sometimes when we use wrapper techniques to extract information from the Web, we will get much similar information. How to detect and consolidate similar information are important problems to help users handle extracted data easily. In this section, we shall introduce some post-process processes of extraction, and discuss research directions about these kinds of problems.

Rule Verification and Maintenance Existing techniques usually generate extraction rules by analyzing structure information of Web documents [Kushmerick, 2000a, Sakamoto et al., 2001] or semantic information [Grumbach and Mecca, 1999]. The extraction rules can describe the structures of data sources and contents embedded in them. Thus, these rules are vulnerable to changes of Web document structures or contents. When the extraction rules failed, we must re-generate and test them.

Example 2.6. Let us back to Example 2.4, this time we want to extract the names of these search engines and the descriptions of these hyperlinks before each search website entry. An basic extraction rule ⟨xx.xx">,</A>,">,</A>⟩ can exactly describe how to extract these information. Now if we move the hyperlinks before each search engine to the place after each engine, the HTML document will be changed to below. Obviously, the basic wrapper will fail when it encounter this change.

```
        ⋮
01  <LI><A HREF=¨http://www.informatik.uni-trier.de/~ley/db/index.
      html¨>Database Systems and Logic Programming</A>
02  <LI><A HREF=¨http://xx.xx¨>icon1</A>
03  <LI><A HREF=¨http://xxx.lanl.gov/archive/cs/intro.html¨>Comput-
      ing Research Repository (CoRR)</A>
04  <LI><A HREF=¨http://xx.xx¨>icon2</A>
05  <LI><A HREF=¨http://www.ncstrl.org/¨>Networked Computer Science
      Technical Reference Library (NCSTRL)</A>
06  <LI><A HREF=¨http://xx.xx¨>icon3</A>
        ⋮
```

In this situation, verifying whether a wrapper is still validated may not be a big problem. For example, we can image that when the wrappers fail, it is very possible that

there will exist HTML tags appear in the extract data or cannot extract anything at all. If the numbers of HTML tags in newly extracted data and historical extracted data varies widely, we can assume the wrapper failed. Based on this assumption, Kushmerick [1999b] suggests a wrapper verification method — RAPTURE.

RAPTURE [Kushmerick, 1999b] is a wrapper verification tool using regression testing techniques to test the validity of wrappers. It counts the number of HTML tags and some other features appearing in extracted data. To test the validity of a wrapper, it selects some features r of data extracted, compares r with these features R of historical data extracted using validated wrappers. If r is very similar to R, RAPTURE treats the wrapper is still validated. RAPTURE defines *equal probability* P to measure the similarity of r and R. We can calculate it using the formula below:

$$P[r, \mu_R, \sigma_R] = \frac{1}{\sigma_R\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{r-\mu_R}{\sigma_R})^2}$$

In this formula, μ_R is the mean of R, and σ_R is the standard deviation of R. RAPTURE assumes that these features distributed according to normal distribution; collects all the features of extracted data; calculates the equal probabilities and then combine them to get an equal probability v. Using v it is easy to evaluate whether the wrappers are still validate or out-of-date. The features exploited by RAPTURE include: (1) HTML density (the fraction of `"<"` and `">"` characters in extracted data), (2) word count (the number of words in extracted data), and (3) mean word length (average length of extracted words). We have mentioned before that if wrappers (such as OCLR class wrappers) fail it will be very possible that the number of HTML tags extracted will change largely. The experiment results of RAPTURE approved this assumption; i.e., only using the HTML density features of extracted data we can archive almost the same or even better veracity comparing with the result of combining several features.

To improve the robusticity of wrappers, we want wrappers to be more flexible. For example, the basic extraction rules ⟨`</A>,<LI><A HREF=",",>,</A>,</LI>`⟩ can exactly extract data in Example 2.4, in the situation of Example 2.6, some fields appear in various order, the wrapper will fail. If we use a regular expression as the wrapper rules that means these text string can appear in multi-ordering, we can process this situation. Some research [Chang and Lui, 2001, Davulcu et al., 2000] has been done to improve their wrappers' flexibility by transforming fixed string extraction rules to regular expressions.

Some wrapper models such as DataPro [Lerman and Minton, 2000] do not use the exact matching techniques to compare rules and data sources. DataPro first encodes HTML documents to sequences of tokens. For example, a text string `"New York Street 10"` is an address, can be encoded to `"Cap-Char Street Number"` where `"Number"`, `"Street` and `"Cap-Char"` are three tokens. DataPro calculates the proportion of strings that are encoded to `"Cap-Char Street Number"`. If the proportion is high, then this token sequence is significant and DataPro treat it as an extraction rule.

In these situations, wrapper verification is more complex than that applied on fixed string extraction rules, because layout changing has littler effect on these wrappers. When the wrappers fail, the number of HTML tags may not change as large as the situations in RAPTURE. In these situations whether the verification performance can be improved by combining more features is an interesting question to be studied.

If the verification result tells us a wrapper is out-of-date, it needs to be re-generated. After re-generating we can also use verification techniques to test the new wrappers and judge whether they are validate.

Data Consolidation Data extracted from heterogeneous resources may have similar structures or overlay contents. For example, news extracted from news websites should have news title, detail description, time, etc. Books data extracted from bookstore websites should include book title, authors, editor, price, publisher, etc.

Data consolidation consists of two tasks: (1) to cluster data with similar structures together, and (2) to diminish overlay contents from extracted data. There are two benefits to consolidate data: (1) If the similar data are clustered together, the efficiency of queries can be improved, and the queries are submitted to restricted clusters only. (2) If the overlay contents are diminished, the size of extracted data will decrease, and consistency of extracted data will improve.

As extraction rules describe how to store extracted data, it is natural to find the similar structures of output data by investigating the extraction rules. Davulcu et al. [2000] introduce efficient algorithms to find this kind of information in extraction rules, and define the concept of unambiguous and maximization of extraction rules that can represented by regular expressions. We also hope that wrappers can extract data as accurate as they can. Based on extracted correct data, we also hope the wrapper rules can process as many data sources as they can. Unambiguous wrapper rules can extract data 100% accurately, and maximization wrapper rules are unambiguous rules that can

process maximum number of documents. Davulcu et al. devise an algorithm that can maximize a large class of unambiguous extraction rules. If we can find the maximized rules of rules applied to different data source, it is possible to maintain only one set of wrappers to extract data from several websites and the extracted data will share the common structures.

To consolidate overlay contents is bit difficult, as the same type of fields in data extracted from different websites may have different names. For example, we can image that the hottest news will be involved in the homepages of both website a and website b. The field "news_title" in data from a and the field "article_title" from b talk about the same thing. Here, the different field names make content consolidation difficult. Consolidation of structures of extracted data gives the basis of consolidation of overlay contents. After the structures of these news extracted are consolidate; i.e., after we know "news_title" and "article_title" contain the same type of information, it is easier to detect overlay contents by comparing summarizition of corresponding fields of data extracted from different websites.

2.4 Related Research Area

So far, we introduced components that may be included in an information concierge system. Some research areas are strongly related to information concierge. If these related techniques can be exploited properly, they can improve components in concierges. This section briefly discusses some related techniques.

2.4.1 Natural Language Processing (NLP)

In this thesis, when we mention Web data extraction, the target data source is semi-structural Web documents downloaded. Traditional Information Extraction (IE) focuses on free text document and largely depends on Natural Language Processing (NLP) techniques. Figure 2.10 is an architecture [Appelt and Israel, 1999] for transitional IE system focusing on free text extraction. From this figure we can see that there is close relationship between IE and NLP techniques. Issues in Web data extraction are obviously different from those issues in IE systems. However, as mentioned before, employing NLP techniques in Web data extraction is also possible.

In free text extraction, tokenization using word segmentation techniques is a prereq-

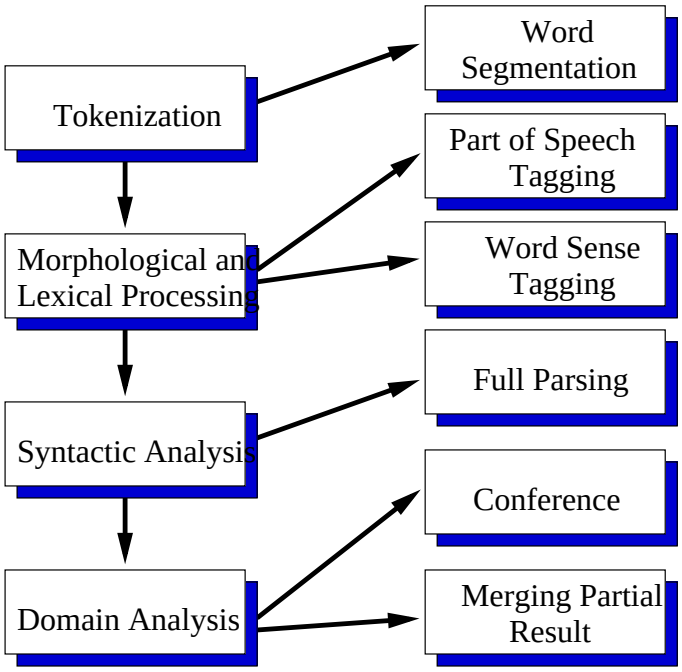


Figure 2.10: Architecture of Free Text Extraction System

uisite to execute further processes. To extract Web documents, if long paragraphs are tokenized first, then the extraction can be done at a finer granularity. Morphological analysis and lexical lookup are techniques used to analyze these tokenized text and find the possible semantic information. They are very important to do exact analysis. For example, if a text string is a name and we have not recognized it, we will stand a big chance to mistake the meaning of the whole sentence. In most Web data extraction tasks, we do not need to understand the semantic information of documents, but sometimes we can use these morphological analysis and lexical lookup techniques. Golgher et al. [2001] discuss how to maintain an object repository and match text appearing in HTML documents with the object attributes in the repository. The matched text strings can be used as clues to judge which parts of documents are to be extracted. We can employ NLP techniques to improve the performance of such kind of methods.

After word analysis, IE system will analyze sentences' syntax and use domain related knowledge to judge which parts are interesting and should be extracted. This task expects the text to be in the form of full, grammatical sentences. It is difficult to ask semi-structured Web documents full of tags meet this requirement. Soderland [1997] discusses the possibility of use the evidence of the Web page layout to reconstruct HTML

documents into grammatical sentences and apply syntactic analysis techniques to them.

2.4.2 Data Mining

There are so many data available in databases that to discover much knowledge from databases is almost an impossible mission for human without aid of computers. The motivation of data mining is to provide methods to find knowledge in large volume data in databases. *Classification, Clustering* and *Association Rule mining* are the common tasks of data mining. Web wrappers process Web data and transform extracted data to structured data. These structure data can be stored in database, XML document or other structured storages. The objective of Web data extraction is to provide a middle layer between heterogeneous Web data sources and users, so that users can concentrate on major data collections and find interesting information easily. If there are huge sizes of extracted data, concierges will meet the same problems as those in traditional databases, and data mining techniques can be directly applied on data extracted.

With the rapid growth of information on the Web, mining knowledge from Web documents becomes an interesting research area. There are three well-known Web mining models commonly called *Web content mining, Web usage mining* and *Web structure mining*.

Normally Web content mining is the task of automatically searching and retrieving information from the Web. Unlike traditional Information Retrieval system such as search engine, Web content mining employ domain related knowledge to improve the precision of retrieval results. For example, it can categorize Web data sources by mining users' bookmarks [Maarek and Shaul, 1996]. Web usage mining is to analyze user access patterns from Web servers. This kind of information is easy to collect for that Web server can save user access logs. These Web mining methods can help us to choose training samples to train extraction rule generator and help us generate strategy to present extracted data.

2.4.3 Database Techniques

Database systems are developed to manage information and query them. To manage Web data is really a huge challenge to database techniques and it is impossible to use database techniques to solve all information management problems on the Web. However, because database techniques are mature, there are still many researches in database community

that address problems of how to use database systems to manage Web data.

Commonly there are three kinds of database concepts related to Web data management:

- Modeling and Querying the Web
- Information Extraction and Integration
- Website construction and restructuring

The Web can be viewed as a directed graph where each node is a Web document and edges are the hyperlinks among documents. How to generate queries that can retrieve accurate documents from the Web is a problem. All the search engines can only execute simple queries that consist of keywords. Many search engines allow users add more constraints on their queries. These constraints can be related to the content or link relation among documents. For example, *Google* provides an advanced search page that allows users to appoint the language and file format, etc. Web documents can also be viewed in smaller granularity; we can say that the task of Web information concierges is to extract the finer granular data from Web documents. Some databases extend themselves by integrating wrapper system [May et al., 1999] and some databases' main data sources are provided by their internal wrappers [Gupta et al., 1997]. Unlike the main task of Web data concierges, the focus task of databases is how to manage the extracted data. Database techniques and Web data mediator techniques are complementary.

2.5 Concierge System Architecture

At the end of this chapter, we will introduce some architecture of Web information concierges. Extraction component is the kernel of information concierges, while various auxiliary components are provided according to different application environments. As the systems introduced in this section are oriented to different users, their architecture varies widely. Lixto [Baumgartner et al., 2001b] focuses on how to build an interactive extraction rule generator and a solid extraction machine. MIA [Beuster et al., 2000] focuses on auxiliary components that can be customized by users, while FLORID [May et al., 1999] concentrates on how to provide a database accessing interface to extraction components to obtain high flexibility.

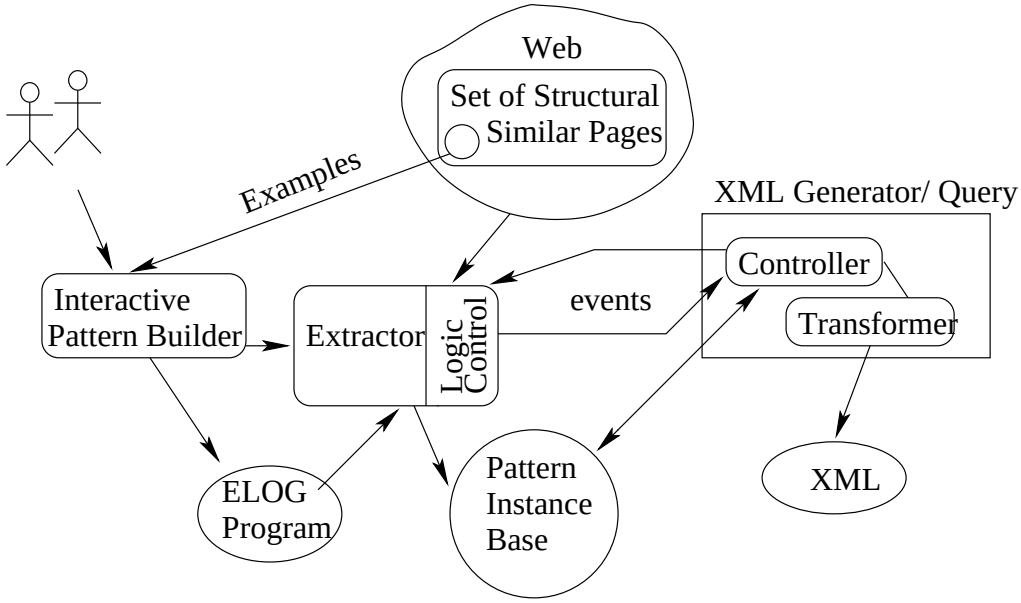


Figure 2.11: Architecture of Lixto

2.5.1 Extraction Components

Lixto [Baumgartner et al., 2001b] is a Web information concierge system using supervised rule generation techniques to build extraction rules. The extraction rules built are roughly equivalent to monadic second-order logic formulae. Thus, it is possible to exploit those logic program interpreters as extraction machine and use optimization approaches for logic programming.

Lixto considered how to provide visual interfaces to a user to ease extraction rule generation and maintenance. As drawn in Figure 2.11, *Interactive Pattern Builder* is the rule generation component with visual interfaces that allows a user to annotate the training samples and the system generate extraction rules based on the sequences of annotation. Learned results are recorded by the internal logic-based declarative language — Elog. Users need not understand Elog and deal with it directly, but Elog provides a potential of control extraction tasks by modifying extraction rules manually.

Elog programs generated by the pattern builder are fed to *Extrator* — the extraction machine of Lixto. Web documents are downloaded and fed to the extractor at the same time. Document fragments that match the patterns defined via Elog are to be extracted. These fragments extracted are stored in *Pattern Instance Base*. *XML Generator* transforms the data in pattern instance base into final XML output documents.

Figure 2.11 shows the relationship among these components. It provides user interfaces to control whole data extraction process. Lixto stores extracted data in XML documents and provides a simple query interface to users. Elog [Baumgartner et al., 2001c], a logic based language like Prolog provides a method to store wrapper rules in a familiar way. For example, the following Elog rule tells system to locate the first table in BBC news websites' homepage.

```
tablesq(S,x) ←documents("www.bbc.co.uk"),S),
             subsq(S, (.body,[ ])),(.table,[ ]),x)
```

Lixto provides prolific interfaces to control execution, but it is not enough sometimes. Let us image an application on the Web: an application service provider (ASP) wants to provide data extraction services to a mobile equipment user. In this application these interfaces are not useful to these end-users. Interfaces of Lixto can facilitate expert users to define wrappers and control the extraction procedure. Those systems oriented to end-users need to be built on more auxiliary components.

2.5.2 Integration of Auxiliary Components

MIA [Beuster et al., 2000] is Web information concierge system serving mobile users. It crawls on the Web and extracts data to send them to mobile equipments such as mobile phone, PDA, etc. MIA can collect users' location information and users' preference and present customized data to users. In the architecture provided by MIA, shown in Figure 2.12, end-users will never see the details of extraction rule generation and execution machine.

The Server module provides a gateway users can access through their mobile equipments. Users' equipments use HTTP and WAP protocols to communicate with the server and the server collect users information, tailors the data extracted corresponding to users' information and send the data to users. Depending on the document type a user appointed, the server can send data in various formats such as HTML, WML or free text.

Users only need to communicate with the gateway, behind the gateway there are several interacting agents: user agents, spider agents, localization agent. Users' operations are analyzed in MatchMaker first and then distributed to different agents. The Spider Agents is the extraction components, like Figure 2.11. It is service providers' responsibility to define how to use spider agents to extract data from the Web.

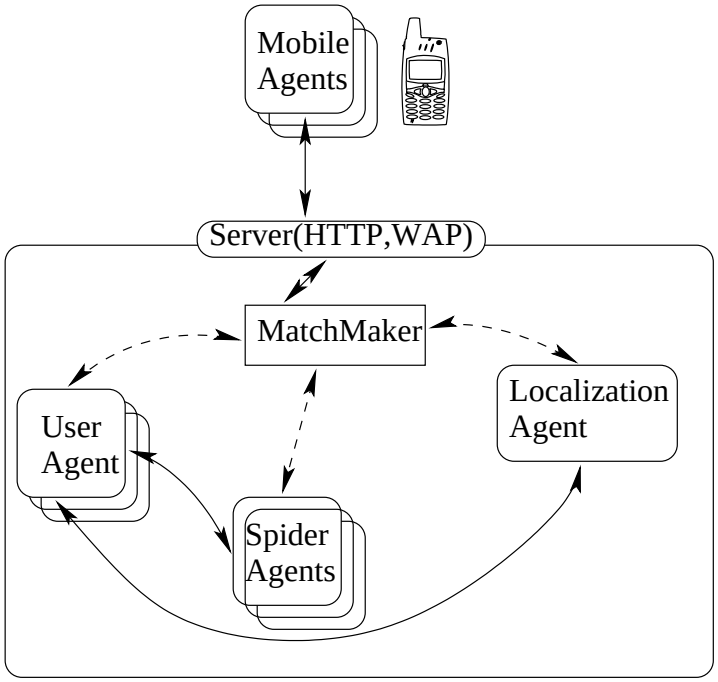


Figure 2.12: MIA Architecture

Lixto extractor directly extracts data from different websites based on extraction rules, while MIA uses another strategy. The spider agents crawl on the Web to extract documents and classify them. Localization Agent detects the current location of users, and User Agents provide interfaces that allow users to define in which type information they are interested. Based on these information, MIA does extraction with finer granularity and send final extraction results to users. In this architecture, what the end-users need to do is to select information topics they are interested in.

2.5.3 Interface to Database Applications

Lixto and MIA provide relative prolific components to support their extraction execution. There is another problem, is it possible to communicate among various information concierge systems? As the ad-hoc formalisms exploited by different systems, the inter-system communication is a problem. For example some systems save extracted data in XML documents and some save to relational databases. If two systems use similar storage strategies, it is possible for them to use data extracted by other system, if they can understand schemata of extracted data each other. Thus, it is important to deliver

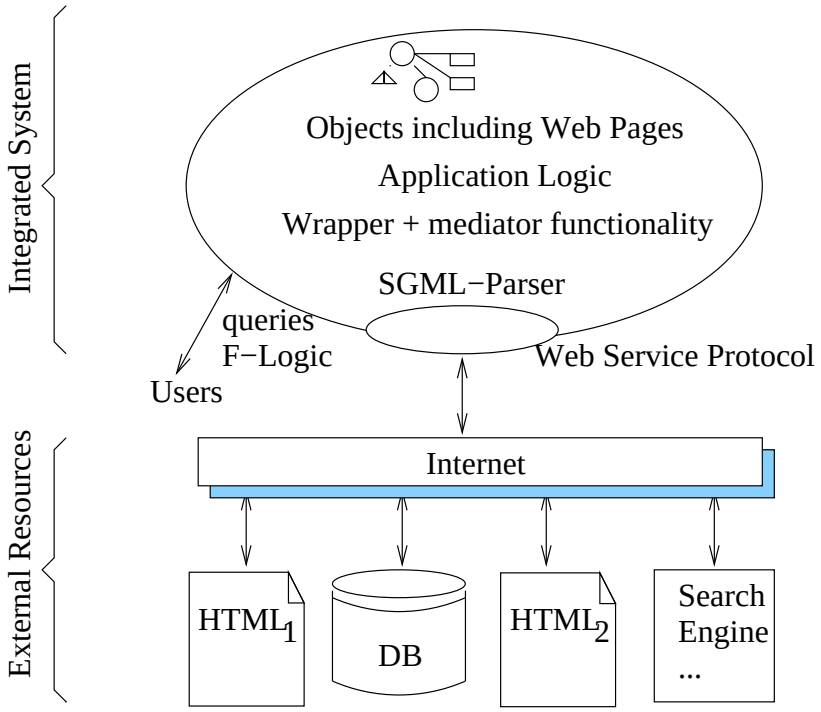


Figure 2.13: Architecture based on Florid

a public schemata over data extracted.

Web information concierge systems are still on the road of evolution. The compatibility among different systems has be seldom considered. May et al. [1999] suggest an integration architecture based on FLORID, shown in Figure 2.13. FLORID is a deductive object-oriented (OO) database system. It employs F-Logic as data definition and query language. Applications that can access objects stored in FLORID can also access Web data extracted. Because OO database techniques are relative more mature and are understood by many users, an architecture based on existing databases is easy to use and control for them. This architecture mainly consists of two parts:

- An appropriate modeling of Web documents,
- A formal language and methodology that can handle the model.

From Figure 2.13 we can see the integrated architecture uses SGML-Parser to process Web documents before transforming them to objects in the database, so the details of different Web services and different document structures are hidden to other components.

FLORID can maintain comprehensive data models that are suitable to describe Web data structure. Ludascher et al. [1998] discuss F-Logic, the internal language of FLORID that supports deduction and OO, they argued that it is an excellent tool to handle these models. In this architecture, wrapper definition, execution, user query interface and data presentation are integrated and controlled using F-Logic.

2.6 Summary

Web data extraction is the kernel of information concierge systems, which is consist of the following basic constituents: (1) data model for target Web documents, (2) output data model, (3) extraction rule model (4) rule generation mechanism, and (5) extraction execution mechanism. In this section, we presented each constituent and discussed their relationship with some auxiliary components in systems including document pre-processing and post-processing technique.

By studying related literature of building information concierge systems, we can find that existing work suffers from the following problems:

- Most work chooses rule generation algorithm, rule model and techniques for other components in a rather ad-hoc way. This makes the comparison among different systems difficult, and makes integration of different systems almost impossible.

- Although most automatic rule generation methods need training sets, these sets are assumed to be pre-defined. Given a large set of documents that do not share similar structures, multiple training sets have to be constructed and extraction rules are learnt from these training sets independently. To the best of our knowledge, automatically construction of training sets has not been addressed.

- Learnt extraction rules cannot be applied to an incoming target document that is not from the pre-defined target set. That is, mapping between a target document and the corresponding extraction rules cannot be achieved automatically.

- Efficiency of extraction is mostly studied from theoretical perspective, or based on relative small corpus in literature.

In the following chapters, we shall first introduce the proposed framework in Chapter 1 in detail. This framework allows us to integrate extraction components and auxiliary

components. The common layers support these components, and make it easy to analyze their capability and efficiency. Then, we shall show the flexibility of the framework by demonstrating several implementation based on tree language theory and logical program. A practical Web data extraction language is introduced and compared with other related techniques. Also, a rule generation method is devised to induce rules in linear time in document size.

Chapter 3

Realization of Web Information Concierge

3.1 Framework Overview

In this chapter, the concierge framework mentioned in Chapter 1 is introduced in detail. The basic idea of the framework and benefits are also discussed.

To build the Web data extraction system demonstrated in Chapter 2, we need to consider five aspects including: (1) target document model (2) output data model (3) rule model (4) extraction approach (5) rule generation approach. In Kushmerick's [1997] work, when a more complexity document model is chosen, other aspects need corresponding changing, vice versa. When we use rule generation algorithm from other system, usually, we also must adapt its document model.

In the proposed framework in Chapter 1, we try to devise a mechanism that allows integration of different approaches from various systems. The basic idea is to use a target document model that can generally represent Web documents and extracted data, while providing a formalism of extraction rule that can describe extraction rules with various expressive powers. Based on this kind of rule formalism, extraction approaches and rule generation approaches are also possible to be compared under the same formal framework.

Our framework models both target documents and extracted data as labeled rooted trees, which is a common data structure that can represent almost all Web documents. An instance layer is built to store cached Web documents and extracted data. The core

of the framework is the *schema layer*. Schemata describe data instances and provide information to the operation layer. The operation layer contains extraction component, rule generation component and can be extended flexibly. The details of instance layer is hidden to the operation layer, and operations can handle data in instance layer in terms of schemata.

In the following part of this chapter, we first give the definition of components in each layer and then a detailed example system based on the framework will be demonstrated in Section 3.5.

3.2 Instance Layer

In this section, we propose a model to represent Web documents and extracted data. Based on this representation, the problem of building instance layer is defined.

Semi-structured documents can be generally modeled as directed graphs [Bergholz, 2000, Florescu et al., 1998, Lian and Cheung, 2004]. For example, a HTML document can be parsed into a directed graph — each tag element is parsed to a node; corresponding to each pair of parent-child tag elements there is a directed edge; and each hyperlink, which describes non-parent-child link relationship between two tag elements, can be parsed into a directed edge. However, as hyperlinks carry less information in Web data extraction, we do not consider them during the process of parsing in this thesis. Without considering hyperlinks, documents can be parsed to trees. Here are some examples:

Example 3.1. Figure 1.1(a) shows a Web page fragment from the Amazon website on 30 Jan 2004. It lists four top sellers of computer books. The topmost seller in this page is rendered from the HTML codes as shown in Figure 3.1(a). DOM (Document Object Model) tree generated from the codes are shown in Figure 3.1(b) (For simplicity, we have not drawn all nodes in the DOM tree; nodes with *folder* icon are non-text nodes and nodes with *paper* icon are text nodes). The right-hand side of pages B and C in Figure 1.1(b) are detailed information about the two topmost hot books respectively. Figure 3.1(c) shows the top four levels of the DOM tree corresponding to the HTML codes for the right-hand side of page B.

In Example 3.1, an element name is mainly used to describe page layout. However, for some semi-structured documents, especially XML, an element name may contain

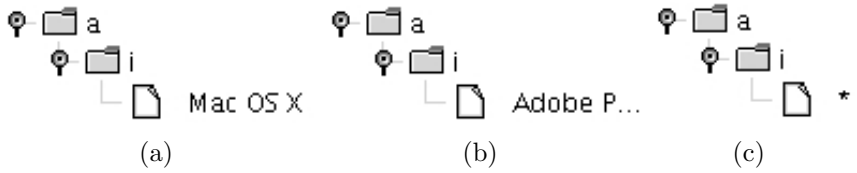
Figure 3.1: Example of HTML Document Fragment and DOM Tree

important information. Thus, in this thesis, we model semi-structured documents as labeled trees — document tree, as defined below:

Definition 3.1 (Document Tree)**.** A document tree is a rooted, labeled tree that can be defined as a 4-tuple: $t = \langle V, E, r, \gamma \rangle$, where V is the set of nodes corresponding to tag elements, E is the set of edges connecting these nodes, r is the root node, $\gamma : V \rightarrow L$ is a function that assigns each node a string label, where L is the label set of t. An edge e is an ordered 2-tuple (u, v) where $u, v \subseteq V$ and u is the parent node of node v. Root node r has no parent node. Each non-root node u in V has exact one parent node.

In this thesis, discussion is restricted to HTML and XML documents so that we can exploit DOM parsers to parse documents to trees. During the parsing processing, we label each non-text node with the name of corresponding element in original documents, and label each text node with its value.

The objective of Web data extraction is to extract fragments from document trees that are relevant to a user's requirements. We refer to these fragments as data instances, as defined below:



(a) (b) (c)

Figure 3.2: Instances and Schemata

Definition 3.2 (Data Instance)**.** Given a document tree $t = \langle V, E, r, \gamma \rangle$, $t_1 = \langle V_1, E_1, r_1, \gamma_1 \rangle$ is a data instance (DI) of t if $V_1 \subseteq V$, $E_1 \subseteq E$ and $\gamma_1 = \gamma$. This relation is denoted as $t_1 \subseteq t$. Given two DIs t_i and t_j, t_i is a sub-DI of t_j if $t_i \subseteq t_j$.

