INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600
USING VIRTUAL DOCUMENTS TO MODEL PERSONAL WEB SPACES

by

Mabel C. Y. Wong

A thesis submitted in conformity with the requirements for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

© Copyright by Mabel C. Y. Wong 1998
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-40752-7
To my grandma and my mother
Using Virtual Documents to Model Personal Web Spaces

Mabel C. Y. Wong

Master of Science

Graduate Department of Computer Science

University of Toronto

1998

Abstract

The World Wide Web has become one of the largest global information resources in today's world. Given this enormous pool of distributed information, manual navigation of the Web is no longer adequate to retrieve information effectively. Moreover, there is a further problem of presenting and organizing the retrieved information in a manner that facilitates understanding, browsing, and subsequent accesses by individuals or groups. It is desirable to develop tools that assist in automatically gathering information of interest from the Web and building customized, integrated views over the information collections for personal use. We call a view constructed in this way a personal Web space.

In this thesis, we propose to use Virtual Documents to model personal Web spaces. A Virtual Document is defined by a declarative specification, and its content and structure are computed dynamically from the results of a search procedure applied to the underlying Web. A concrete Virtual Document is a network of Web pages generated upon a materialization request. An abstract Virtual Document model, which is reusable for a family of related applications, is created to capture the main properties of a personal Web space.

We present the Virtual Document Specification language designed for the model. The language enables users to gather information from the Web by querying the Web as a database, to choose presentation and navigation structures for representing the collections of information as hypertext and to specify an integrated hypertext network structure for organizing and connecting these representations. This thesis specifies the syntax and semantics of the language, describes current prototype implementation, and discusses four applications of Virtual Documents: querying multiple index servers, building customized global views in a Web site, sharing views in a workgroups environment, and organizing information into personal navigation spaces.
Acknowledgements

I would like to express my gratitude to a number of people who contributed to the successful completion of this thesis or helped me to learn and grow during the seven years of my stay in Toronto.

I am grateful to my supervisor, Alberto Mendelzon, for giving me a chance to resume my graduate studies. This is very important to me. I have to thank him for his understanding and tolerance during the long period of my adjustments. Also, many thanks to his invaluable comments on the thesis.

I would also like to thank Charles Clarke for being the second reader of my thesis, and to Eric Harley for proof-reading it.

I owe thanks to other DB group graduate students George Mihaila, Shili Yang, Gustavo Arocena for their help in the development of the project, to Bing He for his encouragement, and to Attila Barta for his warm ‘hello’ whenever we meet.

I want to thank John DiMarco and the cslab staff for their technical support, Kathy Yen for her patience and concerns for me when helping me with administrative procedures in these years, and Debby Repka for her encouraging words. I also gratefully acknowledge the financial support I received from the Commonwealth Scholarship and the advice of my previous supervisor, Songnian Zhou.

I am deeply grateful to Annie, Charles, Luk for their support and help throughout my studies, and also to C.K. Poon and W.K. Lo for their advice.

Moreover, I want to thank Angel, Sue, Irene, Alice, Nastaran, Grace He, Raymond, Don, Ronnie and Chi Lok for their friendship.

Many thanks to the Chinese Baptist Church, the Chinese Presbyterian Church and Knox on Spadina for their spiritual guidance and fellowship. I especially owe thanks to Beatrice and Mr. Choy for their support and advice.

I also have to thank Ms. Tung and Mr. Leung for their kind help.

In addition, I want to express my gratitude to the Yau’s family, for their hospitality and for sharing this learning experience with me with patience and tolerance.

I owe thanks to my old friends in Hong Kong Saminda, Theresa, Grace and Susanna. They show me that a faithful friend is really a strong defense.

Special thanks to my grandma and my mother for always loving me and for enduring all these years during which I have been away from home. I dedicate this work to them, with all my love.

Above all, I have to thank the greatest friend in my life, Jesus. He was with me while I was reading every paper and writing every page of the thesis. Though this thesis might not be the best work I have ever done and these years in Canada might not be the most productive years in my life thus far, He has brought me back to Him through hard lessons and given me a new life, a new perspective and a new meaning to live for. I know that He will continue to help me to become the person He wants me to be. Thank you, Lord.
# Contents

1 Introduction

1.1 Motivation and Objectives ................................................. 1
1.2 Related Work ..................................................................... 3
  1.2.1 Information Discovery and Gathering ................................. 3
  1.2.2 Personal Information Organization and Access ...................... 4
  1.2.3 Information Reuse .......................................................... 6
  1.2.4 Model-Driven Generation of Web Applications ..................... 7
  1.2.5 Hypertextual Views, Structural Views, Scripted Views .......... 7
1.3 Design Principles .............................................................. 9
1.4 Overview .......................................................................... 10
  1.4.1 Data Model .................................................................. 10
  1.4.2 Virtual Document Operations .......................................... 11
  1.4.3 System Architecture ..................................................... 12
1.5 Organization of the Thesis .................................................... 14

2 Data Model

2.1 Overview .......................................................................... 15
2.2 Virtual Documents ............................................................. 17
2.3 Result-Sets ....................................................................... 18
  2.3.1 A Relational model of the WWW ...................................... 18
  2.3.2 Base Result-Sets and the Base of a Virtual Document ........ 21
  2.3.3 Perspectives ................................................................. 22
  2.3.4 Ordering .................................................................. 26
### Table of Contents

4.5 Practical Examples ........................................... 109  
   4.5.1 Query Report on Data Warehousing ......................... 109  
   4.5.2 Web-Team Library ........................................ 114

5 Implementation .................................................. 117  
   5.1 Translation to Java ........................................... 117  
   5.2 Materialization ............................................. 119  
   5.3 CGI Interface ................................................ 121  
   5.4 Stand-alone Application ....................................... 125

6 Summary, Conclusions, and Future Extensions .................. 126  
   6.1 Summary .................................................... 126  
   6.2 Conclusions ................................................ 127  
   6.3 Future Extensions ............................................ 129

A VD Specification Language .................................... 132  
   A.1 Syntax .................................................... 132  
   A.2 Semantics .................................................. 138

Bibliography ...................................................... 145
List of Figures

