A Study on Requirement Adaptable Browsing and Querying Schemes for Structured Documents

Doctoral Program in Engineering

University of Tsukuba

2001, March

SHINAGAWA Norihide

寄贈
品川徳秀氏
平成 年 月 日
01301633

Acknowledgments

I do not have the words to express my deepest gratitude to Professor Hiroyuki Kitagawa, my supervisor. I could not have begun my research if he had not accepted me as one of his students when I was new to this area. I discussed possible subjects for research with him, and I told him I took an interest in document processing and management. Then, he suggested that I focus on structured documents. Without his vision and guidance, I would not have studied the utilization of structured documents. He has given me constructive suggestions, precise criticism, and kind encouragement at all times. Because he always puts his heart into study and education, I have been able to enjoy the research fully.

I have been studying at the database laboratory of the University of Tsukuba. I am grateful to Assistant Professor Yoshiharu Ishikawa. He has given me instructive advice a number of times. I am also thankful to Professor Nobuo Ohbo and Assistant Professor Kiminori Utsunomiya for their constant

encouragement. I would also like to thank Doctor Atsuyuki Morishima and other members of the laboratory. I have asked them a great many things in my laboratory life; ever pleasant, they have always helped me.

Finally, I am grateful to the members of my dissertation committee for offering comments to improve this dissertation. Professor Hiroyuki Kitagawa, Professor Nobuo Ohbo, Professor Yoshihiko Ebihara, Professor Seiichi Nishihara, and Professor Mikio Yamamoto served on my dissertation committee.

Abstract

The rapid advances of world wide web (WWW) technology have made a huge amount of information available over the Internet. Information on the WWW is usually provided as structured HTML and XML documents. It has, therefore, become more and more important to use and manage these structured documents efficiently.

Both browsing and querying are generally used to access and identify useful information contained in a huge number of documents. Individual users have their own application-specific requirements, which should be respected in the querying and browsing environments. In this sense, requirement adaptability is a desirable feature for querying and browsing schemes.

When users browse documents, they have interests related to their aims. The WWW environment comprises a huge number of documents, some of which may be very large. This makes it difficult for users to identify sub-

structures and regions relevant to their interests. A facility is needed that makes it easy to identify descriptions that fit individual user requirements.

When a user issues queries to structured document databases, some queries need functions that depend on the structure and content descriptions of structured documents. Given access to these functions, a user can formulate queries in a more straightforward way. It is, however, very difficult to provide a complete set of functions that can cope with varied requirements. So users need query languages that can be extended to adopt user-defined functions.

This dissertation discusses the requirements for adaptable browsing and querying schemes for structured documents. It includes two subjects: (1) A scheme to support browsing WWW pages and identifying relevant descriptions within them, based on a user's interest, and (2) an XML query language called X^{2}QL featuring extensibility via user-defined foreign functions. The proposed schemes enable users to get needed information within structured documents in a requirement adaptable manner.

On the first subject, a browsing support scheme based on specification of the user's interest is studied. In this scheme, the user's interest is explicitly expressed as a user profile. Virtual WWW pages called *view-pages* tailored to the user's interest are then dynamically generated and presented

when browsing the WWW. View-pages essentially show summaries of existing HTML pages complying with user interests and the level of details specified in the user profile. With this scheme, users having different interests are given different views of the WWW.

We developed a prototype system that provides view-pages in a non-intrusive way in the current WWW browsing environment. This dissertation explains the architecture. The user can enjoy the proposed features using ordinary WWW browsers rather than resorting to special browsers tailored for this purpose.

On the second subject, an XML query language called X^2QL, which features the inclusion of user-defined *foreign functions*, is proposed. Foreign functions are provided as external programs written in programming languages. X^2QL provides the framework for implementing foreign functions. It defines the type system, internal object model and Java binding. By including appropriate user-defined foreign functions, individual users can extend the processing power of X^2QL. This extensibility makes it possible to integrate the processing facilities for application-oriented high-level contents into querying documents.

This dissertation also explains a scheme for implementing X^2QL query processing systems on top of XSLT processors. This involves a translation

scheme for X^2QL queries into XSLT template rules. We have already developed an X^2QL query processing system that follows the scheme.

Adaptability has become more and more important in accessing and using structured documents. The proposed schemes show promise in realizing the adaptability requirements for browsing and querying structured documents.

Contents

Acknowledgments i

Abstract iii

1 Introduction 1

2 Background 10

2.1 Structured Document 10

2.2 Vector Space Model 14

2.3 XML Query Language 16

2.3.1 XML-QL 17

2.3.2 XSLT 20

3 Related Works **26**

3.1 Supporting Schemes for WWW Browsing 26

3.2 Querying XML Documents 30

4 Browsing WWW Pages Based on User Profiles **35**

4.1 Overview 35

4.2 Logical Trees 39

4.2.1 Classification of HTML Tags 39

4.2.2 Derivation of Logical Trees 41

4.3 View-pages 44

4.3.1 Derivation of Feature Vectors 45

4.3.2 Generation of View-pages 47

4.4 Prototype System Development 52

4.4.1 System Architecture 52

4.4.2 Browsing Assistance Engine (BAE) 55

4.4.3 Typical Processing Flow 57

4.4.4 Session Example 59

4.5 Experimental Evaluation 61

4.5.1 Evaluation for Derivation of Logical Tree 62

4.5.2 Evaluation for Generation of View-pages 68

4.5.3 More Experiments in the WWW 73

4.6 Discussion and Future Issues 74

5 X^2QL: An Extensible XML Query Language 78

5.1 Overview 78

5.2 Motivating Examples 81

5.3 X^2QL: An eXtensible XML Query Language 83

5.3.1 Query Syntax 83

5.3.2 Foreign Functions 84

5.3.3 Query Specification Examples 86

5.4 Extensibility via Foreign Functions 88

5.4.1 Type System 89

5.4.2 Internal Object Model 90

5.4.3 Java Binding 92

5.5 Query Processing on XSLT Processors 94

5.5.1 System Architecture 94

5.5.2 Mapping of Foreign Functions 95

5.5.3 Query Translation 97

5.6 Advanced Features 102

5.6.1 Aggregation Functions 102

5.6.2 Variable Binding Filtering 106

5.7 Discussion and Future Issues 111

6 Conclusion **112**

References **114**

List of Tables

4.1 Bias factors β_n associated with style-oriented tags 48

4.2 Coefficients α_c for child nodes 48

4.3 Usage of style-oriented tags 64

4.4 Experimental result for IGD 66

4.5 Experimental result for W3C 66

4.6 Experimental result for DLIB 66

4.7 Averages for all sites 67

4.8 Experimental result for IGD 67

4.9 Experimental result for W3C 67

4.10 Experimental result for DLIB 67

4.11 Averages for all sites 67

4.12 Condition (a): Experiment result using uniform weights α_c . . 71

4.13 Condition (b): Experiment result using non-uniform weights α_c 71

4.14 Condition (c): Experimental result using different weights $\beta_{n,t}$ 71

4.15 Non-uniform weights α_c 72

4.16 Experimental result for keyword set A 75

4.17 Experimental result for keyword set B 75

4.18 Experimental result for keyword set C 75

4.19 Experimental result for keyword set D 76

4.20 Experimental result for keyword set E 76

4.21 Averages for all experimental results 76

4.22 Evaluation of scoring scheme 77

5.1 Translation rules between types 90

List of Figures

2.1 Sample XML document 13

4.1 Overview of the proposed scheme 37

4.2 Derivation rules (a) 42

4.3 Derivation rules (b) 43

4.4 Sample HTML document 44

4.5 Extracted logical tree 45

4.6 Example of view-page generation 51

4.7 HTML document for a view-page 51

4.8 Prototype system architecture 53

4.9 Architecture of BAE 53

4.10 Controller window 60

4.11 Browser window 60

5.1 Query processing scheme 95

5.2 Mechanism of a method invocation 96

Chapter 1

Introduction

The rapid advances of WWW technology have made a huge amount of information available over the Internet. Information on the WWW is usually provided as structured documents written in HTML and XML. XML, especially, has attracted a great deal of attention as a standard format for information exchange for WWW publishing, electronic commerce and storing application data. It has, therefore, become more and more important to use and manage these structured documents efficiently. A variety of schemes and tools addressing the issues have been investigated to date [1] [2].

Both browsing and querying are generally used to access and identify useful information in a huge number of documents. Individual users have their own application-specific requirements, depending on their purpose. Their re-

quirements should be respected by the querying and browsing environments. There can be a variety of requirements in their context, so the environments and their schemes should be adaptable to varied requirements.

When users browse documents to find descriptions related to their aims, they must access a number of documents in, for example, the WWW environment. Beyond that, some of those documents may be very large and may contain many unrelated parts. This makes it difficult for users to identify substructures and regions relevant to their interests, even if WWW search engines return interesting documents. A facility is needed that makes it easy to identify descriptions that fit individual user requirements.

When a user issues queries to structured document databases, some queries need functions that depend on the structure and content descriptions of structured documents. If the query language allows the use of such functions, the user can formulate such queries in a straightforward way. However, document structure is designed in response to the purpose, and contents are written in natural language. This diversity makes it very difficult to provide a complete set of functions that will cope with the varied requirements. Users need query languages that can be extended to adopt user-defined functions.

This dissertation discusses the following two subjects.

1. A supporting scheme for WWW browsing based on the user's interest [3] [4]
2. An XML query language, called X^2QL, featuring extensibility via user-defined foreign functions [5]

The schemes proposed in this dissertation enable users to get needed information within structured documents in a way that is adaptable to varied requirements.

Browsing WWW Pages Based on User Profiles

As mentioned, when browsing WWW pages, it is difficult for users to identify substructures and regions relevant to their interests. The proposed scheme in this subject allows users to identify these substructures and regions within each WWW page written in HTML by providing *view-pages*. View-pages consist of substructures relevant to their requirements and are dynamically generated from specified WWW pages. Each user's requirements are represented as a *user profile*, which contains a keyword set and a detail level of view-pages; it is used to generate view-pages.

Documents generally have hierarchical logical structures comprising meaningful blocks of contents. To provide informative descriptions for in-

dividual users, view-pages should contain few descriptions irrelevant to user requirements, and their original contents and logical structures should be respected. Providing automatically generated mini-summaries is not always suitable to our purpose, which is to help users browse individual WWW pages and roughly understand their contents. This is done rather than have the user select pages from many candidate pages. View-pages are generated by pruning away irrelevant substructures from specified WWW pages based on their logical document structures. Note that generated view-pages must be valid HTML documents. Our procedures to prune away irrelevant substructures and to generate HTML documents ensure validity.

In structured documents, logical structures are indicated by markups embedded as tags. However, HTML tag hierarchies do not always coincide with a document's logical structure. HTML tags construct flat structures and some are used to indicate physical presentation features rather than logical structures. To overcome this shortcoming, we introduce *logical trees*, which represent logical structures. They are derived by derivation rules based on HTML tags. The derivation rules are applicable to most HTML documents.

Relevance of each substructure to a user requirement can be judged using methods of information retrieval. This dissertation uses the vector space model. Each node score of the logical tree is calculated by the similarity of its contents and the keyword set in a given user profile. The score is then

compared with the detail level given by the user profile. Note that some HTML tags are used to emphasize enclosing phrases, and some substructures play important roles in the documents. We take these facts into account when calculating the node score.

A prototype system implementing our scheme has been developed. In this development, we take into account the following points. (1) Users can use ordinary WWW browsers to browse view-pages. It is not easy to develop a new fully featured WWW browser. And even if we could build one, users have a right to use their favorite WWW browsers. Therefore, we provide view-pages through an HTTP proxy server featuring view-page generation. (2) Users can interactively and easily control view-page generation. When view-pages are provided for individual users, they do not necessarily fit their requirements, because appropriate detail levels generally depend on browsed pages. Moreover, it is desirable that a user can browse each page at various detail levels. So we also display the logical tree of the page and its node scores visually using a Java applet. This applet offers clues that allow users to modify their user profiles.

Finally, we experimentally evaluate the proposed scheme against three criteria. (1) First, we evaluate validity of the logical trees derived from each given page. The logical trees must have an appropriate structure to extract substructures for inclusion in view-pages. (2) Second, we evaluate correctness

of the node scores in a given logical tree. When the given logical tree structure and calculated node scores are correct, the generated view-page can contain only the appropriate substructures within the original page by adjusting the detail level of view-pages. (3) Third, we applied the above two evaluation methods to more general WWW pages. In this experiment, we used WWW pages searched by the Goo search engine. We can check effectiveness of the proposed scheme in a more realistic context of WWW page browsing.

X^2QL: An Extensible XML Query Language

XML has attracted a great deal of attention as a standard format for information exchange, and the number of XML documents has been increasing. It has therefore become more and more important to use and manage these structured documents efficiently. Query languages for XML documents play important roles, just as they do in the context of traditional databases, and many XML query languages have been developed.

XML allows users to define their own document structures, and their structures generally introduce data structures. It should be possible, therefore, for structure-dependent functions to be used in queries. Additionally, their contents are written in natural languages, so some queries require content processing functions. Some examples are similarity-based selection,

ranking, automatic summary generation, and other content processing functions. Thus, it is very difficult to provide a complete set of such functions to cope with the varied requirements. This is why XML query languages should be extensible. When we started investigating this subject, there were no XML query languages with excellent extensibility.

This dissertation proposes extensible XML query language X^2QL, which features the inclusion of user-defined *foreign functions*. X^2QL is based on XML-QL, which is the best-known XML query language featuring powerful, tag-based document structure manipulation. We designed X^2QL using XML-QL as a place to start. In X^2QL, foreign functions are written in general programming languages, and the current version of X^2QL supports Java binding.

First, we introduce two kinds of foreign functions: *general functions* and *element methods*. General functions are normal functions: they compute return values from given argument values. Element methods, on the other hand, are associated with element types. Each element method is invoked for an element and its execution depends on the element itself. Element methods enable object-oriented features. To define and implement foreign functions, we define data types that represent pieces of XML documents. Individual element types can also be regarded as data types. Therefore, we treat elements as *element objects* in query processing, and provide interfaces

to invoke element methods and to store objects created by the implementation as *properties.* These interfaces make it possible to treat elements as stateful objects. We then define the mapping of types into Java interfaces and invocation mechanisms of foreign function implementations.

We have developed an X^2QL query processor on top of an XSLT processor. XSLT is a popular XML transformation language, and several XSLT processors are available. The rationale for our implementation approach is three-fold. (1) The approach contributes to rapid development of the query processor. We have been able to build our query processor by developing foreign function invocation mechanisms and query translation into XSLT template rules. (2) An implementation on top of XSLT processors assures a certain level of portability. XSLT allows its processors to have implementation-dependent features, for instance, extension functions. We need them to implement foreign functions on top of XSLT processors. These features are abstracted by the adopter module for each XSLT processor. (3) X^2QL can work as a front end to XSLT processors. XSLT specifications are low-level and procedural, and they are difficult for novice users.

Finally, we introduce advanced features: *aggregation functions* and a *variable binding filtering mechanism.* We call a *tuple* for each set of variable bindings, which is generated by a `where` clause and consumed by a `construct` clause in query processing. Aggregation functions are foreign

functions, which compute a value from multiple tuples, for instance, the maximum, minimum or average value. The variable binding filtering mechanism, on the other hand, is used to control the tuple sequence. This mechanism enables user-defined re-ordering, selection, and other activities, by using foreign functions called *filtering functions.* Java binding of these functions is defined and our implementation approach makes it possible for these features to be implemented using XSLT processors.

Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes the background of the research. Structured documents are explained. The vector space model is then explained in the context of information retrieval. Finally, XML processing languages are described. The description includes XML-QL and XSLT, which are XML query and translation languages. Chapter 3 surveys related works. Chapter 4 explains our supporting scheme for WWW browsing. This includes logical tree derivation, view-page generation, prototype system development, and experimental evaluations. Chapter 5 describes X^2QL. This involves the query syntax, foreign functions, the development of query processor on top of XSLT, and advanced features. Finally, Chapter 6 concludes the dissertation.

Chapter 2

Background

This chapter first explains structured documents. Second, the vector space model for information retrievals is explained. The chapter then gives simplified overviews of XML-QL and XSLT, which are languages to process XML documents.

2.1 Structured Document

In general, documents such as news articles, technical papers and books have internal structures. For example, a news article consists of its headline, date, category, body containing several paragraphs, and so on. To make possible the exchange of structural information as well as text, SGML (Standard

Generalized Markup Language) was developed in 1986 [6]. SGML documents have two main parts: The first part is the *DTD* (Document Type Definition), which lists rules for the logical structure of the document. The second part is tagged text, which conforms to the DTD. SGML received a boost in popularity when the U.S. Department of Defense adopted it as the standard for military documentation. SGML was also adopted as the basis of the Hytime hypermedia description language [7]. However, the best-known application of SGML is HTML (Hyper Text Markup Language), which is the language for coding WWW pages [8] and has been adopted by W3C (World Wide Web Consortium). HTML documents are SGML documents with inner document structures coded against a predefined DTD. In 1997, XML (Extensible Markup Language) was developed as a next generation language for Web page coding [9]. XML is a simplified version of SGML, and allows Web pages to have user-defined document types. In 2000, HTML was reformulated as an XML application and named XTHML (Extensible HTML) [10].