Example 3.2. In Example 3.1, the document tree in Figure 3.1(b) is a DI of the document tree corresponding to Page A in Figure 1.1(a). This instance corresponds to the first book entry in page A. Figure 3.2(a) is a DI of Figure 3.1(b). Figure 3.2(a) corresponds to the title of the first book; while Figure 3.2(b) corresponds to the title of the third book.

Definition 3.2 states that a document tree is also a DI. We therefore do not distinguish between semi-structured document, document tree and data instance in the remaining parts of this thesis. Given definition of *DI*, the problem of *building instance layer* is quite straightforward; i.e., given n Web documents $\mathcal{P} = \{p_1, \ldots, p_n\}$, induce the set of DIs $\mathcal{D} = \{d_1, \ldots, d_n, d_{n+1}, \ldots, d_{n+m}\}$, where d_i is parsed from p_i, and d_{n+j} is a sub-DI of d_k, where $k \in [1, n]$. In our framework, extracted data will be stored in XML documents, thus, these data can also be modeled as DI.

In Figure 3.1, these repeated contents can be parsed to equivalent DIs, as defined below:

Definition 3.3 (Equivalence)**.** Given two DIs $t_1 = \langle V_1, E_1, r_1, \gamma_1 \rangle$ and $t_2 = \langle V_2, E_2, r_2, \gamma_2 \rangle$, t_1 is *equivalent to* t_2, if and only if there exists a bijection $\mathcal{M}$ between V_1 and V_2 such that:

- $\mathcal{M}(r_1) = r_2$;
- $(u, v) \in E_1$ if and only if $(\mathcal{M}(u), \mathcal{M}(v)) \in E_2$;
- $\gamma_1(u) = \gamma_2(\mathcal{M}(u))$.

This definition does not consider the orders among sibling nodes; i.e., in this thesis, when we compared two document trees, we treat them as unordered trees. For schema detection algorithms introduced in later Chapters, an unordered model is more suitable than ordered model. This statement is based on the observation that if the only difference between two DIs is the order of their sub-DIs, hopefully, these two DIs are generated based on the same schemata. For example, a table lists top 10 hot sale books, although order of books may be changed in the table, the table's schema is remain unchanged. Chapter 5 will discuss more in detail.

3.3 Schema Layer

In this section, we formally define schema and discuss problems in building the schema layer.

3.3.1 Schema

Repeated contents in semi-structured documents are usually prominent and easily raise a reader's attention. Most Web data extraction systems [Arasu and Garcia-Molina, 2003, Chang and Lui, 2001, Crescenzi et al., 2001] assume that repeated contents are important and should to be extracted. For example, in the example in Figure 1.1, all books have similar formats and most parts of their HTML codes are repeated, like those in Figure 3.1.

The two instances in Figures 3.2(a) and (b) are sub-DIs of the DI in Figure 3.1, and are almost the same; except for the labels of two text nodes. If we use a label `"*"` (wildcard character) to replace both labels, the two instances will become Figure 3.2(c). A wildcard character `"*"` is a regular expression that generates any string, denoted as $* \vdash \ell$, where ℓ is a label. We assume `"*"` does not appear in label sets of DIs of document trees to be extracted. As introduced before, schemata are objects describing a set of DIs. Thus, we treat the instance in Figure 3.2(c) as a schema. We define a schema below:

Definition 3.4 (Schema)**.** A schema s is a 2-tuple $\langle D, A \rangle$, where D is a set of DI, and γ functions in these k-subtrees label some text nodes with `"*"`. A is a n-tuple $\langle a_1, a_2, \ldots, a_n \rangle$, where a_i is an attribute, $i \in [1, n]$.

Attributes in a schema are important to aid data extraction and will be discussed in Section 3.3.2. Here, we may assume $A = \langle |s| \rangle$, where $|s|$ denotes the size of a schema s; i.e., the cardinality of the node sets of D.

Definition 3.5 (Conformation and Type)**.** A DI $t = \langle V, E, r, \gamma \rangle$ conforms to a schema $s = \langle D, A \rangle$ or s is the schema of t, if and only if there is a DI $\langle V_s, E_s, r_s, \gamma_s \rangle \in D$, and there exists a bijection $\mathcal{M}$ between V and V_s such that:

- $\mathcal{M}(r) = r_s$;
- $(u, v) \in E$ if and only if $(\mathcal{M}(u), \mathcal{M}(v)) \in E_s$;
- $\gamma(u) = \gamma_s(\mathcal{M}(u))$ or $\gamma_s(\mathcal{M}(u)) \vdash \gamma(u)$.

where t conforms to s, denoted as $t \mapsto s$. A set of DIs conforming to the same schema is known as a *type*.

3.3.2 Schema Attributes

Some Web data extraction systems [Arasu and Garcia-Molina, 2003, Chang and Lui, 2001] assume large structure patterns that match large number of structures in documents are important; contents that match these patterns are then extracted. Similarly, we compute schema weight to measure the important of corresponding DIs based on two observations.

Observation 3.1. A type including a large number of DIs are usually important in documents, and its corresponding schema is important.

Observation 3.2. A schema with large size is usually important in documents.

There are two factors influencing the weight of a schema — the cardinality of its corresponding type and its size. Given a document tree t, a type T and a schema s, if DIs in T confirm to s, the *weight* of s in the document is:

$$\omega(s) = \ln \|T\| \times |s| \tag{3.1}$$

where $\|T\|$ is the cardinality of T, also known as the *document frequency* of s in this thesis.

The weight of a schema is an important evidence to decide which DIs should be extracted. However, it is not enough to decide which DIs should be extracted based on schema weight only. One property of semi-structured documents is irregularity — DIs conforming to the same schema may encode different kinds of contents. Another property

of semi-structured documents is that there exist redundant contents; e.g., advertisement bars appearing in many HTML pages. In this section, we introduce attributes of schemata that provide additional information other than the schema weight. Users may control extraction operations based on these attributes.

Given a document set $\mathbb{C}$, we collect the following attributes of a schema:

- Size, Document Frequency (DF) and Weight.
- Set Frequency (SF): Given a set of documents $\mathbb{C}$, if s is the a schema conformed by DIs in type T, SF of s in $\mathbb{C}$ is $\|T\|$.

All documents are organized into clusters, where each cluster is a set of similar documents. Given these clusters, we may identify more attributes of a schema.

Inverse Set Frequency (ISF) of a schema s is denoted with $I(s)$, and $I(s) = log\frac{N}{n}$, where N is the number of document sets and n is the number of document sets containing DIs conforming to s.

Suppose a document set $\mathbb{C}$ contains types T_1 to T_n, the weight of the schema s conformed by DIs in type T_i is $\varpi(s) = \|T_i\|/max_{j=1}^{n}\|T_j\|$ (ϖ is a temporary variable used to calculate Set Weight, and is different from document weight ω), the *Set Weight (SW)* of a schema s is:

$$W(s) = \varpi(s) \times ISF(s) \tag{3.2}$$

To be easily accessed by users, Web documents often include much redundant information. For instance, the same navigation bar may appear in many pages. An entropy measurement of fragments in Web documents was suggested in [Lin and Ho, 2002] to detect redundant information; only fragments with small entropy should be extracted. Inspired by this idea, we calculate entropy of schemata and restrict that only DIs conforming to schemata with entropy smaller than a threshold will be extracted. Given n document sets, *entropy* of a schema s is:

$$Ent(s) = -\sum_{i=1}^{n} \varpi(s) log \varpi(s) \tag{3.3}$$

Usually, a type of DIs appearing closely and regularly is likely to be interesting. Based on this observation, we define two attributes of a schema – mean distance (MD) and standard deviation of distance (SDD) to measure how close and regular those DIs in

a type are. To compute MD and SDD, we need to know distance and the order relation among DIs and orders among DIs.

Distance between DIs t_i and t_j in a document tree is the number of nodes in the shortest path between root nodes of t_i and t_j. If t_i and t_j belong to different document trees, distance between them is 1.

Given a set of document trees $\{d_1, \ldots d_n\}$, a node r_0 is added as the parent node of these trees; the tree rooted from r_0 is denoted as T. The *position* of a DI t is $\theta(t) = m$ if the root of t is the mth nodes accessed in the pre-order traversal of T. An *Order Relation* between a pair of DIs is denoted with $t_i \preceq t_j$. We say $t_i \preceq t_j$ if $\theta(t_i) \leqslant \theta(t_j)$.

Given a type T, we sort all DIs in T, such that $t_i \preceq t_{i+1}$. We say that t_i and t_{i+1} are adjacent DIs. *Mean Distance (MD)* of the schema s conformed by DIs in T, $\mu(s)$, is the mean value of distance between each pair of adjacent DIs belonging to T.

$$\mu(s) = \frac{\sum_{i=1}^{n-1} d(t_i,\, t_{i+1})}{n-1} \tag{3.4}$$

Standard Deviation of Distance (SDD) of s is defined as below:

$$\sigma(s) = \sqrt{\frac{\sum_{i=1}^{n-1} (d(t_i,\, t_{i+1}) - \mu(s))^2}{n-1}} \tag{3.5}$$

We store all attributes detected in a schema s in a tuple $\langle$ *Size*, *DF*, *Weight*, *SF*, *ISF*, *SW*, *MD*, *SDD* $\rangle$.

We know an important property of semi-structured documents is sectional; i.e., contents in different parts of a document may contain different information. Usually, DIs belonging to the same type appear in the same part in a document. Thus, if most DIs belonging to the same type appear in a part and a DI is far away from this part, this DI may be outlier and need not to be extracted. We define *Distance Offset* of a DI t_i as:

$$\Delta(t_i) \;=\; \frac{d(t_{i-1},\, t_i) + d(t_i,\, t_{i+1})}{2\mu(C)} - 1 \tag{3.6}$$

Based on some intuitive observations, we proposed some attributes of schemata. As an initial step of studying these attributes, in Chapter 7, we shall present some statistic results of the relationship between schemata and attribute. In Chapter 6, some attributes are used to prepare training set for our schema detection algorithms, and play important role to improve system efficiency. Actually, the use of these kinds of attributes is not

limited in concierge system. e.g.; Cai et al. [2004] exploit physical location attributes to improve PageRank and HITS. In spite of the studying presented in this thesis, how to exploit these attributes to aid concierge systems still need more research.

3.4 Operation Layer

As shown in Figure 1.2, extraction and rule generation components are placed in operation layer. Once the instance layer and the schema layer are built, we obtain the mapping relationships among DIs and schemata. Both extraction and rule generation can be treated as operation in term of schemata. We give some simple introduction in this section.

Detection of Schemata In our framework, the instance layer contains all Web documents to be extracted. The schema layer contains useful information of DIs in the instance layer. For example, two documents are structurally similar if the sets of schemata corresponding to them are the same. Given a set of n document trees $\{d_1, \ldots, d_n\}$ that contain a set of DIs $\mathcal{D} = \{t_1, \ldots, t_m\}$, *schema detection* is the procedure of inducing the set of schemata $\mathcal{S} = \{s_1, \ldots, s_n\}$, where $s_i \in \mathcal{S}$ if and only if $d_j \in \mathcal{D}$ and $d_j \mapsto s_i$.

The problem of schema detection is to find the set of schemata of all DIs in documents (or subtrees of document trees). This problem can be reduced from the problem of *the largest common subtrees (LCST)* [Akutsu, 1992]. As LCST is P for two trees and NP-hard for more than two trees [Akutsu, 1992], schema detection problem is also NP-hard. The same is applied to the problem of building instance layer. Some approximate solutions of similar problems have been proposed by putting some restrictions on subtrees to be processed; e.g., TreeMiner [Zaki and Aggarwal, 2003] only processes subtrees whose frequency of occurrence exceeds a threshold. We define a constrained version of this problem in Chapter 4 and an efficient algorithm is proposed to solve it in Chapter 5.

Extraction in Terms of Schemata Given schemata detected, we can define an extraction as below:

Definition 3.6 (Matching Query)**.** Given a document T, a matching query $M(s) = \{t | t \mapsto s\}$, where s is a schema and t is a DI of T.

Based on Definition 3.5, DIs that have the same structure as s will be returned by the matching query. This query is a bit naive, but can be extended easily [Bergholz, 2000]. By defining an extraction as such a query, the schemata can be treated as extraction rules and extraction acts as query in relational databases. Unlike detection of schemata, such a query in terms of schemata can be evaluated [Bergholz, 2000] with polynomial complexity.

Other Operations A DI and its sub-DIs conform to a set of schemata; i.e., it is possible to use a set of schemata to describe a document. Chapter 6 will discuss how to manage documents based on the set of schemata conformed by DIs appearing in these documents.

Based on the above introduction, the extraction in terms of schemata can be executed efficiently, while to detect this kind of schemata is hard. We find it is frustrated to obtain a single schema model that can both be detected and used to extract efficiently. Thus, in the following chapters, we shall study in two directions:

- In Chapter 4, we formally define and analyze some classes of schemata that are abstracted from schemata in this chapter. These schemata have higher expressive power, and are hard to be detected automatically, but can be generated by a person easily.
- Chapter 5 introduces a restricted version of the schemata in this chapter. Detection algorithms of these schemata are given and optimized.

3.5 An Example Concierge System

Web Information Collection, Collaging and Programming System (Wiccap) is a Web information concierge system developed in Centre for Advanced Information Systems (CAIS)[1], on which the thesis is based. This section demonstrates the primitive version of Wiccap [Liu et al., 2002b].

To extract Web data using this system roughly needs three steps: (1) Define a common logic data structure to store data extracted from Web sites belonging to the same area, e.g., news Web site and product list Web site; (2) Detect the mapping from physical structures of Web documents to the logic structure; (3) Schedule an extraction engine to

[1] http://www.cais.ntu.edu.sg/

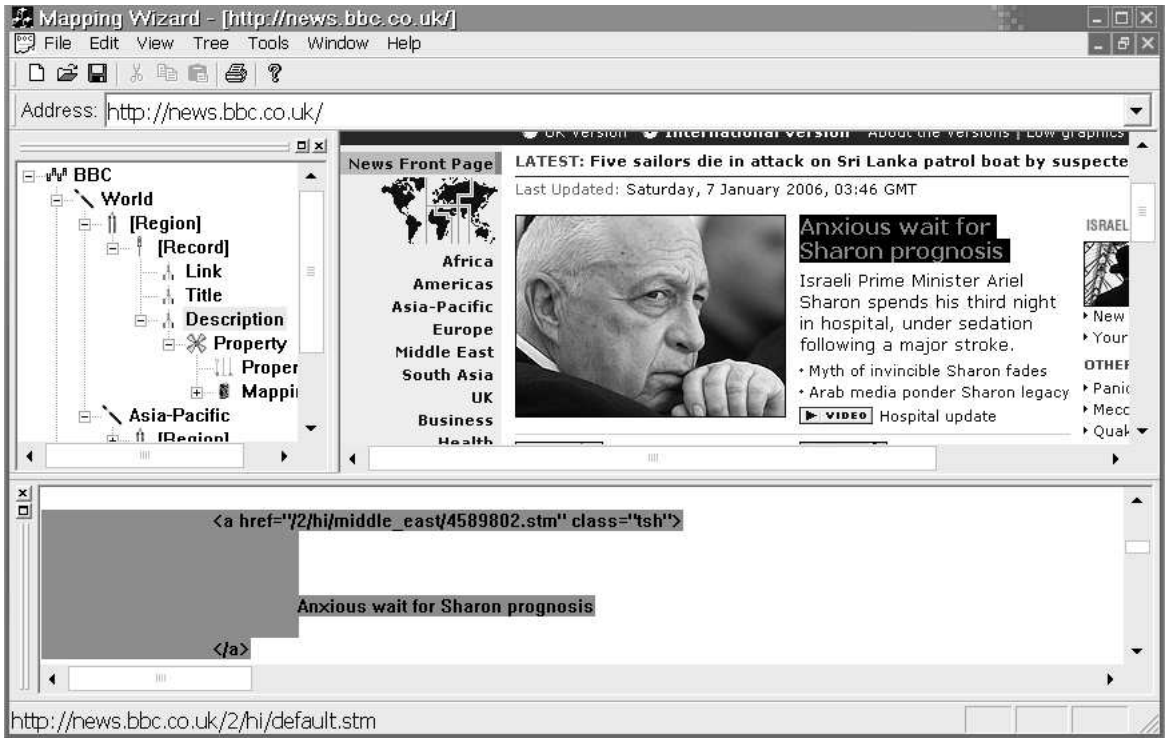


Figure 3.3: Visual Interface of Wiccap

read Web documents, with the mapping relation between physical structures and logic structures, to extract special parts in documents.

Figure 3.3 is the visual interface of Wiccap that aids the first two steps. The left panel is the logic data structure editor. In this figure, the logic data structure describe a common view over news Web sites. Usually, a news Web site organize news in term of world areas, i.e., Asia news, Europe news, etc. In each area, there is a list of news entries, and each entry contains news title, news description, etc.

After designing the logic data structure, a user should define the mapping from physical Web documents to the logic structure. In Figure 3.3, the right panel shows Web pages rendered from Web documents. The user can select a node in the logic structure, and then select texts in the right panel corresponding to the logic node. Then the surrounding texts around the selected text are shown in the bottom windows in Figure 3.3. By recording the surrounding texts with the logic structure, so-called extraction rules, Wiccap can map the logic structure to data to be extracted in Web documents. Based on the extraction rules generated, Wiccap engine is scheduled to parse given rules and do real extraction.

In the process described above, a logic data structure can be formalized as a schema defined in Section 3.3.1. The schema contains only a DI, without any attribute. Web documents can also be formalized as DIs, where each subtree is parsed from texts surrounded by a pair of texts recorded in extraction rules. The extraction engine locates all leave nodes from the DIs and organizes these nodes according the logic structure.

During the process to apply Wiccap on real Web environment, we found some weakness of the primitive version Wiccap: (1) Although Wiccap provides a visual interface to aid schemata generation, schemata are essentially detected manually. (2) A schema is corresponding to whole pages only, and it is not easy to do extraction with finer granularity. (3) A schema contains only one DI, and lack expressive capability, e.g., it is difficult to describe relationships among sibling DIs using a schema. (4) A few operations are provided. Later chapters will introduce further research to overcome these weaknesses.

Chapter 4

Inside The Concierge Framework

In Section 3.5, a sample system is demonstrated to build up a system based on the framework introduced in Chapter 3. The schemata in the sample system is represented as some simple tree structures; and the DIs are parsed from plain text files based on text delimiter pairs defined by users, i.e., a given text file will be parsed to a DI, and texts between a text delimiter pair will be parsed to a sub-DI.

The sample system suffers from lack of high expressive schemata, and there is much space to improve the expressive capability of schemata. In this chapter, to discuss expressive capability, alternative formalisms of instance layer and schema layer are presented, and provide a theoretical framework to compare expressive capability of schemata. When we choose high expressive schemata, the components in operation layers should be changed accordingly, e.g., the extraction engine needs to be revised.

Fortunately, our concierge framework provides a scenario to integrate various techniques. In the background scenario of Webinformation concierge framework, this chapter will discuss the cross-fertilization relationships among tree language, logic program and tree automata: (1) Tree language theory provides a theoretical basis to model DIs and schemata, and compare schemata expressive power; (2) Logic program provides a declarative perspective of DIs and schemata; (3) Tree automata are natural tools to evaluate tree language and do extraction.

Section 4.1 investigates tree language characterization of instances and schemata. From this characterization, a set of DIs is a tree language, and corresponding schemata are grammar of the tree language. By studying related results in grammar induction, we can find to automatically detect general schemata is a hard problem. Based on logic theory, an alternative formalism of instances and schemata is given in Section 4.2. Based

on the formalism given, extraction rules from various systems can be compared.

To put the framework into practical systems, efficient algorithms of extraction are to be devised. In Section 4.3, we propose a new class of tree automata as the computational model of extraction components, which can recognize Web documents efficiently. Some augmented version of standard tree automata are devised to do complicated extraction.

Section 4.4 focuses on a Web data extraction language (Wdel) based on the framework, which is implemented in our Web data extraction system — Wiccap. A program of this language can be treated as extraction rules, and the tree automata introduced later can execute the program.

4.1 Tree Language Characterization

In this section, characterizations of instances and schemata are introduced from the perspective of tree language theory. We first study how a set of DIs can be treated as a tree language; and then the relationships between schemata and tree grammars are discussed. If we exploit tree grammars as schemata, schema detection will become a grammar induction problem. Section 4.1.3 discuss schema detection from the perspective of grammar induction.

4.1.1 Tree Language and Instance

Given a set of symbols, so-called the alphabet Σ, a sequence of symbols in Σ is a string. A set of strings is a string language. Given the simple definition of string language, we can see, if Σ is the character set of HTML and XML, Web documents to be extracted can be treated as some string languages. When Web documents are treated as string languages, we also call them flat texts.

In instance layer, instances are stored as XML (or HTML) documents. By modeling XML documents as flat texts, it is natural to exploit automata to process them. However, in the Web world, to treat all documents as flat texts is too simple and prone to lose important information, as a string language is difficult to represent the hierarchical relationship among contents in XML documents. By using tags, XML documents can be parsed to hierarchical trees. Tree automata are such machines that process trees, like the role of finite automata on string languages.

The basic concepts of tree automata theory includes:

- Symbol: A symbol is an atomic item, which usually is a letter. In this thesis, α, β, β_i and α_i is used to represent symbols, where i is a positive integer.
- Alphabet: An alphabet is a set of symbols, denoted as Σ.
- Terms: Terms over alphabet Σ is denoted as $\mathcal{T}(\Sigma)$, is inductively defined as below:
  - If $\alpha \in \Sigma$, then α is a term,
  - If $\alpha \in \Sigma$ and t_1 through t_n are terms, then $t = \alpha(t_1, t_2, \ldots, t_n)$ is a term, and t_1 through t_n are child terms of t's root node.

  In this definition, the number of child terms has an upper-bound. This kind of terms are also named ranked terms [Comon et al., 1997]. As in XML specification, there is no such a bound to the number of child elements, if we want to use terms to model XML documents, it is more natural to define term as below:
  - If $\alpha \in \Sigma$, then α is a term,
  - If $\alpha \in \Sigma$, then $\alpha(2^{\mathcal{T}(\Sigma)*})$ is a unranked term, where $2^{\mathcal{T}(\Sigma)*}$ is a regular set of sequences of terms; i.e., strings over $\mathcal{T}(\Sigma)$. This means the sequence of children nodes of α is a regular language over terms.
- Tree Language: A tree language is a set of terms. The language of ranked terms is called ranked tree language, and language of unranked terms is called unranked tree language.

A Web document can be represented as a ranked term iff it can be parsed into a document tree. This proposition can be obtained by directly comparing the definition of document tree and term. As there exists no upper-bound of child elements of an element in XML, we shall therefore insist on treating all data instances as unranked tree languages. Without confusion, we shall treat unranked term and document tree as the same in the thesis. A set of trees is also called a *hedge*, following definition from Courcelle [1989], Brüggemann-Klein et al. [2001].

4.1.2 Tree Grammar and Schema

Like the role of string grammar in automata theory, a tree grammar is a set of rules that generate a tree language. Formally, a tree grammar is a 5-tuple $\langle \Sigma, S, N, T, P \rangle$ that consists of a set of start symbols S, a set non-terminal symbols N, a set of terminal symbols

T and a set of production rules P. Rules in P have the form of $\mathcal{L} \rightarrow \mathcal{R}$. With different constraints on $\mathcal{L}$ and $\mathcal{R}$, the tree grammars have different expressive capability. We briefly list some classes of tree grammar, with decreasing order of expressive capability.

Context-Free Tree Grammar $\mathcal{L}$, the left side of a production rule in context-free tree grammar has the form of $n(x_1, x_2, \ldots, x_n)$, where $n \in N$, x_1 through x_n are members of the set of variables $\mathcal{X}$, where x_i can be replaced with an element in $N \cup T$ and $\mathcal{X} \cap (N \cup T) = \emptyset$. $\mathcal{R}$ has the form of $\mathcal{T}(N \cup T \cup \{x_1, x_2, \ldots, x_n\})$; i.e., a term over alphabet $N \cup T \cup \{x_1, x_2, \ldots, x_n\}$.

Regular Tree Grammar In a regular tree grammar, $\mathcal{L} \in N$; i.e., only single non-terminal symbol can appear in the left side of a production rule. $\mathcal{R}$ has the form of $t(2^{\mathcal{T}(N \cup T)*})$, where $t \in T$.

Local Tree Grammar In a regular tree grammar, if two production rules sharing the same right hand side cannot have the same left hand side, the grammar is a local tree grammar.

Given a tree grammar, $G = \langle \Sigma, S, N, T, P \rangle$, a set of terms can be generated according to production rules. A term t_j can be generated from term t_i, denoted as $t_i \rightarrow_G t_j$, if there is a term L appearing in t_i, R appearing in t_j, and $L \rightarrow R \in P$. A term t is generated from a grammar G, if there is a sequence of generation, $s \rightarrow_G t_1, t_1 \rightarrow_G t_2, \ldots, t_i \rightarrow_G t$, denoted as $s \rightarrow_G^+ t$, where $s \in S$. The set of terms belonging to $\mathcal{T}(T)$ that can be generated from G is called the tree languages generated from G, denoted as $L(G)$.

Local tree grammar is strictly less powerful than regular tree grammar. For example, the language of a(b,c),a(c,b) cannot be generated by a local tree grammar, but is generated by the following regular tree grammar:

Σ=`{s,a,b,c}`, `S=N={s}`, `T={a,b,c}` and
`P={s`$\rightarrow$ `a(b,c), s`$\rightarrow$ `a(c,b)}`

As defined in Definition 3.5, a type is a set of trees conforming to a schema. We loose the restriction to define type as a set of trees that can be generated by a tree grammar, and say that trees generated from a tree grammar conform to the grammar. As schemata can be viewed as structure patterns used to identify a type, from this perspective, a tree grammar can be treated as a schema.

Murata et al. [2001] propose a framework based on tree grammar to compare expressive capability of six important XML schema languages. In his framework, finer classifications of tree grammars are defined, including regular tree grammar, restrained competition tree grammar, single-type tree grammar and local tree grammar. The six schema languages are analyzed inside this framework. Ranked on expressive capability by Murata et al. [2001], the languages are RELAX, XDuce, TREX, XML-Schema and DTD. As stated in [Comon et al., 1997], context-free tree grammar exceeds regular tree grammar in term of expressive capability. Thus, most present XML schema languages are strictly less powerful than context-free tree grammar. As the objectives of studying tree grammar in this thesis is to obtain methods describing schemata of Web documents, context-free tree grammars will not be addressed any more.

Based on Definition 3.4, a schema is essentially a set of DIs where some labels are replaced with "*". If the DI set D of a schema s can be generated from a tree grammar G, then the set D_s of all DIs conforming to this schema can also be generated from this tree grammar, with an auxiliary rule $* \vdash \ell$. From this perspective, we can also treat G as the schema of DIs in D_s.

4.1.3 Schema Detection

Before the extraction starts, the extraction rules should be generated. Hand-coded extraction rules is used in some practical Web extraction systems [Baumgartner et al., 2001b, Gupta et al., 1997, Li and Ng, 2004]. Convenient visual interfaces are provided in Wiccap [Li and Ng, 2004], Lixto [Baumgartner et al., 2001b] to help rule generation. However, hand-coded rule generation is very time-consuming, and this section focuses on automatic methods only.

In Section 4.4, we shall introduce the process of rule generation, where schema detection is the main task. Basically, automatic rule generation can be reduced to schema detection problems. When we model schemata as tree grammars, schema detection is the problem of *grammar induction*, which is a sub-problem of *inductive inference*. Based on materials from Gold [1967] and Mitchell [1997], we say inductive inference is a learning problem that consists of four elements: (1) A class of inference objectives. We call them as target grammars. Target grammars can generate Web documents to be extracted. (2) A method of information presentation about the target grammars; e.g., some sample Web documents generated from the target grammars. (3) The spaces of hypothesis of

target grammars learnt from information presentation. (4) The definition of learnability or performance measurement.

Various information presentations are used in literature [Gold, 1967] to induce exact grammars, however, only positive examples are available for Web data extraction system [Crescenzi et al., 2001, Kosala et al., 2003]; i.e., we can obtain information of the target grammars from target Web documents, and these documents are generated from the target grammar. We call these positive examples as the training set or sample documents. Given training sets, a schema detection procedure gives some hypothesis of the target grammars. Here, the hypothesis spaces can be tree grammars. Actually, the choice of hypothesis spaces in literature is rather ad-hoc and various formalisms are given [Chang and Lui, 2001, Gottlob and Koch, 2002a, Kushmerick, 2000a]. We shall introduce logic programs as the hypothesis space in later sections.

Models of exact learnability are well studied in the past decades; e.g., *identification in the limit* [Gold, 1967]. Unfortunately, the results are disappointing to devise practical rule generation methods based on these models. Gold [1967] prove it is impossible to identify a target grammar from positive examples only, except the language generated by the grammar is finite. Valiant's famous approximate learning model *PAC-identification* [Valiant, 1984] is exploited by Kushmerick [Kushmerick, 1997] to generate extraction rules. Although his rules are quite simple, the number of samples needed is very large. Sometimes, it is infeasible to prepare such a large training set. Kosala's [2003] work assumes there exists a representative documents generated from the grammar that can generate all other documents to be extracted, and treats Web documents as k-testable tree languages, which can be identified in the limit from positive examples. This assumption is so strong that the grammars induced are too sensitive to noise in documents.