1.1 The System Architecture .......................... 13
2.1 The Virtual Document Model ........................ 16
2.2 The Base of a Virtual Document .................. 22
2.3 The Graph Structure for a sub-Web ............... 28
2.4 Paths Materialized in the Document-Collection .... 29
2.5 Operations: Grouping and Comparison ............. 30
2.6 Deriving Views From Result-Sets .................. 31
2.7 Different Types of Links in a Virtual Document .... 33
2.8 Navigational Structure in a DocumentList View .... 42
2.9 The VD Graph Structure ........................... 49
3.1 The Base Result Set Used in All the Examples of Simple VD Graph ... 56
3.2 tlist1_simple - The VD Specification ............... 58
3.3 tlist1_simple - Presenting Collected Information as a Table .... 59
3.4 tlist2_simple - Grouping Collected Information into Subtables .... 59
3.5 tlist2_simple - The VD Specification ............... 62
3.6 tlist3_simple - The VD Specification ............... 66
3.7 tlist3_simple - Comparing Two Result Sets .......... 67
3.8 d1_simple - The VD Specification .................. 69
3.9 d1_simple - Creating a Compact View for a Document Collection .... 71
3.10 d1_simple - Contents of the Index Page in the Context Frame .... 71
3.11 d1_simple - Choosing a Document to be Browsed in the Doc-Display Frame 72
3.12 d1_simple - Navigating Outside the View Context in the Doc-Display Frame 72
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>ResultSet Manipulation and View Creation Operators</td>
<td>112</td>
</tr>
<tr>
<td>4.2</td>
<td>$dwh_report$ - RSV Relationship Diagram</td>
<td>112</td>
</tr>
<tr>
<td>4.3</td>
<td>VD Graph Primitives</td>
<td>113</td>
</tr>
<tr>
<td>4.4</td>
<td>$dwh_report$ - VD Graph Diagram</td>
<td>113</td>
</tr>
<tr>
<td>4.5</td>
<td>$web_team_lib$ - RSV Relationship Diagram</td>
<td>116</td>
</tr>
<tr>
<td>4.6</td>
<td>$web_team_lib$ - VD Graph Diagram</td>
<td>116</td>
</tr>
<tr>
<td>5.1</td>
<td>The VD Manager HomePage</td>
<td>121</td>
</tr>
<tr>
<td>5.2</td>
<td>The Form for Editing and Compiling a VD Specification</td>
<td>122</td>
</tr>
<tr>
<td>5.3</td>
<td>The Form for Materializing a Virtual Document</td>
<td>123</td>
</tr>
<tr>
<td>5.4</td>
<td>The Form for Viewing a VD Specification</td>
<td>124</td>
</tr>
<tr>
<td>5.5</td>
<td>The Menu for Viewing the Content of a Virtual Document</td>
<td>125</td>
</tr>
<tr>
<td>A.1</td>
<td>Syntax of VD Specification Language (part 1 of 3)</td>
<td>133</td>
</tr>
<tr>
<td>A.2</td>
<td>Syntax of VD Specification Language (part 2 of 3)</td>
<td>134</td>
</tr>
<tr>
<td>A.3</td>
<td>Syntax of VD Specification Language (part 3 of 3)</td>
<td>135</td>
</tr>
<tr>
<td>A.4</td>
<td>The Hypertext Structure for a DocumentView</td>
<td>140</td>
</tr>
<tr>
<td>A.5</td>
<td>The Hypertext Structure for a DocumentListView</td>
<td>141</td>
</tr>
<tr>
<td>A.6</td>
<td>The Hypertext Structure for a DocumentTreeView</td>
<td>143</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation and Objectives

The ambition of the World-Wide Web (WWW or W3) project [5] to bring a global information universe into existence is being realized. The Web has gained widespread acceptance as a new way of making information publicly available on the Internet, and has become one of the largest global information resources in today’s world. Given this enormous pool of distributed information, manual navigation of the Web is no longer adequate to retrieve information effectively. Moreover, there is a further problem of presenting and organizing the retrieved information in a manner that facilitates understanding, browsing, and subsequent accesses by individuals and groups. It is clearly desirable to develop tools that assist in automatically gathering the portions of the Web that are of interest and building customized, integrated views over the collected information for personal use.

When we talk about views, we refer to hypertextual views that are adapted from database views which have been studied extensively in the context of the relational model [11, 40] and the object-oriented database model [1, 35]. In fact, hypertextual views resemble virtual structures [10] for handling dynamic information in hypertext systems. Virtual structure is defined intentionally by only specifying a description, rather than the exact identity of their components. The actual components are determined by a search procedure that is applied to the underlying base hypertext entity. Whenever the virtual structure is instantiated, its content and structure are recomputed to construct a new
hypertext entity from the results of the search procedure. Similarly, hypertextual views provide the same flexibility for representing dynamic information resources in the context of the Web. In this thesis, we focus on building customized, integrated views of dynamic Web-based information collections in the form of hypertextual views over the Web; such a view is called a personal Web space (PWS).

One of the motivations for a personal Web space, as mentioned before, is to provide automation support for gathering the portions of Web-based information that are of interest to a user or group of users. This issue of resource discovery and gathering was addressed by our group's previous work with WebSQL [28, 29, 31, 41], a declarative language that models the Web as a relational database and allows users to retrieve information from the Web by querying it. We would like to build on this work to provide support for constructing a personal Web space from the information captured in the query results.

In addition, there are several other motivations for a personal Web space, which primarily concern adapting the information to user needs:

1. A personal Web space reduces cognitive overhead and enables more focused and effective access for users by gathering only the information that is of interest and filtering out unwanted information. This could also be used as a security measure for restricting access to a subset of information in the underlying Web to protect against unauthorized access.

2. It provides a flexible way to form an integrated view on a subject domain by grouping arbitrary portions of the Web into a logical unit. Users can freely combine the results of multiple queries that retrieve information from different index servers and/or sub-networks in any accessible Web sites.

3. It allows users to choose their own representations and structures to present and organize the collected information in a manner that facilitates understanding, browsing, and access.

4. To cater to diverse requirements, multiple views can be simultaneously provided in a personal Web space for representing the same set of information differently.
5. Users can control when to re-instantiate a personal Web space to obtain a current snapshot of the underlying Web. Physical pages of the personal Web space can be re-generated every time users access it or the last materialized version can be used until users request re-materialization.

6. It provides a way to create a “stand-alone” Web site from the underlying Web. Representative nodes in a personal Web space can contain hyperlinks that point to the source documents in the Web or they can be constructed by replicating, restructuring, extracting information from the source documents or, in general, deriving new information from them by applying some function. Local caching can also be supported to enable fast access to frequently accessed information.

The objectives of this thesis are to propose a data model of personal Web space, design a specification language for it, implement a system that materializes a personal Web space for a given specification, and suggest its applications.

In the following section, we will first present a survey of some existing approaches that address individual issues of the problem. The design principles of our work will be stated in Section 1.3. We will then give an overview of the proposed solution in Section 1.4. Finally, the organization of the rest of the thesis will be outlined in Section 1.5.

1.2 Related Work

1.2.1 Information Discovery and Gathering

There are many index servers (e.g. AltaVista, Excite, HotBot, WebCrawler, Yahoo) deployed on the Internet that help users locate information of interest. However, individual index servers are insufficient because (1) no index server is best for every request, (2) none of them can index all the documents in the world — a document not indexed in one server may be found in another, (3) different index servers use different proprietary ranking algorithms — a document not returned in the top N results by an index server may be returned in the top N results by another. To alleviate these problems, meta-search tools like MetaCrawler [37] were introduced. They integrate query results from multiple index
servers by using some proprietary algorithms and present the ranked batch of results to users. But these tools lack flexible options for combining and manipulating the query results collected from individual servers in addition to integration. On the other hand, Web query languages: WebSQL [28, 29, 31, 41], W3QL [22, 23], Weblog [24], integrate full-text content-based search with structure and topology-based queries, thus enabling users to search directly from known Web-sites and specify test conditions for selection and filtering. Among them, WebSQL and W3QL also allow users to query individual index servers and provide a uniform interface that hides the peculiarities of query interfaces of different servers. Despite the expressiveness of these languages, they were primarily designed for resource discovery, and their support for personal use of the gathered information is still inadequate. W3QS, the system implementing W3QL, supports storing W3QL queries as view definitions and automatic, periodic update of views. Yet it only has a fixed, limited set of view representations (a table and a file directory tree) and operations applicable on the query results (projection of a column and union of sub-query results). Weblog is most powerful in its capability for information restructuring and generation of new pages from restructured results, but the prolog-like programming approach makes it hard to grasp the hypertext structures being manipulated. Also, the pages are specified individually and it does not have an organization scheme.

In summary, it is desirable to have (1) a query language that provides a uniform user-interface for querying index servers and integrates content-based search with automatic navigation, (2) customizable meta-search service for gathering information from multiple index servers and/or known Web-sites, (3) more flexibility, tailorability, and extensibility in the support of operations for manipulating results of multiple queries and in the support of view representations, and (4) an organization scheme for the collections of information.