Figure 2.1 shows a sample XML document. The text surrounded by "<!DOCTYPE *root*[" and "]>" in the document is the DTD. It is followed by the tagged text part. Tagged text is divided into *elements* enclosed by a *start tag* "<*tag*>" and an *end tag* "</*tag*>", where "*tag*" is a *generic identifier* representing the *element type*. Elements can have *attributes*, which consist of the *attribute names* and *attribute values*. They are described as the form

"*name*="*value*"" in start tags. Elements can be nested within other elements. The DTD prescribes how the elements can be hierarchically constructed by child elements. The root element is specified by the *root* in the DTD. XML allows instance documents without DTDs. Such documents are called *well-formed.* Documents with a DTD are called *valid.*

In Figure 2.1, each line starting with "<!ELEMENT" in the DTD is an *element type definition.* It defines an element name and its *content model,* which is the rule for content structure of the element type. The root `Document` element consists of one or more `Article` elements. An `Article` element is a sequence of `Date`, `Category`, `Location`, `Headline` and `Body` elements. Note that the `Location` element is optional. Additionally, each `Article` element has an attribute named `AID`, whose values give identifiers of `Article` elements. A `Date` element is also a sequence of `Year`, `Month` and `Day` elements. The `Body` element consists of one or more `Paragraph` elements. The element types such as `Year`, `Month`, `Day`, `Category`, `Headline` and `Paragraph` are defined as having no internal structures.

In structured documents, structural irregularity is caused by content models with repetitions by "+" and "*", optional structures by "?", and selections by "|". Thus, they are often said to be semi-structured data [11] [12].

```
<?XML version="1.0" encoding="UTF-8" MD="ALL"?>

<!DOCTYPE Document [
      <!ELEMENT Document  Article+>
      <!ELEMENT Article  (Date, Category, Location?,
                          Headline, Body)>
      <!ATTLIST Article   AID  ID  #REQUIRED
      <!ELEMENT Date     (Year, Month, Day)>
      <!ELEMENT Year      #PCDATA>
      <!ELEMENT Month     #PCDATA>
      <!ELEMENT Day       #PCDATA>
      <!ELEMENT Category  #PCDATA>
      <!ELEMENT Location  #PCDATA>
      <!ELEMENT Headline  #PCDATA>
      <!ELEMENT Body      Paragraph+>
      <!ELEMENT Paragraph #PCDATA>
]>

<Document>
  <Article AID="20000101-001">
    <Date>
      <Year>2001</Year> <Month>1</Month> <Day>1</Day>
    </Date>
    <Category>Financial</Category>
    <Headline>...</Headline>
    <Body>
      <Paragraph> ... </Paragraph>
      <Paragraph> ... </Paragraph>
      ...
    </Body>
  </Article>
</Document>
```

Figure 2.1: Sample XML document

2.2 Vector Space Model

In information retrieval, documents are compared with each other or given search requests, called *queries*, and most similar documents are obtained. The *vector space model* is the best-known scheme for information retrieval systems [13].

In the vector space model, documents are located in a *document space*, and they are represented as vectors from the origin to their locations. The axes of the document space correspond to document features. Generally, features are given by the weights of indexed terms. A query given by a keyword set is represented as a feature vector in the same way. A feature vector V_d of document d is formulated as follows:

$$V_d = (w_{d,t_1}, w_{d,t_2}, ..., w_{d,t_n})$$

where each t_i is an indexed term and w_{d,t_i} is its weight in d.

The most standard method of giving feature weights is *tf·idf*. In this method, the weight $w_{d,t}$ of t in d is given by the following expressions:

$$w_{d,t} \;=\; tf_{d,t} \cdot idf_{d,t}$$

$$idf_t = \log\left(\frac{N}{n_t + 1}\right)$$

where $tf_{d,t}$ is the number of occurrence of the t in d, called the *term frequency factor*, N is the number of all documents, and n_t is the number of documents including t. The idf_t is called the *inverse document frequency factor.*

The similarity between a document d and a given query q is calculated by the *(standard) cosine measure*:

$$similarity(d, q) = cosine(V_d, V_q) = \frac{V_d \circ V_q}{||V_d|| \cdot ||V_q||}.$$

The range of the cosine measure is 0 to 1, and it gives the highest similarity value of 1.0 to the same vectors.

In the standard cosine measure, the lengths of vectors are normalized to 1.0 so that the size of each document is not explicitly taken into account. Therefore, when the collection contains documents of different length, smaller documents tend to have higher similarity values [14]. To deal with this problem, Singhal and others proposed the *C-pivot* measure [15]. According to the C-pivot measure, the similarity between the d and q is calculated as

follows:

$$similarity(d,q) = \frac{V_d}{(1-S)\cdot L + S\cdot U_d} \circ \frac{V_q}{\|V_q\|}$$

where L is the average number of terms in the documents, U_d is the number of distinct terms in d, and S is a slope constant in $[\,0,1\,]$. The slope constant is decided empirically.

2.3 XML Query Language

In database management systems, query languages such as SQL [16] and OQL [17] have been used to access databases. Query languages make it easy to specify required data in databases and to access it using declarative and powerful expressive power. Query languages have been proposed for XML documents and their databases, because it has become more and more important to use and manage these documents efficiently.

This section gives a simplified overview of XML-QL [18], then XSLT [19]. XML-QL is the best-known XML query language and the basis of X^{2}QL. XSLT is the most widely used XML transformation language. XSLT is a low-level and procedural language to process XML documents and is used by our X^{2}QL query processor.

2.3.1 XML-QL

XML-QL [18] is an XML query language proposed to W3C by A. Deutsch and others in 1998. It allows declarative descriptions to extract data from XML documents and integrate them into a resulting document.

XML-QL Query

The basic syntax of an XML-QL query as follows.

```
where       patterns [in source] [, patterns [in source]]*
            [, predicate]*
[order-by   ordering keys [descending]]
construct   construction of each output
```

For example, given the sample XML document in Section 2.1, the query to generate a new XML document in which articles published after 1999 are selected and grouped by the categories can be specified as coded below.

```
where     <Document> </> content_as $x
construct <Document>
  where     <Article>
              <Date> <Year> $y </> </>
              <Category> $c </>
            </> element_as $a in $x,
            $y >= 1999
  order-by  $c
  construct <Group ID=CtgID($c)> $a </>
          </>
```

Where clause: This clause specifies the element patterns, predicates, and variable bindings. Element patterns are given by XML-like forms. Each variable is bound to either an element, the content of an element, an element name, an attribute name, or an attribute value. The expression `element_as $x` binds the variable `$x` to the preceding element, and the expression `content_as $x` binds `$x` to the content of the preceding element. The expression `content_as $x` is abbreviated to the variable `$x` enclosed by tags.

In the above example, `$x` is bound to the content of a `Document` element in the first `where` clause. Variables `$a`, `$c`, and `$y` are bound to an `Article` element, the content of a `Category` element in the `Article` element, and the content of a `Year` element in the `Date` element, respectively, in the second `where` clause. Moreover, the `Year` value must be greater than or equal to 1999.

Construct clause: This clause specifies how to construct an output element for each set of bound variables. The query result is a sequence of output elements. As shown in this example, the `where···construct` clauses may be nested.

Each output element can be given its `ID` attribute value by a Skolem function. The Skolem function gives one-to-one mapping from a set of arguments to an `ID` attribute value. In the query result, elements that have the same element name, parent node, and `ID` value are grouped into a single element. This feature can be used to group elements.

The above query returns a `Document` element that contains the result of the subquery. The subquery returns a sequence of `Group` elements. Each `Group` element has a one-to-one correspondence to a `Catagory` value, and contains `Article` elements of the category. Each `Article` element is given by $a.

Order-by clause: This clause specifies the ordering of output elements. They are sorted by the sort key. In the above query, the output elements are sorted in the ascending order of the value of `$c`, namely the `Category` value.

Function Definition

XML-QL supports functions, which are canned queries with arguments. Function are coded as follows:

```
function function-name( argument-list )
  XML-QL query
end
```

Each function is a query that can include unbound variables given in function arguments. The return value of a function is the result of the query.

2.3.2 XSLT

XSLT [19] is an XML transformation language for XSL (XML stylesheet language) [20] and a W3C recommendation. Several XSLT processors are available, and their use has become popular in the context of XML document processing.

Template Rules

A transformation expressed in XSLT is an XML document, called a *stylesheet*, whose root element is `xsl:stylesheet` containing a set of *template rules*. We explain template rules by describing how document manipulation using the following X²QL query is expressed in XSLT. This query extracts `Item` elements whose `Number` element values are less than 100. Its result is a `Document` element consisting of `Item` elements sorted by the `Number` value.

```
where      <Document></> content_as $d
construct <Document>
  where      <Item> <Number> $n </> </> element_as $i in $d,
             $n < 100
  order-by  $n
  construct $i
          </>
```

A template rule corresponding to the query is as follows:

```
<xsl:template match="/Document">
  <xsl:variable name="d" select="."/>
  <Document>
    <xsl:for-each select="Item/Number[ . < 100 ]">
      <xsl:sort     select="."/>
      <xsl:variable name="n" select="."/>
      <xsl:for-each select="..">
        <xsl:variable name="i" select="."/>
        <xsl:copy-of  select="$i"/>
      </xsl:for-each>
    </xsl:for-each>
  </Document>
</xsl:template>
```

Elements whose tags are in the namespace `xsl` are called *instructions.* A template rule is represented as a nested structure of instructions. Each XML document is modeled as a tree whose nodes are elements, attributes, text (#PCDATA), and so on. Each instruction in the body of a template rule is applied to the nodes selected by a path expression called *location path* [21] (for example, "`/Document`", "`.`", and "`Item/Number[.<100]`"). The selected nodes and their set are called *context nodes* and *context node list.*

Location paths may be relative or absolute. Relative location paths are evaluated on the basis of the current context node. Location paths that appear in this paper are as follows: (1) "." selects the current context node itself, (2) ".." selects the parent node, (3) `a/b` selects the child `b` nodes of `a`, (4) `a/@b` selects the attribute `b` nodes of `a`, and (5) `a[x]` selects `a` nodes that satisfy the condition `x`. The condition `x` is given by location paths and

predicates combined by `and`/`or`. Note that these are abbreviated notations; more sophisticated location paths can be specified. Those notations are not explained. The following briefly explains the major instructions. It then interprets the above template rule.

`xsl:template` A template rule is an element whose tag is this instruction. The `match` attribute specifies a location path to identify the target nodes to which the rule applies. When an element matches the location path of a template rule, the content of the rule, which is a sequence of instructions, is instantiated as the output.

`xsl:variable` This instruction is used to bind a variable. The variable name is given by the `name` attribute, and its value is given as the content of the element or the location path specified in the `select` attribute. In XSLT, the value of a variable is denoted by its name with a prefix "$".

`xsl:for-each` This instruction is used to select the context nodes specified by the `select` attribute, and to construct the output for each context node similarly to `xsl:template`.

`xsl:sort` This instruction is used to sort the context nodes. It must occur first in an `xsl:for-each` element.

`xsl:copy-of` This instruction is used to copy a node set selected by the `select` attribute.

`xsl:if` This instruction is used for conditional processing. When the condition given by the `test` attribute is true, the content is instantiated.

When the template rule given at the beginning of this subsection is applied to a `Document` element in a source XML document, a `Document` element is created as an output. It contains the result generated by other instructions in the template rule. The first `xsl:for-each` instruction selects `Number` elements whose values are less than 100, and the `xsl:sort` instruction sorts them. The second `xsl:for-each` instruction then selects the parent `Item` element of the current `Number` element, and the `xsl:copy-of` instruction copies the current `Item` element. The result document is what the X^2QL query specifies.

Extension Functions

The XSLT specification mentions the use of *extension functions* [19]. These functions are external programs written in programming languages. However, the availability and the usage of extension functions depend on the underlying XSLT processor, because their details are left to the implementation.

In this study, we use LotusXSL [22], which is an XSLT processor developed by IBM alphaWorks. The current version of LotusXSL is a wrapper

library for Xalan [23] developed by the Apache XML project. Extension functions in LotusXSL are defined by `lxslt:component` elements as follows.

```
<lxslt:component prefix="namespace"
       functions="list of the extension functions">
  <lxslt:script lang="javaclass" src="URI"/>
</lxslt:component>
```

Example: Suppose we define an extension function `head()` in the namespace `my-space`. It is implemented as a method of `MyHead` class in Java. It returns the head part of the given string whose length is less than or equal to the specified length.

```
<lxslt:component prefix="my-space" functions="head">
  <lxslt:script lang="javaclass" src="MyHead"/>
</lxslt:component>
```

In template rules, we can call extension functions, for example,

```
<xsl:value-of select="my-space:head( string(.), 10 )"/>.
```

Chapter 3

Related Works

3.1 Supporting Schemes for WWW Browsing

In the WWW environment, to utilize a huge amount of WWW pages, users search candidate pages relevant to their interests, browse them, and get relevant descriptions within them. A variety of schemes and systems to help interactive browsing of the WWW, or in context of digital library and information retrieval, are proposed. By their aims, they can be classified into supports for (1) selecting some documents from a number of searched documents, (2) understanding of topics within each document, (3) identification of relevant parts within each document, and (4) hyperlink navigation to other pages. Most of supporting schemes are classified into (1) or (4). Our pro-

posed scheme is classified into (3).

Schemes in category (1) give support to select relevant documents from many candidate documents. An approach is to give a short summary to each document in the list of searched results, and used in many search services, e.g. AltaVista [24], Google [25] and Yahoo [26]. Another approach is to visualize the document space by associations and clusters of their documents, e.g. Information Visualizer [27], VIBE [28], TileBars [29], Envision [30] and Scatter/Gather [31]. These summaries and visual clues enable users to select easily relevant documents without browsing. However, these schemes do not consider user requirements, or do not change the difficulty to identify relevant descriptions within each document.

Schemes in category (2) give support for rough understanding of topics within individual documents in the browsing process. For a document, these schemes provide an informative summary including enough sentences to represent almost the topics in the document. When summaries are used in this process, it is also important that they should be generated dynamically and automatically whenever new pages are fetched. Miike and others developed an information retrieval system which automatically generates the summary of each document in a document repository [32]. In this system, the detail levels of the summaries can be changed interactively.

Our proposal is to give some support in category (3), namely, support for identification of relevant descriptions within each document, in WWW browsing process. To the best of my knowledge, there are no known proposal classified directly into this category. Some above approaches use automatic summarization techniques [13] [33] [34]. Most of them do not take the user's interest into consideration, so that the summaries are static. Such supports make it easy to narrow candidate documents through rough understanding with/without browsing. However, they do not also change the difficulty to identify relevant substructures. User requirements are important to get relevant descriptions, in particular, in the browsing process. To consider user requirements, we can use passage retrieval [35] [36] [37] [38], and query-biased summary generation [39] [40].

In our scheme, scores of substructures in a document are measured. In this sense, our approach has some similarity with the passage retrieval. However, works on the passage retrieval do not explicitly consider the browsing process. We should more interactively and dynamically provide relevant substructures. Moreover, they usually consider only simpler substructures such as sentences, paragraphs and blocks of fixed length. In our scheme, more complicated substructures are dynamically extracted in our document model.

In HTML documents, their tag hierarchies are not enough to represent

their logical structures. Therefore, some schemes are proposed [41] [42] [43] [44] [45] [46] in context of wrapper generation for WWW pages. They extract their logical structure and contents. Ashish and others derive hierarchical logical structures from HTML documents [41]. They also pay attention to headings of chapters, sections and so on, in the same way with our logical tree derivation rules. Note that their "headings" are not given by heading tags `H1`, `H2`, ..., `H6` in HTML, but rendered as single lines hilighted by `B`, `STRONG`, and so on. The work of [42] extracts only data records by heuristic rules. Wrapper Induction [43] [44] and NoDoSe [45] [46] provide more generalized frameworks by pattern matching rules. Their rules generally depend on targets, and can be applied to only limited pages.

Our schemes provide view-pages, which contain just relevant substructures by pruning irrelevant substructures away. View-pages can be regarded as summaries of existing HTML documents based on user requirements. Query-biased summary generation methods also provide requirement-oriented summaries. However, they only consider flat text, and their viewpoint does not focus on the WWW browsing process. Moreover, they do not consider the case that the targets are HTML documents. To browse summaries on ubiquitous WWW browsers, the summaries must be valid HTML documents. Note that view-page generation may be improved by incorporating query-biased summary generation.

Finally, there are schemes in category (4). There are a number of works to support the hyperlink navigation in the hypertext browsing. WebWatcher [47], Letizia [48], and Syskill and Webert [49] have a feature to suggest hyperlinks which are likely to lead to the proper destinations. WebWatcher and Letizia automatically acquire the user's interest from the navigation patterns. Syskill and Webert offers hyperlinks by using the ratings of hyperlinks given manually to learn a user specific topic profile. Lisa [50] collects WWW pages related to the user's interests and enables him to browse the WWW off line. Such features may work well with our scheme.

3.2 Querying XML Documents

Several languages have been proposed for manipulating XML documents. They can be classified into query languages and more general programming languages [2] [51] [52]. Programming languages can process intricate transformation of XML documents, and have flexible expressive power. However, their programs can be complicated. On the other hand, query languages allow more declarative and simple descriptions. Most of them generally have few extensibility. X^2QL is categorized as a query language and has rich extensibility via foreign functions.