Even when proper training sets are available to induce exact grammar based on the model of identification in the limit, the typical induction algorithm of identification by enumeration [Gold, 1967, Knuutila, 1994] is burdensome for Web data extraction systems, and even the extremely restricted schemata is hard to be detected. Fortunately, recent empirical results show it is possible to construct polynomial heuristic induction algorithms that generate acceptable approximate results. There are some other work [Arasu and Garcia-Molina, 2003, Chang and Lui, 2001, Crescenzi et al., 2001, Rajaraman and Ullman, 2001] detects patterns from documents and heuristically construct target grammars. Compared to polynomial time complexity of these methods, we

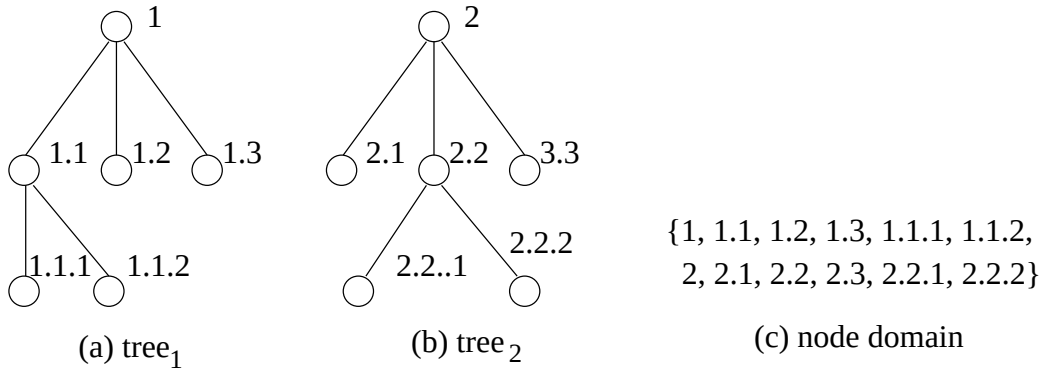


Figure 4.1: Example of Hedge and Node Domain

shall introduce a class of pattern, named k-subtrees, that can be detect in linear time and generate good extraction results. Chapter 5 will introduce detection algorithms that are more efficient than those in literature.

4.2 Logic Characterization

To represent data instances as tree languages provides the possibility of employing tree grammars as schemata. Defining tree grammars as schemata delivers a procedural perspective of DIs; i.e., there is a process that generates DIs from schemata. In this section, we introduce a relational representation of DIs, and use logic programs as schemata. The proof-theoretic interpretation [Ullman, 1988] of rules in logic programs provides another procedural perspective of DIs. At the same time, logic programs also provide a declarative perspective over DIs; i.e., a logic program describes the DIs whose corresponding relational structures make all rules in the program true.

4.2.1 Relational Structure and Instance

A node in a tree can be represented by a sequence of positive integers concatenated by a `"."` character. In $tree_1$ drawn in Figure 4.1, the root node is represented by `"1"`, the first child of the root is `"1.1"`, the second child is `"1.2"`, and so on. All possible positive integer sequences is called the *universe* of tree nodes. For a hedge, all nodes compose a subset of the universe, so-called the domain of nodes in the hedge, such that:

- For empty hedge, the domain is empty, otherwise

- $i \in [1, n]$ is in the domain, if there are n trees in the hedge,
- If $u.j$ is in the domain, then u is also in the domain; if $i < j$, then $u.i$ is also in the domain.

Figure 4.1(c) is the domain of nodes in $tree_1$ and $tree_2$. Given a domain of nodes in a hedge, a relational structure representing the hedge can be built based on parent-child, sibling relationships among nodes and the relationship of nodes and labels.

$$\langle dom(T), \mathit{first\_child}, \mathit{next\_sibling}, \mathit{label}_\alpha \rangle$$

In the structure, $\alpha \in \Sigma$ where Σ is the alphabet, $\langle u, a \rangle \in \mathit{label}_a$ if $u \in dom(T)$ and u is labeled with a, $\langle u, u.1 \rangle \in \mathit{first\_child}$ if $u.1 \in dom(T)$, and $\langle u.i, u.j \rangle \in \mathit{next\_sibling}$ if $j = 1 + 1$ and $u_i, u_j \in dom(T)$.

4.2.2 Logic and Schema

Given the relational representation of data instances, logic program can be exploited to represent a set of DIs. We first introduce some basic notation of logic.

Atom An atom is a predicate of the form $P(x_1, x_2, \ldots, x_n)$, where P is a predicate symbol and x_1 through x_n are variables of the predicate. These variables are called terms. The syntax of atom represented using Backus-Naur Form (BNF) is given as below:

<atom>::=<PredicateSymbol>|<PredicateSymbol>(TermList)
<TermList>::==<Term>|<Term>,<TermList>

Term Syntax of a term can be defined the similar way as a ranked term in tree language

<Term>::=<Constant>|<Variable>|<FunctionSymbol>$(t_1, t_2, \ldots, t_n)$

Term in logic and term in tree language have different meaning; when we mention terms, they are terms in tree language. Term in logic will be referred as logic term henceforth.

Literal A literal is an atom or negative atom, as represented below:

<Literal>::=<Atom>|¬<Atom>

Formula A first-order logic formula is an expression with the following forms:

<Formula>::=<Literal>|¬<Formula>|<Quantifier><Formula>|
<Formula>∨<Formula>|<Formula>∧<Formula>

<Quantifier>::=∀|∃

Before we compare logic and tree grammar in term of expressive capability of representing schemata, we briefly introduce first-order logic (FOL) program and monadic second-order logic (MSOL) program.

FOL FOL formulae are formulae defined above, where a variable is an element in a domain; e.g., the domain of nodes in trees. Such a variable is also called first-order variable.

MSOL MSOL formulae extends FO logic formulae by allowing a variable to be a set of elements in a domain. These variables ranging over sets are called second-order variable, and will be denoted as uppercase characters; e.g., X, Y.

As a classical knowledge of logic theory, FO logic formulae can be normalized to Skolem normal form:

$$\forall x_1, x_2, \ldots x_n M$$

M is a conjunctive normal form; i.e., a formula without quantifiers that has the form like

$$(A_i \vee A_j \vee \ldots) \wedge (B_i \vee B_j \vee \ldots) \ldots$$

A_i through A_j and B_i through B_j are literals. We only consider closed formulae that all variables are quantified, as given domain of variables, closed formulae can be evaluated to true or false without other requirements. If all variables in M are quantified, the prefix quantifiers of Skolem form can be removed. Then, a closed FO logic formula can be transformed to a conjunctive normal form, which is conjunction of a set of disjunctive statements, so-called the logical program of the formula. A FO logical program can be transformed to a Gentzen formula of the form $A_1 \vee \ldots \vee A_k \leftarrow B_1 \vee \ldots \vee B_n$, where A_1 through A_k and B_1 through B_n are atoms. When $k \leqslant 1$, the Gentzen formula is called Horn formula, and a horn formula without functions is called a datalog rule, which is usually written in the form $A :- B_1, B_2, \ldots, B_n$. It is a classical result that datalog and FO logic have the same expressive power in sense of querying relational databases.

A MSO formula can be transformed to monadic datalog is similar way. Gottlob and Koch [2002b] present an important result:

Proposition 4.1. Given a set of built-in predicate so-called the signature over unranked trees:

$$\tau = \langle root, leaf, label_\alpha, first\_child, last\_child, next\_sibling \rangle$$

MSO and monadic datalog are coincides with each other to represent terms.

As Thomas [1997] states, a tree language is regular iff it is definable in MSO, monadic datalog program can be used to define regular tree languages. From the declarative perspective, that means we can represent all trees belonging to a regular tree language with relational structures, and there is a MSO logic program where all rules are true, given the facts in the structures. Such relational structures of the tree language is called the model of logic program.

Like the relationship between tree language and grammar, there is also a sequence of process to generate the relational structures from given logic program, so-called the *proof-theoretic interpretation* [Ullman, 1988] of the rules in the program. For FO and MSO logic programs, The proof-theoretic interpretation of a logic program is also called the minimal model or the least fixed point (LFP) of the program.

Given data instances represented as relational structures, if these structures compose the minimal model of a logic program, the program can also be treated as schemata of these data instances.

Based on logic and tree language characterization of the concierge framework, both declarative and computational model for Web data extraction can be devised. Like the cross-fertilization relationships among automata, logic and database [Neven, 2002a], the different characterizations of our framework will help further development.

4.3 Extraction Machine

Web data extraction needs to figure out document fragments conforming to given schemata. That means, we need a computational model to compute data instances and schemata; i.e., an algorithm that answers whether a data instance conforms to a given schema. In this section, based on the implementation of our framework from perspective of tree language theory, we shall introduce such a computational model, so-called query automata.

Query automata proposed in this section do not only answer whether a data instance conforms to given schemata, it can also output transformed data instances.

4.3.1 Tree Automata

Inside the concierge framework, after schema layer is built, components in operation layer should have the capability to put query in terms of schemata to instance layer. A tree language is a set of trees, thus, the data instances embedded in Web documents to be extracted can be viewed as a tree languages. As introduced in previous section, tree grammars can be used to represent schemata; a query (or extraction) can be treated as the process of recognize data instances that are generated from given tree grammar.

Tree automata are such computational models that accept tree languages generated from given tree grammars; e.g., top-down or bottom-up non-deterministic tree automata (NFTA), top-down or bottom-up deterministic tree automata (DTA) and alternating tree automata [Comon et al., 1997]. Bottom-up NFTA and DFTA can accept tree languages generated from regular tree grammars.

Definition 4.1 (Bottom-Up DTA)**.** Bottom-up deterministic tree automata is a 4-tuple $\langle Q, \Sigma, \delta, F\rangle$, where Σ is an alphabet, Q is a state set, F is a set of final states, $\delta : \Sigma \times 2^{Q*} \rightarrow Q$ is a move function and 2^{Q*} is a regular language over Q. The δ is a function means that with the same $\alpha \in \Sigma$ and regular set 2^{Q*} over Q, $\delta(\alpha, 2^{Q*})$ is a unique state.

Definition 4.2 (Bottom-Up NTA)**.** Bottom-up non-deterministic tree automata is a 4-tuple $\langle Q, \Sigma, \delta, F\rangle$, where Q, Σ and F have the same meaning as them in Bottom-up NTA, and δ here is a many-to-many relation between $\Sigma \times 2^{Q*}$ and Q.

The definition of tree automata can be intuitively connected with tree languages. A bottom-up tree automaton read a hedge from leaf to root. When the automata read in a leaf node, it will move to a state according to the label of the leaf; when all children of a node are read in, the automata will move to states according to the node's label and states associated with its children nodes. If the automata transit into a state in F when after it read the root of a tree, the tree is accepted. If all terms in a hedge is accepted, the hedge is accepted. The hedge containing all terms accepted by an automaton is a tree language.

At any time during reading a tree from bottom to up, a tree automata will be in a set of states. Each state is this set is associated with a node in the tree, which is the set of

current node. As a node can be associated with a state only after all its children nodes have associated states, a pair of ascendant-descendant nodes will not be in the current node set at the same time.

A possible set of current nodes is a cut of the tree. Formally, cut of trees and acceptable tree language of an automata is defined as below:

- A cut of a tree is a set of nodes in the tree, such that there is only one node in the path from root to each leaf. A configuration of automata is the set of states associated with nodes in a cut.

- A bottom-up tree automaton read nodes of a tree from leaves. The set of all leaves is a cut, and the set of states associated with the cut is initial configuration.

- If the current node set is the cut containing root only, and the current configuration is in final states, the tree is acceptable to the automata. If all trees in a hedge is acceptable, the hedge is acceptable.

- The hedge containing all trees acceptable to an automaton is the tree language accepted by the automata.

A regular tree language is the language accepted by a bottom-up tree automaton. We do not distinguish non-deterministic and deterministic tree automata, as there is a classical results [Comon et al., 1997, Gottlob and Koch, 2002b]:

Proposition 4.2. Let L be a regular tree language accepted by a bottom-up NTA. Then there exists a bottom-up DTA that accepts it.

The above tree automata are called bottom-up as they recognize trees from leaves. Another class of tree automata, so-called top-down tree automata, start computing from root node of trees, as defined below:

Definition 4.3 (Top-Down DTA)**.** Top-Down deterministic tree automata is a 4-tuple $\langle Q, \Sigma, \delta, I \rangle$, where I is a set of initial states, $\delta : Q \times \Sigma \rightarrow 2^{Q*}$ is a move function.

The difference between the top-down NTA and DTA is that in a top-down NTA, Σ is a many-to-many relation. From the definition of top-down DTA and local regular tree languages, we can find they have the same expressive capability.

4.3.2 Tree Transducer

Tree automata provide the mechanism to recognize whether hedges are generated from given grammars. This is the basis to extract Web data, as it is an important step to figure out which subtress in a document hedge contains needed information. Furthermore, an extraction task usually is not finished at this step. As in Web document trees, many structures are used to describe visual layouts in browsers, and are not interesting to extraction tasks, approaches is needed to select those nodes sensitive to user requirements from subtrees and transform these selected nodes to data instances conforming to given schemata.

Suppose the nodes to be extracted are known already, we can exploit tree transducer to transform these nodes to given structures. We define a class of transducer as below:

Definition 4.4. Bottom-up non-deterministic tree transducer (NTT) is a 5-tuple $\langle Q, \Sigma, \Sigma', \delta, F, \lambda \rangle$, where Q, F have the same meaning in bottom-up TA, Σ and Σ' are input and output alphabet respectively. $\delta : \Sigma \times 2^{(Q \times \mathcal{T}(\Sigma))*} \rightarrow Q \times \mathcal{T}(\Sigma')*$ is a move relation, $\lambda : \Sigma \times Q \rightarrow \{1, 0\}$ is a function, s.t. $\lambda(a, q) = 1$ is the node associated with q is to be extracted.

Bottom-up NTT is similar with bottom-up TA, except that there is an output term associated with each state, and when NTT move from some source states to a target state, the term associated with each source state is passed to the target state. In the final state, the term associated is the transduced form of the trees accepted. As an example, the move relation δ has the form described as below:

$$\alpha \times (q_1 \times t_1)(q_2 \times t_2) \ldots (q_n \times t_n) \rightarrow q \times t$$

Here, t is a hedge and can be a function on α and t_1 through t_n. In this thesis, we only consider a class of simple function with the form $t = \alpha(t_1, \ldots, t_n)$ if $\lambda(\alpha, q) = 1$, or $t = t_1, \ldots, t_n$ if $\lambda(\alpha, q) = 0$, where t_i can be $\varnothing$. With this restriction, a transduction process can be treated as delete some unwanted nodes from the original tree accepted by the transducer. In Section 4.3.3, we shall introduce how to describe which nodes are wanted and which are not.

4.3.3 Query Automata

In Section 4.3.1, we introduced some tree automata that can accept regular tree languages and have same expressive capability as MSO logical programs. However, the task of extraction only needs to select some nodes in the tree, instead of outputting whole tree. To represent subset of nodes in a tree, MSOL and monadic datalog are strictly more powerful than regular tree grammar.

As an example, we can build a monadic datalog program to represent a tree, based on the built-in predicate set τ and the following extended predicates:

```
right_sibling(X,X):−
right_sibling(X,Y):−next_sibling(X,Z),right_sibling(Z,Y)
child(X,Y):−first_child(X,Y)
child(X,Y):−first_child(X,Z),right_sibling(Z,Y)
descendant(X,Y):−child(X,Z), descendant(Z,Y)
```

A tree that has more than 2 level and whose root is labeled with α can be represented using the following program:

```
root(1):−
label_α(1) :− root(1)
SELECT(x):− descendant(1,x)
```

Given the rule $SELECT(x) :- descendant(1, x)$ in the above program, $\{x|SELECT(x) = true\}$ will be the set of all selected descendant nodes of the root. The program can be treated as a unary query on tree nodes; i.e., query result is a set of nodes. An unary query can also be represented by a tree automata, if we ask the automata output corresponding nodes in specified states. However, no such a bottom-up tree automata that can represent the same query represented by above datalog program. As when the automata process descendants of root, the root's label is unknown, and when the automata go up to the root, descendants' information cannot be accessed in the state.

This example shows that although bottom-up tree automata have the same expressive power as monadic datalog to represent a tree structure, it is less powerful to select nodes in a tree. Neven and Schwentick [1999] suggest a class of tree automata, so-called strong two-way tree automata, that coincides with monadic datalog to represent node queries. We use their formalism, defined as below:

Definition 4.5 (Strong Query Automata)**.** A strong query automata (SQA) is a tuple $\langle Q, \Sigma, \delta_{root}, \delta_{leaf}, \delta_{\downarrow}, \delta_{\uparrow}, \delta_{\_}, F, I, \lambda \rangle$, where Q, Σ, I and F have the same meaning as Q, Σ, I and F in bottom-up and top-down TA, $\lambda : Q \times \Sigma \rightarrow \{0, 1\}$ is a select function, $\delta_{\downarrow}$, $\delta_{\uparrow}$ and $\delta_{\_}$ are down move, up move and stay move functions respectively.

Strong query automata's move function can be divided to five categories. Let U and D be two disjunctive set over of $Q \times \Sigma$, and $U_{\uparrow}, U_{\_} \in U*$, where $U*$ is regular set over U and $U_{\uparrow} \cap U_{\_} = \emptyset$.

1. $\delta_{leaf} : D \rightarrow Q$ is the move function for leaves. If the SQA is processing a leaf node n associated with state q, and there is $\delta_{leaf}(q, label(n)) = q_1$, then the SQA will associate q_1 with n.

2. $\delta_{\downarrow} : D \times N \rightarrow Q*$ is the down move function, where N is the natural number set, and $\delta_{\downarrow}(q, \alpha, i)$ is a string of length i, and $\{\delta_{\downarrow}(q, \alpha, i) | i \in N\}$ is a regular set. If the SQA is processing a node n associated with state q, and there is $\delta_{\downarrow}(q, label(n), m) = q_1 q_2 \ldots q_m$, the SQA will leave n and goto process the m child nodes of n and associate state q_i with n_i, where n_i is the ith child node of n.

3. $\delta_{root} : U \rightarrow Q$ is the move function for root. If the SQA is processing the root node n associated with state q, and there is $\delta_{leaf}(q, label(n)) \rightarrow q_1$, then the SQA will associate q_1 with the root node.

4. $\delta_{\uparrow} : U_{\uparrow} \rightarrow Q$ is a up move function, and $\{\varrho | \delta_{\uparrow}(\varrho) = q\}$ is a regular set. If the SQA is processing node n's child nodes n_1 through n_m, where n_i is associated with state q_i, and there is $\delta_{\uparrow}(\langle q_1, label(n_1)\rangle, \langle q_2, label(n_2)\rangle, \ldots, \langle q_m, label(n_m)\rangle) = q$, the SQA will leave n_1 through n_m to process n, and associate q with n.

5. $\delta_{\_} : U_{\_} \rightarrow Q*$ is the stay move function, which is computed by a generalized string query automata (GSQA) [Neven and Schwentick, 1999]. For child nodes of each node, there is at most one stay move.

   If the SQA is processing node n's child nodes n_1 through n_m, where n_i is associated with state q_i, and there is $\delta_{\_}(\langle q_1, label(n_1)\rangle, \langle q_2, label(n_2)\rangle, \ldots, \langle q_m, label(n_m)\rangle) = q_{1.1} q_{1.2} \ldots q_{1.m}$, the SQA will stay in n_1 through n_m and associate $q_{1.i}$ with n_i.

.

When a SQA begin to run on a tree t, it is in an initial configuration; i.e., the SQA start by processing the root node $root(t)$ only, and associate a state with the node.

According to move functions defined, the SQA will move to a sequence of configurations. If the SQA move to the final configuration; i.e., if state associated with $root(t)$ is $q \in F$, then t is accepted by the SQA.

A SQA selects all those nodes n_i, if in some configuration n_i is associated with q_i, and $\lambda(q_i, label(n_i)) = 1$. The set of nodes selected is the results of unary query defined by the SQA. From Neven and Schwentick [1999], there are two results about the capability of SQA and logic to express unary queries:

Proposition 4.3. A query on strings is expressible by a GSQA iff it is definable in MSO logic.

Proposition 4.4. GSQA coincides with MSO logic to represent unary query over tree nodes.

The strong query automata are used to describe unary queries over tree nodes. In this thesis, our objective is to deliver a machine that can transduce selected nodes to given structures. The query automata need some revision to obtain this objective.

4.3.4 Query Transducer

Definition 4.6 (Query Transducer)**.** A query transducer (QT) is a tuple $\langle Q, \Sigma, \delta_{root}, \delta_{leaf}, \delta_{\downarrow}, \delta_{\uparrow}, \delta_{\_}, F, I, \lambda\rangle$, where Q, Σ, I and F have the same meaning as Q, Σ, I and F in bottom-up and top-down TA, $\delta_{\downarrow}$, $\delta_{\uparrow}$ and $\delta_{\_}$ are down move, up move and stay move function respectively.

Query transducer's move function are defined the same as those of strong query automata, except $\delta_{\uparrow}$ is redefined as below:

$\delta_{\uparrow} : (Q \times \Sigma \times \mathcal{T}(\Sigma)*)* \rightarrow Q \times \mathcal{T}(\Sigma)*$ is a up move function. The value of the function is a tuple, such that $\delta_{\uparrow}(\langle q_1, \alpha_1, t_1\rangle, \ldots, \langle q_m, \alpha_m, t_m\rangle) = \langle q, t\rangle$, where $t = t_{1.1}, \ldots, t_{m.1}$, and $t_{i.1} = \alpha_i(t_i)$ if $\lambda(q_i, \alpha_i) = 1$, else $t_{i.1} = t_i$. If the SQA is processing node n's child nodes n_1 through n_m, where n_i is associated with state q_i, and there is $\delta_{\uparrow}(\langle q_1, label(n_1), t_1\rangle, \ldots, \langle q_m, label(n_m), t_3\rangle) = \langle q, t\rangle$, the SQA will leave n_1 through n_m to process n, associate q with n, and associate t with n.

A QT accepts the same tree language as corresponding SQA. The difference between QT and SQA is that the terms associated with the root nodes in final state in QT is the data extracted. These terms are called the extraction result of the QT from given hedge. The following theorem connects query transducers with MSO logic programs:

Theorem 4.1. An extraction can be defined by MSO logic program, iff it is definable to a query transducer.

Proof: In one direction, to prove MSO logical programs can be simulated by query transducers, we only need to prove that all trees satisfying MSO logic formulae can be accepted by a query transducer. From Definition 4.3.4, the extension to query automata does not change the acceptance condition for a tree language. As query automata coincide with MSO logical program, query transducer can accept those trees described by MSO logic.

In another direction, we need to show a query transducer can be simulated by monadic datalog, as monadic datalog coincides with MSO logic to represent unary queries [Gottlob and Koch, 2002b]. Gottlob and Koch [2002b] present the monadic datalog programs simulating query automata, and we do not repeat the overlay programs in this thesis; i.e., those programs simulating move functions except for $\delta_{\uparrow}$. To simulate $\delta_{\uparrow}$, we need to generate predicates that generate the same tree model as output of query transducers. For each $q \in Q$, $\alpha \in \Sigma$ and $lambda(q, \alpha) = 1$, the following rule is added:

$\texttt{query}(n) \leftarrow \langle x, q\rangle(n), \texttt{label}_{\alpha}(n)$

For each $x \in Q$, $\alpha \in \Sigma$ and $\lambda(q, \alpha) = 0$, there is a rule:

$\texttt{tmpchild}(n_0, n) \leftarrow \langle x, q\rangle(n), \texttt{label}_{\alpha}(n), \langle x_1, x\rangle(n_0)$

Then, the following rules are added to simulate output:

$\texttt{tmpdescendant}(X, Y) \leftarrow \texttt{tmpchild}(X, Y)$

$\texttt{tmpdescendant}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{tmpchild}(X, Z), \texttt{tmpdescendant}(Z, Y)$

$\texttt{outChild}(X, Y) \leftarrow \texttt{child}(X, Y), \texttt{query}(X), \texttt{query}(Y)$

$\texttt{outChild}(X, Y) \leftarrow \texttt{child}(X, Z), \texttt{tmpdescendant}(Z, Y), \texttt{query}(X), \texttt{query}(Y)$

It can be inductively prove, the relational model built is the same as output of the query transducer. ∎

So far, we introduced query transducer as the computational model of extraction machine. The connection between QT and monadic datalog program facilitate the definition of a declarative language to describe an extraction task.

4.4 Wdel Language

In this section, Web data extraction language (Wdel) coinciding with monadic datalog program is defined. In the framework introduced, programs written using Wdel contain

schemata of Web documents to be extracted. As shown in Figure 3.1(b), a document tree may be so complicated that to perceive real data from the document tree is painful for a user. The corresponding schemata of these documents may also be very complicated, thus, to handle documents via schemata is not easy. These structures (and schemata) of documents are named *physical structure (schemata)* in this thesis.

To make accessing to Web data easier, we try to derive logical schemata of the target Web documents and then extract information from the documents based on the schemata. The role of the logic schemata is to relate information from a website in terms of commonly perceived logical structure, instead of physical document structures. The logical schemata here refer to people's perception on the organization of contents in Web documents. Data extracted should conform to the logic schemata. The logical schemata and the mapping from physical schemata to logical schemata are also defined in Wdel programs.

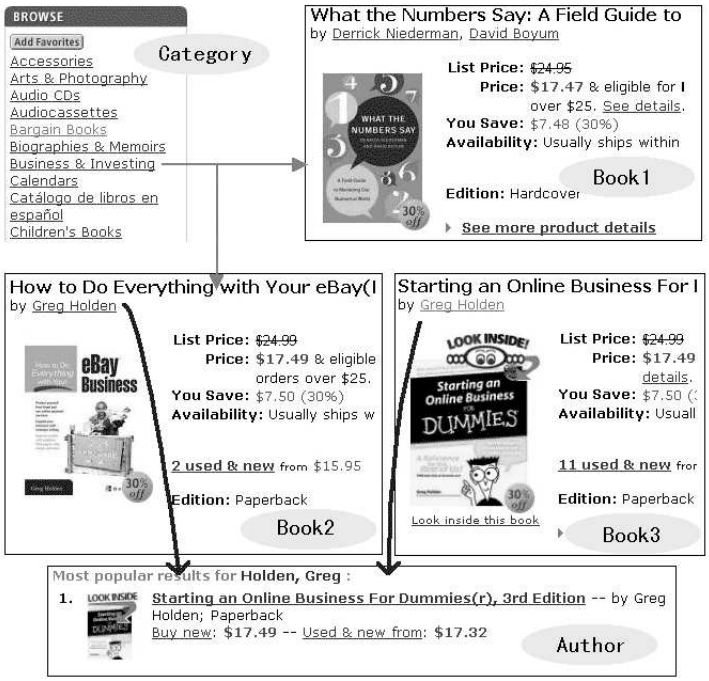
As the equivalent expressive capability of Wdel monadic logic program and query transducer, Wdel programs can be further compiled to low-level codes of QT and executed. Based on the logical view over documents and the physical structures of documents, a user can program Wdel to extract data from documents.

4.4.1 Wdel Introduction

Logic Schema and Physical Schema In this section, we firstly demonstrate an example that is more complex than Example 3.1. The properties of Web documents in the example motivate the concrete definition of Wdel.

Example 4.1. Figure 4.2 includes fragments selected from five different Web pages. The `"category"` is a part of navigator panel of Amazon's homepage. `"Book1"` through `"Book3"` are cut from pages linked from `"category"`. `"Author"` is from the author information page, and it is linked by both `"Book2"` and `"Book3"`.

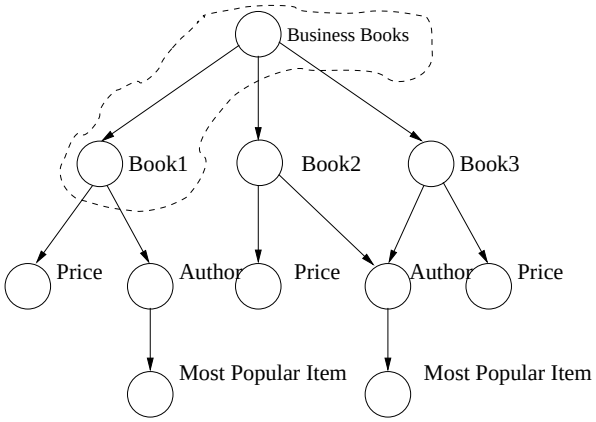
Despite numerous pages and complicated physical structures of Amazon, a user may only care about several books' information, such as those page fragments in Example 4.1. For such a user, his perception of content organization of Amazon usually is much different from physical structure of Amazon documents. From most users' perception, Amazon consists of book categories; e.g., Computer book, Children book, etc. Each category contains a list of books, each of which includes information fields including Price, Title, Author, etc. This hierarchical content organization is a common view of bookstore Web



(a) Web Pages from Amazon

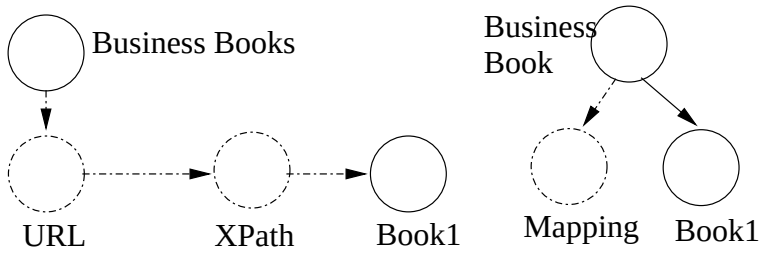
Business	Book1	Price	17.47
		Author	Derrick Niederman
	Book2	Price	17.49
		Author	Greg Holden
	Book3	Price	17.49
		Author	Greg Holden
Arts	...		
...	...		

(b) Output



(c) Logic View

Figure 4.2: Web Pages and Logic View over Web Pages



(a) Full Path (b) Collapse

Figure 4.3: Mapping from Physical Structure to Logic View

sites in most users' mind. If we extract data from a bookstore Web site and organize in this way, a user will be familiar with its structure and can handle extracted data intuitively. Moreover, as a class of Web sites have the same organization structure in users' mind, it is possible to organize data extracted from different Web sites with the same structure.

We call the common user perception of a special class of Web sites as their *logic structure*. Figure 4.2(c) is the logic structure of the five pages in Example 4.1, and Figure 4.2(b) is extracted data stored with this structure.

As introduced before, in our framework, extracted data and target documents share the same data model. Thus, we can also derive schemata of the logic structures. As a logic structure need not to be restricted in a single Web site, we can organize data extracted from multiple Web sites in the same class with the same logic structure, and data extracted from these sites have the same schemata. Thus, a query like *"return all books with schema A"* will return data from various sites. This provide an easy way to integrate Web data.