1.2.2 Personal Information Organization and Access

The problem of remembering where interesting documents are located is addressed in the idea of bookmarks [33] in Netscape. Once appropriate documents are found, users can store the Universal Resource Locators (URLs) associated with them in hierarchi-
cal folder structures for subsequent reference. This basic mechanism has been extended by many systems to build personal information systems. For example, Warmlist [21] augments Mosaic's hot-list (the equivalent of Netscape's bookmarks) by automatically caching the full text of all hot-list entries and organizing them into a hierarchical Unix file directory structure according to a user's preferences. With the aid of another tool, full-text content-based search is supported. On the other hand, WebTagger [19] adopts a semi-automated approach: users assign categories to URLs, and WebTagger automatically builds a lattice indexing structure internally, which is used for information access. One problem of providing new bookmarking services is the implementation of their interfaces to existing browsers without making any changes to the browsers themselves. The Warmlist tool is platform-dependent and they solve the problem by using named pipes in Unix. WebTagger, however, is implemented as a proxy-based system to ensure universal access without installation of plug-ins. Users can configure their Web browsers to use the WebTagger proxy, which intercepts each standard Webpage served to the browser and inserts a set of interface buttons at the top of the page. This approach has the drawback of slowing down Web page delivery, and users may not like the buttons added to every page. We believe that the bookmark mechanism is very useful for collecting documents when they are discovered serendipitously during exploration. But instead of improving it, we would like to experiment on a complementary approach that integrates Web query information gathering and the customized organization of the collected results to construct personal views over the Web. The desired views are not only for facilitating users' access, but are also readily publishable and queriable like any Web documents.

There are other non-bookmarking systems developed for adding structures to Web documents. In the hypermedia community, the VIKI project [27] introduces spatial hypertext, a geometric and visual structuring paradigm, to provide expressive superstructures for organizing and interpreting information. However, like in many other innovative information spaces, the information discovery process is not part of their design. User-supplied URLs need to be pre-processed so that VIKI object may refer to Web documents. Afterwards, users can start Web browsers to access these documents by interacting with VIKI objects. In the Web server community, the problem of imposing navigation struc-
tures to guide the users in browsing is addressed. An experimental WWW server called Perplex [14] supports local configuration files for defining tour structures that facilitate access to remote pages. The server provides a mechanism to retrieve the original document upon request, add hyperlinks to its contents according to its position in the tour structure, and send it to the client. The problem with such configuration files is that nodes in a tour structure are explicitly mapped to the exact URLs of the corresponding Web documents. This makes them difficult to write and maintain, especially in response to rapidly changing information in the Web.

1.2.3 Information Reuse

Sometimes, users do not just want to remember the URLs of the Web documents found, they want to record or reuse parts of their contents. Nabbit [26] provides this support by allowing users to create their own personal Web notebooks using a copy-and-paste paradigm. It works with two Netscape windows, one for browsing (input) and one for collecting information (output). A user can select with the mouse the part of the page he wants to remember, and click on a "Copy" button on the Nabbit’s window. Everything copied is being assembled into one HTML page, which can be viewed on the output window. The whole process is integrated with the browsing process, thus providing convenience. However, since the tool is intended only for taking notes, the assembled pages are static and do not reflect the current state of the original documents. On the other hand, the RIO project adopts a SGML document authoring approach [45] to gather information from multiple, distributed, heterogeneous data sources and combine it dynamically to produce a virtual document [34]. Their approach is targeted for exploiting the internal structure of data sources. A virtual document is generated on demand, and can include pieces of information from other documents or other data sources, transform them, and reflect their updates. The instructions for its construction are stored in a document prescription, which is written as an SGML document and processed by a document interpreter to generate or update it. They also plan to design an editor to facilitate the writing of the prescriptions. Nevertheless, writing SGML code, data queries, and transformation instructions are three non-trivial, non-intuitive tasks. It remains
unclear how the editor can be designed to really make it easier for non-programmers to write documents. Furthermore, their query language acts more like a selection on the results from data sources, as their focus is not on resource discovery and they assume it is done by the data source.

1.2.4 Model-Driven Generation of Web Applications

The problem of automated support for generation of a Web application has been addressed in hypermedia research. HSDL [20], OOHDM [36], RMCae [6] based on the RMM methodology [18], and AutoWeb [9] based on the HDM-lite methodology all provide tools for generating physical pages of a Web application directly from the design model specified for the application. This approach has the advantage of avoiding problems in manual construction of pages such as link inconsistency.

1.2.5 Hypertextual Views, Structural Views, Scripted Views

Hypertextual views over Web sites are supported by Web-site management systems like Araneus [4] and STRUDEL [7]. In Araneus, hypertextual views over the Web can be constructed in four processes. Firstly, data are extracted from the Web sites of interest and mapped to page schemes of their data model by using an Editor language. The page schemes together form the scheme of a structured Web site. Secondly, another language called Ulixes can be used to build relational views over the Web based on the scheme, and store them in a relational database. Thirdly, further database views can be built based on traditional SQL queries. And finally, hypertextual views based on page schemes can be defined from the relational views by using a language PENEOPE. We find Araneus' scheme approach comprehensible and the high-level description it provides makes it easier to reason about data organization. But PENEOPE is targeted for presenting structured data, so it lacks support for creating navigation structures for arbitrary collections of documents gathered from the Web. STRUDEL, however, does not deal with the internal structure of documents — it provides a single graph model in which all data sources are uniformly modeled. Instead of supporting different languages for creating views, a
single query and transformation language is used for defining an internal data graph as views over external sources and for defining site graphs as views over the data graph. An HTML generator is then used to materialize a site graph as a hypertextual view. Since the language ignores internal structure, it cannot be used to define an embedded view representation (e.g., a table) for a collection of data within a page. Both systems can be regarded as heavy-weight solutions that closely couple view support with their own data management approaches. We are more interested in designing a flexible, light-weight tool that can support customized views independently and can be used together with diverse management approaches.

On the other hand, structural views representing the hypertext structure of a document sub-network in the Web are very often used as maps or overview diagrams to facilitate browsing. Hy+ [13] is one example of visualization systems that can be used to visualize navigation history graphs and Web-site maps. The problem with using an external system like Hy+ as a Web-surfing aid is that users have to work with two very different interfaces simultaneously: the browser’s interface and the external system’s interface, which may not be easy for average users. Also, it requires modification of the browser to interface with Hy+. There are many tools around that are specially designed to support Web-site mapping. WebCutter [25] is the only one among them that not only provides structural map of the whole site but also customizable maps for users’ interests. We show how WebSQL was extended to provide this capability in a hypertextual view in Section 2.3.5.

There are also scripted views generated dynamically as virtual documents. MacWeb [32] and Web3Objects [17] are two examples which support scripting languages designed for composing the content of a virtual document from fragments of information stored in their systems. However, these procedural languages are usually difficult for non-experts to use, and it is desirable to find new ways that simplify view specification.
1.3 Design Principles

Before we present our proposed solution, we first outline the design principles we have adopted as follows:

**Reusability** The data model should be reusable for developing a family of personal Web space applications with varying requirements for operations on query results and view representations for the results.

**High-level Abstractions** High-level abstractions should be provided for specifying:
- (1) the portions of information to be gathered from the WWW that are of interest to users,
- (2) a hypertext representation for browsing and organizing the retrieved information, which serves as a customized view/snapshot of the WWW for users.

**Tailorability and Extensibility** The data model should allow application developers to tailor the existing functionality to better match its specific requirements and extend the system with new functionality.

**Maintainability** Structures imposed for presenting and organizing information consistently in a personal Web space should be simple enough to be maintained easily. Also, automated support for generation of its physical pages should be provided.

**Open Industry Standard** A personal Web space should adhere to an open industry standard for publishing documents, and should be queriable like the source Web.

**Flexibility** Flexibility should be provided for users in customizing their personal Web spaces and choosing when to materialize them.

**Interoperability** The tool constructed should be able to be used together with other Web-site construction and maintenance systems as long as documents in their sites are accessible from the Web, making no assumptions on the management approaches chosen for their sites.