There are several query languages which can be applied to XML docu-

ments. Some of them are query languages for XML documents. Most of the remainder are query languages for semi-structured data, and they model XML documents as semi-structured data.

Well-known proposal of XML query languages are XPath [21], XQL [53], XML-QL [18] and Quilt [54]. XPath and XQL are location path languages. XPath have been developed as a part of XSLT, and XQL has been proposed as an extension of previous version of XPath. These are very weak in document structure manipulation, and their queries simply select nodes satisfying the given path expression. They have no extension mechanisms. Because of this simplicity, they are often used in combination with other languages, and as a basic query language in commercial XML servers such as Tamino [55] and eXcelon [56].

XML-QL [18] is a query language for XML documents, and it is used as a basis of X^2QL. In the current version of XML-QL, user-defined functions are just canned queries, as explained in Subsection 2.3.1. The proposal mentions the possible use of user-defined predicates as a future research issue. However, their scope, role, syntax, and semantics are not specified. Foreign functions in X^2QL can represent user-defined predicates. Furthermore, they can be used for a wide range of applications, and can return elements and contents.

Quilt [54] has recently emerged from designing experiments of XML query

languages. It has been designed by borrowing useful features from SQL [16], XPath, XML-QL and OQL [17], and also influenced by Lorel [57] and YATL [58]. Therefore, Quilt has powerful structure manipulation facility. It also has extensibility by user-defined functions, and competes with X^2QL. However, the detail of its extensibility has not been explained yet. Note that Quilt had not appeared when we started to develop X^2QL.

The second category of query languages includes YATL [58], Lorel [57], UnQL [59], and so on.

YATL is a declarative and rule-oriented language for YAT semistructured data model [58]. YATL has a powerful structure manipulation capability based on pattern matching facilities. It supports graphical interface to write queries and composing them. The authors mention the use of user-defined external functions. However, YATL is not dedicated to XML document processing, and their usage of foreign functions is limited to string-based pattern matching.

Lorel [57] [60] is a query language for OEM data model [11] [61], a graph-based semistructured data model. XML documents are represented in OEM and processed. Lorel can be regarded as an extension of OQL [17]. Its data restructuring capability is very limited. UnQL [59] [62] is a query language for a semistructured data model similar to OEM. UnQL has a powerful data

restructuring capability by applying template rules recursively. However, the use of foreign functions is not considered.

Examples of XML programming languages are XSLT [19] and XDuce [52]. XSLT is a low-level and procedural XML transformation language recommended by W3C. XSLT is designed to be used independently of XSL (XML Stylesheet Language) [20]. However, XSLT is not intended as a complete general-purpose XML transformation language, and allows two kinds of extension: extension functions and extension elements. These are implemented by general programming languages such as Java, as is the case with foreign functions of X^2QL. However, implementation of extensions depends on the individual XSLT processor.

XDuce [52] is a statically typed functional programming language for transformation of XML documents. Is is based on Haskell but specialized to the domain of XML processing. Its novel features are regular expression types and a corresponding mechanism for regular expression pattern matching. XDuce can flexibly process XML documents by regular expressions and programmable features.

XML query languages have different expressive power and underlying data models. To establish their foundation, W3C Query Working Group has been discussing XML Query Requirements [63], XML Query Data Model [64]

and XML Query Algebra [65]. XML Query Requirement includes desirable features of XML query languages. XML Query Data Model formally defines the information contained in the input to an XML Query processor. This is based on XML Information Set [66], which provides a description of the information available in a well-formed XML document. It also supports data types in XML Schema [67] [68] [69], which extends an instance of the XML Information Set with more precise type information. XML Query Algebra introduces a formal basis for an XML query language based on XML Query Data Model. X^2QL may have some features which do not precisely coincide with their foundation. X^2QL should support them in the near future.

Chapter 4

Browsing WWW Pages Based on User Profiles

4.1 Overview

The rapid advances of world wide web (WWW) technology have made a huge amount of information available over the Internet. The number of available WWW pages is continually increasing, making it difficult to find and use relevant information. A variety of schemes and tools have been investigated to address this problem [1] [41] [70].

Both querying and browsing are generally required to access and identify relevant WWW pages. In a typical scenario, the user first submits queries

to search engines that return a set of candidate WWW pages. The user then selects and browses candidates from the set returned. The user may then need to browse other linked pages and go back to previous pages. In this way, querying and browsing are used iteratively to reach WWW pages relevant to the user's interest.

The interest of a browsing user is usually implicit, while queries are explicitly expressed. This means that existing WWW pages are usually presented exactly in the same format and layout to all users, even if the users have different interests. It then becomes the user's responsibility to understand the structure and semantics of WWW pages and to identify relevant substructures and regions. This is not, however, an easy task. In the WWW environment especially, the user has to manipulate many documents, most of which are unfamiliar. Beyond that, each WWW page sometimes contains a large amount of information. Most WWW browsers, of course, provide a string search facility to help the user. Simple string matching, however, is not enough to help a user understand page contents. Moreover, a string search facility does not change the environment in which every user is given the same page presentations.

This chapter proposes a new scheme to support the browsing process based on specifications of user interest. In our scheme, the user's interest is explicitly expressed as a user profile. Virtual WWW pages named *view-pages*

tailored to the user's interest are then dynamically generated and presented when the user browses the WWW. View-pages essentially show summaries of existing HTML pages that comply with the user's interest and level of details specified in the user profile. Under this scheme, users of different interests are given different views of the WWW. In the view-page generation, a document model called a *logical tree* is used to model the logical structure of an HTML page and plays an important role. This chapter also shows how view-pages can be achieved in a non-intrusive way in the current WWW browsing environment. With this approach, the user can enjoy the proposed feature on ordinary WWW browsers rather being compelled to use special browsers tailored to this purpose.

Figure 4.1 illustrates our approach.

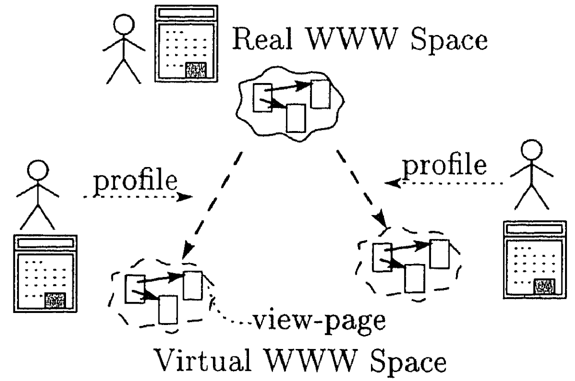


Figure 4.1: Overview of the proposed scheme

In our environment, the user's interests are explicitly specified as a user profile. A user profile consists of keywords to represent the user's interest and

a threshold value that controls the detail level of the presentation. Whenever the user accesses WWW pages, view-pages are generated dynamically based on the user profile for presentation. The pages may contain hyperlinks referring to other WWW pages. This makes it possible for the user to navigate through the virtual WWW space. Each view-page is essentially a summary of the original WWW page from the standpoint of user interest. A page is generated as follows.

(1) We assume that WWW pages are written in HTML. In the first phase, the logical document structure embedded in the target HTML page is extracted, and is modeled in a document model called a logical tree. HTML pages contain tags, some of which are used to construct the logical trees. However, as pointed in [41], the logical document hierarchy does not always coincide with the tag hierarchy.

(2) Similarity between each node in the logical tree and the user profile is calculated.

(3) We mark all nodes whose similarities fall below the threshold value in the user profile. A view-page is constructed by pruning away subtrees whose nodes and ancestors are all marked. This phase must ensure that view-pages are also valid HTML documents.

The remaining part of this chapter is organized as follows: Section 4.2 introduces a logical tree to put logical hierarchies on HTML pages. Section 4.3 presents a scheme to generate view-pages based on a user profile. The logical tree in Section 4.2 is used in the view-page generation. Section 4.4 shows the system architecture to implement the proposed scheme non-intrusively in ordinary WWW browsing environments. Section 4.5 shows experimental evaluations. Finally, Section 4.6 summarizes and discusses the chapter.

4.2 Logical Trees

This section explains derivation of logical trees from HTML pages. For this purpose, first, HTML tags are classified in Subsection 4.2.1. Then, is is shown how tags in some group are used to derive logical trees in Subsection 4.2.2.

4.2.1 Classification of HTML Tags

HTML tags are classified into the following four groups.

a) Structure-oriented tags: Tags in this group are primarily used to indicate document structures such as headings, paragraphs and lists of items. This group includes the following tags: `H1`, `H2`, `H3`, `H4`, `H5`, `H6`, `P`,

BLOCKQUOTE, DIV, UL, OL, DL, TABLE. Generally, it is not easy to extract logical document structures from HTML documents, but tags in this group give clues to extract them. As a matter of fact, they are used to construct the logical tree as explained in Subsection 4.2.2.

b) Style-oriented tags: Tags in this group are mainly used to give special presentation effects. The tags STRONG, EM, TT, I, U, B, BIG, SMALL, STRIKE, S, FONT, DL are included in this group. Note that some of them are deprecated in HTML 4.0 [8], which recommends us to use style sheets instead. Elements enclosed by tags in this group are given predetermined weights when deriving feature vectors of document substructures as explained in Subsection 4.3.1.

c) Media-oriented tags: Tags in this group are used to embed media objects such as images and applets. They include the following tags: IMAGE, FORM, SCRIPT, APPLET, OBJECT, EMBED, MAP. They are ignored in the construction of logical trees. The contents of embedded media objects are also ignored in the derivation of feature vectors.

d) Miscellaneous tags: The other tags are ignored in the construction of logical trees. Existence of tags in this group have no effects on the derivation

of feature vectors, and elements enclosed in those tags are treated as if they are ordinary character strings.

4.2.2 Derivation of Logical Trees

This subsection describes derivation of a logical tree from an HTML document. Structure-oriented tags are used as important clues in the derivation. In some context, similar schemes to derive logical structures inherent in HTML documents are suggested (e.g. [41]).

A logical tree consists of the eight types of nodes: ≪*doc*≫, ≪*desc*(L)≫, ≪*leading*(L)≫, ≪*trailing*(L)≫, ≪*packed*(L)≫, ≪*block*(L)≫, ≪*heading*(L)≫, ≪*paradiv*≫, ≪*paragraph*≫. L represents the hierarchy level of the node, and the values correnpond to the heading levels of `H1`, ..., `H6` tags. Given an HTML document, it is parsed according to rules in Figures 4.2 and 4.3, and then, the logical tree is obtained. After this, "(L)" is omitted when it is obvious.

The ≪*doc*≫ node indicetes whole the document. A ≪*desc*≫ node corresponds to a sequence of substructures such as chapters and sections, and ≪*leading*≫ and ≪*trailing*≫ nodes represent individual substructures. A ≪*heading*≫ node corresponds to the heading of a ≪*trailing*≫ node. Especially, a ≪*leading*≫ node represents a substructure without headings, which

$$
\begin{array}{rcl}
\ll doc \gg & \rightarrow & \texttt{<BODY>} \ll desc(1) \gg \texttt{</BODY>} \\
\ll desc(L) \gg & \rightarrow & \ll leading(L) \gg \ll trailing(1) \gg + \\
 & | & \ll trailing(L) \gg + \\
\ll leading(L) \gg & \rightarrow & \ll block(L) \gg \\
\ll trailing(L) \gg & \rightarrow & \texttt{<DIV>} \ll trailing(L) \gg \texttt{</DIV>} \\
 & | & \ll packed(L) \gg \\
\ll packed(L) \gg & \rightarrow & \ll heading(L) \gg \ll block(L) \gg \\
\ll block(L) \gg & \rightarrow & \texttt{<DIV>} \ll block(L) \gg \texttt{</DIV>} \\
 & | & \ll desc(L+1) \gg \\
\ll heading(1) \gg & \rightarrow & \texttt{<H1>}text\texttt{</H1>} \\
\ll heading(2) \gg & \rightarrow & \texttt{<H2>}text\texttt{</H2>} \\
\ll heading(3) \gg & \rightarrow & \texttt{<H3>}text\texttt{</H3>} \\
\ll heading(4) \gg & \rightarrow & \texttt{<H4>}text\texttt{</H4>} \\
\ll heading(5) \gg & \rightarrow & \texttt{<H5>}text\texttt{</H5>} \\
\ll heading(6) \gg & \rightarrow & \texttt{<H6>}text\texttt{</H6>} \\
\ll desc(7) \gg & \rightarrow & (\ll paragraph \gg \mid \ll paradiv \gg)+
\end{array}
$$

Figure 4.2: Derivation rules (a)

tends to give a summary of the $\ll desc \gg$ node or an introduction for the following $\ll trailing \gg$ nodes. This fact is used in the next section. A $\ll desc(7) \gg$ node represent a sequence of leaf substructures such as paragraphs, lists, tables, and quotes. Nodes of $\ll paradiv \gg$ and $\ll paragraph \gg$ correspond to individual leaf substructures, and $\ll packed \gg$ and $\ll block \gg$ nodes are introduced for convenience.

Given the sample HTML document in Figure 4.4, the derived logical tree shown in Figure 4.5 is derived by the rules. The dotted lines indicate node

$\ll$*paradiv*$\gg$ → <DIV> $\ll$*paradiv*$\gg$ </DIV>
| $\ll$*paragraph*$\gg$
| <BLOCKQUOTE> $\ll$*block*(1)$\gg$ </BLOCKQUOTE>

$\ll$*paragraph*$\gg$ → <P>*text*</P>
| <UL>*text*</UL>
| <OL>*text*</OL>
| <DL>*text*</DL>
| <TABLE>*text*</TABLE>
| "*text−without−structure*"

Note 1: L=1,2,...,6 where $\ll$*desc*(*L*)$\gg$, $\ll$*leading*(*L*)$\gg$, $\ll$*trailing*(*L*)$\gg$, $\ll$*packed*(*L*)$\gg$, $\ll$*block*(*L*)$\gg$.
Note 2: "*text*" is any character string, while "*text−without−structure*" accepts one that is not decomposable by any rule. Thus, "*text−without−structure*" accepts a string which appears immediately inside a BODY, DIV or BLOCKQUOTE element and does not contain any structure-oriented tags.

Figure 4.3: Derivation rules (b)

groups having the same *L*-value. In this example, two $\ll$*heading*(1)$\gg$ are derived from the H1 tags. In the subtree rooted by the left $\ll$*block*(1)$\gg$ node, one $\ll$*leading*(2)$\gg$ node is derived from the P tag enclosing the introduction part of the first chapter, and two $\ll$*heading*(2)$\gg$ nodes are derived from H2 tags.

```
<BODY>
   <H1> 1. heading of chapter </H1>
   <P> introduction of chapter </P>
   <H2> 1.1. heading of section </H2>
   <P> body 1.1.1 </P>
   <P> body 1.1.2 <B> IMPORTANT </B> </P>
   <H2> 1.2. heading of section </H2>
   <P> body 2 </P>
   <H1> 2. heading of chapter </H1>
   <H2> 2.1 heading of subsection </H2>
   <TABLE> table 2.1.1 </TABLE>
   <P> body 2.1.2 </P>
</BODY>
```

Figure 4.4: Sample HTML document

4.3 View-pages

This section explains how to construct view-pages based on the logical tree. We use the vector space model [13] [36] [71], but structure-oriented and style-oriented tags introduced in Subsection 4.2.1 are taken into consideration.

First, a feature vector for each node in the logical tree is derived. Second, the similarity between each node and the user profile using the feature vector is calculated. Next, all nodes whose similarities are less than the threshold value given in the user profile are marked. Finally, the view-page by pruning away the subtrees whose nodes and ancestors are all marked is generated.

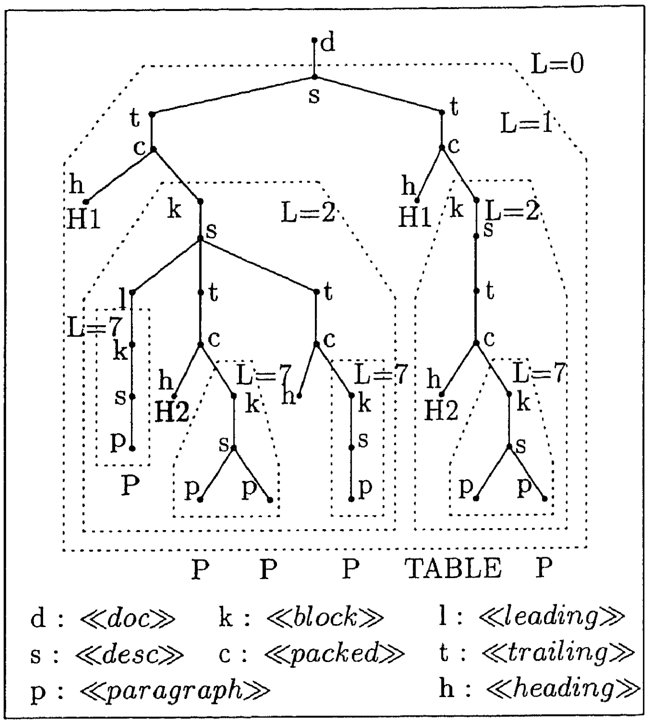


Figure 4.5: Extracted logical tree

4.3.1 Derivation of Feature Vectors

Leaf Nodes

Initially, feature vectors of leaf nodes in the logical tree are derived based on the traditional tf·idf (term frequency $\times$ inverse document frequency) weighting scheme [71]. In the calculation of the idf factor, the target document set has to be fixed. We regard the set of leaf nodes as the target document set in deriving the idf. Namely, the initial tf·idf weights are calculated according

to distribution of terms within the document.