For a Web information concierge, to fill up the gap between logic schemata and physical schemata is as important as to induce physical schemata. Wdel is a formal language defining the mapping from physical schemata to logic schemata. In a short word, Wiccap parse a Wdel program; extract fragments in Web documents conforming to physical schemata in the program; and transform the fragments to data conforming to the logic schemata specified.

As an example, Figure 4.3(a) is a physical structure corresponding to the two nodes inside dash line in logic structure of Figure 4.2(c). When a user extracts Book1 information from Web, he needs first follow URL in the `"category"` page by clicking a URL,

and then locates the root node of data instance of Book1 by following a path. A Wdel program defines a structure like Figure 4.3(b) to describe schemata corresponding to both the logic structure and physical structure.

The idea of programming Wdel is straightforward. Firstly, a user defines a logic schema corresponding to data extracted and a physical schema corresponding to Web documents. For each pair of parent-child nodes in the logic schema, there should be a pair of ascendant-descendant nodes in physical schema. By collapsing the path between these ascendant-descendant node pairs, the physical schemata will be the same as logic schema. Then, a Wdel program is finished by adding a child node to each node in logic schema that describes the path between corresponding node pair in physical schema, shown as Figure 4.3(b).

Thus, Wdel consists of two parts: (1) elements to define logic schemata and (2) elements to define physical paths. The first part is called schema elements and the second part is named *Mapping Element*, which is described in the following sections.

Schema Elements Wdel program is written in normal XML, conforming to some DTD. We shall define the DTD of Wdel in this section.

schemaElement Schema elements in Wdel are used to define the hierarchical structures of schemata. Actually, the schema elements can define both physical schemata and logic schemata. For the sake that Wdel does not directly give physical schemata definition, when we mention schemata, we mean logic schemata in this section. As we use a simple model of data instance to represent schemata, the schema elements to define schemata are also simple. The DTD of schema elements is listed as below:

```
<!ELEMENT schemaElement (Mapping,schemaElement*)>
<!ATTLIST schemaElement Label CDATA "*" #REQUIRED>
<!ATTLIST schemaElement schemaID ID #IMPLIED>
<!ATTLIST schemaElement nextSibling IDREFS #IMPLIED>
<!ATTLIST schemaElement firstChild CDATA "no" #IMPLIED>
    ⋮
```

The schemaElement is basic construction elements of a Wdel program. The XML document tree conforming to this DTD is a valid schema in Definition 3.4 already, if the first two lines are replaced with the below lines:

```
<!ELEMENT schemaElement (schemaElement*)> <!ATTLIST
schemaElement Label CDATA "*" #REQUIRED>
```

The attributes schemaID, nextSibling and firstChild are added to make Wdel to have the same expressive capability as monadic datalog. The relationship between monadic datalog and Wdel will be discussed in Section 4.4.2. The above DTD is a minimal definition, and schemaElement can be customized to include more schema attributes as those defined in Section 3.3.2. For instance, if a user wants only those DI with size larger than 10 are conform to a schema, a line can be inserted to the DTD as below:

```
<!ATTLIST schemaElement Size CDATA "[>10]">
```

Mapping Elements Mapping elements included in the above DTDs are used to represent a path in a physical schema. If there is an edge $\langle a_1, a_2 \rangle$ in a logic schema, and b_1,b_2 are corresponding nodes in a physical schema, then the path between b_1 and b_2 is represented by the mapping element of a_2. For Web document trees, an edge between two nodes may be intra-page or inter-page. An intra-page edge allows locate any descendant node in the current documents, while an inter-page edge leads to another document from a node in current page. We first introduce several types of edges in Wdel and then introduce how to use mapping element to organize these edges to a path.

PathLoc PathLoc is an element representing an intra-page edge, with DTD as below:

```
<!ELEMENT PathLoc (#PCDATA)>
```

The #PCDATA is a XPath expression. For simplicity, we use a subset of XPath; i.e., a linear expression $[/]b_1.1/b_1.2/\ldots/b_2$, where $b_1.1$ are child node of b_1, $b_1.2$ is a child node of $b_1.1$, and so on. Details of more elements representing intra-page edges can be found in Wiccap [Liu et al., 2002b].

Link Link is an element representing a hyperlink, conforming to the following DTD, where the #PCDATA is a valid URI string.

```
<!ELEMENT Link (#PCDATA)>
```

DynaLink DynaLink is also an element representing a hyperlink. Unlike Link elements that contain a static URI strings, a DynaLink contains information to extract a URI strings from target documents.

```
<!ELEMENT DynaLink (#PCDATA)>
```

The #PCDATA is a XPath expression that is defined as the one in PathLoc. The element pointed by the path should be a URI string.

Form As there are many dynamic documents generated by submitting a form on Web, Wdel includes the Form elements to represent a form that can return dynamic documents.

```
<!ELEMENT Form (#PCDATA)>
```

In the DTD of form elements, the #PCDATA is the code of a HTML form. During interpretation of Wdel programs, these #PCDATA will be rendered in browser windows to interact with users, and return dynamic Web documents.

Mapping Given these elements representing edges in Web documents, a Mapping element can organize multiple edges together to compose a path in Web documents, defined as below:

```
<!ELEMENT Mapping ((PathLoc|Link|DynaLink|Form),Mapping?)>
```

4.4.2 Wdel and Monadic Datalog

So far, we have defined Wdel and introduced the rough process to construct Wdel programs. Before looking into Wdel programming, this section will compare Wdel and monadic datalog in term of expressive power. As we stated in Proposition 4.1, monadic datalog has the same expressive power with monadic second order logic, given the certain predication set. The connection between Wdel and monadic datalog make it easy to define extraction rules in a declarative way.

Theorem 4.2. An extraction is definable for monadic datalog over τ iff it is definable for Wdel with minimal schemaElement definition.

Proof: In one direction, we need to prove Wdel programs can be simulated by monadic datalog over τ. Given a Wdel program written in XML document, the simulation objective is to define a Wdel extraction using monadic datalog over τ, which includes two points: (1) the physical structures of documents to be extracted should be acceptable to a monadic datalog; (2) the logic structure of output should be computable to a query transducer, for that query transducer can be simulated by monadic datalog. In term of describing physical structures, any parent-child element pair in a Wdel program contains information about the ascendant node in physical structure corresponding to the parent element, the descendant node in physical structure corresponding to the child node and the path between these two nodes. Such a node pair can be encoded to the following datalog rule:

$$\texttt{ele}_{desc}(X) \text{:}-\texttt{ele}_{ascd}(Y), \texttt{descendant}_{path}(Y, X)$$
$$\texttt{descendant}_{\alpha.path}(X, Y) \text{:}-\texttt{child}(X, Z), \texttt{label}_{\alpha}(Z), \texttt{descendant}_{path}(Z, Y)$$

The firstChild and the nextSibling attributes can be encoded in similar way, and the physical structures described by a Wdel program can be accepted by the datalog program that encodes all parent-child and sibling element pairs.

In another direction, we need to simulate monadic datalog over τ by Wdel programs. To prove this, we use general strong query automata as auxiliary. As proven by Gottlob and Koch [2002b], each transition in a GSQA can be simulated by a sequence of datalog rules. These datalog rules can furthermore be simulated by a Wdel program. ∎

4.4.3 Example of Wdel Program

With Example 4.1 again, this section will illustrate how to program Wdel to execute an extraction task. This task transforms physical structures of Web documents in Figure 4.2(a) to the logic structure like Figure 4.2(c). Figure 4.4(a) is the Wdel program, which is a validated XML documents conforming to DTD introduced in Section 4.4.1.

Firstly, let us consider how to define the schema of the logic structure in Figure 4.4(c). The structure is rooted from `"Business"` book category; there is a list of books belonging to this category; each book has `"title"`, `"author"` and `"price"` information. For each book, `"title"`, `"author"` and `"price"` appear one and only one time, while the number of books there are in `"Business"` category may vary in corresponding Web documents.

Schema of this structure can be represented using a tree grammar, defined as below:

```
1: <schemaElement Label="Business">
2:  <Mapping>
3:    <Link>http://www.amzon.com/</Link>
4:    <Mapping>
5:      <DynaLink>/html/.../a[7]</DynaLink>
6:    </Mapping>
7:  </Mapping>
8:  <schemaElement Label="Book" schemaID="book" nextSibling="book">
9:    <Mapping>
10:      <PathLoc>/html/.../td</PathLoc>
11:    </Mapping>
12:    <schemaElement Label="Title">
13:      <Mapping>
14:        <PathLoc>a[1]/text()</PathLoc>
15:      </Mapping>
16:    <schemaElement Label="Author"
17:       <Mapping>
18:         <PathLoc>text()</PathLoc>
19:       </Mapping>
20:    <schemaElement Label="Price"
21:       <Mapping>
22:         <PathLoc>span/b[1]/text()</PathLoc>
23:       </Mapping>
24:    </schemaElement>
25:  </schemaElement>
26: </schemaElement>
```

(a) A Wdel Program

```
1: <Business>
2:   <Book>
3:     <Title>What the Number ...<Title>
4:     <Author>Derrick ...</Author>
5.     <Price>$17.47</Price>
6.   </Book>
2:   <Book>
3:     <Title>How to do ...<Title>
4:     <Author>Greg Holden</Author>
5.     <Price>$17.49</Price>
6.   </Book>
2:   <Book>
3:     <Title>Starting an online ...<Title>
4:     <Author>Greg Holden</Author>
5.     <Price>$17.49</Price>
6.   </Book>
1: </Business>
```

(b) Extraction Results

Figure 4.4: A Wdel Program and its Output

Business → Book*
Book → Title, Author, Price

This schema can also be defined as the following datalog program:

```
Business(n):– root(n)
Book(X):– Business(Y),child(Y,X)
Title(X):– Book(Y), first_child(Y,X)
Author(X):– Title(Y), next_sibling(Y,X)
Price(X):– Author(Y), next_sibling(Y,X)
```

Corresponding to the five predicates at the left hand side in above datalog rules, there are five schema elements in this program labeled with "Business", "Book", "Title", "Author" and "Price" respectively. The *first_child*, *next_sibling*, *root* predicates can be simulated by the relationships among schema elements in the XML document of Wdel program, as drawn in Figure 4.4(a). The first datalog rule is simulated by the root element. The second rule is simulated by the schema elements labeled with "Business" and "Book", and so on.

4.5 Summarization

In this chapter, we proposed various perspectives over the implementation of concierge framework. As the lack of theoretical framework to compare different Web information concierge systems in literature, the comparison among these perspectives is the initial work that can help us to integrate different techniques into the framework, and aid further research in Web information concierge systems.

Wdel is the language combining the benefits of both tree language and logic program. Wdel describe Web document structures in a declarative way, and can be parsed by the tree transducer introduced. Based on techniques discussed in this chapter, we improve the sample system in Chapter 3, and Figure 4.5 is the visual interface to generate Wdel scripts[1]. In this interface, the left panel is the hierarchical structure corresponding to Wdel script defined; the upper-right panel is an embedded Web browser and the lower-right panel is the DOM tree and tree node property window corresponding to Web documents in the Web browser.

[1]Please refer to http://www.cais.ntu.edu.sg/~lee/wiccap/example-1.html for complete demo.

Figure 4.5: Visual Interface to Generate Wdel

Compared with the sample system demonstrated in Chapter 3, the main different points include:

1. DIs are simplified DOM tree of Web documents. In Figure 4.5, the lower-right panel replaces the bottom window in Figure 3.3, and contains DOM tree of the Web document.

2. Schemata can describe structures of both Web documents and extracted data.

3. Schemata have the same expressive power as monadic datalog.

4. A schema can define structures corresponding to any fragment of a Web document, instead of the whole document.

With the help of the visual interface, a user can define simple Wdel scripts by clicks on the DOM tree, and then the extraction engine can parse the scripts to do real extraction. Moreover, if we can devise algorithms that detect important fragments

in Web documents, the user may ask the Web browser to show only these fragments, generate the schemata automatically, and improve efficiency more. In next Chapter, we shall introduce the algorithms to detect schemata automatically.[da Silva et al., 2002]

Chapter 5

Efficient Schema Detection

5.1 Schema Detection and Rule Generation

In Chapter 4, WDEL is introduced to define extraction rules. The relation between schemata and extraction rules is also presented. The basic constituents of WDEL are mechanisms defining schemata of data instances, and the mapping from physical schemata and logic schemata. Thus, a main task of extraction rule generation in our framework is to detect schemata of data instances.

Schema Detection is divided into two parts: (1) physical schema detection, and (2) logic schema detection. As mentioned in Section 4.3.4, logic schemata are restricted to be transformed from physical schema, and problem of logic schema detection can be reduced to select subsets of nodes in physical schemata. Thus, in this chapter, we only consider how to figure out physical schemata to generate extraction rules.

It is always a straightforward method to ask an expert to observe Web documents in hand, and induce the schemata by hand [Baumgartner et al., 2001b, Gupta et al., 1997, Li and Ng, 2004]. As the manual methods are not scalable, automatic methods are preferred in Web data extraction scenarios. For that physical schemata can be viewed as tree grammars (or equivalent logic rules) that generate data instances in Web documents, grammar induction approaches are exploited in schema detection. In this section, we shall first briefly introduce grammar induction and its application in Web data extraction.

5.2 Schema Detection and Inference Problem

Schema detection is an inference problem, which is a classical problem originated decades ago [Gold, 1967]. Before describing details of schema detection as an inference problem, we introduce the definition of inference problem in recent literature:

The *inference problem* is to find an allowed *algorithm* M that, given an allowable *representation* for (or a sample of) of a language in the *target class* and the possibly present *additional information*, produces a *hypothesis* H from the chosen space $\mathcal{H}$ fulfilling the given definition of learnability — Timo Knuutila [1994]

From the definition above, an inference problem needs to be depicted from at least six aspects: (1) the class of languages to be inferred, named target class. The language to be inferred is named target language. (2) the representation of information about target languages, (3) the form of space of hypothesis of target languages, (4) allowable inference algorithms, (5) other additional information that helps inference, and (6) how to measure whether the inference algorithms are successful. Different choices on these aspects produce various models for inference problem.

5.2.1 Models for Schema Detection

The problem of Web document schema detection can be depicted as an inference problem, where the first three aspects are: (1) The target languages are Web documents to be extracted. (2) The representation of information is a set of sample documents of the target language. (3) The hypothesis space consists of Web document schemata.. When the schemata in the hypothesis space are represented as language grammars, the inference problem is also called grammar induction.

In a general inference problem, information representation may contain negative examples; i.e., those examples that do not belong to target language. However, in the scenario of concierge systems, usually only positive examples are available as information representation; i.e., sample documents of the target language [Crescenzi et al., 2001, Kosala et al., 2003]. Although the choice of information representation is limited, various models of target languages and hypothesis can be exploited. As introduced in Chapter 4, target documents can be relational data, local tree languages, regular tree languages, etc. Schemata of documents can be tree grammars introduced before, or be

represented as automata, logic programs, and variants in existing work [Chang and Lui, 2001, Gottlob and Koch, 2002a, Kushmerick, 2000a].

Chapter 4 introduced various schemata with different expressive power; e.g., FO logic program, MSO logic program, regular tree grammar, local regular tree grammar, etc. In Web information concierge systems, the following grammar variants are exploited:

- The basic model in Chapter 2 assumes Web documents have tabular structures. The schemata of documents are sequences of delimiter strings and field strings. These sequences can be mapped to relational schemata directly.

- Compared with the simple relational grammar, many systems [Chang and Lui, 2001, Crescenzi et al., 2001] use regular word grammar as document schemata. The schemata are some regular expressions, and Web documents to be extracted should be generated from these expressions.

- Modeling Web documents as trees is more intuitive. Tree patterns are used as schemata in some systems [Arasu and Garcia-Molina, 2003, Rajaraman and Ullman, 2001], although they do not explicitly represent these patterns as tree grammars. The k-testable tree grammar is a kind of tree grammar with weak expressive power, and is used by Kosala [2003].

5.2.2 Inference Algorithm Evaluation

After choosing proper models for the first three aspects, the hard problem is how to devise inference algorithms, and improve the algorithms based on additional information. In this chapter, we shall present our algorithms for schemata detection. Our implementation includes both offline and online algorithms; i.e., offline algorithms need all information representation to be available before executing, while online algorithms allow examples fed on-the-fly. Another important point to be considered in inference problem is how to evaluate whether the algorithms are successful.

There are many evaluation methods in literature. Most existing evaluation approaches are not suitable to schema detection. We briefly introduce some of them.

Identification in the limit Suppose there is a sequence of n information representations R_1 through R_n of a target language, and this sequence is fed to inference algorithm one by one. Based on the given sequence where n may be infinity, an inference algorithm

should make some hypothesis. If there exists a number m, such that after R_m is fed, the algorithms will produce no more new hypothesis and the hypothesis obtained coincides with the target language, we say the target language is *identified in the limit* by this algorithm. The algorithm making a target language to be identified in the limit is successful.

The requirements of identification in limit [Knuutila, 1994] are burdensome for Web data extraction systems, and even the extremely restricted schemata is hard to be detected. When there exists an oracle that can answer whether a given example is positive and whether two grammar are equivalent, regular languages can be identified in the limit [Angluin and Smith, 1983]. However, when only positive examples exist, Gold [1967] prove that it is impossible to identify a target grammar, except the language generated by the grammar is finite.

The result of Gold [1967] is a bit disappointing to induce schemata in the limit. More constructive results show it is possible to identify some languages; e.g., k-testable tree languages [Garcia and Vidal, 1990, Rico-Juan et al., 2000], suitable to be document schemata in the limit based on positive example only. Kosala [2003] models Web documents as k-testable tree languages, and introduces the polynomial inference algorithms.

Another problem of identification in the limit is the size of example set needed, as we cannot assume there are enough sample documents to make the inference algorithm convergent. This motivates an evaluation method of success algorithms based on probability theory.

PAC learning model Identification in the limit asks the algorithms to give a hypothesis equivalent to the target language. PAC-identification [Valiant, 1984] is a learning model based on probability theory, and it looses the success condition of inference algorithms.

PAC-identification assumes examples are fed randomly to the learning algorithm, the success of an algorithm is measured by (1) *error parameter*, and (2) *confidence parameter*. Error parameter is the probability that instances in the hypothesis differ from those in the target language. Confidence is the probability of correct hypothesizes are generated. Given the two parameters, an algorithm is successful if it can produce hypothesis with less error and higher confidence.

Kushmerick [1997] exploits PAC learning model to induce extraction rules, and analyzes how large the example set is needed. Although his rules are quite simple, the

number of samples needed is very large. Sometimes, it is infeasible to prepare such a large training set.

5.2.3 Heuristic Methods for Schema Detection

Based on the evaluation methods above, these exact and probable algorithms mostly are not practical in Web information concierge systems. For example, usually, there are not enough sample documents available, and when there is no enough examples, to induce proper schemata from Web documents may be difficult. For example, both Kosala, Kushmerick's [2003, 1997] algorithm will be too fragile if small set of example is used to inference; i.e., the grammar learnt cannot generate Web documents with noise data.

Fortunately, recent empirical results [Arasu and Garcia-Molina, 2003, Chang and Lui, 2001, Crescenzi et al., 2001, Rajaraman and Ullman, 2001] shown it is possible to construct polynomial heuristic inference algorithms that generate acceptable extraction results. Unlike using grammars as schemata of the whole Web documents that asks documents can be generated from these grammars, some approaches assume there are frequent patterns appearing in documents, and the schemata need not to be conformed by the whole documents. Those patterns are used as schemata, and only those fragments conforming to the patterns should to be extracted. Thus, these algorithms strongly rely on pattern detection techniques.

Compared to polynomial time complexity of these methods, we shall introduce a class of pattern, named k-subtrees, that can be detect in linear time and generate good extraction results.

5.3 k-Schema: A Class of Simple Schemata

In our concierge framework, instance layer contains data extracted and all those Web documents to be extracted, which may be cached locally. Schema layer contains schemata, which are structure patterns of documents in instance layer and some attributes of these patterns. Operation layer does not directly process items in instance layer, it handle items in schema layer to do extraction, knowledge discovery, etc. Although Web documents are changing from time to time, the structures containing interesting data may remain relative static. Thus, to detect those structure patterns matching with interesting fragments of documents as schemata is more stable to handle data in instance layer.

In this framework, Web documents is viewed as rooted labeled trees, which is a 4-tuple $t = \langle V, E, r, \gamma \rangle$. The set V contains the nodes corresponding to tag elements, E is the set of edges connecting these nodes, r is the root node, $\gamma : V \rightarrow L$ is a function that assigns each node a string label, where L is the label set of t. An edge e is an ordered 2-tuple (u, v) where $u, v \subseteq V$ and u is the parent node of node v. Root node r has no parent node. Each non-root node u in V has exact one parent node.

Data instance (DI) in instance layer includes Web documents and fragments of these documents extracted. A fragment of document is a data instance if given a document tree $t = \langle V, E, r, \gamma \rangle$, the fragment can be represented by $t_i = \langle V_1, E_1, r_1, \gamma_1 \rangle$ and $V_1 \subseteq V$, $E_1 \subseteq E$ and $\gamma_1 = \gamma$. This relation is denoted as $t_i \subseteq t$, and t_i is a sub-DI of t_j if $t_i \subseteq t_j$. Two DIs $t_1 = \langle V_1, E_1, r_1, \gamma_1 \rangle$ and $t_2 = \langle V_2, E_2, r_2, \gamma_2 \rangle$ are equivalent, iff exists a bijection $\mathcal{M}$ between V_1 and V_2 such that: (1) $\mathcal{M}(r_1) = r_2$. (2) $(u, v) \in E_1$ if and only if $(\mathcal{M}(u), \mathcal{M}(v)) \in E_2$. (3) $\gamma_1(u) = \gamma_2(\mathcal{M}(u))$.

Like tables in a relational database, which is a set of tuples sharing the same schema, data extracted from Web documents can also be organized as sets corresponding to schemata. In this thesis, a schema also contains DIs, where some nodes are labeled with wildcard char `"*"`, and a DI t matches a schema s, denoted as $t \mapsto s$, if t is equivalent to one DI of s, suppose for any node v, $* = \gamma(v)$ is true. The set of all DIs that match a schema is a type.

In a Web document, the same type of data usually are organized in disjoint subtrees at the bottom. For example, in Amazon Web site, data about each book is represented as a subtree at the bottom of a Web document, and those subtrees of different books are disjointed. Based on this observation, we suppose data to be extracted are those subtrees at the bottom. The set of all subtrees at the bottom of a DI t is denoted as $s = \cup_{j=1}^{n} s_k(t)$, where $s_k(t)$ is the set of subtrees with height k at the bottom. In tree t, whose root have m subtrees from t_1 to t_m, $s_k(t)$ is defined as below:

$$s_k(t) = \cup_{i=1}^{m} s(t_i) \cup \begin{cases} \varnothing & if\ height(t) \neq k \\ t & otherwise \end{cases}$$

The subtrees in $s_k(t)$ is named k-subtrees in document tree t. In the following parts of this chapter, we only consider how to extract those k-subtrees containing user sensitive data. This is an important heuristic that leads to possibility of efficient schema detection, as the detection of all subtrees from multiple documents is a NP-hard problem, as stated in [Li and Ng, 2004]. We also observed that the subtrees containing similar contents in

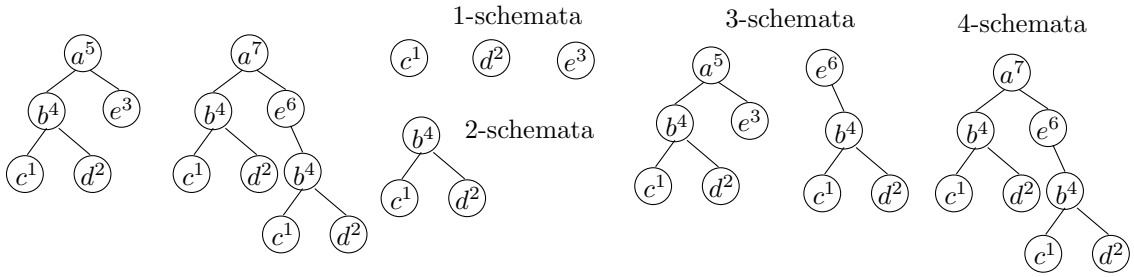


Figure 5.1: Naive Detection

the same Web document trend to share the same structure. Empirical data in [Li and Ng, 2004] confirmed this observation. Thus, it is reasonable to assume that k-subtrees in the same type contain similar contents. As in Web documents, especially HTML documents, data is enclosed in text nodes, we use a structure pattern that is equivalent to subtrees in a type, except that those labels of text nodes are `"*"` as the schema of this type. Such a schema of a type of k-subtrees is named k-schema.

Definition 5.1 (k-schema)**.** A k-schema s is a 2-tuple $\langle D, A\rangle$, where D is a set of k-subtree of Web DIs, and γ functions in these k-subtrees label some text nodes with `"*"`. A is a n-tuple $\langle a_1, a_2, \ldots, a_n\rangle$, where a_i is an attribute, $i \in [1, n]$.

Once the schemata of types of interesting data of Web documents are obtained, a user can easily submit a query to a document t; e.g., a query $M(s) = \{t | t \mapsto s\}$, where s is a k-schema and t is a k-subtree of t. The set of all k-schemata is denoted as $S_k(t)$. Figure 5.1 is a sample of a forest including tree t_1, t_2 and these k-schemata in them. In the following sections, we shall introduce the details of k-schema detection.

5.4 Approximate k-Schema Detection Algorithm

As the problems to detect general schemata, building instance layer and schema layer are intractable, we only consider a special class of DIs in document trees based on an observation:

Observation 5.1. Most interesting contents appear near leaf nodes in many document trees. Text information to be extracted is mainly embedded in text elements, especially for HTML documents.

Based on this observation, we define a class of DIs as follows:

Definition 5.2 (Bottom Data Instance)**.** Given a document hedge T, a bottom data instance (BDI) is a k-subtree of T.

The only difference between a DI and a BDI is that the leaf nodes of a BDI are also the leaf nodes of the document tree containing the BDI. Thus, $S_k(t)$ is the schemata of BDIs in document t. In the following parts of this chapter, we only consider BDI and detection of k-schemata. Given this restriction, we shall introduce an algorithm with polynomial time complexity to detect k-schemata. Before introducing the algorithm details, we provide an alternative definition of *conformation* to make our algorithm easier to understand.

Definition 5.3 (Conformation)**.** Given a data instance $t = \langle V, E, r, \gamma \rangle$, its **type vector** is $\varphi(t) = \langle \gamma(r), o_1, \ldots, o_n \rangle$ where o_1 to o_n are sorted schema ID of n child DIs of r. A DI $t = \langle V, E, r, \gamma \rangle$ conforms to a schema $s = \langle V_s, E_s, r_s, \gamma_s, A_s \rangle$ if $\varphi(t) = \varphi(s)$.

Note that Definition 5.3 is equivalent to Definition 3.5. From the definition, it is possible to traverse document trees from the bottom to top and detect all schemata in one traversal.

Unlike schemata introduced in Chapter 4, this kind of schemata is quite simple and can only perform well when document fragments containing the same kind of information can be parsed to the same structure. Unfortunately, the real situation is more complicated. For example, in Figure 1.1(a), the first and the third books in Page A are rendered from document fragments that can be parsed into the same type of DIs. This is ideal; however, the second data instance of book information does not conform to the same schema as them. Unlike other books, the book titled "*C++ GUI Programming with QT3*" has no second-hand price. There are some other properties of semi-structured documents that make data instances describing the same kind of information different and cannot be put into the same type.

- Missing attributes: Information of a book may include author, price, etc. Sometimes, some books have second-hand price, and others do not have such information.
- Multi-valued attributes: A book may have more than one author.
- Disjunctive delimiters: A document may use different delimiters to mark the same attribute. For example, the titles of hot sales or the special price may appear in bold format.

To resolve the problem that the same kind information is embedded in different types, we relax the condition in Definition 5.3 and place two DIs in the same type if most sub-DIs appearing in them are equivalent.

Definition 5.4 (Approximate Equivalence, Conformation and Type)**.** Given a DI $t = \langle V, E, r, \gamma \rangle$, there are m sub-DIs t_1 to t_m rooted from m child-nodes of root r. The DIs t_1 to t_m conform to schemata s_1 to s_k respectively, and $\{s_1, \ldots, s_k\}$ is the child schema set of t, denoted as $\mathcal{S}_c(t)$. Given a DI $s = \langle V_s, E_s, r_s, \gamma_s \rangle$, suppose $\mathcal{S}_{in} = \mathcal{S}_c(t) \cap \mathcal{S}_c(s)$, and $\mathcal{S}_{un} = \mathcal{S}_c(t) \cup \mathcal{S}_c(s)$, type vectors $\varphi(V)$ and $\varphi(V_s)$ are approximate equivalent, if

$$\sum_{s_i \in \mathcal{S}_{in}} |s_i| > r \times \left(\sum_{s_i \in \mathcal{S}_{un}} |s_i| \right) \tag{5.1}$$

denoted as $\varphi(V) \approx \varphi(V_s)$, where $r \in [0, 1]$. DIs t and s are approximate equivalent if $\varphi(V) \approx \varphi(V_s)$, denoted as $t \approx s$. If s is a schema, we say t approximately conforms to s, denoted as $t \succsim s$. We call a set of DIs that approximately conforms to the same schema as an approximate type.

Based on the preliminary introduction above, we provide a procedure to build the instance layer and schema layer in Algorithm 3. In the schema layer, the schemata detected in each document set are stored in a schema table. Each tuple in a schema table consists of a schema's ID, the schema and its type vector, denoted as $\langle id, s, T \rangle$. In the instance layer, the approximate types detected in each document set are stored in a type table. Corresponding to each approximate type, there is a tuple that consists of the schema ID of this type, its type vector and the list of DIs belonging to this type, denoted as $\langle id, T, D \rangle$. Given a document set, Algorithm 3 first initiates the schema table and type table; then reads nodes from bottom to top, while detecting the schema of the DI rooted from each node. New schemata, DIs and types detected will be appended to the schema table and type table correspondingly. As the schema table and type table of a document set are stored persistently, when the document set is changed; e.g., a new document is added, Algorithm 3 only needs to read nodes from changed parts and modify the schema table and type table. We know that DIs in an approximate type may approximately conform to multiple schemata. Line 13 of Algorithm 3 assigns an *id* to an approximate type; this *id* is the same as the *id* of the schema that is conformed by the first DI inserted to the type. We call this schema the *representative schema* of this approximate type.