**Accessibility and Ease of Use** No specialized browser or plug-in should be required for browsing the personal Web space generated. Users only need to have knowledge of the standard browser interface.
1.4 Overview

We propose to use Virtual Documents (VD) to model personal Web spaces. In the context of our work, a Virtual Document is defined by only specifying a description, and its content and structure are computed dynamically from the results of a search procedure applied to the underlying WWW. A concrete Virtual Document is a graph of HTML pages generated upon a materialization request. We use an abstract Virtual Document model, which is reusable for developing a family of personal Web space applications, to capture the main properties of a personal Web space. In what follows, we first present the Virtual Document model in Section 1.4.1, then describe the Virtual Document operations in Section 1.4.2, and finally outline the system architecture in Section 1.4.3.

1.4.1 Data Model

The proposed model is based on three key abstractions: result-sets, views, and VD Graphs. In what follows, we briefly explain what these abstractions are used for in specifying a Virtual Document, leaving their definitions and the details of the model for the subsequent chapters.

Result-sets

The first thing to specify in a Virtual Document is what information a user wants to get from the Web. We allow him to view the Web as a database and retrieve information by querying it. He can specify a result-set to collect the information returned by a query. In addition, a tailorable and extensible set of operations for manipulating result-sets is supported. The user can specify what operations are to be applied to any result-sets to derive new result-sets that facilitate analysis or organization of the query answers.

Views

The second thing to specify is how the user wants to represent the information captured in a result-set as hypertext. We allow him to specify the hypertext representations of result-sets as views. Views are supported as a variety of view types, each with a unique
presentation and navigational structure for a result-set. The user can simply choose the most appropriate view type supported to represent a result-set. The set of view types supported is also tailorable and extensible.

**VD Graphs**

Lastly, the user has to specify how he wants to structure the information collected by organizing and connecting these views into an integrated hypertext network. We allow him to specify the structure of this hypertext network as a VD Graph.

### 1.4.2 Virtual Document Operations

To simplify Virtual Document authoring, our strategy is to separate the high-level specification of a customized Virtual Document from the design and programming of the underlying components that support all Virtual Document constructions in a specific application. We provide a high-level VD specification language for a user of medium expertise to specify preferences for customizing a Virtual Document: (1) the information to be collected from the Web into result-sets, (2) any operations to be applied on the result-sets, (3) view types to be used to represent result-sets, and (4) the VD Graph structure for organizing and integrating the views. As for the underlying components, they are supported in a VD construction framework which is supposed to be built and maintained by expert users with knowledge in application design and programming. The framework should be relatively stable, and modifications are only made when there are changes in the functionalities to be supported. In addition, we expect specific sites to design special HTML forms, templates or graphical user interfaces to allow even a novice user to specify a personal Web space; and the system could automatically generate a VD specification for documentation and the program for constructing the customized Virtual Document.

Thus, VD operations may involve four categories of people: (1) application architects and programmers, who design and implement tailored or new components that satisfy application-specific requirements. (2) VD designers who are specially assigned to author VD specifications that satisfy users' preferences. (3) Webmasters who are responsible
for Web-site administration and Web-site services support. They will decide when to
generate or update a Virtual Document. (3) users who use the materialized Virtual
Documents as their personal Web spaces for browsing.

Note that one person can play several of the above roles. For example, in cases where
the user is an expert, he himself can author his own VD specifications, or even design
and implement operations and view types to build a custom application. Moreover, a
user can be given the option of deciding when to materialize a Virtual Document.

To define a Virtual Document for a personal Web space, a VD designer authors a VD
specification file. The specification is then translated to a Java source program, which
is compiled to an executable Java class. The Java class is stored for use together with
other system-supplied Java classes in the VD construction framework. Afterwards, the
Webmaster can issue a request to materialize or re-materialize the Virtual Document
whenever he wants. The content of the Virtual Document can be viewed by a user using
a standard Web browser.

1.4.3 System Architecture

The system consists of four components: the VD specification translator, the VD con-
struction driver, the VD construction framework, and the WebSQL system. A brief
description of each component is given below and the system architecture is depicted in
Figure 1.1.

VD Specification Translator It takes a VD specification file as input and generates
the corresponding VD construction driver dynamically as a Java program, which
is then compiled into a Java class to be stored for use.

VD Construction Driver After the VD specification is converted into a Java class
of VD construction driver, the driver can be used together with the other Java
classes in the VD construction framework. Upon a materialization request, it issues
construction instructions to the construction framework which generates a Virtual
Document in the form of HTML pages.
VD Construction Framework According to the instructions given by the VD construction driver, it performs the following tasks: (1) issuing queries to the WebSQL system to gather information from the Web into result-sets, (2) applying operations on them to derive new result-sets if specified in the instructions, (3) constructing views from the result-sets and finally, (4) using the views to build the VD Graph structure. This structure is then materialized as a Virtual Document.

WebSQL System The system takes a query as input and evaluates it. Depending on the conditions specified in the query, this involves either sending a request to an index server and/or performing a depth-first traversal of a document sub-network. Results are then returned to the query supplier.
1.5 Organization of the Thesis

The rest of the thesis is organized as follows:

- Chapter 2 presents the Virtual Document model of personal Web space. The three key abstractions of the model: result-sets, views, and VD Graphs are defined.

- Chapter 3 introduces the VD Specification Language designed for this model. We explain a list of illustrating examples of Virtual Documents constructed using the specification language, with an emphasis on the design of view types and their user interfaces.

- Chapter 4 discusses applications of Virtual Documents in different areas to demonstrate the reusability of the model. Two practical examples are given to further illustrate how Virtual Documents can be utilized to model personal Web spaces.

- Chapter 5 briefly describes the prototype implementation.

- Chapter 6 presents a summary and the conclusions of our work. At the end, we explore possibilities in supporting personal Web spaces and suggest future extensions.

- Appendix A describes the syntax and semantics of the VD Specification Language.
Chapter 2

Data Model

2.1 Overview

In this chapter, we present the Virtual Document model of personal Web space. The proposed model is based on three key abstractions: Result-Sets, Views, and VD Graphs. Before we define these abstractions and explain the details of the model, we first describe what they are used for in specifying a Virtual Document, as mentioned in Chapter 1.

Result-sets The first thing to specify in a Virtual Document is what information a user wants to get from the Web. We allow him to view the Web as a database and retrieve information by querying it. He can specify a result-set to collect the information returned by a query. In addition, a tailorable and extensible set of operations for manipulating result-sets is supported. The user can specify what operations are to be applied to any result-sets to derive new result-sets, which can then be used to facilitate analysis or organization of the query answers.

Views The second thing to specify is how the user wants to represent the information captured in a result-set as hypertext. We allow him to specify the hypertext representations of result-sets as views. Views are supported as a variety of view types, each with a unique presentation and navigational structure for a result-set. The user can simply choose the most appropriate view type supported to represent a result-set. The set of view types supported is also tailorable and extensible.
**VD Graphs** Lastly, the user has to specify how he wants to structure the information collected by organizing and connecting these views into an integrated hypertext network. We allow him to specify the structure of this hypertext network as a VD Graph.

The relationships between these abstractions is depicted in Figure 2.1.

![Diagram](image)

**Figure 2.1: The Virtual Document Model**

As shown in the diagram, we use result-sets to model the sets of information to be retrieved from the WWW that are of interest to the user. They constitute the base
of a Virtual Document from which new result-sets can be derived by applying local operations to it. Views are used to model hypertext structures for presentation and navigation of any base or derived result-sets. Finally, we use the notion of a VD Graph to assemble these views into an integrated hypertext network, which represents a coherent, organized information architecture that serves as the user's personal Web space. A Virtual Document, in summary, is a VD Graph of views derived from the base result-sets.

The rest of this chapter covers the following topics. We first give a definition of Virtual Document in Section 2.2. To further explain the components in a Virtual Document, we organize the main concepts of the abstractions into three sub-models. In Section 2.3, we present the result-set sub-model which is built upon a relational model of the WWW. We then describe the view sub-model that defines a conceptual framework for application architects and programmers to construct view types in Section 2.4. Finally, the structure used in the VD Graph sub-model for organizing and connecting views is explained in Section 2.5. We summarize this chapter by pointing out in Section 2.6 how the proposed data model satisfies the design principles.