Then, style-oriented tags are processed. They are usually used to emphasize some terms. Therefore, when terms occur within elements enclosed by style-oriented tags, their weights in the feature vectors are multiplied depending on the types of tags. The bias factors associated with style-oriented tags are shown in Table 4.1.

For each leaf node n, its feature vector is formulated as follows. When a term t occurs in multiple elements enclosed by different style-oriented tags, their maximum bias factor is used:

$$v_n = (tf_t \cdot idf_{n,t} \cdot \beta_n)_{t \in \mathcal{T}}, \tag{4.1}$$

where β_n is the maximum bias factor associated with the style-oriented tags enclosing the term t in node n.

Non-Leaf Nodes

The feature vector of each non-leaf node is given as an aggregation of feature vectors of its child nodes.

Generally, titles, headings, abstracts and introductions give good summaries of their bodies and contain many important terms [35] [36]. In the logical tree, headings appear as ≪*heading*≫ nodes. Moreover, ≪*leading*≫ nodes which appear as the first child node of the ≪*desc*≫ nodes tend to play a role of abstract and introduction, as mentioned in Subsection 4.2.2.

To reflect this property, the feature vector of each non-leaf node n is calculated as weighted sums of those of its child nodes:

$$v_n = \sharp C \cdot \frac{\Sigma_c \alpha_c \cdot v_c}{\Sigma_c \alpha_c}, \tag{4.2}$$

where c is a child node of n and $\sharp C$ is the total number of the child nodes. The coefficient α_c is given in Table 4.2.

4.3.2 Generation of View-pages

To generate view-pages, we must decide substructures which are likely to be relevant to the user profile. The score of each node is given by the similarity of each node in the logical tree to the user profile. it is calculated from its feature vector and the profile vector derived from the given keyword set. In the calculation, we have to note that the length of text (namely, the numbers of terms) associated with each node differs depending on nodes. To take this

Table 4.1: Bias factors β_n associated with style-oriented tags

tag	β_n	semantics
STRONG	5	strong emphasis
EM	3	emphasis
BIG	3	big letters
U	2	underline
B	2	bold
I	2	italic
DT	2	defined term in DL

Table 4.2: Coefficients α_c for child nodes

node	α_c	node	α_c
$\ll \mathit{heading} \gg$ (H1)	15	$\ll \mathit{desc} \gg$	1
$\ll \mathit{heading} \gg$ (H2)	11	$\ll \mathit{leading} \gg$	5
$\ll \mathit{heading} \gg$ (H3)	8	$\ll \mathit{trailing} \gg$	1
$\ll \mathit{heading} \gg$ (H4)	6	$\ll \mathit{packed} \gg$	1
$\ll \mathit{heading} \gg$ (H5)	4	$\ll \mathit{block} \gg$	1
$\ll \mathit{heading} \gg$ (H6)	3	$\ll \mathit{paragraph} \gg$	1

into account, the C-pivot measure is used rather than the traditional cosine measure.

View-pages are generated by pruning away subtrees whose all nodes and ancestors have similarity scores lower than the threshold value in the user profile. This process has to ensure that view-pages are valid HTML documents, which the user can browse on the ordinary WWW browsers. Note that careless pruning procedure may bring about document structures which do not form valid HTML documents. For instance, an HTML document must have a BODY element and it must not be empty. Therefore, we cannot

delete the BODY or its subtree even if scores of all its member nodes are less than the threshold value.

Another concern is to help the user understand the document structures through view-pages. Since the headings are important clues in understanding the whole document structures, we have decided to maintain all headings in the original documents in view-pages. In addition, some appropriate indications should be given as notes when some parts or substructures have been pruned. Moreover, simple pruning may bring about unexpected rendering of the view-page. For instance, pruning the P element away from "text <P> paragraph </P> text" brings about "text text". In this case, two "text" segments are accidentally concatenated to form a single text segment.

Based on the above consideration, in the view-page generation, each sequence of elements which are to be pruned away is replaced by a "(snip)" enclosed by a pair of DIV tags. The complete procedure to generate view-pages is given as followings:

1. Calculate the similarity score of every node in the logical tree.
2. Mark the nodes whose scores are less than the threshold value. Here, ≪*doc*≫ and ≪*heading*≫ nodes are always excluded even if their scores are low, since the BODY element is mandatory and H1, ..., H6 elements are important.

3. Identify the maximal subtrees whose nodes and ancestors are all marked.

4. Replace each sequence of HTML elements associated with the subtrees with "<DIV> (snip) </DIV>".

5. Generate the output as a view-page.

Theorem *Given an arbitrary valid HTML document, the above procedure generates a valid HTML document.*

(Proof)

It is obvious from the derivation rules in Figures 4.2 and 4.3 that each sequence corresponds to the `BODY` element or a sequence of child elements in a `BODY`, `DIV` or `BLOCKQUOTE` element. According to the HTML syntax, the `BODY` element cannot be replaced by a `DIV` element, but this case is avoided in Step 2. `BODY`, `DIV` and `BLOCKQUOTE` elements can contain `DIV` elements. Therefore, the replacement in Step 4 preserves the conformity to HTML. Thus, the above procedure derives a valid HTML document. Q.E.D.

Figure 4.6 illustrates the generation of a view-page. The circles are nodes whose scores are greater than or equal to the threshold value. The subtrees indicated by dotted lines are replaced by "<DIV> (snip) </DIV>" elements.

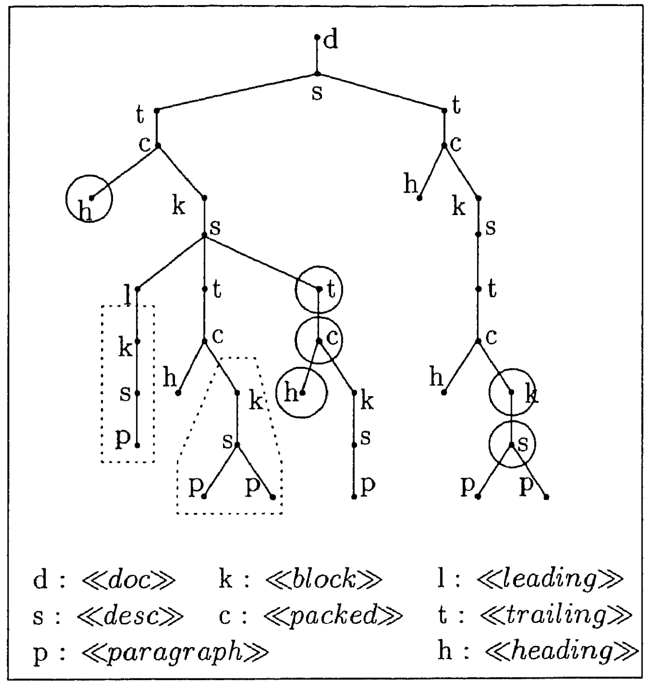


Figure 4.6: Example of view-page generation

```
<BODY>
   <H1> 1. heading of chapter </H1>
   <DIV> (snip) </DIV>
   <H2> 1.1. heading of section </H2>
   <DIV> (snip) </DIV>
   <H2> 1.2. heading of section </H2>
   <P> body 2 </P>
   <H1> 2. heading of chapter </H1>
   <H2> 2.1 heading of subsection </H2>
   <TABLE> table 2.1.1 </TABLE>
   <P> body 2.1.2 </P>
</BODY>
```

Figure 4.7: HTML document for a view-page

Figure 4.7 shows the HTML document corresponding to the derived view-page.

4.4 Prototype System Development

This section presents the system architecture to implement the proposed scheme, and show a sample session in the prototype system.

4.4.1 System Architecture

Overview

One of criteria for our prototype system design has been to respect user-friendliness provided in the current WWW browsing environment. The prototype system architecture is shown in Figure 4.8. It consists of three modules.

- Browser
- Controller
- Browsing Assistance Engine

Solid arcs indicate requests and method invocations, and dotted arcs indicate responses and value returns.

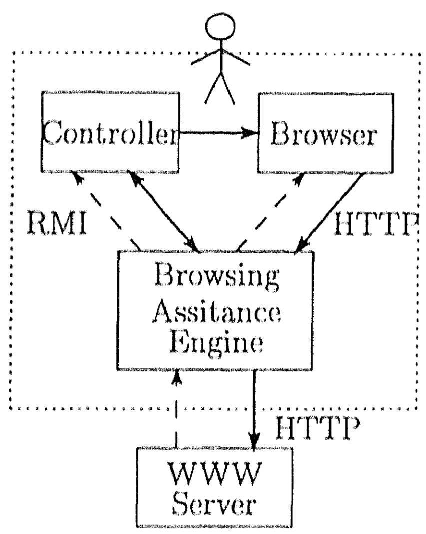


Figure 4.8: Prototype system architecture

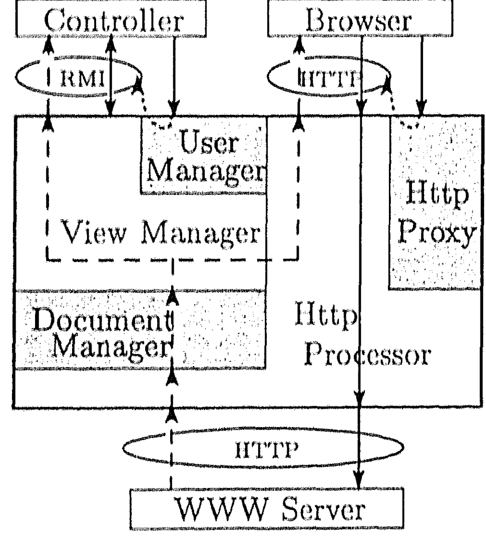


Figure 4.9: Architecture of BAE

The user can use an ordinary WWW browser, such as Netscape Communicator and Internet Explorer, without any modification. The Controller and the Browsing Assistance Engine (BAE) are original to our browsing environment. For each user, a pair of the Browser and Controller instances works as a front-end module and the BAE works as a back-end module.

The BAE is the central module in our environment, and provides the view-page for each WWW page. It works as an HTTP proxy server for the Browser, and can serve multiple pairs of the Browser and Controller instances. The BAE gets the user profile from the Controller and maintains it. It generates view-pages and their logical trees using the user profile for each user. The user can disable the view-page generation feature, when he/she wants to browse WWW pages in an ordinary manner.

The Browser displays the view-page generated by the BAE. When view-pages include hyperlinks, the user can visit other pages. Then, view-pages for the next pages will be generated and presented.

The Controller interacts with the user to get his/her user profile. When the user modifies the profile, it notifies the BAE of the modification. The Controller visually displays the logical tree of the current view-page, as described in Subsection 4.4.4. It helps the user to identify relevant substructures in each page.

In the prototype system, the Controller is implemented as a Java applet and the BAE runs as a Java application program. The presentation on the screen given by the Browser and Controller is synchronized. The applet context is used for the synchronization.

4.4.2 Browsing Assistance Engine (BAE)

Figure 4.9 shows the internal architecture of the BAE in It consists of the following five modules.

- Http Proxy
- User Manager
- Document Manager
- Http Processor
- View Manager

The first three modules are created when the BAE starts its execution. The Http Processor and View Manager are instantiated and destroyed based on demands from other modules. All the modules except the Document Manager run as threads.

The Http Proxy listens to a TCP port. When the Browser requests an HTTP resource, it instantiates an Http Processor and delegates the request to it.

The Http Processor fetches the requested HTTP resource. When the requested HTTP resource is an HTML document, the Http Processor parses it, creates the DOM object, and generates the logical tree and the view-page, by calling the Document Manager and the View Manager. When the Http Processor finishes processing the request, it is destroyed automatically.

The Document Manager is responsible for parsing documents and creating the DOM objects and logical trees. It also caches them on the memory resident hash table. URI is the hash key.

The User Manager waits for the RMI connection from a Controller instance and creates a View Manager for each connection (see below).

The View Manager communicates with the Controller to maintain the user profile and generates view-pages. View-pages are generated from the DOM objects and logical trees managed in the Document Manager. The View Manager also provides the Controller with information about the current view-page such as its URI, logical tree structure and node scores.

4.4.3 Typical Processing Flow

Starting-up Before invoking the Browser, we have to start the BAE, and a WWW page containing the Controller applet has to reside on the host. The user has to use the BAE as an HTTP proxy server on the WWW browser.

First, the user accesses to the WWW page containing the Controller applet. Then, the Controller is downloaded and executed in the window of the original WWW browser. It creates a new window and launches another instance of the WWW browser as the Browser. The Controller can control the Browser through the applet context. Next, the Controller interacts with the User Manager in the BAE, and its View Manager is instantiated by the User Manager.

Processing an HTTP Request After the above step, the user can browse WWW pages in an ordinary manner. When a new WWW page is requested, The Http Proxy instantiates an Http Processor to process it. If the requested resource is an HTML document, the URI is registered in the View Manager as the current URI of the Browser. Otherwise, it fetches the resource and simply returns it to the Browser.

When processing a request to an HTML document, the Http Processor

checks whether it is in the cache maintained by the Document Manager. If not, it fetches and parses the the HTML document. Next, the Http Processor generates a view-page by calling the View Manager and returns it to the Browser. If the requested URI is different from the last URI, the View Manager notifies the Controller to make the presentation of the logical tree up-to-date.

Change of User Profile The user can change keywords and the threshold value in his/her user profile. In this case, the Controller notifies the View Manager. When the keywords are changed, the View Manager computes the score of each node in the logical tree again.

Next, the Controller gets the current URI from the View Manager, and makes the Browser reload the current page through the applet context. The reload request is caught by the BAE, and the View Manager generates the view-page based on the updated user profile.

Changes to the threshold value in the user profile are processed in a similar manner.

4.4.4 Session Example

this subsection shows sample usage of our browsing environment. Suppose that we want documents which are related to this chapter under the W3C WWW site (http://www.w3.org/). Here, we select {*browsing, user, profile, interest, view, paper*} as a keywords set.

First, we turn off the view-page generation feature of the BAE, and we search candidate WWW pages querying the index server existing at the W3C site. To check highly ranked WWW pages, we turn on the view-page generation feature. Figures 4.10 and 4.11 show the Controller window and the Browser window, respectively, when we browse the first document in the list.

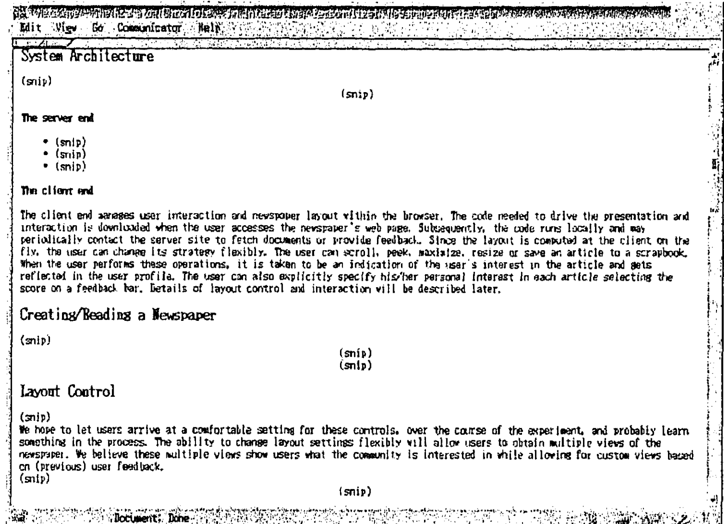
At the top of the Controller window, there are components to input and modify the user profile. The Controller window displays the logical tree of the current page. The color of each node is determined by its score. Nodes with the highest score are displayed in red and nodes with score 0 are in blue. Nodes with scores between the extremes are displayed in orange, yellow, green and cyan.

Nodes whose scores are greater than or equal to the threshold value are tagged with large red icons. Nodes whose scores are less than the threshold value are tagged with small cyan icons. The view-page consists of the contents

browsing user profile interest view paper
Enabled 0.0 0.27550417 Update

- System Architecture (0.0)
- Figure 1 shows the system archit ... (0.1657983)
- [2.3.1] (0.0)
- [2.3.2] (0.0148752555)
- [2.3.3] (0.2217892)
- The client end (0.0)
- The client end manages user inte ... (0.2738361)
- The client end manages user inte ... (0.2738361)
- [2.4] (0.041442294)
- [2.5] (0.0362022)
- Layout Control (0.0)
- Figure 4 shows a typical screen ... (0.21462202)
- Figure 4 shows a typical screen ... (0.15960306)
- ... (0.0)
- Figure 4. Example of the Krak ... (0.0)
- We will not discuss the layout p ... (0.0)
- Articles should be laid out in t ... (0.0)
- An important feature that differ ... (0.082203634)
- User score vs. community score. ... (0.0)
- Figure 5. Layout parameters ... (0.0)
- We hope to let users arrive at a ... (0.24345845)
- In the Krakatoa Chronicle, this ... (0.08603037)
- ... (0.0)
- Figure 6. Layout control mecha ... (0.0)
- [2.6] (0.076608784)
- Interaction (0.0)
- The Krakatoa Chronicle provides ... (0.23238862)

Figure 4.10: Controller window



Edit View Go Communicator Help

System Architecture

(snip)

(snip)

The server end

- (snip)
- (snip)
- (snip)

The client end

The client end manages user interaction and newspaper layout within the browser. The code needed to drive the presentation and interaction is downloaded when the user accesses the newspaper's web page. Subsequently, the code runs locally and may periodically contact the server site to fetch documents or provide feedback. Since the layout is computed at the client on the fly, the user can change its strategy flexibly. The user can scroll, peek, maximize, resize or save an article to a scrapbook. When the user performs these operations, it is taken to be an indication of the user's interest in the article and gets reflected in the user profile. The user can also explicitly specify his/her personal interest in each article selecting the score on a feedback bar. Details of layout control and interaction will be described later.