In Figure 5.2, we provide a running example of Algorithm *SchemaDetector*. Here, we

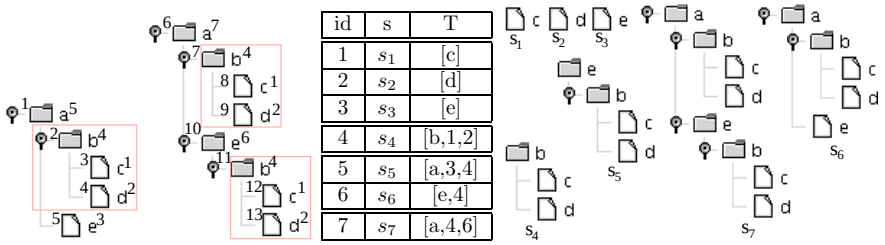
Algorithm 3 LayerBuilder

Require: a document forest F.

initiate a dequeue of nodes $\mathcal{D}$;
read the schema table $\mathcal{S}$ from the schema layer, each tuple is $[id, s, T]$;
read the type table $\mathcal{T}$ from the instance layer, each tuple is $[id, T, D]$;
push all the leaf nodes in F to $\mathcal{D}$;
while $\mathcal{D}$ is not empty **do**
  pop the first node d from $\mathcal{D}$, t=$TreeRootedFrom(d)$;
  search $\mathcal{S}$ for $\mathcal{S}[i]$: $\varphi(t) \approx \mathcal{S}[i].T$;
  if found **then**
    insert t to $\mathcal{T}[i].D$;
  else
    let $id_d = |\mathcal{S}|$, s is t's schema;
    insert $[id_d, s, \varphi(t)]$ to $\mathcal{S}$;
    insert $[id_d, \varphi(t), \{t\}]$ to $\mathcal{T}$;
  end if
  search sibling nodes of d in $\mathcal{D}$;
  if failed **then**
    push the parent node of d to $\mathcal{D}$;
  end if
end while

set r in Equation 5.1 to 1. Figures 5.2(a) and (b) are input DIs where `"a"`, `"b"`, `"c"`, `"d"`, `"e"` are labels of nodes. The number at the left-upper corner of each node is its pre-order traversal position in the input document forest. The number at the right-upper corner of each node is the id of schema to which the DI rooted from the node conforms. The SchemaDetector performs its operation in the following order:

- Detect schemata of leaf nodes `"c"`, `"d"`, `"e"`, delete corresponding nodes from $\mathcal{D}$, and insert the detected schemata into $\mathcal{S}$; i.e., the first three tuples.
- Detect schema of subtree rooted at `"b"` as schemata of all its child nodes have been detected, delete node `"b"` from $\mathcal{D}$, and insert the schema into $\mathcal{S}$; i.e., the fourth tuple.
- Similarly, detect schema of subtrees rooted at `"a"` (with pre-order traversal position 1) and `"e"` respectively, delete the two nodes from $\mathcal{D}$, and insert the detected schemata into $\mathcal{S}$; i.e., the 5th and 6th tuples.
- Detect schema of subtree rooted at `"a"` (with pre-order traversal position 10), delete



id	s	T
1	s_1	[c]
2	s_2	[d]
3	s_3	[e]
4	s_4	[b,1,2]
5	s_5	[a,3,4]
6	s_6	[e,4]
7	s_7	[a,4,6]

(a) DI_1 (b) DI_2 (c) Final Status of $\mathcal{S}$ (d) Schema Detected

Figure 5.2: Running Example

the node from $\mathcal{D}$, and insert the detected schema into $\mathcal{S}$; i.e., the last tuple.

Figure 5.2(c) is the final status of $\mathcal{S}$.

SchemaDetector accesses each node only once. For a set of DIs containing n nodes, it assigns class id to those nodes in n iterations. In the ith iteration, the complexity of all statements except line 7 is $O(1)$. Statement 7 searches in a table; and its complexity is $O(n)$ in the worst case; i.e., all sub-DIs accessed conform to different schema. Thus, the complexity of SchemaDetector is $O(n^2)$ in the worst case. However, in a large document set, the number of all schemata is often much smaller than the number of nodes; thus, the complexity of SchemaDetector is near $O(n)$.

5.5 DocItem Tree for k-Schema Detection

In this section, we shall introduce an algorithm with linear time complexity to detect all k-schemata from a given document hedge. The difference between the linear algorithm and Algorithm 3 is that the linear algorithm only detects those exact schemata with much higher performance, while the Algorithm 3 can detect those approximate schemata. As introduced in Section 5.4, those approximate schemata are more flexible to be used to extract complicated Web documents. A user can choose to apply which algorithm based on compromise between flexibility and performance.

5.5.1 From k-Schema to DocItem Schema

In previous chapters, documents are modeled as labeled hedge, which is intuitive to represent hierarchical structured data. The k-schemata are tree structure patterns embedded

in the hedge. Section 5.4 presented the k-schema detection algorithm, and how to exactly match document fragments with a k-schema. Approximate matching among k-schemata and document fragments can also be executed efficiently. However, to parse documents to a hedge is rather expensive in real implementation.

In this section, we insist on treating Web documents as flat texts; i.e., string languages. We shall introduce an algorithm to detect k-schema without the process to parse documents to hedge. This algorithm detects string patterns that can be parsed to k-subtrees from flat texts. Firstly, we use a subset of XML specification to model Web documents as sequences of symbols from a given alphabet, described as following:

```
CharData = [^<]* - ([^<]* ‘]]>’ [^<]*);
STag = ‘<’ CharData ‘>’;
ETag = ‘</’ CharData ‘>’.
```

$CharData, STag$ and $ETag$ are atoms of documents.

```
Element = STag Content ETag
Content = CharData?  (Element CharData?)*;
DocItem = CharData | Element;
DocItemList = DocItem*;
```

In the above document model, CharData are some strings; those CharData organized by nested tag pairs are elements; each Web document is a DocItem, which is an element or a CharData. Each DocItem in this model can be parsed to a k-tree defined in Section 5.3. If all CharData in a DocItem is replaced with `"*"`, the DocItem can be parsed to a k-schema, so-called the DocItem schema. Hereafter, when we mention a document, it is the document where CharData is replaced with `"*"`.

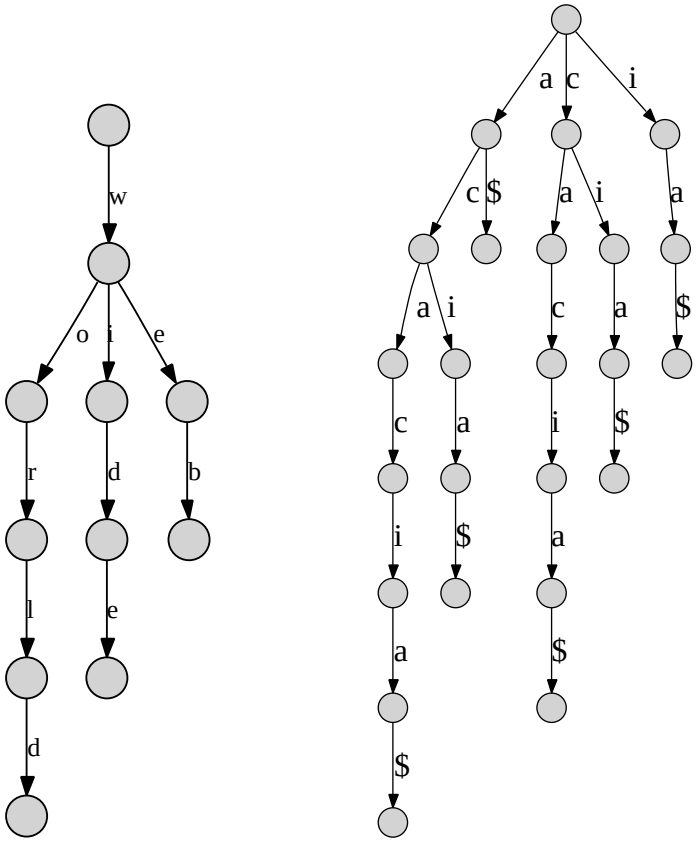
Unlike modeling Web documents as hedge, one benefit of exploiting DocItem to model documents is it is possible to build a DocItem schema repository where each schema can be found in linear time. Inspired by the idea of applying trie as an index of word dictionary, we shall introduce a trie data structure containing DocItem schemata. Search in this structure can be done in time linear in the size of DocItem.

Consider that main workload of Algorithm 3 is brought by searching the schema repository, the complexity of Algorithm 3 can be decreased, if we can search schema more efficiently. The problem is whether the schema repository can be built efficiently, such that, the building process does not increase the whole complexity, while the overhead

of searching is decreased. We shall present the linear time algorithm to detect k-schemata and to build the schema repository.

5.5.2 DocItem Trie Construction

Before introducing the algorithm to detect DocItem schemata and build trie of DocItem schemata, we briefly introduce trie for suffix strings of a CharData.



(a) Trie for {World Wide Web}

(b) Suffix trie of 'acacia$'

Figure 5.3: Exmaple of Trie

A trie is a ranked tree, where each edge is labeled with a symbol from give alphabet, and there exist no two edges from a node have the same label, thus the possible maximum degree of a trie of substrings in a CharData is the size of the alphabet of the CharData. The set $\sigma(T)$ denotes the set of sequences of labels from root to each leaf in trie T; i.e.,

the path labels of leaves. Figure 5.3(a) is a trie T, and $\sigma(T) = \{world, wide, web\}$. We say p is end at l, if a leaf l has path label p.

If $\xi = uvw$ is a CharData, u, v, w are prefix, factor and suffix CharData of ξ respectively. if u, v, w are ElementList, we say they are prefix, factor and suffix elements of ξ. If $\sigma(T) = \{w | w \text{ is a suffix CharData of } \xi\$\}$, T is called the suffix trie of $\xi\$$, where \$ is a symbol does not appear in ξ. Figure 5.3(b) is the suffix trie of `"acacia$"`. The reason to append \$ is to make each suffix appear only one time in ξ. Thus, each suffix can end at a leaf node; otherwise, a suffix may end at an internal node. Usually, suffix trie for $\xi\$$ is easier to be read by human than the one for ξ.

Various linear time suffix trie construction algorithms are devised by McCreight [1976], Ukkonen [1995], Weiner [1973]. With suffix trie of a CharData, any string pattern can be matched with the CharData in time linear in the pattern size, independent of the size of the CharData size. As introduced in Section 5.5.1, the problem now is whether there is an algorithm to construct trie for DocItem schemata in linear time.

Basic Idea of Efficient Trie Construction This section will introduce preliminary notations of DocItem trie, and a naive algorithm to detect DocItem schemata and construct DocItem tries for Web documents. In later sections, based on this naive algorithm, application of some tricks will decrease its computational complexity to linear time.

Generally, a Web document is an element. A n-length document D is a string $d_1 d_2 \dots d_n$. If there is a DocItem start from d_i, d_i is a $SChar$. The atom start from a SChar is a SAtom. For $1 \leqslant i \leqslant n$, if d_i is a $SChar$, D_i denotes the DocItem start from d_i. For $D_i = d_i, \dots, d_j$, d_j is named the $EChar$ of D_i, and the atom ended at d_j is an $EAtom$. If a sequence of atoms $\xi_i, \dots, \xi_j$ is a DocItem, ξ_i and ξ_j are SAtom and EAtom of the DocItem. Given a document D, $\sigma(D) = \{D_i | 1 \leqslant i \leqslant n \text{ and } d_i \text{ is a } SChar\}$

As introduced in Chapter 3, we can parse Web document into trees. Each DocItem in document D can be parsed to a k-subtree in the DOM tree of D. Thus, to detect all k-subtrees from document trees is reduced to detection of of all DocItem from documents. Hereafter, the set of all DocItem in a document hedge T is denoted as $\sigma(T)$.

In the process of parsing a document D to a document tree T, each symbol in D can be mapped to a node in the T. For a symbol d_i in D, $level(d_i)$ denotes the level of d_i in corresponding DOM tree, where root node has level 0, and child nodes' level is equal to 1 plus the level of their parent node. The number of atoms in D is denoted as $length(D)$.

Trie is a data structure that usually used as a dictionary of strings, where a string

can be located in linear time. A DocItem trie of a document T is an augmented trie representing $\sigma(T)$, defined as below:

Definition 5.5. A DocItem trie of document T, denoted as $DTrie(T)$, is a tree that can be defined as a tuple $\langle \Sigma, \mathcal{S}, E, r, \gamma, f, s \rangle$, where Σ is the alphabet of T; $\mathcal{S}$ is a finite set of nodes; E is the set of edges; r is the root node; $\gamma : \mathcal{S} \rightarrow \Sigma*$ is a function that labels each edge with a document atom; $path(x)$ denotes the sequence of edge labels from root to node x; f is transition function $f : \mathcal{S} \times \Sigma* \rightarrow \mathcal{S}$, $f(x, \xi) = y$ if and only if there is a document atom ξ, $path(y) = path(x)\xi$ and $\exists E \in \sigma(T)$ s.t. $path(y)$ is a prefix string of E; s is DocItem link function $s : \mathcal{S} \rightarrow \mathcal{S}$. DocItem link function will be defined in Definition 5.6

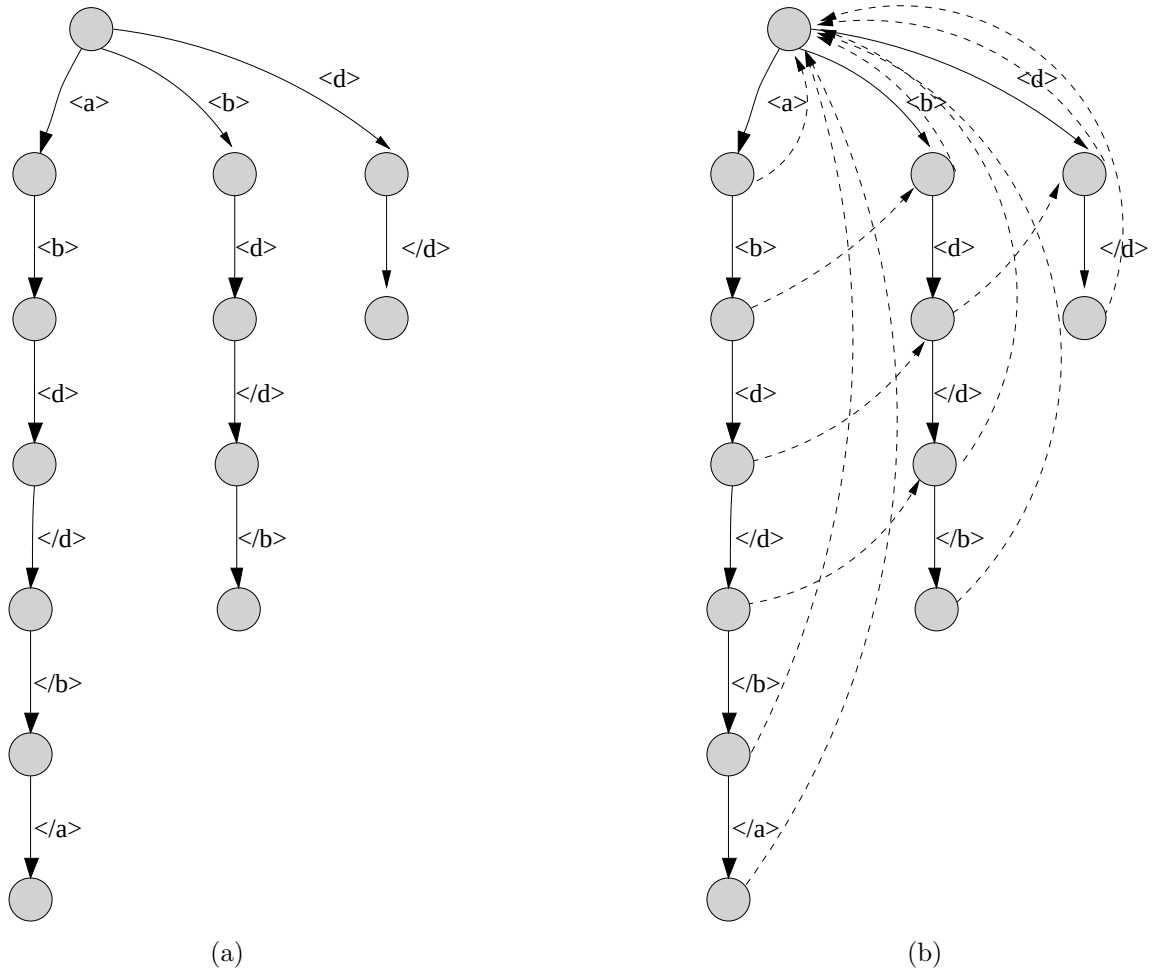


Figure 5.4: Example of DocItem Trie

Figure 5.4(a) is the DocItem trie for document <a><d></d>, without DocItem links.

Ukkonen's well-known algorithm [Gusfield, 1999, Ukkonen, 1995] builds a trie of the set of suffix strings (suffix trie) in linear time to the trie size. Given a string $d_1d_2 \ldots d_n$, Ukkonen's algorithm build suffix tree $\mathcal{T}_i$ (the suffix trie for $d_1d_2 \ldots d_i$) based on $\mathcal{T}_{i-1}$. Given $\mathcal{T}_{i-1}$, suffix strings of $d_1d_2 \ldots d_{i-1}$ are already in the trie. To build suffix trie of $d_1d_2 \ldots d_i$, the algorithm only needs to check whether there is a transition $f(\eta, d_i)$ from the end point η of each suffix string in $\mathcal{T}_{i-1}$; if no then create a node η_i and $f(\eta, d_i) \leftarrow \eta_i$.

Algorithm 4 CreateDTree

Require: A document $D = \xi_1, \ldots, \xi_n$, where ξ_i is an atom for $i \in [1, n]$.
Ensure: $\mathcal{T} = DTree(D)$ created.

```
1: Create T_1
2: for i = 2 to length(D) do {/*phase loop*/}
3:    T_i ← T_{i-1}
4:    for j = 1 to i and k ⩾ i do {/*extension loop*/}
5:       if ξ_j is a SAtom then {/*ξ_k is the EAtom of ξ_j*/}
6:          search node v with path ξ_j, ..ξ_{i-1}
7:          new node u and f(v, ξ_i) ← u if f(v, ξ_i) is not defined
8:       end if
9:    end for
10: end for
```

Differing from Ukkonen's [1995] algorithm, we need to build a trie of DocItem in a document hedge. In despite of the difference, we can share similar idea with Ukkonen's approach. Algorithm 4 depicts the procedure framework of our approach, which does not consider DocItem links yet. The basic idea of Algorithm 4 is to build DocItem trie $\mathcal{T}_i$ of $d_1d_2 \ldots d_i$ from $\mathcal{T}_{i-1}$. All path start from SChar d_j to d_{i-1} should be included in $\mathcal{T}_{i-1}$, thus, given $\mathcal{T}_{i-1}$, Algorithm 4 only needs to check whether there is a transition $f(v, d_i)$, if no, create such a transition.

Using concepts from Gusfield [1999], we call each time of outer loop in Algorithm 4 as a phase, and each time of inner loop as an extension. The Statements 6-7 inside the extension loop will be executed $O(i^2)$ times, and in each extension, Statement 6 locates a path in $O(i)$ time. Statement 7 only needs to add a transition to the trie in constant time. Thus, complexity of the naive algorithm is $O(i^3)$. How to exploit some tricks to improve its performance will be introduced step by step.

DocItem Trie Construction Linear to Trie Size In Algorithm 4, most overheads are brought by Statement 6, thus, it is intuitive to decrease complexity of Statement 6 to improve Algorithm 4. Inspired by Ukkonen's usage of suffix link that guarantee suffix strings can be located in a suffix tried in constant time, we exploit DocItem links to make Statement 6 can be done in constant time.

Definition 5.6. Let $\mu\upsilon\xi$ denote an arbitrary string, where μ is a STag, υ is an DocItemList (maybe empty), ξ is a sequence of document atoms inside a DocItem (maybe empty), a DocItem link is a pointer from the node x with path $\mu\upsilon\xi$ to the node y with path ξ, denoted as $s(x) = y$, where y is root node, if ξ is empty.

Figure 5.4(b) is the complete DocItem trie for document `<a><b><d></d></b></a>`.

To search a path in a trie without DocItem links, Statement 6 in Algorithm 4 need match edge labels form root one by one. With DocItem links, the searching can be done in constant time. In ith phase, the first node added definitely is labeled with $\xi_1, \ldots, \xi_i$, we can store the node with label $\xi_1, \ldots, \xi_{i-1}$ in $(i-1)$th phase to save the time to search $\xi_1, \ldots, \xi_{i-1}$, i.e; Statement 6 can be done in constant time. For the following extensions, Statement 6 can be done in constant time is guaranteed by the lemma below:

Lemma 5.1. In each phase, the DocItem link that starts from the node with path p_i found in an extension points to the node with path p_j to be searched in next extension.

Proof: Suppose current extension needs to search $p_i = \xi_i, \ldots, \xi_k$. Based on the definition of DocItem trie and Algorithm 4, p_i is a prefix of a DocItem, thus $\xi_i, \ldots, \xi_k = \xi_i, \xi_{i+1}, \ldots, \xi_j, \xi_{j+1}, \ldots, \xi_k$, where ξ_i is a STag; $\xi_{i+1}, \ldots, \xi_j$ is a DomItemList and $\xi_{j+1}, \ldots, \xi_k$ is a sequence of document atoms inside a DocItem. Next extension will search $\xi_{j+1}, \ldots, \xi_k$, as j is the only position that fulfills conditions in Statement 4. Based on Definition 5.6, the lemma is hold. ∎

Now, the problem is how to obtain the DocItem link function, and whether there always exists a DocItem link starting from node v located in current extension. To solve this problem, the basic idea is to add a DocItem link pointing to the node created in current extension from the node created in the previous extension. More details about adding DocItem links are depicted in Algorithm 5.

In this algorithm, we need not finish all extensions. If in the jth extension of nth phase, the path $\xi_j, \ldots, \xi_n$ is already in the DocItem trie, the phase can be finished, as based on the definition of DocItem trie, the paths to be added in the following extensions are already in the trie.

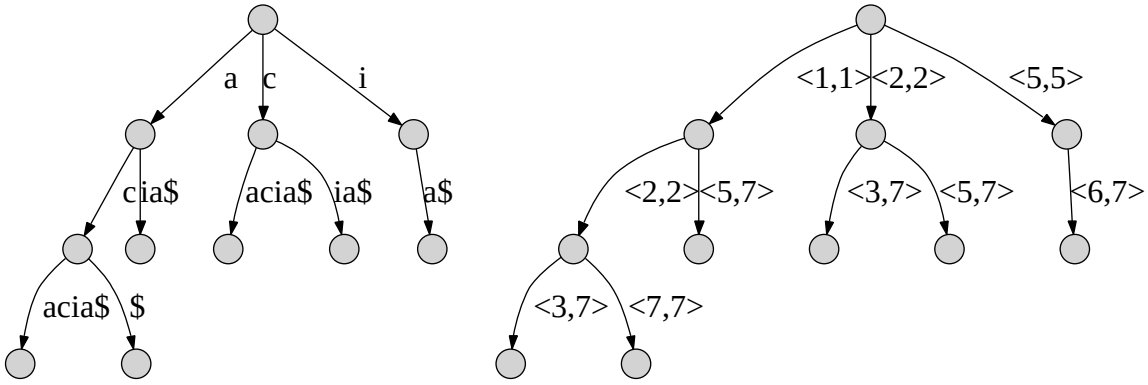


Figure 5.5: Example of Suffix Tree

Based on introduction above, Algorithm 5 gives the complete DocItem trie construction procedure.

Theorem 5.1. Algorithm 5 is linear in the size of the trie.

Proof: As for each node newly created, only constant time needed. ∎

The size of a DocItem trie of a document T is $O(|T|^2)$, as the number of DocItems is $O(T)$, and for each DocItem, there is a path in the DocItem trie, which contains $O(T)$ nodes. Thus, with Theorem 5.1, we can know the complexity of Algorithm 5 is $O(|T|^2)$. In next section, we shall compact DocItem trie to get a data structure, so-called DocItem tree, while a construction algorithm with linear complexity to the size of documents is introduced.

5.5.3 DocItem Tree Construction Linear to Document Size

This section first introduces a data structure of compact DocItem trie — DocItem tree, whose size is linear in the size corresponding documents. Then, some tricks are exploited to further decrease complexity of Algorithm 5.

Compact DocItem Trie to DocItem Tree As the output trie size Algorithm 5 is $O(|T|^2)$, and an algorithm cannot be less complex than its output size, there is no other DocItem trie construction algorithm that can produce lower complexity. Fortunately, it is possible to compact a trie, without losing important information. For example, the trie in Figure 5.3(b) can be compacted to Figure 5.5(a), where a trie path without branch is

replaced with an edge, and the edge label is the concatenation of labels on the trie path. This compacted trie has only $O(|T|)$ nodes, however, its edge labels still has $O(|T|^2)$ size. As each edge label is a substring in document T, it can be replaced by the start position and end position of the substring in T; e.g., Figure 5.5(a) is changed to Figure 5.5(b). As edge labels in Figure 5.5(b) have $O(|T|)$ too, the trie has $O(|T|)$ size. Such a compact DocItem trie is also called DocItem tree hereafter. These nodes left in DocItem tree are called explicit nodes in original trie, and those removed nodes are called implicit nodes.

In a DocItem tree, the transition function should be redefined as $f : \mathcal{S} \times N \times N \to \mathcal{S}$, $f(x, \langle r, l \rangle) = y$ if and only if there is a list of atom $\xi = \xi_r, \ldots, \xi_l$, s.t. $path(y) = path(x)\xi$.

The problem now is what information is lost in the process of compacting a DocItem trie to a DocItem tree. The following lemma states the targets of DocItem links from explicit nodes will not be removed in the compact process.

Lemma 5.2. The target of DocItem links from an explicit node is also an explicit node.

Proof: An explicit v has at least two outgoing edges, thus, $path(v)$ is a prefix of at least two different substrings in the original document T. From Definition 5.6, $path(s(v))$ is a suffix of $path(v)$, thus, $path(s(v))$ is also a prefix of at least two different substrings of T, which means there are at least two outgoing edges from $s(v)$, and $s(v)$ is an explicit node. ∎

Thus, we know only these DocItem links among implicit nodes are lost in the compacting process.

The DocItem Tree Construction Algorithm As a path in a DocItem trie may be collapsed to an edge in the corresponding DocItem tree, Algorithm 5 cannot directly construct a DocItem tree. There are four main different points between constructions of DocItem trie and DocItem tree.

1. In a DocItem trie, $path^{-1} : P \to S$ is a function, and $path^{-1}(p) = v$ if $path(v) = p$; i.e., a path can be identified by a node. As those implicit nodes in DocItem trie are removed, a path cannot be identified by a node in DocItem tree. Insteadly, we can use a node and a position in its outgoing edge to identify those implicit nodes, i.e. $f'(v, \langle r, j \rangle)$ denotes a prefix of DocItem $p = \xi_i, \ldots, \xi_j$. Thus, in a DocItem tree, a path can be identified as $f'(v, \langle r, j \rangle)$.

2. A DocItem prefix in a DocItem tree may not end at a node. For example, in Figure 5.5(a), the end of `"acaci"` is not an explicit node. Thus, in i-th extension

of j-th phase, if $p' = \xi_i, \ldots, \xi_{j-1}$ is located and $p = \xi_i, \ldots, \xi_j$ is not defined in DocItem tree, whether and how to create a new node identifying p depends on some conditions:

(a) If p' is identified by an explicit node v and v is a leaf node, then we only need to replace $f(u, \langle r, j-1\rangle) = v$ with $f(u, \langle r, j\rangle) = v$, instead of creating a new node.

(b) Else, if p' is identified by an explicit node v and v is not a leaf node, then create a leaf node w and let $f(v, \langle j, j\rangle) = w$.

(c) Else, if p' is identified by an implicit node $f'(v, \langle r, j-1\rangle)$, and there is a transition $f(v, \langle o, m\rangle) = u$, then create a new internal node v' and a leaf node w, replace $f(v, \langle o, m\rangle) = u$ with $f(v, \langle r, j-1\rangle) = v'$, let $f(v', \langle j, j\rangle) = w$ and $f(v', \langle (o+j-r), m\rangle) = u$.

Summarizing these differences, Algorithm 6 is the procedure to create new nodes.

3. Like the building process of DocItem trie, if in the i-th extension of the j-th phase, the path $p = \xi_i, \ldots, \xi_j$ is in the DocItem tree already, this phase can be finished. In $(j+1)$-th phase, the i-th extension is the first extension that may need to create a new node.

4. In Algorithm 5, DocItem links are used to help search strings in the following extension in constant time. In the DocItem tree, a path found in current extension may be an implicit node without DocItem link starting from it. To search the node in next extension, we need to exploit DocItem links from explicit nodes. Algorithm 7 gives the pseudocodes of the procedure to locate a node.

Lemma 5.3. Overhead of *LocateNode* is $O(|T|)$ in whole execution process of Algorithm 8.

Proof: In each running, *LocateNode* first traces back to parent node from current node, follows DocItem link to the target node from the parent node, and then searches from the target node in the direction to leaf nodes. In next running, *LocateNode* treats this target node as current node and repeats the process. Thus, we know all the nodes accessed by *LocateNode* cannot be more than the height of the DocItem tree. As the height of the DocItem tree is $O(|T|)$, the execution time of *LocateNode* cannot be larger than $O(|T|)$. ∎

Summarizing these different points, Algorithm 8 is the modified algorithm that builds DocItem tree of a given document.