### 2.2 Virtual Documents

A Virtual Document is defined as follows:

**Definition 2.1** Given a Web query language $WQL$, a set of result-set operations $OP_{rs}$, a set of embedded view types $VT_e$, and a set of page view types $VT_p$, a Virtual Document is a tuple $VD = (RS_b, RS_d, N, V_e, V_p, Embed, Contain, CrossRef)$, where

- $RS_b$ is the base of the Virtual Document defined by Web queries of $WQL$,
- $RS_d$ is a finite set of derived result-sets, each computed from the base by applying one or more of the supported result-set operations $op \in OP_{rs}$,
- $N$ is a finite set of simple or aggregate nodes,
- \( V_e \) is a finite set of embedded views, each representing some \( r \in RS_b \cup RS_d \) by one of the supported embedded view types \( vt \in VT_e \),

- \( V_p \) is a finite set of page views, each representing some \( r \in RS_b \cup RS_d \) by one of the supported page view types \( vt \in VT_p \),

- \( Embed \subseteq N \times V_e \) is the embedment relation,

- \( Contain \subseteq N \times (N \cup V_p) \) is the containment relation,

- \( CrossRef \subseteq (N \cup V_p) \times (N \cup V_p) \) is the cross-reference relation.

- \((N, V_e, V_p, Embed, Contain, CrossRef)\) is a well-formed VD Graph.

The concepts mentioned will be explained in the subsequent sections of this chapter.

2.3 Result-Sets

In this section, we present the result-set sub-model. To provide a high-level abstraction for the user to specify the portions of information to be retrieved from the Web that are of interest to him, we adopt an approach that is similar to the query approach used in conventional database: the Web is modeled as a database, and a SQL-like query language is used to pose queries. Based on this metaphor, a result-set is introduced as an abstraction of the information returned by a query. The world view of this approach is outlined in Section 2.3.1 as background. We then define base result-sets and the base of a Virtual Document in Section 2.3.2. Section 2.3.3 explains how we can view a result-set from different perspectives. Section 2.3.4 describes the ordering of tuples in a result-set. Section 2.3.5 explains how a result-set can capture the hypertext structure of a sub-Web. Finally, we present the result-set operations in Section 2.3.6.

2.3.1 A Relational model of the WWW

A relational model of the WWW was proposed in the work on the WebSQL query language [28, 29, 31, 41], a declarative language for specifying information to be retrieved
from the Web. With this model, the Web is viewed as a relational database composed of two virtual relations: *Document* and *Anchor*. The Document relation has one tuple for each document in the Web, which can be an HTML document [44], a postscript file, image, audio or any type of document currently supported by the WWW, whereas the Anchor relation has one tuple for each hypertext link of an HTML document in the Web. In current implementation of WebSQL, the two virtual relations are defined as follows:

1. **Document**[^url, title, text, type, length, modif, ...]

   - url represents the Uniform Resource Locator (URL) which identifies a document in the Web.
   - title and text are attributes specific to an HTML document.
   - type, length, modif (last modification date) are additional information provided by Web Servers.

   The url attribute is the key of the Document relation, and all other attributes in the relation can be null. Among the attributes mentioned above, all are character strings, except that the url attribute has a type ‘URL’, the length attribute has a type ‘Integer’, and the date attribute has a type ‘Date’ as defined in the WebSQL API.

2. **Anchor**[^base, href, label, ...]

   - base is the URL of the HTML document containing the link.
   - href is the URL of the referred destination document.
   - label is the link description.

   All of the above-mentioned attributes in the Anchor relation are character strings.

   There are other attributes not mentioned above which are used for more advanced queries. We will only draw attention to them when we explain concepts that involve their use later.
Computable Sub-domains vs Conventional Relational Database

Before we can query the WWW based on this relational model, we need to first clarify the “virtual” nature of the two relations. In a conventional relational database, the entire contents of the database is assumed to be available to the query engine, and standard relational operations like selection, projection, and Cartesian product can be executed, in the worst case by enumerating all of the tuples. In the case of the Web, since the Document and Anchor relations are completely virtual and there is no way to enumerate all the documents in the Web, we cannot operate on them as we would do with relations in a conventional relational database. Thus, in order to query the Document and Anchor relations as traditional relations, one must first restrict the domain from a large, non-computable set of documents and links to a computable set. Computable sub-domains [28, 29, 31] can be defined either using keyword matching in querying index servers such as AltaVista, HotBot, or through controlled navigation starting from known URLs. WebSQL allows users to specify computable sub-domains of interest as part of a WebSQL query. It hides the peculiarities of query interfaces of different index servers, and integrates full-text content-based search with structure and topology-based queries. Once they are defined, we may view these materializable subsets of the two virtual relations as traditional relations, and we may use ordinary relational algebra to manipulate them. The WebSQL system supports three relational operators: select, project, and Cartesian product.

More in-depth study in computability issues of Web queries was covered in [30]. It states that, from a theoretical point of view, index servers evaluate keyword-matching queries on pre-computed indices, so they do not provide true associative retrieval from the whole Web — they do not guarantee currency of the index, nor do they guarantee every Web document has been indexed. However, computable sub-domains based on querying index servers are very useful approximations in practice, especially when users want to search certain information from the Web and no URLs for that purpose are known beforehand. We therefore include this capability of WebSQL in applications of Virtual Documents.
2.3.2 Base Result-Sets and the Base of a Virtual Document

In this subsection, we proceed to define our result-set sub-model based on this relational model of the WWW.

Given a Web database $W$ composed of the virtual relations $Document(D)$ and $Anchor(A)$, we assume there exists a Web relational machine like the WebSQL system, which materializes and operates on computable sub-domains of this database. Queries can then be posed to retrieve information from the Web. We define a Web query as follows:

**Definition 2.2** A Web query $Q$ is a relational query mapping Web database instances to sets of tuples over the values in the instances, such that the Web relational machine computes $Q$ on any Web and terminates with a set of tuples as answer.

The answer set of a Web query is modeled as a base result-set. In what follows, we will give definitions of base result-sets and the base of a Virtual Document in terms of Web queries.

**Definition 2.3** A base result-set $RS$ is a relation defined by a Web query $Q$ where

1. The schema of $RS$ is derived from the attributes $a_j$ of the Web database relations $Document(D)$ and $Anchor(A)$ such that for a query $Q$:

   ```
   SELECT
   a list of attribute expressions of the form $x_i.a_j$ where
   $x_i$ is a computable sub-domain of $D$ or $A$ specified in the FROM clause
   and $a_j$ is some attribute of $x_i$
   FROM
   some set of computable sub-domains $x_i$ of $D$ or $A$, $\{x_1, x_2, ..., x_i, ..., x_n\}$
   WHERE
   some boolean expression applied to expressions $x_i.a_j$ and constants.
   ```

   The attributes of the schema of $RS$ are named the same as the attribute expressions in the SELECT clause of $Q$, i.e. the schema of $RS$ is $RS(..., x_i.a_j, ...)$. Each $x_i.a_j$
is an attribute of $D$ or $A$: if $x_i$ is a computable subset of $D$, $x_i.a_j$ is an attribute of $D$, otherwise $x_i.a_j$ is an attribute of $A$. And the type of $a_j$ is as defined in the schema of the corresponding Web database relation.

2. An instance of $RS$ is associated with a finite set of tuples, $RS = Q(W) = \{t_1, ..., t_m, ..., t_q\}$, which is the result of applying $Q$ by a Web relational machine to some Web database instance $W = (D, A)$.

Definition 2.4 A base of a Virtual Document is a finite set of base result-sets $B = \{RS_1, RS_2, ..., RS_i, ..., RS_n\}$ specified for a Virtual Document through a set of Web queries $\{Q_1, Q_2, ..., Q_i, ..., Q_n\}$ where the following two conditions hold.