Creating/Reading a Newspaper

(snip)

(snip)
(snip)

Layout Control

(snip)
We hope to let users arrive at a comfortable setting for these controls, over the course of the experiment, and probably learn something in the process. The ability to change layout settings flexibly will allow users to obtain multiple views of the newspaper. We believe these multiple views show users what the community is interested in while allowing for custom views based on (previous) user feedback.
(snip)

(snip)

Document: Done

Figure 4.11: Browser window

corresponding to the nodes tagged with the large red icons. The user can control the threshold value interactively with the aid of the colored logical tree.

The view-page displayed in the Browser window helps our understanding the contexts. Especially, we can easily identify descriptions related to the keywords. When we want to browse the next document, we go back to the document list in the query result and select it. Then, the Browser displays its view-page. We can browse other pages similarly. When view-pages include hyperlinks, we can visit the linked pages as usual. In this way, our environment supports browsing view-pages tailored to the user's viewpoint as if they were existing WWW pages.

4.5 Experimental Evaluation

This section experimentally evaluates effectiveness of the proposed scheme from the three standpoints described below. Note that these experiments excluded WWW pages designed with frames and large tables

First, validity of the logical trees derived from each given page is evaluated. The logical trees must have an appropriate structure to extract substructures for inclusion in view-pages.

Second, correctness of the node scores in a given logical tree is evaluated. When the given logical tree structure and calculated node scores are correct, the generated view-page can contain only the appropriate substructures within the original page by adjusting the detail level of view-pages.

Third, the above two evaluation methods are applied to more general WWW pages. In this experiment, we used WWW pages searched by the Goo search engine. We can check effectiveness of the proposed scheme in a more realistic context of WWW page browsing.

The rest of this section describes these experiments.

4.5.1 Evaluation for Derivation of Logical Tree

To evaluate validity of the derived logical trees, three sets of ten pages are randomly selected from each of the following sites:

- IGD (http://www.igd.fhg.de/www/www95/)
- DLIB (http://www.dlib.org/)
- W3C (http://www.w3.org/).

These sites provide large, well-structured WWW pages such as technical papers and specifications, and have distinct policies for embedding HTML tags. The structures of logical trees are compared with the hierarchical structures of chapters, sections, paragraphs, and other structures.

The selected pages had the following peculiarities regarding the use of the structure-oriented tags used to derive the logical trees:

- Headers and footers with their logotypes were given in individual ways.
- The lowest level headings in DLIB on some pages were not indicated by heading tags such as `H5` and `H6` but by `B` enclosed by `P`. An example is, "<P><B>*someheading*</B></P>".
- In IGD and W3C, the list of bibliographies for each page was coded by `OL` or `UL`. In some DLIB pages, the lists of their bibliographies were not coded by such tags, and each item of them was indicated by `P`.
- Most pages in W3C followed recommendations of the HTML 4.0 specification. In IDG, on the other hand, there were some misuses and the meaningless use of tags generated by WWW authoring tools.

It was also judged whether phrases enclosed in style-oriented tags contained important terms to generate view-pages. Table 4.3 lists the ratios

Table 4.3: Usage of style-oriented tags

Site	Use of tags	Important phrases	Ratio
IGD	167	112	0.670659
DLIB	341	286	0.838710
W3C	211	153	0.725118

of the number of occurrences of tags to the number that enclose important phrases.

We look at hierarchical structure with seven levels, which consist of chapter, section, subsection, paragraph and so on. Such semantic boundaries within the WWW pages are manually identified at each hierarchical level, and used as right answers. These boundaries are decided on the basis of presentations rendered by WWW browsers. Here, the top level is indicated by $h1$, the second level is by $h2$, ..., and the lowest level is by *block*. Paragraphs, lists and tables are always treated as *block* level. The sets of boundaries are coded as A_i $(i = h1, ..., h6, block)$.

Validity of a derived logical tree is evaluated as follows: At each level $L = 1, ..., 6$ corresponding to $i = h1, ..., h6$, let B_i be the set of boundaries of $\ll leading(L) \gg$ and $\ll trailing(L) \gg$ nodes in the logical tree. Beyond that, let B_{block} be the set of boundaries of $\ll paradiv \gg$ and $\ll paragraph \gg$ nodes. The precisions P_i and recalls R_i are then calculated for substructures at each

hierarchical level in each given logical tree.

$$P_i = \frac{|A_i \cap B_i|}{|B_i|}$$

$$R_i = \frac{|A_i \cap B_i|}{|A_i|} \qquad (i = h1, .., h6, block).$$

The results for the three sites are listed in Tables 4.4 to 4.6, and the average of the three are listed in Table 4.7. In the tables, *Manual* and *Rules* show the total number of the boundaries decided manually and by derivation rules. *Precision* and *Recall* are the averages of precisions and recalls of boundaries. Note that, to calculate precisions and recalls at each level, we use only pages that have boundaries produced manually and by derivation rules at the respective level. *Pages* are the number of pages used.

With the highest three and the *block* levels, in most cases, *Precisions* and *Recalls* were over 90%. At other hierarchical levels, they were about 25% to 70%. Substructures at these levels, however, seldom occur. This result indicates that the derivation rules can derive logical trees that are similar to human-judged structures.

Additionally, Tables 4.8 to 4.11 lists correlations between the use of heading tags and the descriptions of headings. In the tables, *Use of Tags* is the

Table 4.4: Experimental result for IGD

Level	*Manual*	*Rules*	*Precision*	*Pages*	*Recall*	*Pages*
$h1$	20	20	1.000	10	1.000	10
$h2$	61	60	1.000	9	0.984	9
$h3$	63	63	1.000	9	1.000	9
$h4$	19	19	1.000	2	1.000	2
$h5$	3	5	0.500	2	1.000	1
$h6$	0	0	—	0	—	0
block	549	715	0.773	10	0.998	10

Table 4.5: Experimental result for W3C

Level	*Manual*	*Rules*	*Precision*	*Pages*	*Recall*	*Pages*
$h1$	17	17	0.850	10	0.850	10
$h2$	72	70	0.917	9	0.930	9
$h3$	68	69	0.911	9	0.933	9
$h4$	71	71	0.933	5	0.933	5
$h5$	0	0	—	0	—	0
$h6$	0	0	—	0	—	0
block	665	738	0.852	10	0.937	10

Table 4.6: Experimental result for DLIB

Level	*Manual*	*Rules*	*Precision*	*Pages*	*Recall*	*Pages*
$h1$	3	3	1.000	3	1.000	3
$h2$	27	27	1.000	9	1.000	9
$h3$	88	87	0.846	10	0.936	9
$h4$	46	32	0.800	5	0.667	6
$h5$	19	8	0.125	4	—	0
$h6$	0	20	0.000	7	—	0
block	659	877	0.717	10	0.947	10

Table 4.7: Averages for all sites

Level	*Manual*	*Rules*	*Precision*	*Pages*	*Recall*	*Pages*
*h*1	40	40	0.935	23	0.935	23
*h*2	160	157	0.972	27	0.971	27
*h*3	219	219	0.916	28	0.956	27
*h*4	136	122	0.889	12	0.821	13
*h*5	24	13	0.250	6	0.667	3
*h*6	0	20	0.000	7	—	0
block	1873	2330	0.781	30	0.961	30

Table 4.8: Experimental result for IGD

Heading Level	*Use of Tags*	*Descriptions of Headings*
*h*1	1.000	1.000
*h*2	1.000	0.990
*h*3	1.000	1.000
*h*4	1.000	1.000
*h*5	0.333	1.000
*h*6	—	—

Table 4.9: Experimental result for W3C

Heading Level	*Use of Tags*	*Descriptions of Headings*
*h*1	1.000	1.000
*h*2	0.998	1.000
*h*3	0.923	1.000
*h*4	1.000	1.000
*h*5	—	—
*h*6	—	—

Table 4.10: Experimental result for DLIB

Heading Level	*Use of Tags*	*Descriptions of Headings*
*h*1	1.000	1.000
*h*2	1.000	1.000
*h*3	0.816	0.938
*h*4	1.000	0.738
*h*5	0.000	0.000
*h*6	0.000	—

Table 4.11: Averages for all sites

Heading Level	*Use of Tags*	*Descriptions of Headings*
*h*1	1.000	1.000
*h*2	1.000	0.995
*h*3	0.879	0.940
*h*4	1.000	0.763
*h*5	0.290	0.039
*h*6	0.000	—

ratio of the number of phrases enclosed by heading tags to the number that describe actual headings. *Descriptions of Headings* are the ratio of the number of descriptions of headings to the number indicated by heading tags.

These tables allow us to obtain correlations between *Use of Tags* and *Precision*, and between *Descriptions of Headings* and *Recall*. Misuses of lower level heading tags to emphasize enclosed phrases seem to reduce precision. On the other hand, the use of non-heading tags to indicate headings seems to reduce recalls. Therefore, lower level heading tags should not be used to derive logical trees from actual WWW pages.

These experiments revealed that higher and *block* level substructures can be extracted by the derivation rules from large pages, which comprise the main target of the proposed scheme. We further think that other lower level substructures can be extracted by this scheme if corresponding heading tags are used validly. In future research, we will modify our scheme to consider the use of actual tags in the current WWW.

4.5.2 Evaluation for Generation of View-pages

In the second experiments, the correctness of the node scores are evaluated. This evaluation is performed as follows under an assumption that target pages are valid HTML documents and the derived logical trees are correct.

First, a human evaluates the nodes of each logical tree according to a five-grade system in which five is the best grade and one is the worst. Our system then scores them using the scoring scheme in Section 4.3. The correctness is evaluated by comparing two sequences of nodes ranked by these scores.

We used ten valid HTML documents manually generated from our published papers and two keyword sets for each document. For each document, five words that appeared frequently in a few regions of the document were decided as a keyword set. Next, for each pair of an HTML document and a keyword set for the document, we gave human-judged grades and calculated scores to nodes in the derived logical tree.

Correctness of the scores of their nodes are evaluated on the basis of Bertell's measure J [72]. Here, we describe the human-judged grade by $G(n)$ and the calculated score by $R(n)$ for a node n. Given a total order $\succ$ for the node set, the measure J is given by following formula:

$$J(R) = \frac{\Sigma_{n \succ n'}(R(n) - R(n'))}{\Sigma_{n \succ n'}|R(n) - R(n')|}.$$

This measure calculates similarity of the sequence ranked by the scoring function R to the sequence ranked by the order $\succ$. It gives 1.0 for the same rankings and -1.0 for reverse rankings.

We determine the total order $\succ$ based on $G(n)$. however, we cannot uniquely determine the sequence based on the human-judged grades because there are many nodes with the same grades. This evaluation uses three sequences instead of a unique one for each logical tree: optimistically, pessimistically and randomly generated sequences based on the grades. The first two sequences are given by following total orders:

(I) $G(n) > G(n') \vee (G(n) = G(n') \wedge R(n) > R(n'))$

(II) $G(n) > G(n') \vee (G(n) = G(n') \wedge R(n) < R(n'))$.

Definition (I) gives an optimistic order because R always gives correct orders for nodes with the same grades. Definition (II) gives a pessimistic order because R always gives wrong orders. We call J based on (I) and (II) J_1 and J_2. We also generate sequences with nodes having the same grades ordered randomly. Let J_3 be the average of the J based on such one hundred sequences.

Table 4.12: Condition (a): Experiment result using uniform weights α_c

Weights	J_1	J_2	J_3
all-1	0.89812	0.54273	0.71924
all-5	0.90482	0.56939	0.73591
all-10	**0.90733**	0.58532	**0.74731**
all-15	0.90652	**0.58581**	0.74581
all-20	0.90141	0.56719	0.73291
Difference	0.00921	0.04308	0.02807
Improvement(%)	1.01507	7.35392	3.75614

Table 4.13: Condition (b): Experiment result using non-uniform weights α_c

Weights	J_1	J_2	J_3
A	0.90198	0.56822	0.72916
B	0.90538	**0.59328**	**0.74710**
C	0.90293	0.58371	0.73198
D	0.90431	0.57121	0.73939
E	**0.90562**	0.58931	0.74583
Difference	0.00364	0.02506	0.01794
Improvement(%)	0.40356	4.41026	2.46037

Table 4.14: Condition (c): Experimental result using different weights $\beta_{n,t}$

Weights	J_1	J_2	J_3
all 1	0.90381	0.58181	0.74194
$\times$ 1	**0.90538**	**0.59328**	**0.74710**
$\times$ 2	0.90321	0.57619	0.73981
$\times$ 3	0.90298	0.57581	0.73973
Difference	0.00240	0.01747	0.00737
Improvement(%)	0.26579	3.03399	0.99631

Tables 4.12 to 4.14 list the averages of results by these measures using some of the weight variations listed in Table 4.1 and 4.2. *Differences* are the difference of the maximum and minimum values; *Improvements* are the

Table 4.15: Non-uniform weights α_c

Node Type	A	B	C	D	E
$\ll$*heading*(1)$\gg$	15	15	20	10	15
$\ll$*heading*(2)$\gg$	15	15	20	8	11
$\ll$*heading*(3)$\gg$	10	10	15	6	8
$\ll$*heading*(4)$\gg$	10	10	15	5	6
$\ll$*heading*(5)$\gg$	5	10	10	4	4
$\ll$*heading*(6)$\gg$	5	10	10	3	3
$\ll$*leading*$\gg$	5	5	5	5	5

ratio of $Differences$ to the minimum values. The weight of $\ll$*leading*$\gg$ is fixed at five, and the slope constant S is fixed at 0.2 [1].

First, to investigate the effects of bias factors β_n in Table 4.2, we fixed coefficients α_c to Table 4.1 and applied the uniform weights. Table 4.12 lists the results. In these weights, all the heading nodes are given the same weights, 1, 5, ..., 25. We call each of them *all*-1, *all*-5, ..., *all*-25. The result showed a little improvement in *all*-10 and *all*-15. The weights *all*-1 always showed the worst results. This showed that assigning weights corresponding to node types is effective.

Second, to decide more appropriate bias factors, we also fixed weights to Table 4.1 and applied the non-uniform weights. The results are listed in Table 4.2. These weights were about 10 to 15 based on the best results in Condition (a). In these case, weights B and E showed relatively good results. In particular, B showed better results than *all*-10 and *all*-15. This reveals

[1] This slope constant value is efficient for some data collections of TREC [15].

that non-uniform weights are effective.

Finally, we applied variations of coefficients based on Table 4.1 and fixed bias factor β_n to weights B. Weights *all*-1 gives 1 to all style-oriented tags, and "$\times$N" gives N times of the weights listed in Table 4.1. Weights $\times 1$ yielded the best results, but they provided little improvement from *all*-1.

The average values for J_1, J_2 and J_3 were about 0.91, 0.59, 0.75, respectively. Through all experiments, J_1 showed good. And although J_2 is a pessimistic measure, the results it yielded were not bad. This suggests that our scoring scheme works well for HTML documents with appropriate tags, and can be improved with appropriate weights.

4.5.3 More Experiments in the WWW

The experiments in Subsection 4.5.1 and 4.5.2 were effective for WWW pages that had been slightly controlled. Real WWW pages, however, do not necessarily follow W3C recommendations. This subsection covers our application of the two evaluation methods to a more realistic context of WWW page browsing.

First, we decided a five keyword set consisting of five words related to the whole target domain and a more local topic. We then selected five sets of

ten WWW pages searched by the Goo search engine with the keyword sets. They included small and unstructured pages as well. As mentioned at the beginning of this section, WWW pages comprising frames and large tables were excluded.

Tables 4.16 to 4.21 list the evaluated results of validity on the derived logical trees. The averages of precisions and recalls were over 70% in most cases. Because there were many misused tags and tag use was inconsistent, some precisions and recalls were reduced. We need to discover headings by using clues other than heading tags. We also need to be able to identify misused tags.

Table 4.22 shows the evaluated results of correctness of node scores. The averages for J_1, J_2 and J_3 were 0.80 to 0.95, 0.56 to 0.80, 0.69 to 0.87, respectively. With these results, we can conclude that our scheme also works for ordinary WWW pages.