Theorem 5.2. The complexity of Algorithm 8 is linear in the size of T.

Proof: In Algorithm 8, to create each node in a DocItem tree, each statement will be finished in constant time, except for Statement *LocateNode*. As Lemma 5.3, we know Statement *LocateNode* spends only $O(n)$ in the running process of Algorithm 8. Thus, the complexity of Algorithm 8 is linear in the size of T. ∎

5.6 Summary

This chapter presents enhanced schema detection and extraction component in a comprehensive framework of Web data extraction systems introduced in previous chapters. Algorithms for both schema detection and extraction that are linear in the size the documents to be extracted are demonstrated. Although the theory upper-bound the algorithms is improved from $O(n \log n)$ to $O(n)$ only, the improvement of real implementation is quite significant. This is because the previous work needs to parse documents into trees, unlike we can treat documents as linear symbol sequences. The detected schemata are stored in a data structure like suffix tree, which can acts as a dictionary of schemata. As locating any schemata in this dictionary need only linear time, it is possible to be exploited by other applications more efficiently.

Chapter 7 will present the experiments of these algorithms running on Web documents. Although the experiment results show the performance is quite good, there is a weakness of these algorithms based on suffix trees, i.e., suffix trees can be efficiently built from small datasets, but suffix tree construction based on large datasets is much more difficult due to memory limit [Tata et al., 2004]. The datasets used in this thesis is relative small, compared with huge volume of Web data, thus, it is an interesting direction to improve the suffix tree based algorithms to support on-disk construction.

Algorithm 5 CreateDTree1

Require: A document D
Ensure: $\mathcal{T} = DTree(D)$ created.

Create $\mathcal{T}$ {init}
$r_{init} \leftarrow root$
$\{/*path(r_{init}) = \xi_j, \ldots */\}$
for $i = 2$ to $length(D)$ **do**
  $r \leftarrow r_{init}\{/*path(r) = \xi_j, \ldots */\}$
  while $f(r, \xi_i) = nil$ and $k \geqslant i$ **do** $\{/*\xi_k$ is the EAtom of $\xi_j */\}$
    if ξ_j is a SAtom **then**
      Create r'
      $f(r, \xi_i) \leftarrow r'$
      if $r \neq r_{init}$ **then**
        if $level(\xi_i) \geqslant level(\xi'_i) - 1$ **then** $\{\xi_{i'}$ is the first atom in $path(r')\}$
          $s(oldr') \leftarrow root$
        else
          $s(oldr') \leftarrow r'$
        end if
      end if
      $oldr' \leftarrow r'$ {a virtual node linked from root is added}
      if $r \neq$ root **then**
        $r \leftarrow s(r)$
      end if
    end if
  end while
  if $r \neq$root **then**
    $s(oldr') \leftarrow f(r, \xi_i)$
  else
    $s(oldr') \leftarrow$ root
  end if
  $r_{init} \leftarrow f(r_{init}, \xi_i)$
end for

Algorithm 6 CreateNewNode

Require: node $v \in \mathcal{T}$ identifying path $p = \xi_i, \ldots, \xi_j - 1$
Require: ξ_j
Ensure: *newLeafNodeCreated* and *newIntNodeCreated* are set according to whether there are leaf or internal nodes created
  if $p\xi_{j+1} \in \mathcal{T}$ **then** {$\mathcal{T}$ is the DocItem tree in construction}
    newIntNodeCreated $\leftarrow$ nil;
    newLeafNodeCreated $\leftarrow$ nil;
  else if $v = f(u, \langle r, j-1\rangle)$ is a leaf **then**
    replace $f(u, \langle r, j-1\rangle)$ with $f(u, \langle r, j\rangle)$
    newLeafNodeCreated $\leftarrow$ nil;
  else if $v = f(u, \langle r, j-1\rangle)$ is NOT a leaf **then**
    create node w and $f(v, \langle j, j\rangle) \leftarrow w$
    newLeafNodeCreated $\leftarrow w$;
  else if v is an implicit node **then**
    create node v' and w
    replace $f(v, \langle o, m\rangle) = u$ with $f(v, \langle r, j-1\rangle) = v'$
    $f(v', \langle j, j\rangle) \leftarrow w$ and $f(v', \langle (o + j - r), m\rangle) \leftarrow u$
    newIntNodeCreated $\leftarrow v$; newLeafNodeCreated $\leftarrow w$;
  end if

Algorithm 7 LocateNode

Require: node $v \in \mathcal{T}$
Ensure: node u pointed by DocItem link starting from v in corresponding DocItem trie
  if v is an explicit node **then**
    $u \leftarrow s(v)$
  else {/*$v = f'(v', \langle r, j\rangle)$*/}
    $nodeV \leftarrow s(v')$
    $k \leftarrow j - r$
    while $k > y - x$ **do** {/*$f(nodeV, \langle x, y\rangle)$ is defined and $\xi_x = \xi_r$*/}
      $nodeV \leftarrow f(nodeV)$
      $k \leftarrow k - y + x$
    end while
    $u \leftarrow f'(nodeV, \langle j - k, j\rangle)$
  end if

Algorithm 8 CreateCompactDTree

Require: A document D
Ensure: $\mathcal{T} = CDTree(D)$ created.
  Create $\mathcal{T}_1$
  $r \leftarrow root$
  for $i = 2$ to $length(D)$ **do**
    $oldIntNode \leftarrow nil$
    while $f(r, \langle i, i \rangle) = nil$ and $k \geqslant i$ **do** {/*here, we do not distinguish f and f'*/}
      if ξ_j is a SAtom **then**
        CreateNewNode(r, ξ_i)
        if $newIntNodeCreated \neq nil$ and $oldIntNode \neq nil$ **then**
          if $level(oldIntNode) \geqslant level(\xi'_i) - 1$ **then**
            $s(oldIntNode) \leftarrow root$
          else
            $s(oldIntNode) \leftarrow newIntNodeCreated$
          end if
          $oldIntNode \leftarrow newIntNodeCreated$
        end if
        $r \leftarrow LocateNode(r)$
      end if
    end while
    if $oldIntNode \neq nil$ **then**
      $s(oldIntNode) \leftarrow r$
    end if
  end for

Chapter 6

Auxiliary Operations in Framework

As introduced in Chapter 1 and 3, one objective of our concierge framework is to provide a method of conceptual modeling over Web data and various operations. Chapter 4 gives a closer study of how to model Web documents and extraction operation. Extraction is kernel component of operation layer, however, a practical system may need aid of more auxiliary components, and we shall do initial studying in this Chapter about how to integrate more components into our framework.

6.1 Document Management

Based on operations introduced in the concierge framework, Figure 6.1 presents a typical Web data extraction process consisting of three phases:

- Phase 1: documents in training set are parsed to structural representations such as graphs, trees, etc. A training set is a set of documents sharing similar structures.
- Phase 2: extraction rules are induced from the training set. Extraction rules include

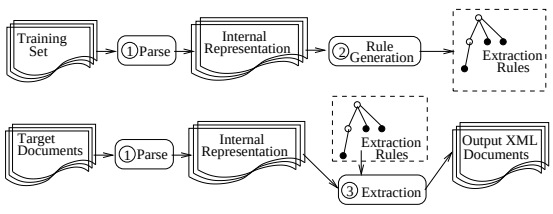


Figure 6.1: Web Data Extraction Process

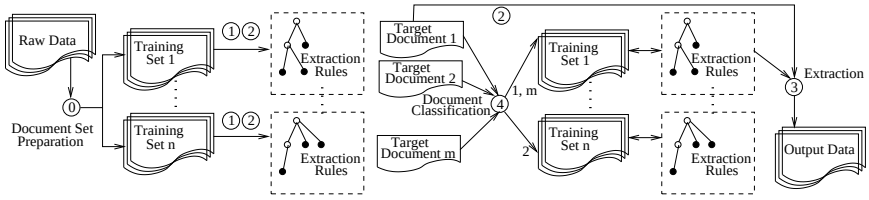


Figure 6.2: Enhanced Web Data Extraction Process

knowledge about structure patterns appearing in these documents.

- Phase 3: given the target set consisting of documents to be extracted, parts in these documents matching with structure patterns are extracted and transformed to structured format. Documents in the target sets are assumed to have similar structures with documents in the training set.

This process suffers from some problems. Firstly, the training set and the target set are assumed to be pre-defined. Given a large set of documents that do not share similar structures, multiple training sets have to be constructed and extraction rules are learnt from these training sets independently. To the best of our knowledge, automatically construction of training sets has not been addressed. Secondly, the learnt extraction rules cannot be applied to the documents that are not from the pre-defined target set. That is, mapping between a document and the corresponding extraction rules cannot be achieved automatically.

This section introduces some operations of document preparation and document classification processes. With the aid of these operations, the Web data extraction process is enhanced and formalized as shown in Figure 6.2. In this enhanced process, phase 0 automatically clusters structural similar documents together to construct multiple training sets. Given a new document, phase 4 finds the most structural similar training set using classification techniques. The extraction rules learnt from the set are then used to extract data from the document.

As introduced in Section 5, schema detection is to induce grammars of a training document set. Documents in a training set should share large number of common structures. Usually, the training set is using manual methods, which are time-consuming. This section introduces how to automatically prepare training sets by clustering documents that are similar in terms of embedded schemata. We also introduce document classification operation that classifies a document to a training set. This operation can help to de-

DI_1=[1 1 1 1 0 1 0]
DI_2=[1 1 1 1 1 0 1]
(a)Documents

$$M = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}$$
(b)Document Space

Figure 6.3: Vector Space of Document

cide which extraction rules should be applied to a document automatically. These two operations significantly improve efficiency of Web data extraction. In this section, we first introduce how to measure structural similarities among documents, followed by the details of these two operations.

6.1.1 Document Similarity Measurement

To measure similarity among documents, we first represent them as vectors of schema weights and compute document similarities by computing the similarity among these vectors. Corresponding to each schema embedded in a DI, there is an item in the vector recording the weight of the schema. Suppose the weight of a schema is 1 if the schema appears in a document tree, and 0 if it does not appear; the vectors of DI_1 and DI_2 in Figure 5.2 are shown in Figure 6.3(a). So far, we can represent all documents in a vector space. For example, DI_1 and DI_2 can be represented using the matrix in Figure 6.3(b).

Given two documents t_1 and t_2, suppose there are n representative schemata, we represent $t_1 = \langle \omega_1(s_1), \ldots, \omega_1(s_n) \rangle$ and $t_2 = \langle \omega_2(s_1), \ldots, \omega_2(s_n) \rangle$, where $\omega_i(s_j)$ is the weight of s_j in t_i. The similarity between t_1 and t_2 can be computed using Equation 6.1.

$$\Omega_C(t_1, t_2) = \frac{\sum_{i=1}^{k} (\omega_1(s_i) \times \omega_2(s_i))}{\sqrt{\sum_{i=1}^{k} \omega_1(s_i)^2 \times \sum_{i=1}^{k} \omega_2(s_i)^2}} \tag{6.1}$$

6.1.2 Document Set Preparation and Document Classification

Given the structural similarity among documents, we now introduce how to solve problems in the phases of document set preparation and document classification.

Document Set Preparation We prepare training set by clustering similar documents together. In the similarity calculation phase, we can obtain similarity square matrix $\mathbb{M}_{m \times m}$, where m is the number of document trees and $\mathbb{M}_{i,j}$ is the similarity between document t_i and document t_j. Given the similarity matrix, we choose bisecting k-means

clustering algorithm that clusters documents into k clusters (k is predefined by a user), from $\mathbb{C}_1$ to $\mathbb{C}_k$, such that the maximum value of the following formula is obtained:

$$\sum_{i=1}^{k} \sqrt{\sum_{t_1,t_2 \in \mathbb{C}_i} \mathbb{M}(t_1, t_2)} \tag{6.2}$$

Document Classification As introduced before, we may detect schema attributes in terms of document sets, and extract data based on those attributes. Given a new document, it is an issue on how to exploit the known attributes to extract data. We propose to classify the new document to an existing training set and apply extraction rules learnt from the set on this document. We referred to XRules [Zaki and Aggarwal, 2003] for some initial work of documents classification based on tree structure of documents. XRules mines rules like $T \Rightarrow \mathbb{C}$, which describe the relationship between the appearance of a tree structure t in a document and the document belonging to class $\mathbb{C}$. During the classification phase, XRules combines the evidence of structures appearing in a document and suggests a class to the document. However, XRules does not consider the occurrence frequency of structure t in the documents to be classified. Here, we suggest a more general approach to classify document trees. From the results of the document set preparation phase, we use a vector $\mathbb{W}_{\mathbb{C}} = \langle \omega_{\mathbb{C}}(s_1), \ldots, \omega_{\mathbb{C}}(s_n) \rangle$ to represent a training set where $\omega_{\mathbb{C}}(s_i)$ is the weight of a representative schema s_i in training set $\mathbb{C}$. Previously, we have introduced how to represent a document tree using a weighted vector. Assume that $\mathbb{W}_t = \langle \omega_t(s_1), \ldots, \omega_t(s_n) \rangle$ is the weight vector of document tree t, the similarity between a document tree and a document set is:

$$\Omega_S(t, \mathbb{C}) = \frac{\sum_{i=1}^{k} (\omega_t(s_i) \times \omega_{\mathbb{C}}(s_i))}{\sqrt{\sum_{i=1}^{k} \omega_t(s_i)^2 \times \sum_{i=1}^{k} \omega_{\mathbb{C}}(s_i)^2}} \tag{6.3}$$

We may classify a document tree t to the training set that is most similar if the similarity value is larger than a threshold; otherwise, t should be treated as an outlier.

6.2 WDEE: Web Data Extraction by Example

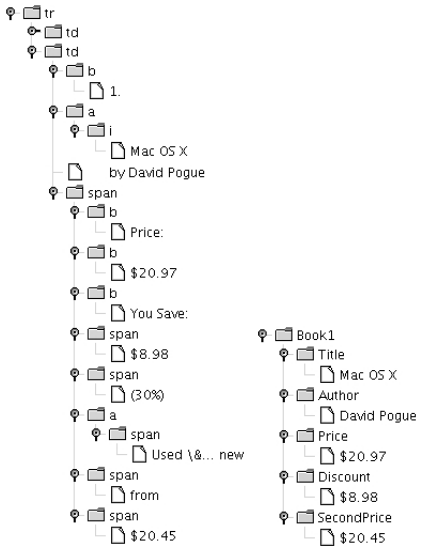
Web data extraction systems are the kernel components of concierge systems between users and Web data resources. In the beginning, the main purpose of Web data extraction systems is to transform semi-structured Web documents to relational databases [Gupta et al.,

1997] or object-oriented databases [May et al., 1999]. These systems need users to code by hand using formal languages to extract data from Web documents. We also define Wdel as extraction language in Chapter 4.

Those formal extraction languages provide firm basis for Web data extraction. However, manually programming is always time-consuming and error-prone. The appearance of automatic extraction rule generation techniques partially solves this problem. Kushmerick [Kushmerick, 2000a] presents the method using grammar induction techniques to learn extraction rules. His method assumes documents to be extracted are organized like relational tables. IEPAD [Chang and Lui, 2001], RoadRunner [Crescenzi et al., 2001] use various approaches to learn regular expressions as extraction rules. Those methods treat documents as flat text. Thus, it is easy to lost structural information. ExAlg [Arasu and Garcia-Molina, 2003] and Skeleton [Rajaraman and Ullman, 2001] can detect tree patterns, which are more intuitive and accurate. Wiccap [Liu et al., 2002b] and Lixto [Baumgartner et al., 2001b] proposed supervised learning approaches based on visual interface to generate extraction program. Chapter 5 introduced our heuristic algorithms to help automatic extraction rule generation.

Although extraction efficiency can be improved obviously using these methods, extracted data are still not easy to be accessed. To exploit extracted data, a user needs to program instead of browse the easy-to-read original document because of two reasons: (1) It is difficult to reverse visual effect from patterns detected by them; (2) Lack of management of documents, schemata and extracted data. Wang and Liu [2000] move the first step to use structured patterns in semi-structure documents to manage them. In recent years, more detailed structure-based document management techniques are devised. XRules [Wang and Liu, 2000] can classify semi-structured documents based on embedded subtrees very well. There are some papers discussing how to measure similarity among XML documents [Flesca et al., 2002, Nierman and Jagadish, 2002]. It is easy to cluster documents based on their similarity. These approaches provide new ideas of semi-structured document management. However, the structure patterns considered by them are not easy to be rendered in Web browsers and generate examples like those in our system. Moreover, there is still no literature addressing how to exploit those techniques to help Web data extraction.

In this section, we shall introduce a method, so-called Web data extraction by example (Wdee) based on our concierge framework, which provides mapping relationships among data instances and schemata. As an example, the first book in page A in Figure 1.1(a)



(a) Codes of Book1 (b) Extracted Data

Figure 6.4: Web Page and Logic View over Web Page

is rendered from the HTML codes in Figure 6.4(a). The corresponding data instances extracted is drawn in Figure 6.4(b). The concierge framework can maintain the mapping relationship between these two data instances. Similarly, the mapping among data instances, Web document schemata and extracted data schemata can also be maintained.

To illustrate the usage of these relationships, we first consider how to generate a query to instance layer without these relationships. As introduced before, a schema describes a type of DIs. A user may write a simple query like "select * from *schemaA*" returns all DIs matching with *schemaA*. Thus, a schema itself can be treated as a query that will return all data instances that match it. However, this kind of queries need a user to handle schemata directly.

These mapping relationships are to be used to generate queries to instance layer in a manner that resembles the browsing of Web pages. Inspired by query languages like QBE [Zloof, 1977] and QEByE [da Silva et al., 2002], Wdee is devised. As there exist mapping among Web documents, document schemata and data instances extracted, when a user wants to query some instances, Wdee will generated visual examples of Web documents corresponding to these instances. By handling these visual examples, queries are generated to handle these corresponding schemata.

Unlike QBE and QEByE that prepares examples w.r.t. the relational query and the nested table query [da Silva et al., 2002] respectively, schema defined here allows us to

generate hierarchical examples.

6.2.1 Query Operations by Schema Matching

In this section, we introduce the query language of Wdee. The Wdee language consists of a set of schemata enhanced based on Definition 3.4, that has less expressive expressive capability of Wdel introduced in Chapter 4, and may be used to represent *selection* and *projection* queries. Wdee language are augmented with set oriented features: union, join and difference.

We begin by defining matching query, followed by its extensions:

Definition 6.1 (Matching Query)**.** Given an extracted document T, a matching query $M(s) = \{t | t \mapsto s\}$, where s is a schema and $\eta(t)$ is a DI of T.

Based on Definition 3.4, DIs that have the same structure as s will be returned by the matching query. This query is a bit naive and we draw ideas from Bergholz [2000] to improve the expressive capability of the query by augment schema with predicates, as defined below:

Definition 6.2 (Schema and Matching)**.** A schema s is a 4-tuple $\langle V, E, r, \gamma \rangle$, where $\langle V, E, r, \gamma \rangle$ is a DI and γ labels each node with a predicate formula $P(x)$. A DI $t = \langle V_t, E_t, r_t, \gamma_t \rangle$ match with s, denoted as $t \mapsto s$, if $\langle V, E, r \rangle = \langle V_t, E_t, r_t \rangle$ and $P(v)$ is true for $v \in V$.

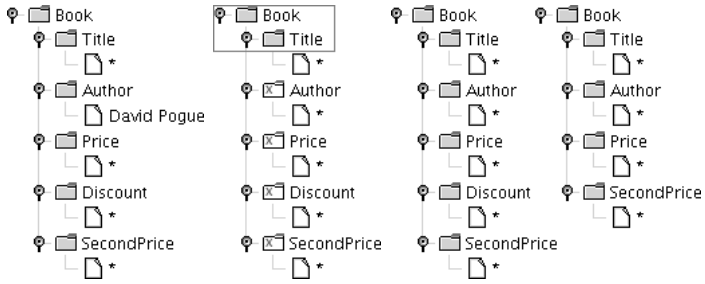
The predicates supported by Wdee are three types:

$x = c$ where x is the label of corresponding node in a DI, c is a constant string; e.g., `"David Pogue"`.

$e \vdash x$ where e is a regular expressive and e can generate x. We restrict that predicates of this type appear on only leaf nodes, such that Definition 6.2 is equivalent to Definition 3.4 when e is `"*"`, where `"*"` is a wildcard generating any string label.

$x = X$ where x is the label of corresponding node in a DI and X is a variable name. This predicate is important to construct join operation, as introduce in later part of this section.

In the following part of this section, we present how to use a matching query to represent those query operations: *selection*, *project*, *union*, *difference* and *join*.



(a) Sel (b) Proj (c) Union

Figure 6.5: Examples for Query Operations

Selection (σ) Selection is an unary operation on a schema, denoted as $\sigma_{condition}(s)$. It returns all DIs that match the schema and fulfill the given condition.

Suppose *Book* is a schema of the DI in Figure 6.4(c), $\sigma_{author=\text{“}DavidPogue\text{”}}(Book)$ is a select operation that returns all data of books in Figure 6.4(a) with author “David Pogue”. To represent this operation, we use a matching query with the schema in Figure 6.5(a). The constant string label c in the figure is the short form of $x = c$ and a regular expression label e is the short form of $e \vdash x$. This schema matches with the first DI corresponding to the first book in Figure 6.4(a). Thus, the matching query will return the first DIs.

Projection (π) Projection is also an unary query operation on a schema, denoted as $\pi_{\langle fieldlist \rangle}(s)$.

The $\langle fieldlist \rangle$ is a list of field with syntax of *field*:=*path*‘.’*schema*, *path*:=*item*‘.’. . .‘.’*item*, *item*=*regexp*|*regexp*‘[’*idx*‘]’, where *regexp* is a regular expression over node labels, *idx* is a natural number and $l[i]$ means the i-th child nodes with label l of a node.

Definition 6.3. A schema $s_p = \langle V_p, E_p, r_p, \gamma_p \rangle$ is a projection schema of $s = \langle V, E, r, \gamma \rangle$, iff $\eta(s_p) = s$; i.e., s_p maps to s. A projection $\pi_{\langle path_1.s_1,\ldots,path_n.s_n \rangle}(s)$ return a set of DIs $\{t | t \mapsto s_p\}$, where $e_p = \langle v_i, v_j \rangle \in E_p$ iff $\langle v_m, v_n \rangle \in E$, or v_m and v_n is connected by $path_i$ in E, v_m is root of s and the tree rooted from v_n match s_i.

Suppose the schema for the book DI in Figure 6.4(c) is s_B and the schema for book title is s_T, a projection $\pi_{\langle Book.s_T \rangle}(s_B)$ will return DIs containing only book’s titles. The schema in gray box of Figure 6.5 represents this query.

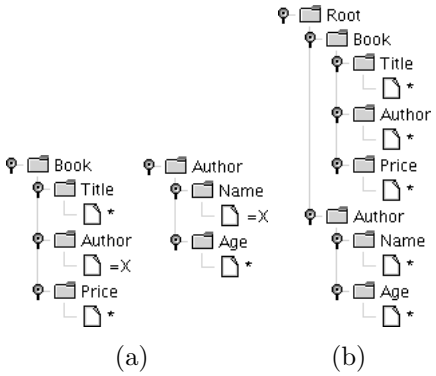


Figure 6.6: Examples for Join Operations

The selection and projection operations are both matching queries. Wdee distinguishes them by the schemata to be processed. A selection or a projection operation returns a set of DIs matching with the given schema. We define three set-oriented operations below:

Union ($\cup$) Union is a binary query operation on two schema, denoted as $s_1 \cup s_2$.

A union operation merges two DI sets returned by $M(s_1)$ and $M(s_2)$. It may be represented as shown in Figure 6.5(c) and consists of two schemata corresponding to two types of DIs about book information. If s_1 and s_2 match with each other, the two sets are *union compatible*. If two DI sets are union compatible sets of DIs, the union operation needs to detect overlay DIs to merge them, otherwise, the union operation simply copy two sets DIs together.

Difference ($-$) Difference is a binary query operation, denoted as $s_1 - s_2$.

A difference operation ask s_1 and s_2 match with each other; it simply removes overlay DIs from the DI set returned by $M(s_1)$. The difference operation and join operation introduced below may both be represented using schemata like those in Figure 6.5(c). As the three set operations cannot be distinguished only based on the schemata, a user needs to explicitly choose the type of operation.

Join ($\bowtie_{condition}$) Join is a binary operation, denoted as $s_1 \bowtie s_2$.

To join two DI sets, s_1 and s_2 first need to be combined to s. To combine the two schemata or DIs, Wdee simply adds a root node as the parent node of both root

nodes of them. For example, the two schemata in Figure 6.6(a) will be transformed to Figure 6.6(b). Wdee allows the join condition of two text nodes of two DIs share the same label. As an example in Figure 6.6(a), to join book DIs and author DIs sharing the same author name, a user may change the label as Figure 6.6(b), where $=X$ is the short form of $x{=}X$.

We have shown that with different schemata, we may execute various query operations on extracted documents. By parsing the schemata of extracted DIs to trees like those in Figure 6.5, a user can easily modify these trees and generate schemata to query extracted documents.

6.2.2 Wdee Running Example

In this section, we present how to build examples corresponding to Wdee query operations. An example in Wdee consists of two parts: *tree skeletons* and a *browser view*. A tree skeleton is a visualized schema. To submit a query to an extracted document, a user may give some condition inside the skeleton, while the browser view presents a sample of Web documents where the document is extracted from.

The principle merit of Wdee language is that a Wdee program may be generated within a specific Web browser. When a user chooses to query a certain document extracted, a corresponding Web document will be rendered in the Web browser. Given the Web page shown, a user may generate a Wdee program by selecting texts that are interesting to him.

Concisely, to generate a Wdee program by visual interface consists of three phases: (1) Select schemata extracted documents to be queried; obtain a sample of the original Web documents to which the schemata map and generate a document tree whose text nodes can be modified by a user, so-called a tree skeleton. The Web page rendered from the sample Web document and the skeleton together is called an example. (2) Generate a basic Wdee program by selecting texts in the Web browser. (3) Input conditions in text nodes of tree skeleton to refine the Wdee program.

As an instance, we present how to generate a Wdee program for an example query, as described below:

QueryBook Return book title, price and discount of the books written by `"David Pogue"`.

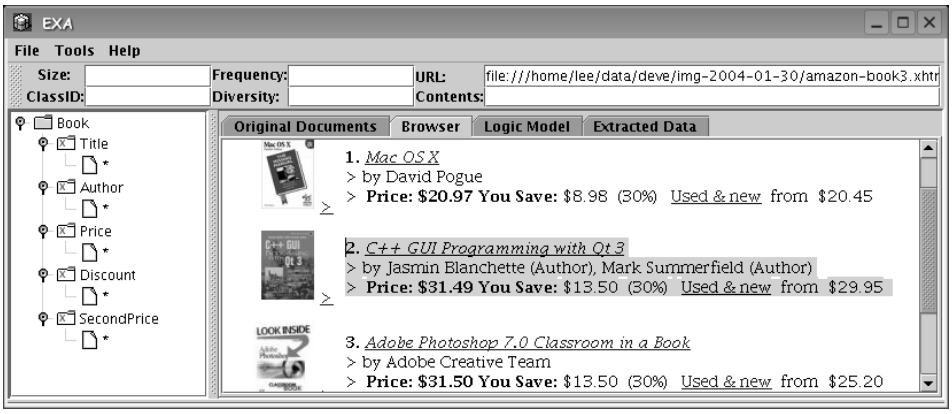


Figure 6.7: Wdee example

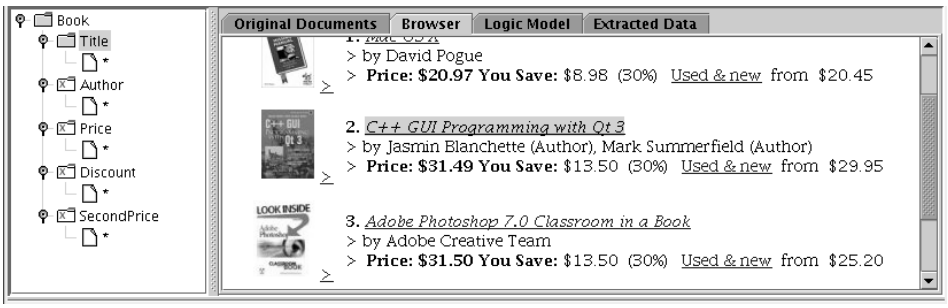


Figure 6.8: Wdee example

The detailed operations follow the 3-phase Wdee program generation procedure, as described below:

1. A user may select a schema using the menu. Suppose the schema *book* is selected, a tree skeleton of this schema will be shown, as the one in the left panel of Figure 6.7. A schema of Web documents $\eta(book)$ will be obtained by search the mapping relation. In turn, a random original Web document containing this Web schema is selected and shown in the Web browser, as shown in the right panel of Figure 6.7. The two panels is an example.

2. The Web page contains some books' information. A user may denote which parts should be returned, by selecting texts in the page. Here, book title, price and discount are data to be returned. Initially, all nodes in the tree skeleton is unselected (denoted as white folder icons). After a user selects the book title by a mouse in

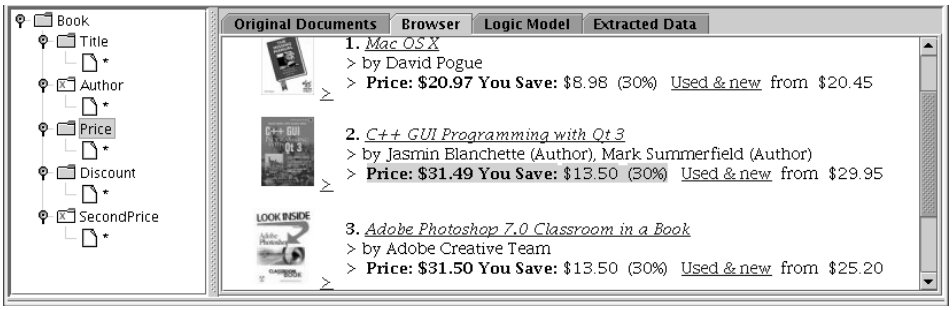


Figure 6.9: Wdee example

the Web browser, as Figure 6.8, corresponding parts in the tree skeleton is selected (those gray folder icon). If the text selection covers more than one field in the give schema, as Figure 6.9, multiple nodes in the tree skeleton will be selected. So far, the schema corresponding to the skeleton can be used to generate a projection query.