1. The schema of every $RS_i \in B$ is derived from the attributes $a_j$ of the Web relations $D$ and $A$ by naming after attributes in the SELECT clause of query $Q_i$ (see definition 2.3).

2. An instance of $B$ is associated with a finite set of tuples $RS_i = Q_i(W)$ for every $RS_i \in B$, which is the result of applying every $Q_i$ by a Web relational machine to some database instance $W = (D, A)$.

Figure 2.2 depicts the base of a Virtual Document.

![Figure 2.2: The Base of a Virtual Document](image)

2.3.3 Perspectives

The information sets gathered from the Web can be applied from different perspectives for different purposes. In general, a result-set is viewed as a set of tuples over the attributes values in some Web database instance, and each tuple may consist of attributes
of more than one object, be it a Document object or an Anchor object. We refer to this perspective of a result-set as a tuple-collection. In some cases, a tuple-collection could be the result of the Cartesian product of independent table variables over the Web relations, just for the convenience of retrieving more than one independent computable sub-domain in a single query. However, more often it is used to capture some meaningful relationship between Document objects and/or Anchor objects that satisfy certain conditions or patterns. Consider the following two examples:

Result-set $RS_1$ —

SELECT $x.url, x.title, y.url, y.title$

FROM documents $x$ in AltaVista mentioning "Computer Science", and documents $y$ that are linked to $x$ through paths of length two or less.

The result-set $RS_1(x.url, x.title, y.url, y.title)$ captures the neighborhood relationship "Linked Through Paths of Length Two or Less" between Document objects $x$ indexed by AltaVista with the keyword "Computer Science" and their neighboring Document objects $y$.

Result-set $RS_2$ —

SELECT $x.url, x.title, a.href$

FROM documents $x$ reachable from some known URL "http://start.html" by a path containing only local links, and anchors $a$ whose source document is $x$

WHERE the protocol of the destination document of anchor $a$, $a.href$, is http AND $a.href$ points to null.

The second result-set $RS_2(x.url, x.title, a.href)$ captures the relationship "Has Broken Link" between Document objects $x$ in a Web server which are reachable from some known URL by a local path and Anchor objects $a$ that are the broken links found in $x$.

Apart from capturing relationships between objects, extracting specific attributes of a Document or Anchor object is a common use of the tuple-collection perspective. Schema attributes of these result-sets are usually restricted to attributes of a single Web relation.
variable. For example, the user may gather the text, type, length, and modification dates of the Document objects that match a certain keyword from an index server, or, he may list the labels and destination URLs of the Anchor objects found in a document of known URL that point to postscript files.

However, in cases where the user's primary objective is to build a personal collection of references to the actual documents, he is more interested in accessing and navigating those documents pointed to by the URL values in the result-sets rather than analyzing their detailed attributes or the relationships between objects, and better interface support for browsing the collection is expected. We refer to this perspective of a result-set as a document-collection. Unlike the tuple in a tuple-collection which may consist of attributes from multiple objects, a stricter restriction is applied: each tuple in a document-collection maps to a single document in the Web, either through the url attribute of a Document object, or through the base or href attribute of an Anchor object.

E.g. To gather a Database collection from HotBot, we use Document objects:

Result-set $RS_3$ —
SELECT $x.url$, $x.title$
FROM documents $x$ in HotBot mentioning “Database”.

E.g. To find references to documents in other servers, we use Anchor objects:

Result-set $RS_4$ —
SELECT $a.href$
FROM documents $d$ reachable from some known URL “http://start.html” by a path of links, and anchors $a$ whose source document is $d$
WHERE the server of the destination document of anchor $a$, $a.href$, is NOT equal to the server of $d.url$.

There are also scenarios where the user wants a mixture of both perspectives, i.e., he wants to obtain information about relationships between documents while building document collections upon multiple URLs captured in each tuple for browsing and subsequent references. We provide the entity-collection perspective for such purposes. A tuple in an entity-collection may map to one or more entities and each entity may be composed
of multiple document objects referred to by a subset of URL attributes in the tuple. As the restriction imposed on result-sets for the document-collection perspective is released, we can view any general result-sets from an entity-collection perspective to support the additional functions that a simple tuple-collection perspective cannot provide. A simple example is \( RS_1 \), which we have used in explaining the tuple-collection perspective. We could define the pair of Document objects \( x \) and \( y \) for each tuple in \( RS_1 \) as a single entity with a hypertext representation designed for the required functional support and view \( RS_1 \) as an entity-collection.

Having explained the relationship between an entity-collection perspective and a document-collection perspective, we define them as follows:

Let \( Attr(Re1n) \) be the set of all attributes defined in the schema of a relation \( Re1n \).

**Definition 2.5** An entity-collection perspective of a result-set \( RS(x_1.a_1, \ldots, x_i.a_j, \ldots, x_p.a_q) \) maps \( RS \) to \( RS_{entity}(E_1, \ldots, E_g, \ldots, E_n) \) such that

1. \( E_g = (e_1, \ldots, e_k, \ldots, e_m) \) where every \( e_k \) is some attribute in \( Attr(RS) \).

2. for every \( x_i.a_j \in Attr(RS) \), there is a unique \( E_g = (e_1, \ldots, e_k, \ldots, e_m) \in Attr(RS_{entity}) \) where \( e_k = x_i.a_j \).

3. if \( E_g \in Attr(RS_{entity}) \) has an attribute \( e_k = x_i.a_j \) for some Document or Anchor object \( x_i \), then there does not exist an \( E_{h,h \neq g} \in Attr(RS_{entity}) \) that has any attribute \( x_i.a_l \) of object \( x_i \).

**Definition 2.6** A document-collection perspective \( RS_{Doc} \) of a result-set \( RS \) is a special case of an entity-collection perspective with a single entity by imposing the following pre-conditions on \( RS \):

1. \( x_i.a_j \in Attr(RS) \) implies \( x_k.a_j, k \neq i \notin Attr(RS) \).

2. There exists \( x_i.a_j \in Attr(RS) \) which is the only attribute whose type is URL.

In this thesis, we will focus on the support of tuple-collection and document-collection. For completeness, the definitions given for the VD data model also cover entity-collection. We will refer to result-sets that may consist of attributes of more than one object as tuple-collections, when the meaning is clear from context.
2.3.4 Ordering

A base result-set may be ordered or unordered depending on the nature of the query that defines the result-set and the implementation of the search engine of the Web relational machine. We assume two possible types of ordering to be associated with a result-set: (1) ranking, and (2) traversal order. Tuples can be ranked relatively according to their relevance to a query, usually as a result of some proprietary ranking function performed by individual index servers. In WebSQL’s implementation, the query answers for a document-collection defined using keyword-matching by an index server are returned in their relative ranking order determined by the server. Tuples can also be gathered in the traversal order of the search engine robot when computing a sub-domain defined by controlled navigation from some known URL. For example, WebSQL’s implementation adopts a depth-first search strategy and returns query answers in this order. In other cases when the semantics of the ordering is undefined, result-sets are unordered. Examples of unordered result-sets are: result-sets that are formed by the Cartesian product of multiple independent sub-domains; tuple-collections for capturing relationships between documents returned by index servers and their neighboring documents, where the Web machine does not implement additional ranking for the related objects.

Without requiring the user to be aware of the underlying implementation, we could assume that a function $TypeOfOrdering$ is provided. Given a query $Q$ for specifying $RS$ as input, the function returns one of the three possible enumeration types $t$:

$$RS.ordering = TypeOfOrdering(Q) = t, t \in \{\text{ranking}, \text{traversal order}, \text{undefined}\}$$

Note that this function is not supported by the current WebSQL implementation, and users are assumed to have the knowledge required for differentiating the type of ordering with which a result-set is associated. We introduce the function to facilitate explanation of our data model.