4.6 Discussion and Future Issues

This chapter presented a new scheme to support WWW browsing by dynamically providing HTML pages tailored to a user profile. This scheme makes it easy for the user to identify interesting parts and to understand the contents

Table 4.16: Experimental result for keyword set A

Level	*Manual*	*Rules*	*Precision*	*Pages*	*Recall*	*Pages*
*h*1	18	16	0.875	8	0.778	9
*h*2	48	27	0.717	6	0.465	9
*h*3	30	33	0.926	5	0.833	6
*h*4	9	3	1.000	1	0.500	2
*h*5	0	0	–	0	–	0
*h*6	0	0	–	0	–	0
block	289	375	0.720	10	0.972	10

Table 4.17: Experimental result for keyword set B

Level	*Manual*	*Rules*	*Precision*	*Pages*	*Recall*	*Pages*
*h*1	26	20	0.912	9	0.688	8
*h*2	23	23	0.821	5	0.985	7
*h*3	33	25	0.826	6	0.633	6
*h*4	12	10	1.000	2	0.402	3
*h*5	3	5	0.731	2	0.837	2
*h*6	0	0	–	0	–	0
block	267	325	0.814	10	0.912	10

Table 4.18: Experimental result for keyword set C

Level	*Manual*	*Rules*	*Precision*	*Pages*	*Recall*	*Pages*
*h*1	19	18	0.958	8	0.932	9
*h*2	16	10	0.715	5	0.373	8
*h*3	20	19	0.894	5	0.712	6
*h*4	1	0	0.000	1	–	0
*h*5	0	0	–	0	–	0
*h*6	12	11	0.712	4	0.632	2
block	218	238	0.832	10	1.000	10

Table 4.19: Experimental result for keyword set D

Level	*Manual*	*Rules*	*Precision*	*Pages*	*Recall*	*Pages*
*h*1	20	20	1.000	10	1.000	10
*h*2	28	26	0.893	7	0.893	7
*h*3	13	14	0.813	3	0.872	3
*h*4	0	4	–	0	0.000	1
*h*5	3	3	1.000	1	1.000	1
*h*6	0	0	–	0	–	0
block	764	893	0.819	10	1.000	10

Table 4.20: Experimental result for keyword set E

Level	*Manual*	*Rules*	*Precision*	*Pages*	*Recall*	*Pages*
*h*1	23	20	0.871	8	0.773	9
*h*2	48	27	0.717	6	0.535	9
*h*3	21	38	0.451	3	0.858	6
*h*4	8	5	0.790	3	0.682	2
*h*5	0	0	–	0	–	0
*h*6	0	0	–	0	–	0
block	289	375	0.720	10	0.972	10

Table 4.21: Averages for all experimental results

Level	*Manual*	*Rules*	*Precision*	*Pages*	*Recall*	*Pages*
*h*1	106	94	0.942	43	0.842	45
*h*2	163	113	0.835	29	0.686	40
*h*3	117	129	0.881	22	0.723	27
*h*4	30	22	0.612	7	0.374	8
*h*5	6	8	0.829	3	0.912	3
*h*6	12	11	0.712	4	0.632	2
block	1827	2206	0.873	50	0.957	50

Table 4.22: Evaluation of scoring scheme

Keyword Set	J_1	J_2	J_3
A	0.94922	0.79706	0.87235
B	0.88103	0.62138	0.75131
C	0.87391	0.55931	0.71648
D	0.79913	0.58138	0.69025
E	0.84679	0.69719	0.77281

of WWW pages. To this end, we have introduced a document model called a logical tree. In our scheme, logical document structures embedded in HTML documents are extracted in the logical tree, and summaries biased by the user profile are generated for browsing. We have also shown that the proposed scheme can be implemented non-intrusively in the current WWW browsing environment. We also have some experimental evaluations and have obtained good results.

Remaining research issues include detailed performance analysis, and improvement of the prototype system. We also plan to extend our scheme to make the best of hyperlinks and referenced page contents in view-page generation. Applications of the scheme to XML documents is another interesting research issue.

Chapter 5

X^2QL: An Extensible XML Query Language

5.1 Overview

With the recent and rapid advance of the Internet, management of structured documents such as XML documents and their databases has become more and more important [73]-[74]. A number of query languages for XML documents have been proposed up to the present.

XML-QL [18] and XQL [53] are well-known XML query languages. Although Lorel [57], UnQL [59], StruQL [75], and YATL [58] are proposed as query languages for semistructured data, XML documents are their potential

targets. They allow the user to describe queries in a declarative manner. Especially, XML-QL and many query languages for semistructured data enable tag-based powerful document structure manipulation such as restructuring element hierarchies and joins. However, they are very weak in document contents processing.

Here, the document contents processing implies the similarity-based selection, ranking, summary generation, topic extraction, and so on [76]-[77]. Some of the aforementioned query languages have certain kinds of string-based pattern matching facilities. However, they are not enough to perform, for instance, summary generation.

Requirements for document contents processing vary depending on applications, target document types, and target element types. For example, several techniques have been proposed for summary generation [76]. Even in similarity-based search, measures proposed so far have their own strong and weak points [78]. Thus, it is very difficult to provide a complete set of contents processing facilities in advance to cope with the variety of requirements. A promising approach to this difficulty is to make a query language extensible to adopt user-defined contents processing functions.

In this chapter, we propose an extensible XML query language X^{2}QL, which features inclusion of user-defined foreign functions to process document

contents in the context of XML-QL-based document structure manipulation. XML-QL is the most well-known XML query language featuring tag-based powerful document structure manipulation. This is the reason we have designed X^2QL taking XML-QL as a starting basis. Foreign functions are given as external programs written in programming languages. By including appropriate user-defined foreign functions, each user can extend the processing power of X^2QL. This extensibility makes it possible to integrate application-oriented high-level contents processing facilities into querying documents. We also describe an implementation of an X^2QL query processing system on top of XSLT [19]-[21] processors.

The remaining part of this chapter is organized as follows. In Section 5.2, we show some examples of queries, in which inclusion of document contents processing is beneficial. Section 5.3 explains the main constructs of X^2QL, and shows its query examples. Section 5.4 describes the extension mechanism via foreign functions. Section 5.5 describes the implementation of the X^2QL query processing system on top of XSLT processors. Section 5.6 explains an advanced feature of X^2QL. Finally, we summarize and discuss this chapter in Section 5.7.

5.2 Motivating Examples

This section shows two examples of XML queries, in which inclusion of document contents processing is beneficial. These query examples are specified in X^2QL in Section 5.3.

Suppose that an XML document including newspaper articles is given, in which each article consists of its publication date, headline, and news body. Its DTD is given as follows.

```
<!-- DTD of the input document -->
<!ELEMENT Document  Article+>
<!ELEMENT Article  (Date, Category, Headline, Body)>
<!ELEMENT Date     (Year, Month, Day)>
<!ELEMENT Year      #PCDATA>
<!ELEMENT Month     #PCDATA>
<!ELEMENT Day       #PCDATA>
<!ELEMENT Category  #PCDATA>
<!ELEMENT Headline  #PCDATA>
<!ELEMENT Body      Paragraph+>
<!ELEMENT Paragraph #PCDATA>
```

Example 1 We generate a new XML document in which articles published after 1999 are selected and grouped by the category, and their summaries are added. The DTD of the output document is shown below. This query requires both the document structure manipulation to select `Article` elements

published after 1999 and group them by the Category values and the document contents processing to generate summaries of Article elements.

```
<!-- DTD of the output document -->
<!ELEMENT Document   DateGroup+>
<!ELEMENT Group      Article+>
<!ATTLIST Group      Category CDATA #REQUIRED>
<!ELEMENT Article   (Date, Headline, Summary, Body)>
<!ELEMENT Date      (Year, Month, Day)>
<!ELEMENT Year       #PCDATA>
<!ELEMENT Month      #PCDATA>
<!ELEMENT Day        #PCDATA>
<!ELEMENT Headline   #PCDATA>
<!ELEMENT Summary    #PCDATA>
<!ELEMENT Body       Paragraph+>
<!ELEMENT Paragraph  #PCDATA>
```

Example 2 We give a set of keywords and get the top N articles related to the keywords. The ranking order of each Article element is determined by the similarity with the keywords. The similarity measure is given by such contents processing function.

5.3 X^2QL: An eXtensible XML Query Language

The syntax of X^2QL is similar to that of XML-QL. However, X^2QL can incorporate user-defined foreign functions. The implementation of foreign functions is given as external programs. First, Subsection 5.3.1 gives the syntax of X^2QL. Then, foreign functions are explained in Subsection 5.3.2. Subsection 5.3.3 shows how example queries in Section 5.2 are specified in X^2QL.

5.3.1 Query Syntax

As aforementioned, the syntax of X^2QL is based on XML-QL as follows.

```
where        patterns [in source][,patterns [in source]]*[,predicate]*
[rank-by     ranking keys  top number]
[order-by    ordering keys [descending]]
construct    construction of each output
```

The `where`, `order-by`, and `construct` clauses are same as in XML-QL, except that they can contain foreign function calls as explained in Subsection 5.3.2. Each `where` clause binds variables according to the specified conditions. For convenience, we call a *tuple* each set of variable bindings. Then, the

construct clause generates the result for each tuple. Finally, if an order-by clause is specified, the generated results are sorted by the given *ordering keys*.

A rank-by clause is used to specify at most the top *number* tuples are selected based on the value of the *ranking keys*. This selection is performed before the construct clause is processed. The order of selected tuples is not affected by this clause, namely, the order of output elements is based on the order in the input document. In the document contents processing, it is often necessary to rank target elements by some similarity or importance measure derived by the foreign function and to select the top N elements based on their ranks. To facilitate such processing, we have included the rank-by clause as an extension.

5.3.2 Foreign Functions

As explained in Subsection 2.3.1, functions defined in XML-QL are just canned queries. In X^2QL, users can define foreign functions whose implementation is given as external programs. In the current version of X^2QL, we assume that they are written in Java. See Section 5.4 to implement foreign functions.

Basic foreign functions are classified into *general functions* and *element methods*. The other types of foreign functions are described in Section 5.6.

General functions are defined in the global namespace. On the other hand, each element method is associated with a specific element type, namely, defined in its local namespace. Therefore, element methods of the same name can be defined for different element types. When a foreign function is called, the element type of the target element is checked, then an appropriate element method is invoked depending on the target element.

Let us consider foreign functions required for the examples in Section 5.2. For Example 1, we need a element method (named `abstract()`) associated with the `Article` element type to summarize each `Article` element and to return a `Summary` element. The element method generates a summary of an `Article` element considering the target dependent property such that terms appeared in its `Headline` element are important.

For Example 2, we need a function to measure the similarity between the given keywords and an element content. Here, let us define it as a general function, say `sim_cosine()`, that calculates similarities based on the traditional cosine measure [13]. By applying the function to an `Article` element, its similarity value is returned. [1]

In our environment, foreign functions are defined as follows.

[1]Similarity measures often need document-dependent global factors (e.g. idf). The implementations of foreign functions are responsible to calculating them in their first invocation.

```
function   type-name general-function-name( argument-list )
defined-by "URI of the implementation"

function   type-name element-type-name
                    .element-method-name( argument-list )
defined-by "URI of the implementation"

argument-list ::= type-name argument-name [,
                          type-name argument-name]*
```

In the definition of a foreign function, the data types of arguments and return values are specified. The data types `number`, `string`, `element`, `content` and any element types are allowed. Variables bound by `element_as` and `content_as` are associated with `element` and `content` values, respectively.

The definitions of `abstract()` and `sim_cosine()` are given as follows.

```
function   Summary Article.abstract()
defined-by "http://fqdn/path/pkg.ArticleMethods#abstract"

function   number sim_cosine( content target, string keyword )
defined-by "http://fqdn/path/common.vecspace#cosine"
```

5.3.3 Query Specification Examples

The query specification for Examples 1 and 2 in Section 5.2 are given as follows.

```
where     <Document> $x </>
construct <Document> in "articles.xml"
  where     <Article>
              <Date> <Year> $y </> </> element_as $d
              <Category></> content_as $c
              <Headline></> element_as $h
              <Body></>     element_as $b
            </> element_as $a in $x,
            $y >= 1999
  order-by  $c
  construct <Group ID=CtgID($c) Category=$c>
              <Article>
                $d  $h  $a.abstract()  $b
              </>
            </>
          </>
```

Query Specification 1

```
where     <Document> $x </>
construct <Document> in "articles.xml"
  where   <Article> </> element_as $a in $x
  rank-by   sim_cosine( $a, keywords )  top N
  construct $a
          </>
```

Query Specification 2

In Query Specification 1, the method `abstract()` associated with the `Article` element type is used. The query returns a `Document` which is

a sequence of Group elements. The Group elements are sorted by the Category values. Each Group element groups Article elements with the same Category value. A Summary element is created by abstract() and inserted between the Headline and Body elements. The Date element is also included in the Article element.

In Query Specification 2, a general function sim_cosine() is used. It gives the similarity of an Article element with respect to the given keywords. With the use of the rank-by clause, only the top N Article elements are contained in the result. They appear in the same order as in the source document.

5.4 Extensibility via Foreign Functions

Users writing foreign function implementations need to understand the foreign function framework. This framework consists of the type system, internal object model and Java bindings.

5.4.1 Type System

There are five built-in types in X^2QL: `string`, `number`, `boolean`, `content` and `element`. Beyond that, each element type in XML documents can be used as a type in X^2QL. It is treated as a subtype of `element` type, which is the generic built-in type for any element in an XML document.

When binding a variable, the value type is decided according to the location from which the value was extracted. The types are `element`, `content` or `string`. When evaluating an expression, the type of the result value is given by the type returned by the function.

When the type of a value differs from the type desired by a function in an expression, the value is translated into the desired type if possible. If translation is impossible, then the query is not processed. This translation is defined in Table 5.1. In this table, a function p_q $(p, q \in \{s, n, b, c, e\})$ maps a value of the type q into a value of the type p. Letters s, n, b, c and e represent `string`, `number`, `boolean`, `content` and `element`, respectively, and "—" indicates that translation is unnecessary or undefined.

Functions s_c and s_e translate a content item and an element into a `string` value by removing tags within them, and s_n and s_b returns the string representation of a given value. When translating into `number` and `boolean`

Table 5.1: Translation rules between types

from \ to	string	number	boolean	content	element
string	—	$n_s(x)$	$b_s(x)$	$c_s(x)$	—
number	$s_n(x)$	—	$b_n(x)$	$c_s(s_n(x))$	—
boolean	$s_b(x)$	$n_b(x)$	—	$c_s(s_b(x))$	—
content	$s_c(x)$	$n_s(s_c(x))$	$b_s(s_c(x))$	—	$e_c(x)$
element	$s_e(x)$	$n_s(s_e(x))$	$b_s(s_e(x))$	$c_e(x)$	—

values, the values are first translated into a `string` value, then into `number` and `boolean` values by b_s and n_s. Functions c_s and c_e return a `content` value that contains only the given value. Only `content` values can be translated into `element` values. Function e_c returns the first element within a given `content` value if the first item is an element.

5.4.2 Internal Object Model

This subsection explains how to manage elements in XML documents and implementations of foreign functions.

The associations between each element type and implementations of element methods are given by foreign function definitions. These associations are managed by the *function table* for each element.

At the first invocation of an element method for a given element, an

instance object called *element object* is created for the element. The implementation is then executed through the corresponding function table, and the element object is passed to the implementation. Instantiated element objects stay in memory while processing a query. After this, when an element method for the same element is invoked, the element object that has already been instantiated is used. All general functions are managed by the anonymous element object.

Element objects have an extended interface of W3C DOM `Element`. Therefore, implementations can be accessed to the element itself, its parent and child nodes and the document with the element. They also serve functions to

- invoke an element method associated with an element, and
- store objects created by the implementation as *properties.*

The first feature is realized by the `invoke` method of element objects. By this feature, implementations can invoke element methods of other element objects. This invocation is enabled by its internal function table. The second feature is provided by their `setProperty` and `getProperty` methods. These methods realize the implementation of foreign functions whose results differ from or depend on the results of previous invocations. Properties are named by string keys and kept in a dictionary collection within the element object.

5.4.3 Java Binding

As mentioned earlier, the implementations of foreign functions are written as Java programs. In Java binding, data types `number`, `string`, `element` and `content` are mapped into `XNumber`, `XString`, `XElement` and `XContent` interfaces in the package provided for X^2QL. All element types are treated as `element` values. These interfaces are subinterfaces of `XObject`. Note that `XElement` and `XContent` are extensions of W3C DOM `Element` and `DocumentFragment` interfaces. `XElement` represents element objects. Especially, additional methods of element objects are defined as follows in Java:

```
package jp.ac.tsukuba.is.kde.x2ql.objects;

public interface XElement extends org.w3c.dom.Element
  {
  public XObject invoke( String name, XObject[] args );
  public void    setProperty( String name, Object value );
  public Object  getProperty( String name );
  }
```

When an element method is to be implemented, a Java class is created and the implementation is given as its method. When an element method is invoked for an element, the element is added as the first argument `self` in the argument list of the Java function. As an example, the Java implementation for the method `abstract()` is given in Subsection 5.3.2. The query processor

checks the element type of the element bound to `self` before the method invocation.

Implementations can create new values using an instance object with the `XObjectFactory` interface. The instance is provided by `SystemModule`, which is a singleton object.

```
// The class file should be located at http://fqdn/path/.
package pkg;
import jp.ac.tsukuba.is.kde.x2ql.objects.XElement;
import jp.ac.tsukuba.is.kde.x2ql.util.SystemModule;

public class ArticleMethods
  {
  public XElement abstract( XElement self )
    {
    XElement result = null;
    ...
    result = SystemModule.getXObjectFactory()
                         .createXElement( ... );
    return result;
    }
  ...
  }
```

5.5 Query Processing on XSLT Processors

We have developed an X^2QL query processing system on top of an XSLT processor. The rationale here is three-fold. First, the approach contributes to rapid development of the prototype of an X^2QL query processing system. Second, an implementation on top of XSLT processors assures certain level of portability. Third, X^2QL can work as a front end to XSLT processors. As shown below, XSLT specifications are low-level and procedural, and they are difficult for novice users. With X^2QL, we can specify document manipulation more declaratively.