3. After a user selects all texts that are interesting, he can give condition that must be fullfiled by the returned DIs. Here, a user may input `"David Pogue"` in the tree skeleton, like the one in Figure 6.5(a). The schema corresponding to the modified skeleton can be used to generate a selection query.

The skeletons generated in the above steps may be directly saved as a Wdee query program. However, only a subset of the Wdee query language can be generated using the visual interfaces. For example, in Step 2, the fields generated contain only strings of labels and will not contain regular expression; i.e., the syntax of *item* of $field$ defined in Section 6.2.1 will be changed to $item{:=}labelstring|labelstring`['idx`]'$. To generate Wdee program with full features, a user may modify or exploit some techniques [Chang and Lui, 2001] to post-process generated Wdee query program.

6.2.3 Expressive Capability and Evaluation Complexity

The objective of Wdee is not to provide a general XML query language like XPath or XQuery. Wdee's expressiveness of selection and projection operations is strictly less than XPath. However, by introducing the concept of schema into Wdee, it is easier for a user to generate query program quickly in the majority situations of the Web. Beside the efficiency of query generation, how to improve the efficiency of query execution is

another consideration of Wdee.

The evaluation of a Wdee query is a computational process that interprets Wdee query and returns special parts in Web documents. To evaluate a Wdee query, we can translate Wdee selection and projection operations to location paths of XPath in linear time. As shown in [Gottlob et al., 2003], location paths of XPath can be evaluated in polynomial time in the size of query. Thus, the upper bound to evaluate Wdee selection and projection query evaluation is at most polynomial time. As introduced in Section 6.2.1, the union and difference query both can be evaluated in linear time. Join query can be evaluated within $O(|S_1| \times |S_2|)$, where $|S_1|$ and $|S_2|$ are size of two DI sets to be joined. As the size of a DI set is less than node number, Wdee queries can be evaluated in polynomial time.

The limited form of Wdee generated by the visual interface can be translated to Monadic Datalog over the signature introduced in [Gottlob and Koch, 2002a]: $\tau_r = \langle root, leaf, (child_k)_{k \leqslant K}, (label_a)_{a \in \sum} \rangle$ in linear time. Thus, the evaluation complexity of a limited form Wdee query is $O(|D| \times |Q|)$, where D is the size of documents to be queried and Q is the size of query.

Chapter 7

Experiments

In this chapter, we shall present and analyze the experiment results. As introduced in previous chapters, the objective of our work is to produce a comprehensive Web information concierge system. We conducted experiments to study whether the techniques we discussed can help.

Based on our concierge framework, we introduced a sample operation procedure in Figure 6.2. Similar to most Web data extraction procedure in literature, the key processes in the procedure are to detect schema and to extract data. Compared with work in literature, Chapter 6 also indicates the procedure is more comprehensive, in term of auxiliary processes, e.g., automatic training set preparation and extraction rule assignment to Web documents to be extracted.

The techniques discussed in previous chapters should help these processes, and the experiments are designed to test how much they can help. Here, the experiments basically contain four parts: (1) performance of schema detection and extraction, (2) accuracy of extraction rules generated using introduced inference algorithms, (3) accuracy of document clustering based on schema information, (4) accuracy of document classification based on schema information. We also do some initial study about distribution of schemata attributions. For each part of the experiments, our techniques are also compared with other famous systems.

7.1 Performance of Schema Detection

This sectioin demonstrates the experiment that compares the performance of Algorithm 3 and those of two recent tree structure miners — TreeFinder and TreeMiner. The objec-

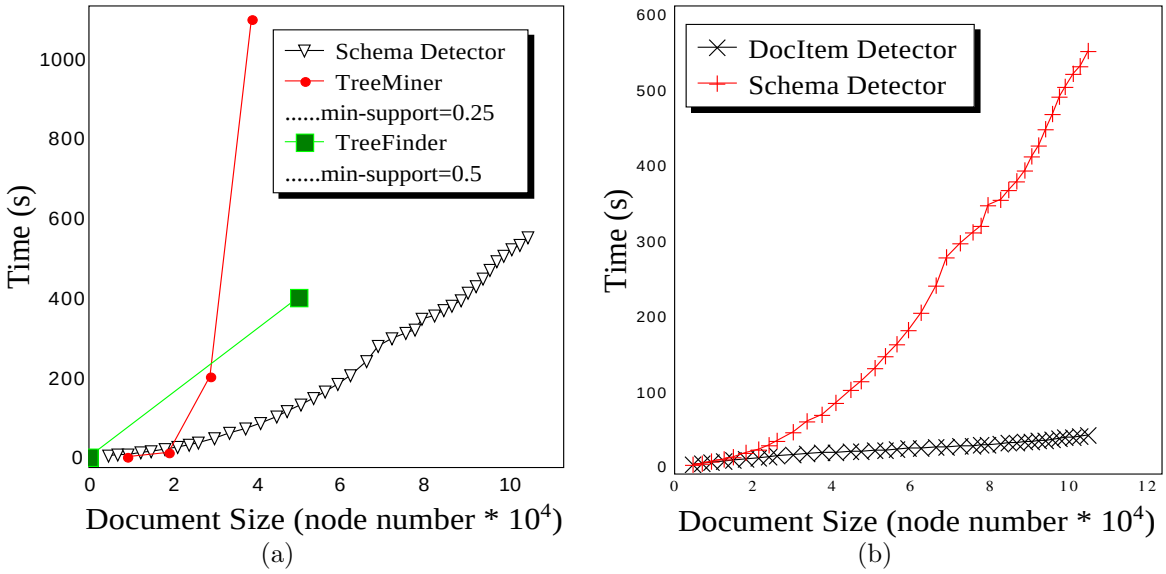


Figure 7.1: Performance Comparison

tive of Algorithm 3 is to detect frequent sub-structure from document trees, and this objective is similar with that of structural mining. The reason to choose them is because structural mining is a relative new research area, and TreeFinder [Termier et al., 2002] and TreeMiner [Zaki and Aggarwal, 2003] are two important methods in this area.

For Web data extraction, it is important to process large-scale documents quickly. For example, if users want to trace price changes in an e-commerce Web site, long refresh period is not accepted. The design of Algorithm 3 is to obtain low time complexity, and the comparsion with other methods focuses on their performance.

The data sets used by TreeFinder and TreeMiner is not open for public use, thus, we try to generate similar data sets. As the two structural miner use XML documents with small size, we generated the **ETC** (Equivalent Tree Class) data set that is larger and more complicated than those of TreeFinder and TreeMiner. Web pages in ETC data set were crawled from Amazon. All these downloaded pages were transformed to the XHTML format using HTMLTidy toolkit. The average size of these documents is 550K bytes, and the number of nodes in the document trees is about 110 thousands. ETC data set is also used in following experiments, and will be introduced in more detail later.

We evaluated the complexity of SchemaDetector on ETC dataset. We conducted experiment on a PIII933 laptop with 512M memory, which is similar with experiment platform of TreeMiner. In Figure 7.1, we draw the complexity of these three algorithms. The

results of TreeFinder and TreeMiner were collected from related papers [Termier et al., 2002, Zaki and Aggarwal, 2003]. The document size of TreeMiner is measured by the string length of documents, not measured by the number of nodes. We plot the point of TreeMiner there, because in ETC data set, the string length of documents containing 50K nodes is about 1M, which is approximately equal to the size of the data set used by TreeMiner. TreeMiner and TreeFinder will discard structures that appear fewer times than a minimum support value, when the value decreases the complexity of them increases rapidly. It is easy to see from Figure 7.1(a) that our algorithm outperformed them when the minimum support is set to small value was TreeFinder (5%) and TreeMiner (0.25%). We have analyzed that when the number of schemata is much smaller than the number of subtrees, the complexity of SchemaDetector is near linear; otherwise, its worst complexity is $O(n^2)$. From the diagram, we can see the empirical results verify that. When the data set is small, the number of schemata is large compared with the number of DIs; thus, the complexity grows quickly. With the increase in size of the data set, the complexity is near linear, as the number schemata is smaller compared with the number of DIs.

Performance of DocItem detector is also compared with SchemaDetector's. Figure 7.1(b) shows the running time of both algorithms on the ETC data set. The complexity of DocItem detector is linear to the size of documents. The experiment result coincides with our complexity analysis in Chapter 5. DocItem detector largely improves the performance of schema detection, compared with SchemaDetector, and is a big step to detect schemata efficiently. As DocItem detector can also be used as extraction engine, Figure 7.1(b) can represent the performance of Web data extraction.

7.2 Clustering Accuracy

Given the similarity between each pair of documents, we choose bisecting K-means method to cluster them. Clustering results based on structural information are compared with results from CLUTO (A famous software package for clustering high-dimensional datasets) [Karypis, 2002, Steinbach et al., 2000], which also uses bisecting k-means method, and is very suitable to cluster free-text documents, according to experiment data comparing with other methods [Steinbach et al., 2000]. The difference is CLUTO does not consider the structural information in documents.

The comparison between our method and CLUTO focuses on their accuracy. The

Name	URL
IEPAD	http://140.115.156.70/iepad/
RoadRunner	http://www.dia.uniroma3.it/db/roadRunner/index.html
RISE	http://www.isi.edu/info-agents/RISE/
WIEN	http://www.cs.ucd.ie/staff/nick/home/research/wrappers/
ExAlg	http://www-db.stanford.edu/~arvind/extract/

Table 7.1: Data Sets

DataSet	E1	E2	W	R1	R2	Average
Our Method(%)	90	100	100	100	89.6	95.7
CLUTO(%)	47.5	100	72.5	66.7	70.8	71.2

Table 7.2: Clustering Accuracy

input to these methods is a set of documents, where each document is assigned a class ID, which is unkown for the methods; and the ouput is some subsets of the document set. The accuracy of clutersing results is the rate of the number of documents with the same document ID clustered into the same subset to the number of all documents.

We use the datasets taken from previous literature listed in Table 7.1. With these datasets, we can easily assign class ID to documents based on their original datasets. To evaluate clustering methods, we mix documents from various categories to generate **MIX** dataset. For RoadRunner, we randomly choose 4 documents from each of its 12 categories. We randomly choose 5 documents from each of the 6 categories in RISE (Repository of Online Information Sources Used in Information Extraction Tasks), 4 documents from each of the 10 categories in WIEN. For ExAlg, they provided 3 categories, 132 documents except some documents from the RoadRunner. We choose 50 documents from 3 categories of ExAlg.

Table 7.2 lists comparison results obtained on MIX dataset including documents selected from several datasets. Row 1 indicates these datasets: E1 (ETC), E2 (ExAlg), W (WIEN), R1 (RISE), R2 (RoadRunner). CLUTO also uses bisecting K-means method but only considers texts in documents. CLUTO is very suitable to cluster free-text documents [Steinbach et al., 2000]. From this table, we can see that except for documents from ExAlg, our method was much better than CLUTO. The reason is that CLUTO only performed well when documents contain long free-text paragraphs; e.g., documents in ExAlg. However, our method delivered good results consistently.

DataSet	E1	E2	W	R1	R2	Average
Our Method(%)	97.5	100	100	100	100	99.5
Rainbow(%)	77.08	86.78	89.58	89.58	87.66	86.14

Table 7.3: Classification Accuracy

7.3 Classification Accuracy

We used the same dataset introduced in Section 7.2, and we also generated the **ETC** (Equivalent Tree Class) dataset that is more complicated than above datasets. Web pages in ETC dataset were crawled from Amazon. These pages have been classified by Amazon into categories, thus we can easily verify clustering and classification results. To generate ETC dataset, we only follow hyperlinks in the category list on the left side of the homepage of Amazon. Each hyperlink links to a homepage of a category of commodities. We randomly choose 10 hyperlinks in this area, and in the homepage of each category, we downloaded 3 to 4 pages following hyperlinks in the navigation area.

We exploited a 4-fold cross validation strategy to evaluate our classification method; i.e., divide each dataset to 4 parts, each time we choose 3 of them as training sets and one as test set. After training with training sets, the method try to assign each document in test set to a training set. The accuracy of each time is the rate of the number of documents classified to the right set to the number of all documents. The final result is the average accuracy of results in four times. Our method is compared with Rainbow from CMU[1]. Comparison shows that our methods are better on semi-structured document sets.

Table 7.3 lists the comparison results of Rainbow and our classification method. The accuracy results obtained by Rainbow vary from 77.08% to 89.58%. Our method classifies documents based on Formula 6.3. It delivers 100% accuracy on all dataset except ETC, and the average accuracy of our method is about 13% higher than Rainbow. On the ETC dataset including documents with complicated structure, Rainbow's results are not good, although in other datasets it can receive accuracy larger than 85%. Our method is almost not affected by the difference among datasets. On the **ETC** dataset, one page was classified to the wrong class by our methods. The reason is that the page is a page linking other sub-categories and it is difficult to judge to which class it belongs.

[1]http://www-2.cs.cmu.edu/~mccallum/bow/

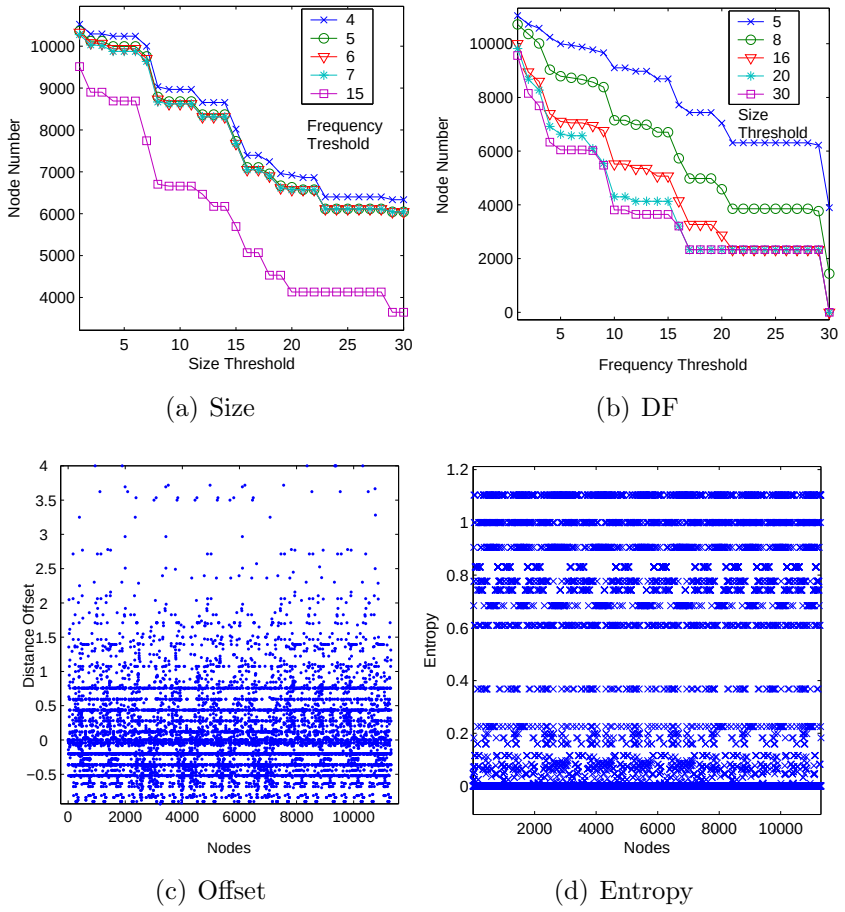
DataSet	RecNum	Extracted	Accuracy
Alta	100	90	0.9
Cora	100	70	0.7
Excite	100	100	1
Galaxy	200	199	1
Hotbot	100	95	0.95
L.A. Weekly	81	76	0.94
Magellan	100	92	0.92
Metacrawler	200	179	0.9
Northenlight	100	97	0.97
Openfind	200	185	0.93
SavvySearch	150	140	0.93
StptCom	100	97	0.97
Webcrawler	250	141	0.56
Total	1781	1561	0.88

Table 7.4: Extraction Accuracy

7.4 Extraction Accuracy

We evaluated our extraction method on IEPAD dataset including Web documents returned by 10 search engines listed in the first column of Table 7.4. We first manually annotated DIs in these documents; the method is quite straightforward; i.e., each entry returned by those searches is treated as a DI. The numbers of hand-annotated DIs ("RecNum") are listed in the second column in Table 7.4.

We executed SchemaDectector to assign schema ID to each DI in document trees parsed from documents of IEPAD. As the result, most annotated DIs from the same search engine are assigned with the same schema ID. In Table 7.4, the second column ("Extracted") is the number of annotated DIs belonging to the type that contains the largest number of annotated DIs. We observe that our method accurately identifies these important contents in documents. The accuracy values in column 3 are the rate of "Extracted" to "RecNum". For document trees returned by most search engines, our accuracy rate is greater than 90%. The reason for receiving poor accuracy results in documents from "Cora" and "WebCrawler" is that these search engines highlight some of their search results using various formats. If we consider the type that contains the second largest set of DIs, the accuracy values will exceed 90% too.



(a) Size (b) DF

(c) Offset (d) Entropy

Figure 7.2: Distribution of Schema Attributes

7.5 Discussion on Schema Attributes

As introduced before, we adjst parameters to select which types and DIs should be extracted. Figure 7.2(a) shows the relation between size threshold and the number of nodes extracted; i.e., only those DIs conforming to schema with size larger than the threshold are extracted. As the threshold increases, the number of extracted nodes decreases. In some ranges, the changes are very slow. For example, when the size threshold changes from 25 to 28, the number of extracted nodes almost does not change, since there are many DIs belonging to the same schema that has 28 nodes. In Figure 7.2(b), when we change the DF threshold from 0 to 30, the number of extracted nodes changes as shown in Figure 7.2(a). In Figure 7.2(c), we plot the distance offset of each DI. The X-axis is the position of their root nodes in traversal and the Y-axis corresponds to distance offset.

Most offsets are located from -1 to 1.5. Empirically, those DIs with large offset are noise. As introduced before, high entropy also means noise sometimes. Figure 7.2(d) shows that DIs conforming to schemata with low entropy is very easily distinguished from those with high entropy. We may control extraction results by combining the parameters. For example, in the AltaVista documents of **IEPAD**, when we set the size threshold to 6 and frequency threshold to 89, most child-DIs of the root node in extracted XML document are those DIs we annotated in the original documents. Figures 7.2(a) and (b) show the effect of combining two parameters.

Chapter 8

Conclusion and Future Work

Information overload motivates the development of concierge systems. A Web information concierge provides a structural accessing interface to Web data resource for end users. Efficiency and expressive power of concierge systems are main concerns of our research. Here, the contributions of this thesis are summarized and discussed. Some improvements of our systems is left for further study.

8.1 Summarization

Much research has been done to optimize the performance of Web data extraction, improve extraction accuracy, and deliver formatted results. However, the lack of generic conceptual modeling over Web documents, extraction methods and extraction rule generation methods makes it difficult to communicate among different systems and extend current systems. The conceptual framework for information concierge system proposed in this thesis is to overcome this problem.

Our framework contains three layers, which are corresponding to physical models of Web data, logic models of Web data and models of the possible operations a user can use to handle Web data. Unlike those ad-hoc methods, our framework separates research issues and puts them in their due perspectives. Similar with the organization of relational database, the separation of physical Web data and logic view over Web data (schema) provides an abstract layer between operations and physical data. A simple example is presented in Chapter 3 to demonstrate the relationships among different layers.

The success of relational database can partially owe to the solid theoretical basis of relationship schema. Chapter 4 presents various perspectives over Web data and

schemata, and proposed some classes of schemata with different expressive power. The subtle classification of the schema based on our unified framework provides a basis to compare various systems in literature.

Other than the theoretical analysis of Web data and their schemata, Chapter 5 concentrates on how to improve the performance of kernel operations in our framework. Some promising results are presented in Chapter 5. To show the flexibility of our framework, some auxiliary operations are introduced in Chapter ch:aux.

8.2 Conclusion

This section lists main contributions of this thesis:

Flexible Concierge Framework This thesis presents a comprehensive framework for Web data extraction system that provides a consistent view over various operations in Web data extraction. Operations including document set preparation, document classification and data extraction are all conducted in schema-based representation of Web documents. To support these operations, similarity measurements for semi-structured documents based on schema are proposed. In our experiments on real world dataset, compared with the methods that do not consider document structures, much better clustering results were achieved using schema-based representation in terms of clustering accuracy. Moreover, we have also demonstrated better document classification results using schema-based representation.

Wdel **and Query Transducer** The proposed framework is investigated from various perspectives, including its characteristics of tree language, logic program and tree automata. Based on these characteristics, expressive capability of extraction rules, the construction of extraction engines and the model of Web documents are analyzed from a unique view. A Web data extraction language — Wdel that is roughly equivalent to MSO logic is defined. Wdel is encoded in XML documents to present an intuitive hierarchical view over Web documents, and can be easily generated via a visual interface implemented in Wiccap. Due to the connection between automata and language, a kind of automata — query transducer is introduced to interpret Wdel and extract data from Web documents. Wdel has high expressive capability and is easy to be generated manually with the aid from visual interface.

Wdel Generation Algorithm Manually coded Wdel programs are accurate to extraction contents, as a user can adjust them in modify-test-modify manners. In despite of this benefit, manual methods are time-consuming. We compromise the expressive capability of Wdel and devise some algorithms that can generate a subset of Wdel. Two types of algorithms are introduced. The first type parses Web documents to trees and detects frequent tree patterns from them. These patterns can be encoded to Wdel programs. The second type share similar ideas like those suffix tree construction algorithm, and need not to parse documents to trees. It can detect some substring patterns that can be parsed to trees, and transformed to Wdel. We presented Algorithms with time complexity linear to document size. The first type is less efficient than the second one, but can produce programs to extract more complicated documents. The second type over-performs present pattern detection algorithms, due to its linear complexity.

Auxiliary Components Besides the main components of extraction engine and extraction rule generator, some auxiliary components inside the concierge framework are introduced, to show the flexibility of the framework. Based on these structural patterns very efficiently detected, we introduced methods to measure document similarity, cluster and classify documents. The clustering and classification results are compared with other methods based on structural information. A visual interface like QBE is also demonstrated.

8.3 Future Work

In this thesis, we discussed technique about various aspects of Web information concierge systems. All the work in this thesis is purely based on structure information embedded in Web documents. The experiment results also show that structural information is very important to process Web documents, and is strongly related to semantic information of Web documents. However, whether to exploit semantic information can improve our work is not addressed in this thesis yet. In the future, how to combine semantic and structural information will be studied.

As the extraction language introduced is oriented to XML documents, and there is a trend to use XML documents as databases of complex data; e.g., bioinformatic data, the application Wdel may be extended to these databases. Due to the high overlay

information is such kind of complex data, how to exploit structural information with semantic information to detect redundant data is to be studied.

Bibliography

Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. of 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

Alfred V. Aho and Margaret J. Corasack. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

Tatsuya Akutsu. An RNC algorithm for finding a largest common subtree of two trees. *IEICE Trans. Information and Systems*, E75-D:95–101, 1992.

Tatsuya Akutsu and Magnús M. Halldórsson. On the approximation of largest common subtrees and largest common point sets. *Theoretical Computer Science*, 233(1-2):33–50, 2000.

Shurug Al-Khalifa, Cong Yu, and H. V. Jagadish. Querying structured text in an xml database. In *Proc. of the 2003 ACM SIGMOD*, pages 4 – 15, 2003.

Natasha Alechina, Stéphane Demri, and Maarten de Rijke. Path constraints from a modal logic point of view (extended abstract). In *Proceedings of 8th Workshop on Knowledge Representation meets Databases (KRDB'01)*, volume 45, 2001.

Natasha Alechina, Stéphane Demri, and Maarten de Rijke. A modal perspective on path constraints. *Journal of Logic and Computation*, 6(3):939–956, 2003.

Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *Computing Surveys*, 15(3):237–269, 1983.

Douglas E. Appelt and David Israel. Introduction to information extraction technology. In *in Tutorial of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)-99*. Artificial Intelligence Center SRI International, 1999.

Arvind Arasu and Hector Garcia-Molina. Extracting structured data from web pages. In *Proc. of the 2003 ACM SIGMOD*, pages 337 – 348, San Diego, California, 2003. ACM Press.

Hiroki Arimura, Hiroki Ishizaka, and Takeshi Shinohara. Learning unions of tree patterns using queries. *Theoretical Computer Science*, 185(1):47–62, 1997.

Luigi Arlotta, Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Automatic annotation of data extracted from large web sites. In *Proceedings of the Eleventh Italian Symposium on Advanced Database Systems*, 2003.

Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroki Arimura, Hiroshi Satamoto, and Setsuo Arikawa. Efficient substructure discovery from large semi-structured data. In *Proceedings of the 2nd Annual SIAM Symposium on Data Mining*, 2002.

Tatsuya Asai, Hiroki Arimura, Takeaki Uno, and Shin ichi Nakano. Discovering frequent substructures in large unordered trees. Technical Report 216, University of Kyushuu, 2003.

Thomas Ball and Fred Douglis. Tracking and viewing changes on the web. In *Proceedings of 1996 USENIX Technical Conference*, pages 165–176, 1996.

Hideo Bannai, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Inferring strings from graphs and arrays. In *Proc. 28th International Symposium on Mathematical Foundations of Computer Science (MFCS 2003)*, pages 208–217, 2003.

Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Declarative information extraction, web crawling, and recursive wrapping with lixto. In *Proceedings of 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, 2001a.

Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with lixto. In *Proc. of 27th International Conference on Very Large Data Bases*, pages 119–128, Roma, Italy, 2001b. Morgan Kaufmann.

Robert Baumgartner, Sergio Flesca, and Georg Gottlob. The elog web extraction language. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001*, volume 2250 of *Lecture Notes in Computer Science*, pages 548–560, Havana, Cuba, December 3-7 2001c. Springer. ISBN 3-540-42957-3.

André Bergholz. *Querying Semistructured Data Based On Schema Matching*. PhD thesis, Humboldt-University Berlin, 2000.

André Bergholz and Johann Christoph Freytag. Querying semistructured data based on schema matching. In *Revised Papers from the 7th International Workshop on Database Programming Languages: Research Issues in Structured and Semistructured Database Programming*, pages 168 – 183, 1999.

Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May:34–43, 2001.

Elisa Bertinoa, Giovanna Guerrinib, and Marco Mesiti. A matching algorithm for measuring the structural similarity between an xml document and a DTD and its applications. *Journal of Information Systems*, 29(1):23–46, March 2004.

Gerd Beuster, Bernd Thomas, and Christian Wolff. MIA - A ubiquitous multi-agent web information system. In *Proceedings of International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'2000)*, 2000.

D. Bianchi. Learning grammatical rules from examples using a credit assignment algorithm. In *Proceedings of the First Online Workshop on Soft Computing (WSC1)*, pages 113–118, 1996.

Daniel M. Bikel, Scott Miller, Richard Schwartz, and Ralph Weischedel. Nymble: a high-performance learning name-finder. In *Proceedings of Applied Natural Language Processing (ANLP)*, pages 194–201, 1997.

Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1 (2001). Technical report, HKUST, 2001.

Peter Buneman, Wenfei Fan, and Scott Weinstein. Query optimization for semistructured data using path constraints in a deterministic data model. In *Revised Papers from the 7th International Workshop on Database Programming Languages: Research Issues in Structured and Semistructured Database Programming*, pages 208 – 223, 1999.

Robert Burbidge. An introduction to support vector machines for data mining. In *YOR 12*, 2001.

Deng Cai, Xiaofei He, Ji-Rong Wen, and Wei-Ying Ma. Block-level link analysis. In *Proc. of The 27th Annual International ACM SIGIR*, 2004.

Julien Carme, Joachim Niehren, and Marc Tommasi. Querying unranked trees with stepwise tree automata. In Vincent van Oostrom, editor, *International Conference on Rewriting Techniques and Applications, Aachen*, volume 3091 of *Lecture Notes in Computer Science*, pages 105–118. Springer, June 2004.

Ashok K. Chandra and David Harel. Structure and complexity of relational queries. In *IEEE Symposium on Foundations of Computer Science*, pages 333–347, 1980.

Chia-Hui Chang and Shao-Chen Lui. IEPAD: information extraction based on pattern discovery. In *Proc. of the 10th International WWW Conference*, pages 681 – 688, Hong Kong, 2001.

Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proc. of the 5th ACM SIGMOD international conference on Management of data and Symposium on PODS*, pages 493 – 504, 1996.

Yan Chen, Sanjay Kumar Madria, and Sourav S. Bhowmick. Diffxml: Change detection in xml data. In *Proc. of the 9th Database Systems for Advanced Applications (DASFAA)*, pages 289–301, 2004.

Yih-Farn Chen, Fred Douglis, Huale Huang, and Kiem-Pong Vo. Topblend: An efficient implementation of htmldiff in java. In *Proceedings of WebNet 2000*, pages 88–94, San Antonio, Texas, 2000.

Yun Chi, Yirong Yang, and Richard R. Muntz. Indexing and mining free trees. In *Proc. of the 3th International Conference on Data Mining*, 2003.

Boris Chidlovskii, Jon Ragetli, and Maarten de Rijke. Wrapper generation via grammar induction. In *Proceedings of 11th European Conference on Machine Learning (ECML)*, volume 1810, pages 96–108, Barcelona, Catalonia, Spain, May 2000. Springer, Berlin.

K.W. Church and R.L. Mercer. Introduction to the special issue on computational linguistics using large corpora. *Computational Linguistics*, 19(1):1–24, 1993.

Gregory Cobena and Serge Abiteboul. Detecting changes in xml documents. In *Proc. of 18th International Conference on Data Engineering*, 2002.