How can we make use of the tuple ordering in designing a personal Web space? Since relative relevance ranking of documents is widely used in index servers, we can provide a view (see Section 2.4) as a uniform user interface to take advantage of the various precomputed indices and ranking algorithms available by presenting the answers from a server in their original order. Our data model also allows the implementation of new operations
on result-sets, and we will talk more about operations later in this section. Custom rank-integration algorithms can be designed for integrating multiple ranked result-sets from different index servers into new result-sets to be presented as consolidated views. On the other hand, a document-collection gathered from traversing a network of documents by following links from a known URL may be used to build a guided-tour view, with the list of documents in traversal order as the predefined navigation path through the document network structure. For convenience, we will refer to a portion of the Web which is a network of documents connected by hyperlinks as a sub-Web, and we will explore its structure in the next subsection.

2.3.5 The Hypertext Structure for a sub-Web

In addition to ordering, some base result-sets capture the structure of a sub-Web of documents defined by the links. This applies to any document-collection with $TypeOfOrdering(Q) = \text{traversal order}$, and its hypertext structure is modeled by a column of path attributes of Document objects in the result-set. The path attribute was specifically added into the Web database Document table for this purpose. The semantics of $d\.path$ is the path from the starting point of traversal (some known URL specified in the query) to the document represented by the Document object $d$, if this document is reachable by following links that match the conditions or patterns defining the associated computable sub-Web. Each path to the document $d$ can be expressed as a stack of Anchor objects in the form of

\[
[\text{AnchorTuple}[\text{source}\_\text{url}, \text{label}_1, \text{url}_2], \text{AnchorTuple}[\text{url}_2, \text{label}_2, \text{url}_3, \ldots], \\
\text{AnchorTuple}[\text{url}_i, \text{label}_i, \text{url}_{i+1}], \ldots, \text{AnchorTuple}[\text{url}_n, \text{label}_n, \text{this}\_\text{url}]]
\]

where

1. $\text{source}\_\text{url}$ is some known URL as starting point of traversal,
2. $\text{this}\_\text{url}$ is the attribute value of $d\.url$, the destination of the path,
3. $\text{AnchorTuple}[\text{url}_i, \text{label}_i, \text{url}_{i+1}]$ is the $i$th Anchor object in the path from $\text{source}\_\text{url}$ to $\text{this}\_\text{url}$ such that $\text{url}_i$ is the URL of the document containing the anchor represented by this $i$th Anchor object; $\text{label}_i$ is the
text description of that anchor, and \( url_{i+1} \) is the URL of the next document in the path to which that anchor points. In a path of \( n \) anchors, 
\[
url_1 = \text{source-url} \quad \text{and} \quad url_{n+1} = \text{this-url}.
\]

There can be more than one possible path between a source URL and the document \( d \) since the underlying sub-Web structure can be a cyclic graph, when the Web is modeled as a graph whose nodes are the documents in the Web and whose edges are the hyperlinks in the Web. We do not attempt to capture all possible paths leading to a document \( d \) in the sub-Web from the given source, but rather choose one among them. The path selected as the value of the path attribute \( d.path \) depends on the implementation of the Web relational machine. In WebSQL, the tuples in a document-collection with \( TypeOfOrdering(Q) = \text{traversal-order} \) are returned in the depth-first search order, and the first path that reaches the document \( d \) determines the value of \( d.path \). We will use the following example to illustrate how WebSQL works. This explains what a user can expect from a query that involves the path attribute, and what structure can be built to represent a graph of documents in a view.

Suppose we have a hypertext graph of documents as shown in Figure 2.3,

![Graph Structure for a sub-Web](image)

Figure 2.3: The Graph Structure for a sub-Web

we can capture the sub-Web structure in the following document-collection:

Result-set \( RS_5 \) —

```
SELECT d.url, d.path
FROM documents d reachable from URL "http://www/a.html" by following links.
```
Figure 2.4 shows the materialized paths returned by WebSQL for every document node in the hypertext structure. Note that for the document `e.html`, we choose the path `a to b, b to c, c to e` instead of `a to e`; similarly, for `d.html`, the path `a to b, b to c, c to e, e to d` and the path `a to e, e to d` are both ignored.

A spanning tree can be reconstructed for every sub-Web of this nature using the column of path attributes in the document-collection. Each tree corresponds to the depth-first traversal tree over the rooted graph of documents in the sub-Web, and we could utilize this simplified topology of the underlying hypertext network to build a view of navigational structure for the document-collection. Such views can also serve as maps for the portions of the Web of interest to users. For further details, readers can refer to the DocumentTreeView in Section 3.1.8 as an example. The above illustration facilitates understanding of which hyperlinks in a given graph of documents will form the tree structure of the view.

### 2.3.6 Operations

In any application, it is often useful (or necessary) to provide operations for manipulation of result-sets to facilitate analysis or organization of the query answers collected. We
therefore support **result-set operations** as an abstraction of functions which derive a new result-set instance from available result-set instances. To distinguish these new result-sets from the base result-sets that are defined by a Web query, we refer to them as **derived result-sets**. Both base and derived result-sets can be input to a result-set operation. The abstraction of operations gives application architects and programmers extensibility and tailorability in the design and implementation of custom operation libraries for specific applications.

Useful result-set operations include relational algebraic operations with standard semantics like union, difference, projection, and intersection. Reorganizing result-sets may also be important. For example, partitioning a result-set can help categorize answers, and composing result-sets into a larger result-set can facilitate comparison of answers. We suggest the support of the following operations for these purposes: grouping operations that split a result-set horizontally by rows according to certain criteria (1 result-set to n subsets), and comparison operations that combine result-sets vertically into columns and compare them according to certain criteria (n result-sets to 1 result-set)(see Figure 2.5). In the context of integration of result-sets, available rank-integration algorithms may be implemented, and tailored integration functions may be developed to satisfy application-
specific needs. The basic operations we chose to support in our prototype are: union, difference, projection, intersection, group_by_field and comparison. More details will be covered in the next chapter when we present the VD Specification Language based on this data model.

In the next two sections, we will explain the view sub-model and the VD Graph sub-model for constructing a hypertext representation of the result-sets defined in a Virtual Document. Since a Virtual Document may be a network of HTML documents, we introduce the term Webpage to mean a single document in the Web to avoid confusion with a Virtual Document.

2.4 Views

![Figure 2.6: Deriving Views From Result-Sets](image)

While result-sets are the abstract information units for manipulation of the sub-Webs of interest to the user, views are the abstract hypertext units for representation of the information captured in result-sets as parts of a personal Web space (see Figure 2.6). To support hypertext units at both the within-Webpage level and Webpage-network level,
we further classify views into two categories: (1) embedded views, and (2) page views. In short, an embedded view is an abstraction of a segment of a Webpage whereas a page view is an abstraction of a hypertext network of one or more Webpages connected by hyperlinks. Just as the operation abstraction provides extensibility and tailorability for application architects and programmers, the view abstraction enables them to design customized hypertext representations for individual applications. Views are supported in the implementation of the model as a variety of concrete view types, each with a unique presentation and navigational structure for a result-set. This allows the VD designer to simply select the most suitable view type supported to represent a result-set. Moreover, as we adopt the model-view approach of object-oriented programming, different views can be specified for the same result-set in a Virtual Document, tuned to different applications of the personal Web space or different classes of users.

In what follows, we present the view sub-model which defines a framework for application architects and programmers to build new view types. The abstract class View models the class of all hypertext representations of the information captured in a result-set, and it is further specialized into two abstract classes EmbeddedView and PageView, each with its own set of constraints imposed on the characteristics of the hypertext representations defined by concrete objects of the class. Before we present their definitions, we first introduce in Section 2.4.1 the conceptual hypertext primitives of the model used in construction of view types. The terms explained in that section are used in the definitions in the rest of the chapter. We then give the definitions of the abstract class View and concrete view types in Section 2.4.2. Embedded views and page views are defined in Section 2.4.3 and Section 2.4.4 respectively. Finally, we outline the considerations to be made in the design of view types for specific applications in Section 2.4.5.