5.5.1 System Architecture

Figure 5.1 shows Query processing in our prototype system. The translator translates each X^2QL query into XSLT template rules in stylesheets. The XSLT processor then executes the query by interpreting the stylesheets.

The current implementation has the following restrictions.

- The query does not include joins of multiple documents.
- No regular path expressions containing closures of tag names are used.

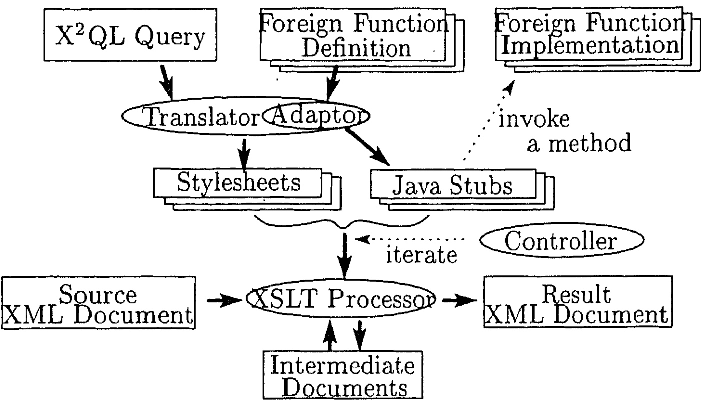


Figure 5.1: Query processing scheme

- No subblocks[2] are used.
- No elements with IDs specified by skolem functions are nested.

5.5.2 Mapping of Foreign Functions

Foreign functions in X^2QL are mapped to extension functions. As mentioned before, implementation of extension functions depends on the underlying XSLT processor. In this study, we concentrate on the implementation on LotusXSL [22] [23] to make our discussion concentrate. The implementation of foreign functions of X^2QL are written in Java.

The definition of each foreign function is processed before the stylesheet generation. This is done by the adaptor module inside the translator. Imple-

[2]A subblock is an XML-QL subquery enclosed '{' and '}', and is used for outer joins.

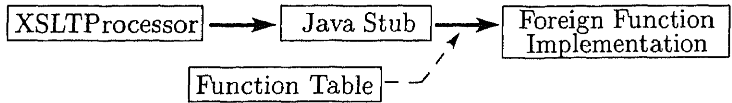


Figure 5.2: Mechanism of a method invocation

mentation of the adaptor module depends on the underlying XSLT processor.

The LotusXSL adaptor module refers to the foreign function definitions and generates stub classes in Java. A stub class is created for each element type associated with method [3]. A stub class has stub methods which work as extension functions in LotusXSL. Foreign function calls in X^2QL queries are translated into calls for appropriate stub methods. When a certain stub method is called, the stub method translates the argument type in LotusXSL into the Java type according to the Java mapping for the X^2QL, and calls the implementation of the corresponding foreign function by using the function table (See Figure 5.2.) The type of the return value is translated in the reverse direction. We have also developed a class loader to load the implementation of foreign functions through the network, and the Java Reflection API is used to manipulate classes and methods dynamically.

[3]In the current implementation, general functions are treated as if they were methods associated with a dummy class.

5.5.3 Query Translation

Basic Translation

First, we describe the translation of queries without `rank-by` and `order-by` clauses. In this case, a query is translated into a stylesheet with a corresponding template rule.

Basic constructs in `where` and `construct` clauses in X^2QL are translated into their counterparts in XSLT as illustrated in Subsection 2.3.2. Basically, template rules are generated along the element structure specified in `construct` clauses, because template rules describe the output document structure. Namely, the tags used in the outermost construction enclose the others. Element hierarchy patterns given in `where` clauses are expressed as location paths, and variables are bound as occasion demands.

Each variable binding is done with `xsl:for-each` and `xsl:variable` instructions. Generally, in template rules, some auxiliary variables are introduced in addition to ones specified in the given query. They are used to simplify the location path derivation, and to keep values depending on the current context node list.

The order of variable bindings primarily coincides with the order in which

they are used in the output construction in the query. When it gives the same order to multiple variables, their usage in Skolem functions and sorting keys is taken in consideration.

Translation of Grouping by Skolem Functions

The return value of the Skolem function has a one-to-one correspondence to the set of values given in arguments. Based on this, a grouping specified using the Skolem function is translated as shown below.

Here, `Article` elements are grouped by the `Category` values. First, an `Article` element is selected. Its `Category` value is bound to the variable `$c`. The `xsl:if` instruction says that, if it is the first `Article` element having a `Category` value, the `Group` element, collecting `Article` elements sharing the same `Category` value, is the output. If this condition is not met, no `Group` element is created. Thus, a grouping of `Article` elements by the `Category` values is attained.

```
<!-- A given query -->
where     <Document> $x </> in "articles.xml"
construct <Document>
  where     <Article>
              <Category> $c </>
            </> element_as $a in $x
  construct <Group ID=CtgID($c)> $a </>
          </>
```

(a) An example query including grouping

```
<!-- The template rule to group Articles -->
<xsl:template match="/">
  <xsl:for-each select="Document">
    <xsl:variable name="x" select="."/>
    <Document>
      <!-- Select only the first Article
           with each Category value -->
      <xsl:for-each select="Article[ not ( Category
              = preceding-sibling::Article/Category ) ]
                                        /Category">
        <xsl:variable name="c" select="node()"/>
        <xsl:variable name="CtgID">
          <xsl:value-of select="$c"/>
        </xsl:variable>

        <!-- Create the Group element -->
        <Group>
          <xsl:attribute name="Category">
            <xsl:value-of select="$c"/>
          </xsl:attribute>
          <xsl:for-each select="$x/Article[ Category = $c ]">
            <xsl:variable name="a" select="."/>
            <xsl:copy-of select="$a"/>
          </xsl:for-each>
        </Group>
      </xsl:for-each>
    </Document>
  </xsl:for-each>
</xsl:template>
```

(b) Translation Result

Translation of Rank-by and Order-by Clauses

Generally, queries with `rank-by` and `order-by` clauses are translated into two and three stylesheets, respectively. They are interpreted in the XSLT processor one by one, and the controller in Figure 5.1 controls the interpretation.

Queries having only `order-by` clauses are processed in the following two steps, and a stylesheet is associated with each step.

1. Construct the output as explained above. The difference is that the sort key values are added to elements to be sorted.
2. Sort the target elements, and remove the added sort key values.

Queries having `rank-by` clauses are processed in three steps. They are translated into three stylesheets, each with a template rule. Here, we show how Query 2 in Section 5.3 is translated. The first stylesheet is shown below. Other two stylesheets are omitted.

The template rule in the first stylesheet generates an `x2ql:order` element for each `Article` element. It has `rank` and `pos` attributes. The `rank` attribute gives the similarity of the `Article` element calculated by the foreign

function `sim_cosine()`. The `pos` attribute has the relative position of the `Article` element. It has the `Article` element as its sub-element. Moreover, we generate an `x2ql:sort` element which contains the `x2ql:order` elements. It is used in the second and third steps.

```
<!-- The first template rule -->
<xsl:template match="/">
  <xsl:for-each select="Document">
    <xsl:variable name="x" select="." />
    <Document>
      <x2ql:sort>
        <xsl:for-each select="Article">
          <xsl:variable name="a" select="." />
          <xsl:variable name="tmp1" select="position()" />
          <x2ql:order rank="{sim_cosine( $a )}" pos="{$tmp1}" >
            <xsl:copy-of select="$a" />
          </x2ql:order>
        </xsl:for-each>
      </x2ql:sort>
    </Document>
  </xsl:for-each>
</xsl:template>
```

The second template rule selects the `x2ql:order` elements, and sort them by the `rank` value. Then, we output the first N `x2ql:order` elements.

The last template rule sorts the remaining N `x2ql:order` elements by the `pos` value, and remove unnecessary elements.

5.6 Advanced Features

This section discusses advanced extension features. These features include aggregation functions and filtering functions.

5.6.1 Aggregation Functions

Grouping facility by `group-by` clause and built-in aggregation functions will be supported in XML-QL in future [18]. This subsection introduces user-defined *aggregation functions*, which calculate a value from a given set of values. Their definition is given by the following syntax, which is like the syntax for general functions.

```
aggregation   type-name aggregation-name( argument-list )
defined-by    "URI of the implementation"
```

The Java binding is also the same as with general functions.

They can be used in `construct` clauses. To group tuples, optional `group-by` clauses can be used. When a `group-by` clause is specified, tuples are grouped by the argument values, then aggregation functions compute a value from all the tuples of each group. In the absence of `group-by` clause,

aggregation functions are applied to all the tuple, so that the construct clause generates one result. Note that the variables in a group-by clause must be used as arguments to the Skolem function in the outermost tag in the construct clause.

In a query including aggregation functions, first, tuples are bound based on *patterns* specified in where clauses. For each tuple, aggregation functions are called. The return values in these calls are ignored. When there are no more tuples, they are called again with null argument values and their return values are used as the aggregated results.

For instance, a query counting articles of each year is given as follows:

```
aggregation number count( element item )
defined-by  "http://fqdn/path/funcs.Counter#count"
```

and,

```
where     <Document> $x </> in "articles.xml"
construct <Document>
  where     <Article>
              <Date> <Year> $y </> </>
            </> element_as $a in $x
  group-by  $y
  construct <Number ID=YearID($y) Year=$y>
              count( $a )
            </>
```

```
</>
```

The count is implemented as follows:

```
package funcs;
...
public class Counter
  {
  int number = 0;  // To keep intermediate result
  public XNumber count( XElement item )
    {
    XNumber result = null;
    if( item != null )
      ++ number;
    else
      {
      result = SystemModule.getXObjectFactory()
                              .createXNumber( number );
      number = 0;  // initialize itself
      }
    return result;
    }
  }
```

The above query is translated into the following XSLT template rule. This template rule is generated as follows. First, the variables in group-by clause are bound, and then aggregation functions are invoked. The rest of the query is translated in the same way as grouping queries. Note that a extension function `count_get()` is used internally. This extension function calls the aggregation function with null argument values.

```
<xsl:template match="/">
  <xsl:for-each select="Document">
    <xsl:variable name="x" select="."/>
    <Document>
      <!-- variables in the group-by clause -->
      <xsl:for-each select="Article[ not ( Year
            = preceding-sibling::Article/Year ) ]/Year">
        <xsl:variable name="y" select="node()"/>
        <xsl:variable name="YearID">
          <xsl:value-of select="$y"/>
        </xsl:variable>

        <!-- intermediate calls -->
        <xsl:for-each select="$x/Article[ Year = $y ]">
          <xsl:variable name="a" select=".">
            <xsl:variable name="tmp1" select="count( $a )"/>
          </xsl:variable>
        </xsl:for-each>
        <!-- the result of the aggregate function -->
        <xsl:variable name="tmp1" select="count_get()"/>

        <!-- the rest of the query -->
        <Number>
          <xsl:attribute name="Year">
            <xsl:value-of select="$y"/>
          </xsl:attribute>
          <xsl:copy-of select="$tmp1">
        </Number>
      </xsl:for-each>
    </Document>
  </xsl:for-each>
</xsl:template>
```

5.6.2 Variable Binding Filtering

X^2QL provides the variable binding filtering mechanism as an advanced feature. This mechanism gets the sequence of tuples from the `where` clause, generates a new sequence of tuples from it, and passes the result to the `construct` clause. This mechanism can be used by the `filter-by` clause instead of the `rank-by` and `order-by` clauses.

In sequence generation, one type of foreign functions, called *filtering functions*, are used. Their definition is given by the following syntax:

```
filter      filter-name( argument-list )
defined-by  "URI of the implementation".
```

Each filtering function is called with a queue to output filtered tuples, an input tuple, and argument values specified in its definition. The function puts tuples into the given queue based on the arguments. Note that given tuples are immutable in this process. The end of input tuples is indicated by an empty tuple.

For simple instances, a filtering function `greater` selecting tuples when the given `value` is greater than the given `base`, is defined as follows:

```
filter     greater_than( number value, number base )
defined-by "http://fqdn/path/funcs.FilterFunc#greaterThan"
```

A query using this function is specified as follows.

```
where     <Document> $x </> in "articles.xml"
construct <Document>
  where     <Article>
              <Date> <Year> $y </> </>
            </> element_as $a in $x
  filter-by greater_than( $y, 2000 )
  construct $a
          </>
```

This query is equivalent to the following query:

```
where     <Document> $x </> in "articles.xml"
construct <Document>
            <Article>
              <Date> <Year> $y </> </>
            </> element_as $a,
            $y > 2000
construct   $a
          </>
```

They are implemented as the following code in Java binding.

```
package funcs;
...
public class FilterFunc
  {
  public void greaterThan( TupleQueue queue, Tuple tuple,
                           XNumber value, XNumber base )
    {
    if( tuple == null )
      queue.close();
    else if( /* value is greater than base */ )
      queue.push( tuple );
    }
  }
```

Queries with `filter-by` clauses are processed as follows. First, a template rule is generated. In this template rule, all the tuples are generated, and output as `x2ql:tuple` elements with `x2ql:variable` elements representing each variable binding. Second, the query processor applies filtering functions to the result internally, and generates the filtered XML document. The structure is the same with the output of the first template rule. Finally, the second template rule which processes the rest of the query is applied.

For example, the above query is processed by following template rules.

```
<!-- The first template rule -->
<xsl:template match="/">
  <xsl:for-each select="Document">
    <xsl:variable name="x" select="."/>
    <x2ql:tuple>
      <x2ql:variable name="x">
        <copy-of select="$x" />
      </x2ql:variable>

      <xsl:for-each select="Article">
        <xsl:variable name="a" select="."/>
        <xsl:for-each select="Date/Year">
          <xsl:variable name="y" select="."/>
          <x2ql:tuple>
            <x2ql:variable name="a">
              <xsl:copy-of select="$a"/>
            </x2ql:variable>
            <x2ql:variable name="y">
              <xsl:copy-of select="$y"/>
            </x2ql:variable>
          </x2ql:tuple>
        </xsl:for-each>
      </xsl:for-each>
    </x2ql:tuple>
  </xsl:for-each>
</xsl:template>
```

```
<!-- The second template rule -->
<xsl:template match="/">
  <xsl:for-each select="x2ql:tuple">
    <xsl:variable name="x"
                  select="variable[@name="x"]/node()"/>
    <Document>
      <xsl:for-each select="x2ql:tuple">
        <xsl:variable name="a"
                      select="variable[@name="a"]/node()"/>
        <xsl:variable name="y"
                      select="variable[@name="y"]/node()"/>

        <xsl:copy-of select="$a"/>
      </xsl:for-each>
    </Document>
  </xsl:for-each>
</xsl:template>
```

A filtering function sorting tuples is another example. This function needs to access all the tuples. In such cases, the implementation stores the passed tuples into a collection object at each invocation. Then, when an empty tuple is passed, it moves all sorted tuples from the collection into the result queue. Therefore, the `rank-by` and `order-by` clauses can be regarded as a specialized version of this mechanism.

5.7 Discussion and Future Issues

Some XML query languages provide powerful, tag-based document structure manipulation. They are, however, generally weak in processing document contents, such as similarity-based selection, ranking, summary generation, and topic extraction. Requirements for processing document contents vary with application context, so it is very difficult to provide a complete set of content processing facilities.

This chapter proposed X^2QL (eXtensible XML Query Language), which features the inclusion of user-defined foreign functions to process document contents in the context of XML-QL-based document structure manipulation. This chapter also explained the extension mechanisms. It also covered implementation of the X^2QL query processing system on top of XSLT processors. We have been developing the prototype system as a command line Java application.

Future work includes relaxation of the restriction on X^2QL queries mentioned in Section 5.5. Another important research issue is the development of an X^2QL query processing system without need for an XSLT processor. Inheritance of element methods and other object-oriented extensions are also interesting future research topics.

Chapter 6

Conclusion

In the use of structured documents and their databases, requirement adaptability has become increasingly important. This dissertation has proposed (1) a scheme to support WWW browsing based on user profiles describing user interests, and (2) an XML query language, called X^2QL, featuring the inclusion of user-defined foreign functions. These proposed schemes enable users to get needed information within structured documents based on adaptation to requirements. This is the main contribution of this dissertation. More detailed contributions are summarized below.

Scheme to support WWW browsing

A new scheme to support WWW browsing was discussed in Chanter 4. In this scheme, the interests of individual users are explicitly represented as a user profile, and the user is presented with a view-page generated from each specified WWW page based on the user profile. View-pages make it easy for the user to quickly identify the relevant substructures.

For this purpose, first the logical tree, which represents the logical structure of the WWW page, is derived. Substructures irrelevant to the user profile are then pruned away. Finally, a view-page is generated from the pruned logical tree. The view-pages generated are ensured to be valid HTML documents. In these steps, some HTML characteristics are also used.

This chapter also has shown how view-pages can be realized in a non-intrusive way in the current WWW browsing environment. Beyond that, the experimental evaluation for this scheme has shown good results. These results reveal that the proposed WWW browsing scheme is feasible.

Extensible XML query language

X^2QL, which is an extensible XML query language based on XML-QL, is

proposed in Chapter 5. This extensibility is made possible by the inclusion of user-defined foreign functions written in programming languages. This extensibility makes it possible to integrate application-oriented, high-level content processing facilities into querying documents.