William Cohen and Lee Jensen. A structured wrapper induction system for extracting information from semi-structured documents. In *Proceedings of International Joint Conference on Artificial Intelligence IJCAI-2001 Workshop on Adaptive Text Extraction and Mining*, Seattle, August 2001.

William W. Cohen. Recognizing structure in web pages using similarity queries. In *Proc. of the 6th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, pages 59–66, 1999. ISBN 0-262-51106-1.

Michael Collins and Nigel Duffy. Convolution kernels for natural language. In *Neural Information Processing Systems*, volume 14, pages 625–632, 2001.

Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. release October, 1rst 2002, 1997.

Bruno Courcelle. *Resolution of equations in algebraic structures*, chapter On recognizable sets and tree automata, pages 93–126. Academic Press, 1989.

V. Crescenzi and G. Mecca. Grammars have exceptions. *Information Systems*, 23(8): 539–565, 1998.

Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. RoadRunner: Towards automatic data extraction from large web sites. In *Proc. of 27th International Conference on Very Large Data Bases*, pages 109–118, 2001.

Altigran S. da Silva, Irna M.R. Evangelista Filha, Alberto H. F. Laender, and David W. Embley. Representing and querying semistructured web data using nested tables with structural variants. In *Proc. of the 21st International Conference on Conceptual Modelling*, 2002.

Silvano Dal Zilio and Denis Lugiez. XML schema, tree logic and sheaves automata. In *Proc. of 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 246–263. Springer-Verlag, June 2003.

Nilesh Dalvi and Dan Suciu. Indexing heterogeneous data. Technical Report 04-01-01, University of Washington, 2004. available from www.cs.washington.edu.

Evgeny Dantsin and Andrei Voronkov. Expressive power and data complexity of first-order logic over trees. *Journal of Computer and System Sciences*, 2005.

Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)*, 33(3):374 – 425, 2001.

Brian D. Davison. Unifying text and link analysis. In *IJCAI-03 Workshop on Text-Mining & Link-Analysis (TextLink)*, 2003.

Hasan Davulcu, Guizhen Yang, Michael Kifer, and I. V. Ramakrishnan. Computational aspects of resilient data extraction from semistructured sources. In *Proceedings of the Nineteenth PODS*, pages 136–144, 2000.

John December. *The World Wide Web Unleashed*. Sams Publishing, 1994.

Anhai Doan, Pedro Domingos, and Alon Halevy. Learning to match the schemas of data sources: A multistrategy approach. *Machine Learning*, 50(3):279 – 301, 2003.

J. E. Doner. Decidability of the weak second-order theory of two successors. *Notices Amer. Math. Soc.*, 12:365–468, 1965.

J. E. Doner. Tree acceptors and some of their applications. *Computer and System Sciences*, 4(5):406–451, 1970.

Line Eikvil. Information extraction from world wide web - a survey. Technical Report 945, Norweigan Computing Center, 1999.

Thomas Eiter, Georg Gottlob, and Yuri Gurevich. Normal forms for second-order logic over finite structures, and classification of np optimization problems. *Ann. Pure Appl. Logic*, 78(1-3):111–125, 1996.

D. Embley, S. Jiang, and Y. Ng. Record-boundary discovery in web documents. In *Proc. of the 1999 ACM SIGMOD*, pages 467–478, 1999a.

David W. Embley, Douglas M. Campbell, Y. S. Jiang, Stephen W. Liddle, Yiu-Kai Ng, Dallan Quass, and Randy D. Smith. Conceptual-model-based data extraction from multiple-record web pages. *Journal of Data & Knowledge Engineering*, 31(3):227–251, 1999b.

Mary F. Fernandez, Jerome Simeon, and Philip Wadler. An algebra for xml query. In *Proceedings of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science*, 2000.

Mirtha-Lina Fernández and Gabriel Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letters*, 22(6/7):753–758, 2001.

Sergio Flesca, Giuseppe Manco, Elio Masciari, Luigi Pontieri, and Andrea Pugliese. Detecting structural similarities between xml documents. In *Proceedings of 5th International Workshop on the Web and Databases*, Madison, Wisconsin, USA, June 2002.

Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

Francesco De Francesca, Gianluca Gordano, Giuseppe Manco, Riccardo Ortale, and Andrea Tagarelli. A general framework for xml document clustering. Technical report, ICAR-CNR, 2003.

Dayne Freitag. Multistrategy learning for information extraction. In *Proc. of 15th International Conf. on Machine Learning*, pages 161–169. Morgan Kaufmann, San Francisco, CA, 1998.

Dayne Freitag and Nicholas Kushmerick. Boosted wrapper induction. In *Proc. of the 7th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, pages 577–583, 2000.

T. Frhwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. 6th IEEE Symp. Logic in Computer Science, Amsterdam*, pages 300–309, 1991.

Alain Frisch. Regular tree language recognition with static information. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004.

R. Gaizauskas and Y. Wilks. Information extraction: Beyond document retrieval. *Journal of Documentation*, 54(1):70–105, 1998.

Harald Ganzinger, Christoph Meyer, and Christoph Weidenbach. Soft typing for ordered resolution. In *Conference on Automated Deduction*, pages 321–335, 1997.

P. Garcia and E. Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(9):920–925, 1990.

Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. XTRACT: a system for extracting document type descriptors from XML documents. In *Proc. of the 2000 ACM SIGMOD*, 2000.

Thomas Gärtner. A survey of kernels for structured data. *SIGKDD Explorations*, 5(1): 49 – 58, July 2003.

Jeroen Geertzen. String alignment in grammatical inference – what suffix trees can do. Master's thesis, Tilburg University, 2003.

David Gibson, Jon Kleinberg, and Prabhakar Raghavan. Inferring web communities from link topology. In *Proceedings of the ninth ACM conference on Hypertext and hypermedia*, pages 225 – 234, 1998.

E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5): 447–474, 1967.

Paulo Braz Golgher, Altigran Soares da Silva, Alberto H. F. Laender, and Berthier A. Ribeiro-Neto. Bootstrapping for example-based data extraction. In *Proc. of the 10th ACM CIKM International Conference on Information and Knowledge Management*, pages 371–378, Atlanta, Georgia, USA, 2001. ACM. ISBN 1-58113-436-3.

Georg Gottlob and Christoph Koch. Monadic queries over tree-structured data. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 189 – 202, 2002a.

Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. In *Proc. of the 21th ACM SIGMOD-SIGACT-SIGART symposium on PODS*, pages 17 – 28, 2002b.

Georg Gottlob, Christoph Koch, and Reinhard Pichler. Xpath processing in a nutshell. *SIGMOD Record*, 32(1):12–19, 2003.

Martin Grohe. Parameterized complexity for the database theorist. *SIGMOD Record*, 31 (4):86 – 96, December 2002.

Stéphane Grumbach and Giansalvatore Mecca. In search of the lost schema. In *Proc. of the 7th International Conference of Database Theory*, volume 1540, pages 314–331, 1999.

Jean-Robert Gruser, Louiqa Raschid, M. E. Vidal, and Laura Bright. Wrapper generation for web accessible data sources. In *Proc. of Conference on Cooperative Information Systems*, pages 14–23, 1998.

Ashish Gupta, Venky Harinarayan, and Anand Rajaraman. Virtual database technology. *SIGMOD Record*, 26(4):57–61, 1997.

Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*, chapter Linear-Time Construction of Suffix Trees, pages 94–121. Cambridge University Press, 1999.

Joachim Hammer, Hector Garcia-Molina, Junghoo Cho, Arturo Crespo, and Rohan Aranha. Extracting semistructured information from the web. In *Proc. of Workshop on Management fo Semistructured Data*, pages 18–25, 1997a.

Joachim Hammer, Héctor García-Molina, Svetlozar Nestorov, Ramana Yerneni, Marcus Breunig, and Vasilis Vassalos. Template-based wrappers in the TSIMMIS system. In *Proc. of the 1997 ACM SIGMOD*, pages 532–535, 1997b.

Lisa Hellerstein, Krishnan Pillaipakkamnatt, Vijay Raghavan, and Dawn Wilkins. How many queries are needed to learn. *Journal of the ACM*, 43(5):840 – 862, 1996.

D. Hiemstra and A. de Vries. Relating the new language models of information retrieval to the traditional retrieval models. Technical Report TR–CTIT–00–09, Centre for Telematics and Information Technology, May 2000.

Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4):664–675, 1977.

Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68 – 95, January 1982.

Theodore W. Hong and Keith L. Clark. Using grammatical inference to automate information extraction from the web. In *Proceedings of the 5th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 216 – 227, 2001.

Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for XML. *ACM SIGPLAN Notices*, 36(3):67–80, 2001.

C. Hsu and C. Chang. Finite-state transducers for semi-structured text mining. In *Proc. of 16th International Joint Conference on Artificial Intelligence IJCAI-99 Workshop on Text Mining*, 1999.

Chun-Nan Hsu and Ming-Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Journal of Information Systems*, 23(8):521–538, 1998.

Guy Jacobson and Kiem-Phong Vo. Heaviest increasing/common subsequence problems. In *Proceedings of Combinatorial Pattern Matching (CPM), Third Annual Symposium,*, pages 52–66, 1992.

H. V. Jagadish, Nick Koudas, and Divesh Srivastava. On effective multi-dimensional indexing for strings. In *Proc. of the 2000 ACM SIGMOD*, volume 29(2), pages 403–414, May 2000.

Gerald Penn Jianying. Flexible web document analysis for delivery to narrow-bandwidth devices. In *Proc. of 6th International Conference on Document Analysis and Recognition (ICDAR01)*, pages 1074–1078, Seattle, WA, USA, September 2001.

George Karypis. A clustering toolkit. Technical Report TR#02-017, Univ. of Minnesota, 2002.

Stephan Kepser. A simple proof for the turing-completeness of xslt and xquery. In *Extreme Markup Languages*, 2004.

Sanjeev Khanna, Rajeev Motwani, and Frances F. Yao. Approximation algorithms for the largest common subtree problem. Technical Report CS-TR-95-1545, Stanford University, 1995.

Thomas Kistler and Hannes Marais. WebL - a programming language for the web. In *Proc. of the 7th International WWW Conference*, volume 30(1-7) of *Computer Networks*, pages 259–270, 1998.

Nils Klarlund. *Logics for Emerging Applications of Databases*, chapter XML: Model, Schemas, Types, Logics and Queries, pages 1–42. Springer Verlag, 2003.

Dan Klein and Christopher D. Manning. Distributional phrase structure induction. In *Proceedings of CoNLL*, 2001.

Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

Timo Knuutila. *On the Inductive Inference of Regular String and Tree Languages*. PhD thesis, Turku Centre for Computer Science (TUCS), 1994. Technical Report R-94-14.

Raymond Kosala. *Information Extraction by Tree Automata Inference*. PhD thesis, Katholieke Universiteit Leuven, 2003.

Raymond Kosala and Hendrik Blockeel. Instance-based wrapper induction. In *Proc. of the 10th Belgium-Dutch Conference on Machine Learning*, pages 61–68, 2000.

Raymond Kosala, Jan Van den Bussche, Maurice Bruynooghe, and Hendrik Blockeel. Information extraction in structured documents using tree automata induction. In *Proc. of the 6th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 299–310, 2002.

Raymond Kosala, Maurice Bruynooghe, Hendrik Blokceel, and Jan Van den Bussche. Information extraction from web documents based on local unranked tree automaton inference. In *Proc. of the 18th International Joint Conferences on Artificial Intelligence*, 2003.

Gregory Kucherov, Dieter Hofbauer, and Maria Huber. Some results on top-context-free tree languages. *19th International Colloquium on Trees in Algebra and Programming, Lecture Notes in Computer Science*, 787:157–171, 1994.

Stefan Kuhlins and Ross Tredwell. Toolkits for generating wrappers – a survey of software toolkits for automated data extraction from web sites. In *Proc. of the Main Conference Net.ObjectDays 2002: Objects, Components, Architectures, Services and Applications for a Networked World*, pages 190–204, Erfurt, October 2002.

Karen Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

Yin-Hung Kuo and Man Hon Wong. Web document classification based on hyperlinks and document semantics. In *Procceding of PRICAI Workshop on Text and Web Mining*, pages 44–51, 2000.

N. Kushmerick. Gleaning the web. *IEEE Intelligent Systems*, 14(2):20–22, 1999a.

Nicholas Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1-2):15–68, 2000a.

Nicholas Kushmerick. *Wrapper induction for information extraction*. PhD thesis, Dept of Computer Science & Engineering, Univ of Washington, 1997. Technical Report UW-CSE-97-11-04.

Nicholas Kushmerick. Wrapper verification. *World Wide Web*, 3(2):79–94, 2000b.

Nicholas Kushmerick. Regression testing for wrapper maintenance. In *Proc. of the 6th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, pages 74–79, 1999b.

Nicholas Kushmerick and Bernd Thomas. Adaptive information extraction: Core technologies for information agents. In *Intelligent Information Agents R&D in Europe: An AgentLink perspective*. Springer, 2002.

Alberto H. F. Laender, Berthier A. Ribeiro-Neto, Altigran S. da Silva, and Juliana S. Teixeira. A brief survey of web data extraction tools. *SIGMOD Record*, 31(2):84 – 93, June 2002.

Dongwon Lee and Wesley W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record*, 29(3):76–87, 2000.

Timothy Robert Leek. Information extraction using hidden Markov models. Master's thesis, UC San Diego, 1997.

Kristina Lerman and Steven Minton. Learning the common structure of data. In *Proc. of the 7th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, 2000.

Kristina Lerman, Craig Knoblock, and Steven Minton. Wrapper maintenance: A machine learning approach. *Journal of Artificial Intelligence Research*, 18:149–181, 2003.

Michael Y. Levin. Compiling regular patterns. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, 2003.

Alon Y. Levy and Daniel S. Weld. Intelligent internet systems. *Artificial Intelligence*, 118(1-2):1–14, 2000.

Leon S. Levy and Aravind K. Joshi. Skeletal structural descriptions. *Information and Control*, 39(2):192–211, 1978.

Feifei Li, Zehua Liu, Yangfeng Huang, and Wee Keong Ng. An information concierge for the web. In A. Min Tjoa and Roland Wagner, editors, *12th International Workshop on Database and Expert Systems Applications (DEXA)*, pages 672–676, Munich, Germany, September 2001. IEEE Computer Society. ISBN 0-7695-1230-5.

Wenyuan Li, Wee-Keong Ng, and Ee-Peng Lim. Spectral analysis of text collection for similarity-based clusting. In *Proc. of 20th International Conference on Data Engineering*, 2004.

Zhao Li and Wee Keong Ng. Wiccap: From semi-structured data to structured data. In *Proc. of 11th IEEE International Conference on the Engineering of Computer-Based Systems*, 2004.

Zhao Li, Wee Keong Ng, and Aixin Sun. Web data extraction based on structural similiarity. *Knowledge and Information Systems*, 8(4):438 – 461, 2005. Published online first: 2 February 2005, DOI: 10.1007/s10115-004-0188-z.

Wang Lian and David Wai-Lok Cheung. An efficient and scalable algorithm for clustering xml documents by structure. *IEEE Transactions on Knowledge and Data Engineering*, 16(1):82– 96, Junuary 2004.

Shian-Hua Lin and Jan-Ming Ho. Discovering informative content blocks from web documents. In *Proc. of the 8th International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2002.

Bing Liu, Robert Grossman, and Yanhong Zhai. Mining data records in web pages. In *Proc. of the 9th International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 601 – 606, 2003.

Ling Liu, Calton Pu, and Wei Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *Proc. of 16th International Conference on Data Engineering*, pages 611–621, 2000.

Xin Liu, Yihong Gong, Wei Xu, and Shenghuo Zhu. Document clustering with cluster refinement and model selection capabilities. In *Proceedings of the 25th ACM SIGIR*, pages 191 – 198, 2002a.

Zehua Liu, Feifei Li, and Wee Keong Ng. WICCAP data model: Mapping physical websites to logical views. In *Proc. of the 21st International Conference on Conceptual Modelling*, 2002b.

Zehua Liu, Wee Keong Ng, Feifei Li, and Ee-Peng Lim. A visual tool for building logical data models of websites. In *Proc. of Fourth ACM CIKM International Workshop on Web Information and Data Management (WIDM'02)*, McLean, Virginia, USA, November 8 2002c.

Alejandro López-Ortiz. Linear pattern matching of repeated substrings. *ACM SIGACT News*, 25(3):114 – 121, 1994.

Chin Lung Lu, Zheng-Yao Su, and Chuan Yi Tang. A new measure of edit distance between labeled trees. In *7th Annual International Conference of Computing and Combinatorics*, 2001.

A.-K. Luah, Wee Keong Ng, Ee-Peng Lim, W.-P. Lee, and Yinyan Cao. Locating web information using web checkpoints. In *Database and Expert Systems Applications (DEXA) Workshop*, pages 716–720, 1999.

B. Ludäscher, P. Mukhopadhayn, and Y. Papakonstantinou. A transducer-based xml query processor. In *Proc. of 28th International Conference on Very Large Data Bases*, August 2002.

Bertram Ludascher, Rainer Himmeroder, Georg Lausen, Wolfgang May, and Christian Schlepphorst. Managing semistructured data with FLORID: A deductive object-oriented perspective. *Journal of Information Systems*, 23(8):589–613, 1998.

Y.S. Maarek and I.Z. Ben Shaul. Automatically organizing bookmarks per contents. In *Proc. of the 5th International WWW Conference*, Paris, France, May 1996.

Sanjay Kumar Madria, Sourav S. Bhowmick, Wee Keong Ng, and Ee-Peng Lim. Research issues in web data mining. In *Data Warehousing and Knowledge Discovery*, pages 303–312, 1999.

Wim Martens and Frank Neven. Typechecking top-down uniform unranked tree transducers. In *Proceedings of the 9th International Conference on Database Theory*, pages 64 – 78, 2003.

Wolfgang May, Rainer Himmeröder, Georg Lausen, and Bertram Ludäscher. A unified framework for wrapping, mediating and restructuring information from the web. In *Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, pages 307–320, 1999. ISBN 3-540-66653-2.

Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262 – 272, 1976.

Dennis McLeod. The translation and compatibility of sequel and query by example. In *Proceedings of the 2nd international conference on Software engineering*, pages 520 – 526, 1976.

Giansalvatore Mecca and Paolo Atzeni. Cut and paste. *Journal of Computer and System Sciences*, 58(3):453–482, 1999.

Giansalvatore Mecca, Paolo Atzeni, Alessandro Masci, Paolo Merialdo, and Giuseppe Sindoni. The araneus web-base management system. In *Proc. of the 1998 ACM SIGMOD*, pages 544–546, 1998.

Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the nineteenth PODS*, 2000.

Jack Minker. Logic and databases: A 20 year retrospective. In *Logic in Databases*, pages 3–57, 1996.

Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

Tom M. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR-117, Rutgers University, New Brunswick, New Jersey, 1980.

Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88, 2002.

Dan I. Moldovan and Roxana Girju. An interactive tool for the rapid development of knowledge bases. *International Journal on Artificial Intelligence Tools*, 10(1-2):65–86, 2001.

Etsuro Moriya. On two-way tree automata. *Information Processing Letters*, 50(3):113–172, 1994.

Donald R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

Pierre-Alain Muller, Philippe Studer, and Jean Bézivin. Platform independent web application modeling. In *UML*, pages 220–233, 2003.

Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, 2001.

I. Muslea. Extraction patterns: from information extraction to wrapper generation. Technical report, Information Sciences Institute, University of Southern California (ISI-USC), 1998.

Ion Muslea. Extraction patterns for information extraction tasks: A survey. In *Proc. of Workshop on Machine Learning and Information Extraction (AAAI-99)*, pages 1–6, Orlando, Florida, 1999.

Ion Muslea, Steve Minton, and Craig Knoblock. A hierarchical approach to wrapper induction. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proc. of the Third International Conference on Autonomous Agents (Agents'99)*, pages 190–197, Seattle, WA, USA, 1999. ACM Press.

Ion Muslea, Steven Minton, and Craig A. Knoblock. Selective sampling with redundant views. In *Proc. of the 7th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, pages 621–626, 2000.

Ion Muslea, Steven Minton, and Craig A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4 (1/2):93–114, 2001.

Frank Neven. Automata, logic, and xml. In *Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic*, pages 2 – 26, 2002a.

Frank Neven. Automata theory for xml researchers. *SIGMOD Record*, 31(3):39 – 46, September 2002b.

Frank Neven and Thomas Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. of the 9th ACM SIGMOD international conference on Management of data and Symposium on PODS*, pages 145 – 156, 2000.

Frank Neven and Thomas Schwentick. Automata-and logic-based pattern languages for tree-structured data. In *Semantics in Databases*, pages 160–178, 2001.

Frank Neven and Thomas Schwentick. Query automata. In *Proc. of the 8th ACM SIGMOD international conference on Management of data and Symposium on PODS*, pages 205 – 214, 1999.

Wee Keong Ng, Zehua Liu, Zhao Li, and Ee Peng Lim. Personalized web information extraction via web views. In *Web Information Systems*. Idea Group Inc, Hershey, Pennsylvania, USA, 2003.

Andrew Nierman and H. V. Jagadish. Evaluating structural similarity in xml documents. In *Proceedings of 5th International Workshop on the Web and Databases*, 2002.

Siegfried Nijssen and Joost N. Kok. Efficient discovery of frequent unordered trees. In *Proceedings of First International Workshop on Mining Graphs, Trees and Sequences*, 2003.

Marcello Pelillo, Kaleem Siddiqi, and Steven W. Zucker. Matching hierarchical structures using association graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(11):1105–1120, 1999.

Pallavi Priyadarshini, Fengqiong Qin, Ee-Peng Lim, and Wee Keong Ng. WEDAGEN - a synthetic web database generator. In *Database and Expert Systems Applications (DEXA) Workshop*, pages 744–748, 1999.

Anand Rajaraman and Jeffrey D. Ullman. Querying websites using compact skeletons. In *Proc. of the 12th ACM SIGMOD-SIGACT-SIGART symposium on PODS*, pages 16 – 27, 2001.

R. Ramesh and I. V. Ramakrishnan. Nonlinear pattern matching in trees. *Journal of the ACM*, 39(2):295 – 316, April 1992.

Berthier A. Ribeiro-Neto, Alberto H. F. Laender, and Altigran Soares da Silva. Extracting semi-structured data through examples. In *Proc. of the 8th ACM CIKM International Conference on Information and Knowledge Management*, pages 94–101. ACM, 1999. ISBN 1-58113-146-1.

Juan Ramn Rico-Juan, Jorge Calera-Rubio, and Rafael C. Carrasco. Probabilistic k-testable tree-languages. In *Proc. of 5th International Colloquium, ICGI 2000*, 2000.

Stefan Riezler, Tracy King, Ronald Kaplan, Richard Crouch, John T. Maxwell III, and Mark Johnson. Parsing the wall street journal using a lexical-functional grammar and discriminative estimation techniques. In *Proc. of the 40th Annual Conference of the Association for Computational Linguistics (ACL-02)*, Philadelphia, PA, 2002.

Eric Sven Ristad and Peter N. Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, 1998.

Arnaud Sahuguet and Fabien Azavant. Building intelligent web applications using lightweight wrappers. *Journal of Data & Knowledge Engineering*, 36(3):283–316, 2001.

Yasubumi Sakakibara. Recent advances of grammatical inference. *Theoretical Computer Science*, 185(1):15–45, 1997.

Hiroshi Sakamoto, Yoshitsugu Murakami, Hiroki Arimura, and Setsuo Arikawa. Extracting partial structures from html documents. In *14th International Florida Artificial Intelligence Research Symposium (FLAIRS'2001) Conference*, pages 264–268. AAAI, 2001.

Jason Sankey and Raymond K. Wong. Structural inference for semistructured data. In *Proceedings of the tenth international conference on Information and knowledge management*, 2001.

Alexandr Savinov. Logical navigation in the concept-oriented data model. *Journal of Conceptual Modeling*, 36, 2008.

Flesca Sergio, Manco Giuseppe, Masciari Elio, Pontieri Luigi, and Pugliese Andrea. Structural document similarity for integrated crawling and wrapping. Technical report, Istituto di Calcolo e Reti ad Alte Prestazioni, 2003.

Kristie Seymore, Andrew McCallum, and Roni Rosenfeld. Learning hidden Markov model structure for information extraction. In *AAAI 99 Workshop on Machine Learning for Information Extraction*, 1999.

Ron Shamir and Dekel Tsur. Faster subtree isomorphism. In *Proceedings of the Fifth Israel Symposium on the Theory of Computing Systems*, 1997.

Stephen Soderland. Learning to extract text-based information from the world wide web. In *Proc. of the 3th International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 251–254, 1997.

Stephen Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

Stephen Soderland, David Fisher, Jonathan Aseltine, and Wendy Lehnert. CRYSTAL: Inducing a conceptual dictionary. In Chris Mellish, editor, *Proc. of the 14th International Joint Conferences on Artificial Intelligence*, pages 1314–1319, San Francisco, 1995. Morgan Kaufmann.

Michael Steinbach, George Karypis, and Vipin Kumar. A comparison of document clustering techniques. In *Proceedings of KDD Workshop on Text Mining*, 2000.

Jun Suzuki, Yutaka Sasaki, and Eisaku Maeda. Kernels for structured natural language data. In *Neural Information Processing Systems*, volume 16, 2003.

Jun Suzuki, Yutaka Sasaki, and Eisaku Maeda. Kernels for structured natural language data. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, Cambridge, MA, 2004. MIT Press.

Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422 – 433, July 1979.

Sandeep Tata, Richard A. Hankins, and Jignesh M. Patel. Practical suffix tree construction. In *Proc. of 30th International Conference on Very Large Data Bases*, 2004.

Alexandre Termier, Marie-Christine Rousset, and Michél Sebag. Treefinder: a first step towards xml data mining. In *Proc. of the 2th International Conference on Data Mining*, Maebashi City, Japan, December 2002.

Bernd Thomas. Token-templates and logic programs for intelligent web search. *Journal of Intelligent Information Systems*, 14(2-3):241–261, 2000.

Bernd Thomas. Anti-unification based learning of t-wrappers for information extraction. In *Proc. of the Workshop on Machine Learning for Information Extraction*, 1999a.

Bernd Thomas. Logic programs for intelligent web search. In Zbigniew W. Ras and Andrzej Skowron, editors, *Proc. of Foundations of Intelligent Systems, 11th International Symposium (ISMIS)*, volume 1609 of *Lecture Notes in Computer Science*, pages 190–198, Warsaw, Poland, June 8-11 1999b. Springer. ISBN 3-540-65965-X.

Wolfgang Thomas. *Handbook of formal languages, vol. 3: beyond words*, chapter Languages, automata, and logic, pages 389 – 455. Springer-Verlag New York, 1997.

Esko Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 1995(14):249–260, 1995.

Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988. ISBN 0-7167-8158-1.

Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134 – 1142, 1984.

Gabriel Valient. An efficient bottom-up distance between trees. In *Proc. of 8th Symposium on String Processing and Information Retrieval*, pages 212–219, 2001.

Jean-Philippe Vert. A tree kernel to analyse phylogenetic profiles. *Bioinformatics*, 18: 276 – 284, 2002.

S.V.N. Viswanathan and Alexander Smola. Fast kernels for string and tree matching. In *Neural Information Processing Systems*, 2002.

W3C. Rdf schema specification 1.0. online, Mar 2000. http://www.w3.org/RDF/.

Igor Walukiewicz. Monadic second-order logic on tree-like structures. *Theoretical Computer Science*, 275(1-2):311 – 346, 2002.

Jason T. L. Wang and Kaizhong Zhang. Finding similar consensus between trees: An algorithm and a distance hierarchy. *Pattern Recognition*, 34(1):127–137, 2001.

Jiying Wang and Fred H. Lochovsky. Data extraction and label assignment for web databases. In *Proceedings of the twelfth international conference on World Wide Web*, 2003.

Ke Wang and Huiqing Liu. Discovering structural association of semistructured data. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):353–371, 2000.

Yalin Wang, Jianying Hu, and I. V. Ramakrishnan. A machine learning based approach for table detection on the web. In *Proc. of the 11th International WWW Conference*, 2002.

Yuan Wang, David J. DeWitt, and Jin yi Cai. X-diff: An effective change detection algorithm for xml documents. In *Proc. of 19th International Conference on Data Engineering*, pages 519–530, Bangalore, India, March 5-8 2003.

Peter Weiner. Linear pattern matching algorithms. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Gio Wiederhold. Mediators in the architecture of future information systems. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, pages 185–196. Morgan Kaufmann, San Francisco, CA, USA, 1997.

Khin-Myo Win, Wee-Keong Ng, and Ee-Peng Lim. Enaxs : Efficient native xml storage system. In *The 5th Asia Pacific Web Conference (APWeb2003)*, Xian, China, April 2003.

Jian Xu, Ee-Peng Lim, and Wee Keong Ng. Cluster-based database selection techniques for routing bibliographic queries. In *Database and Expert Systems Applications*, pages 100–109, 1999.

Yiming Yang, Sean Slattery, and Rayid Ghani. A study of approaches to hypertext categorization. *Journal of Intelligent Information Systems*, 18(2-3):219–241, 2002.

Mohammed J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the eighth ACM SIGKDD*, pages 71 – 80, Edmonton, Alberta, Canada, 2002. ACM Press.

Mohammed J. Zaki and Charu C. Aggarwal. Xrules: An effective structural classifier for xml data. In *Proc. of the 9th International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 316 – 325, 2003.

Klaus Zechner. A literature survey on information extraction and text summarization. Term paper, Carnegie Mellon University, 1997.

Kaizhong Zhang. A constrained edit distance between unordered labeled trees. *Algorithmica*, 15(1):205–222, 1996.

Kaizhong Zhang and Tao Jiang. Some max snp-hard results concerning unordered labeled trees. *Information Processing Letters*, 49(1):249–254, 1994.

Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245 – 1262, December 1989.

Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42(3):133–139, 1992.

Ying Zhao and George Karypis. Evaluation of hierarchical clustering algorithms for document datasets. In *Proc. of the 11th ACM CIKM International Conference on Information and Knowledge Management*, pages 515–524, 2002.

Moshé M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16 (4):324–343, 1977.

Index