General functions and element methods, which are basic foreign functions, are defined. Specifically, element methods have brought about some object-oriented features. Additionally, aggregation functions and variable binding filtering mechanism via filtering functions as advanced features have also been introduced. Their Java bindings have been defined.

This chapter also discussed a scheme for implementing an X^2QL query processor on top of an XSLT processor. This includes translation procedures of X^2QL queries into XSLT template rules. Implementations of foreign functions are independent from features depending on individual XSLT processors.

The proposed schemes show promise in realizing requirement adaptability in browsing and querying structured documents. Most important is that users can manipulate structured documents in their own way. I hope that in feature requirement adaptability, all users will be able to flexibly use structured documents.

Bibliography

[1] P. Atzeni, A. Mendelzon, and G. Mecca, editors. *The World Wide Web and Databases.* Springer-Verlag, 1998.

[2] D. Suciu and G. Vossen, editors. *proceedings of Third International Workshop on the Web and Databases (WebDB 2000).* Springer-Verlag, 2000.

[3] N. Shinagawa and H. Kitagawa. Dynamic generation and browsing of virtual www space based on user profiles. In *Proceedings og 5th International Computer Science Conference (ICSC'99), LNCS 1749*, pages 93–108. Springer-Verlag, 1999.

[4] H. Kitagawa N. Shinagawa and J. Kawada. Dynamic generation and browsing of www view-pages based on user profiles (in japanese). *IPSJ Transactions on Databases*, 41(SIG 6 (TOD7)):22–362, 2000.

[5] H. Kitagawa N. Shinagawa and Y. Ishikawa. X^2ql: An extensible xml query language supporting user-defined foreign functions. In *Proceedings of 2000 ADBIS-DASFAA Symposium on Advances in Databases*

and Information Systems, LNCS 1884, pages 251–264. Springer-Verlag, 2000.

[6] ISO. Information processing – text and office system – standard generalized markup language (SGML), 1986. ISO 8879.

[7] ISO. Hypermedia/time-based structuring language (Hytime), 1992. ISO/IEC 10744.

[8] *HTML 4.01 Specification*, 1999. http://www.w3.org/TR/html401.

[9] *Extensible Markup Language (XML) 1.0 (Second Edition)*, 2000. http://www.w3.org/TR/REC-xml.

[10] *XHTML 1.0: The Extensible HyperText Markup Language – A Reformulation of HTML 4 in XML 1.0*, 2000. http://www.w3.org/TR/xhtml1.

[11] S. Abiteboul. Querying semi-structured data. In *Proceedings of 6th International Conference on Data Theory (ICDT'97)*, pages 1–18, 1997.

[12] P. Buneman. Semistructured data. In *Proceedings of 16th ACM Symposium on Principles of Database Systems (PODS'97)*, pages 117–121, 1997.

[13] G. Salton. *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer.* Addison-Wesley, Reading, MA, 1989.

[14] D. K. Harman. Overview of the first trec text retrieval conference. *Proceedings of TREC-1*, pages 1–20, 1992.

[15] A. Singhal, C. Buckley, and M. Mitra. Pivoted document length normalization. *Proceedings of the 19th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 21–29, 1996.

[16] ISO. Database language (sql), 1992. ISO/IEC 9075.

[17] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann Publishers, 2000.

[18] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for xml. *Proceedings of the Eighth International World Wide Web Conference (WWW8), Computer Networks*, 31(11-16):1155–1169, 1999.

[19] *XSL Transformations (XSLT) Version 1.0*, 1999. http://www.w3.org/TR/xslt.

[20] *Extensible Stylesheet Language (XSL) 1.0 (Candidate Recommendation)*, 2000. http://www.w3.org/TR/xsl/.

[21] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0*, 1999. http://www.w3.org/TR/xpath.

[22] IBM alphaWorks. Lotusxsl. http://www.alphaWorks.ibm.com/tech/LotusXSL/.

[23] Apache Software Foundation. Xalan. http://xml.apache.org/xalan/.

[24] AltaVista. http://www.altavista.com/.

[25] Google. http://www.google.com/.

[26] Yahoo. http://www.Yahoo.com/.

[27] G. G. Robertson amd J. D. Mackinlay S. K. Card. The information visualiser, an information workspace. In *Proceedings of Human Factors in Computing Systems Conference on Reaching through Technology (CHI'91)*, pages 181–187, 1991.

[28] M. B. Spring K. A. Olson, R. R. Corfhage. K. M. Sochats and J. G. Williams. Visualization of a document collection: The vibe system. *Information Processing and Management*, 29(1):69–81, 1993.

[29] M. Hearst. Tilebars: Visualization of term distribution information in full text information access. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*, pages 59–66, 1995.

[30] L. S. Heath L. T. Nowell, R. K. France. D. Hix and E. A. Fox. Visualizing search results: Some alternatives to query-document similarity. In *Proceedings of the 19th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 67–75, 1996.

[31] M. A. Hearst and J. O. Pedersen. Reexamining the cluster hypothesis: Scatter/gather on retrieval results. In *Proceedings of the 19th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 76–84, 1996.

[32] S. Miike, E. Itoh, K. Ono, and K. Sumita. A full-text retrieval system with a dynamic abstract generation function. *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 152–161, 1994.

[33] C. D. Paice. Constructing literature abstracts by computer: Techniques and prospects. *Information Processing and Management*, 26(1):171–186, 1990.

[34] C. D. Paice and P. A. Jones. The identifiation of important concepts in highly structured technical papers. In *Proceedings of the 16th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 69–78, 1993.

[35] R. Willkinson. Effective retrieval of structured documents. *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 311–317, 1994.

[36] G. Salton, J. Allan, and C. Buckley. Approaches to passage retrieval in full text information systems. *Proceedings of the 16th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 49–58, 1993.

[37] M. Kaszkiel and J. Zobel. Passage retrieval revisited. *Proceedings of the 20th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 21–29, 1997.

[38] J. Zobel, A. Moffat, R. Wilkinson, and R. Sacks-Davis. Efficient retrieval of partial documents. *Information Processing and Management*, 31(3):361–377, 1995.

[39] A. Tombros and M. Sanderson. Advantage of query biased summarization in information retrieval. *Proceedings of the 21th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 2–10, 1998.

[40] H. Mochizuki M. Okumura and H. Nanba. Query-biased summarization based on lexical chaining. In *Pacific Association for Computational Linguistics (PACLING'99)*, pages 324–334, 1999.

[41] N. Ashish and C. A. Knoblock. Wrapper generation for semi-structured internet sources. *ACM SIGMOD Records*, 26(4):8–15, 1997.

[42] Y. Jiang D. W. Embley and Y.-K. Ng. Record-boundary discovery in web documents. In *Proceedings ACM SIGMOD International Conference on Management of Data '97*, pages 8–15, 1997.

[43] R. B. Doorenbos N. Kushmerick, D. S. Weld. Wrapper induction for information extraction. In *Proceedings of the Fifteenth International*

Joint Conference on Artificial Intelligence (IJCAI '97), volume 1, pages 23–29, 1997.

[44] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1–2):15–68, 2000.

[45] B. Adelberg. Nodose - a tool for semi-automatically extracting semi-structure data from text documents. In *Proceedings of ACM-SIGMOD International Conference on Management of Data '98*, pages 283–294, 1998.

[46] B. Adelberg and M. Denny. Nodose version 2.0. In *Proceedings ACM SIGMOD International Conference on Management of Data '99*, pages 559–561, 1999.

[47] T. M. Mitchell T. Joachims, D. Freitag. Webwatcher: A tour guide for the world wide web. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI 97)*, volume 1, pages 770–777, 1997.

[48] A. S. Vivacqua H. Lieberman, N. W. Van Dyke. Lets browse: A collaborative browsing agent. In *International Conference on Intelligent User Interfaces*, pages 65–68, 1999.

[49] J. Muramatsu M. J. Pazzani and D. Billsus. Syskill and webert: Identifying interesting web sites. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of*

Artificial Intelligence Conference (AAAI 96/IAAI 96), volume 1, pages 54–61, 1996.

[50] M. Balabanovic and Y. Shoham. Learning information retrieval agents: Experiments with automated web browsing. In *AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments*, pages 213–18, 1995.

[51] M. Marchiori (chair), editor. *QL'98 - The Query Languages Workshop.* W3C, 1998. http://www.w3.org/TandS/QL/QL98/.

[52] H. Hosoya and B. C. Pierce. Xduce: An xml processing language (preliminary report). In *In Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, pages 111–116, 2000.

[53] J. Robie, J. Lapp, , and D. Schach. Xml query language (xql). In *The Query Languages Workshop (QL'98)*, 1998. http://www12.w3.org/TandS/QL/QL98/pp/xql.html.

[54] J. Robie D. D. Chamberlin and D. Florescu. Quilt: An xml query language for heterogeneous data sources. In *Proceedings of the Third International Workshop on the Web and Databases (WebDB 2000)*, 2000. 53–62.

[55] Tamino. http://www.tamino.com/.

[56] excelon. http://www.excelon.com/.

[57] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[58] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data convention! *Proceedings of ACM-SIGMOD International Conference on Management of Data '98*, pages 414–425, 1998.

[59] P. Bunemank, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. *Proceedings of ACM-SIGMOD International Conference on Management of Data '96*, pages 506–516, 1996.

[60] J. McHugh and J. Widom. Query optimization for xml. In *Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases*, Edinburgh, Scotland, September 1999.

[61] H. Garcia-Molina Y. Papakonstantinou and J. Widom. Object exchange across heterogeneous information sources. In *Proc. 11th Data Engineering Conference*, pages 251–260, 1995.

[62] M. Fernandez P. Buneman and D. Suciu. Unql: A query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.

[63] M. Marchiori P. Fankhauser and J. Robie (eds.). Xml query requirements (working draft), 2000. http://www.w3.org/TR/xmlquery-req.

[64] M. Fernandez and J. Robie (eds.). Xml query data model (working draft), 2000. http://www.w3.org/TR/query-datamodel.

[65] J. Simeón P. Fankhauser. M. Fernandez. A. Malhotra, M. Rys and P. Wadler. The xml query algebra (working draft), 2000. http://www.w3.org/TR/query-algebra.

[66] J. Cowan and R. Tobin. Xml information set (working draft), 2000. http://www.w3.org/TR/xml-infoset.

[67] D. C. Fallside (ed.). Xml schema part 0: Primer, 2000. http://www.w3.org/TR/xmlschema-0/.

[68] M. Maloney H. S. Thompson, D. Beech and N. Mendelsohn. Xml schema part 1: Structures, 2000. http://www.w3.org/TR/xmlschema-1/.

[69] P. V. Biron and A. Malhotra. Xml schema part 2: Datatypes, 2000. http://www.w3.org/TR/xmlschema-2/.

[70] T. Joachims R. Armstrong, D. Freitag and T. Mitchell. Webwatcher: A learning apprentice for the world wide web. *AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments*, pages 6–12, 1995.

[71] G. Salton, A. Wong, and C. S. Yang. A vector space model for information retrieval. *Journal of the American Society for Information Science*, 18(11):613–620, 1975.

[72] R. K. Belew B. T. Bartell, G. W. Cottrell. Optimizing similarity using multi-query relevance feedback. *Journal of the American Society for Information Science (JASIS)*, 49(8):742–761, 1998.

[73] R. Sacks-Davis, T. Arnold-Moore, and J. Zobel. Database systems for structured documents. *International Symposium on ADTI '94*, pages 272–283, 1994.

[74] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. *International Journal on Digital Libraries*, 1(1):54–67, 1997.

[75] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the boat with strudel: Experiences with a web-site management system. *Proceedings of ACM-SIGMOD International Conference on Management of Data '98*, pages 414–425, 1998.

[76] I. Mani and M. T. Maybury, editors. *Advances in Automatic Text Summarization.* MIT Press, 1999.

[77] H. A. Hearst. Subtopic structuring for full-length document access. *Proceedings of ACM-SIGIR '93*, pages 59–68, 1993.

[78] J. Zobel and A. Moffat. Exploring the similarity space. *ACM SIGIR Forum*, 32 No. 1:18–34, 1998.

研究業績リスト

(1) 学術雑誌論文

1. 品川徳秀, 北川博之, 川田純. ユーザプロファイルに基づくビューページの動的生成によるWWW 閲覧支援, 情報処理学会論文誌：データベース, Vol.41, No.SIG 6 (TOD7), pp.22-36, 2000 年 10 月.

(2) 査読付き国際会議論文

1. Norihide Shinagawa and Hiroyuki Kitagawa. Dynamic Generation and Browsing of Virtual WWW Space Based on User Profiles, *Proc. 5th International Computer Science Conference (ICSC'99)*, Hong Kong, China, Springer-Verlag, LNCS 1749, pp.93-108, Dec. 1999.

2. Norihide Shinagawa, Hiroyuki Kitagawa, and Yoshiharu Ishikawa. X^2QL: An eXtensible XML Query Language Supporting User-defined Foreign Functions, *Proc. 2000 ADBIS-DASFAA Symposium on Advances in Databases and Information Systems*, Prague, Czech Republic, Springer-Verlag, LNCS 1884, pp. 251-264, Sep. 2000.

(3) 査読付き国内会議論文

1. 品川徳秀, 北川博之, 石川佳治. 変換規則と記述内容処理関数に基づく構造化文書操作記述方式, アドバンスト・データベース・シンポジウム (ADBS'99), 情報処理学会シンポジウムシリーズ, Vol.99, No.19, pp.123-132, 1999 年 12 月.

2. 品川徳秀, 北川博之, 石川佳治. 拡張可能 XML 問合せ言語 X^2QL とその処理系, 電子情報通信学会データ工学研究会, 第 11 回データ工学ワークショップ論文集 (DEWS2000) (CD-ROM), 2000 年 3 月.

(4) 研究会発表

1. 品川徳秀, 北川博之. 内容解析に基づく文書構造の自動抽出, 情報処理学会第 116 回データベースシステム研究会・電子情報通信学会データ工学研究会合同研究会, 情報処理学会研究報告 98-DBS-116(2), Vol.98, No.58, pp.157-164, 1998 年 7 月.

2. 品川徳秀, 北川博之. ユーザ視点に即した仮想 WWW ページの動的生成による閲覧支援, 情報処理学会第 119 回データベースシステム研究会・電子情報通信学会データ工学研究会合同研究会, 情報処理学会研究報告 99-DBS-119, Vol.99, No.61, pp.425-430, 1999 年 7 月.

3. 倉垣公一, 品川徳秀, 北川博之. 拡張可能 XML 問合せ言語 X^2QL を用いた WWW アプリケーション統合, 情報処理学会第 122 回データベースシステム研究会・電子情報通信学会データ工学研究会合同研究会, 電子情報通信学会技術報告 DE2000-71～90, Vol.100, No.228, pp.71-78, 2000 年 7 月.

4. 品川徳秀, 北川博之, 石川佳治. 拡張可能 XML 問合せ言語 X^2QL における外部関数の評価モデル, 情報処理学会第 122 回データベースシステム研究会・電子情報通信学会データ工学研究会合同研究会, 電子情報通信学会技術報告 DE2000-41～70, Vol.100, No.227, pp.9-16, 2000 年 7 月.

5. 倉垣公一, 品川徳秀, 北川博之, 石川佳治. XML に基づく WWW アプリケーション統合のための処理記述方式, 電子情報通信学会データ工学研究会, 情報情報通信学会技術報告 DE2000-91 〜101, Vol.100, No.351, pp.17-24, 2000 年 10 月.

(5) 大会発表

1. 品川徳秀, 藤原譲. 概念の意味構造とその表現モデル unigram, 情報処理学会 第 54 回全国大会, 講演論文集 (3), pp.281-282, 1997 年 3 月.

2. 品川徳秀, 森嶋厚行, 北川博之. 構造化文書とデータベースの統合利用方式の研究— ランキングを含む問合せの記述とその処理方式 —, 情報処理学会 第 55 回全国大会, 講演論文集 (3), pp.11-12, 1997 年 9 月.

3. 品川徳秀, 北川博之. 文書構造解析に基づく部分文書検索, 情報処理学会 第 57 回全国大会, 講演論文集 (3), pp.95-96, 1998 年 10 月.

4. 品川徳秀, 北川博之, 石川佳治. 文書間の類似度の概念を含む構造化文書操作記述方式, 情報処理学会 第 59 回全国大会, 講演論文集 (3), pp.43-44, 1999 年 9 月.

5. 永井孝明, 北川博之, 品川徳秀, 石川佳治. 拡張可能 XML 問合せ言語 X^2QL を用いた G-XML データ処理, 情報処理学会 第 61 回全国大会, 講演論文集 (3), pp.61-62, 2000 年 10 月.

6. 川田純, 品川徳秀, 北川博之, 石川佳治. 拡張可能 XML 問合せ言語 X^2QL 処理系の開発, 情報処理学会 第 61 回全国大会, 講演論文集 (3), pp.17-18, 2000 年 10 月.

(6) 特許出願

1. 北川博之, 品川徳秀. XML 文書問い合わせ言語処理装置, 特願 2000-165800, 2000 年 6 月.

(7) その他

1. 北川博之, 品川徳秀, 石川佳治. 文書構造変換規則と記述内容処理関数を用いた構造化文書操作系, 筑波大学「東西言語文化の類型論」特別プロジェクト研究報告書, pp.237-258, 2000 年 3 月.