2.4.1 Conceptual Hypertext Primitives

First of all, we briefly explain the content and the structure of this section.

To facilitate the design of view types, we classify the hypertext objects (hyperlinks, Webpages, other HTML primitives/constructs supported in a Webpage) used in building them into conceptual hypertext primitives, which are described informally in this section.
Chapter 2. Data Model

We first present the link types supported in the model: outward link, interconnection link, internal link, interior link, and frame link. We then explain how the URLs of selected Webpages in a Virtual Document are used as entry points and exit points for defining the interfaces of hypertext units in the document. This is followed by the introduction of region objects for supporting construction of embedded views. And then, we present the special node types supported in the model for building page views: index node, context node, exit node, frame-composition node and function node. Transformation functions are also introduced when we present function nodes. Finally, we remark on concrete view frameworks.

The view abstraction together with the VD Graph abstraction can be seen as a structuring mechanism for hypertext representations of result-sets. A view is a hypertext module in a personal Web space, similar to a program module in a software system. A VD graph is the overall organization and interconnection structure of views in the Web space, similar to the software architecture of modules in the software system.

![Figure 2.7: Different Types of Links in a Virtual Document](image)

Links

In this context (see Figure 2.7), the hyperlinks to be generated in a Virtual Document are classified into the following conceptual primitives:

1. An **outward link** is a hyperlink with a Webpage in the Virtual Document as its source and a Webpage outside the Document as its destination.
2. An interconnection link is a hyperlink that joins Webpages in two hypertext units where each unit could either be a view or an aggregate node in the Virtual Document. (An aggregate node is for gathering related views in the VD Graph sub-model).

3. An internal link is a hyperlink whose source and destination are distinct Webpages in the same view. There are also special-purpose internal links for supporting the document-collection perspective of result-sets:

- An index link is an item in a listing of elements of the document-collection used for constructing the view. Each link provides access to a Webpage which represents the corresponding document element in the view. Note that an outward link can also be an index link when the index is designed to provide the user direct access to the original document element instead of a generated Webpage representing the document element in the view.

- A structural link constitutes a navigational structure imposed on all the elements in the document-collection. Examples of structural links are: previous, next, up, down.

4. An interior link is a hyperlink with source Webpage equal to the destination Webpage. It points from an offset in a page to another offset within the same page.

In addition to the above-mentioned links, there is a class of links that are not hyperlinks but are used for connecting two related Webpages to support the frame mechanism as defined by Netscape: hierarchies of frames can be built using frame links such that every parent frame can be divided into two child frames. The content of each frame is defined by a Webpage associated with it, and the Webpage of a parent frame contains two frame links\footnote{Though frame links are not hyperlinks, hyperlinks in Netscape can be specified with a TARGET attribute, NEW_FRAME, for starting a new frame hierarchy. The URL of the Webpage associated with the new root frame is given by the destination URL of the link.} referring to the URLs of the Webpages that define its child frames; all leaf Webpages in a frame hierarchy are ordinary Webpages.
Entry Points and Exit Points

The interface of a page view with other hypertext units in the Virtual Document is defined by its entry points and exit points. Entry points are the “exported” URLs of the view which are made public for reference by other components to allow direct access to that particular set of Webpages within the view. Exit points are the “imported” URLs from other hypertext units referenced by the view to allow direct exit from the view to those Webpages. The rest of the Webpages within the view are encapsulated and could only be accessed indirectly after the user enters the view through one of its entry points and navigates via internal links. On the other hand, the interface of an embedded view with other hypertext units is defined by the interface of the parent Webpage encapsulating the view: (1) the view can only be reached by first getting to the beginning of the parent Webpage whose URL acts as the entry point and then jumping to the view via an interior link, (2) all embedded views in the parent Webpage share a common set of exit points for that page.

Regions

At the within-Webpage level, a region is a segment of a Webpage that consists of contiguous objects specified by a string of one or more HTML primitives/constructs within the body of an HTML document: a region can comprise a simple table, or it can be a layout of contiguous images, texts, and hyperlinks. In addition, regions can be defined within a region, i.e., nested. For example, we can define the layout for the content of each table cell within a table, where the segment for each table cell is treated as a subregion inside the region containing the table. Since the building of more sophisticated navigational structures and exploration aids for the documents referred to by the URLs in the result-set generally requires more than one Webpage, we provide no special support for document-collections at the within-Webpage level. Instead, the tuple-collection perspective is assumed.
Nodes

At the Webpage-network level, we provide the following special-purpose Webpages for supporting the document-collection perspective as well as other conceptual primitives for more general result-sets.

1. An **index node** is a Webpage which contains a listing of elements of the document-collection associated with the view. Each item in the listing has an index link referring to either of the possible destinations depending on the view design: the original URL of the corresponding document element in the collection or the generated URL of the Webpage that represents the element in the view.

2. A **context node** is generated for every element of the document-collection associated with the view to implement a navigational structure imposed upon the collection. Each context node of a document element in the navigational structure is a Webpage that captures all the internal links (the structural links) that refer to the element's immediate neighboring document nodes in the structure.

3. An **exit node** is a Webpage where a set of exit points of the view is centralized. Each exit point is defined by an interconnection link with the exit node as its source and any other view or aggregate node in the VD Graph as its destination.

4. A **frame-composition node** is a Webpage which defines the composition of a parent frame from its two child frames. It contains the frame links we introduced earlier that refer to the URLs of the Webpages defining the constituent frames, which in turn could be frame-composition nodes themselves. Thus, frame-composition node and frame link together support the nested frame mechanism in Netscape. Frame composition allows composite nodes to be assembled from primitive nodes mentioned above: index node, exit node, context node, and from the original document elements of the collection associated with the view such that constituent leaf nodes will be displayed in nested frames that constitute a “composite Webpage” in the browser window.
5. A transformation function is a function which processes the contents of the Webpages referred to by a given subset of URLs in a result-set to generate a new set of Webpages for representing them in the view. We refer to the new Webpages formed as function nodes. One simple example is a concatenate function which links the body of given Webpages to produce a combined Webpage. More complex functions may involve pattern substitution or extraction of page contents. In the implementation of the data model, both generic and application-specific transformation functions can be supported.

Readers interested in the construction of page view types from the above nodes together with the internal links and frame links mentioned earlier can refer to the hypertext structures of DocumentView (Figure A.4), DocumentListView (Figure A.5), and DocumentTreeView (Figure A.6) in Appendix A.

Concrete View Frameworks

Concrete view frameworks for specific applications can further expand the set of conceptual primitives introduced above when other useful abstractions are discovered or special functional needs are identified. Based on given primitives as building blocks, application architects and programmers can design view types under the constraints we set for embedded views and page views, as covered in Section 2.4.3 and Section 2.4.4 respectively. We try to balance the need for constraints to impose a standard, consistent, and easy-to-maintain structure and the need for freedom and creativity in view type construction. Considerations to be made in the design of view types for specific applications are outlined at the end of this section.

2.4.2 Views and View Types

Having introduced the conceptual primitives, we can now give the definitions in the view sub-model. We start with the abstract class View and the concrete view types in this section. Embedded views and page views will be defined in the following two sections.

Different representations (views) of a result-set reflect the values of its attributes
according to the needs of a given application by using HTML primitives such as text, hyperlinks, images, list, table, and HTML constructs built from different combinations of the primitives. Examples of HTML constructs are: a "record" of primitives that specifies a layout of HTML objects on a Webpage, and HTML pages containing interconnecting hyperlinks that specifies a network of Webpages.

**Definition 2.7** View is the abstract class that models the class of all hypertext representations of a result-set. The View class is further specialized into two abstract subclasses EmbeddedView and PageView, each with its own set of constraints imposed on the characteristics of the hypertext representations defined by concrete objects of the class.

**Definition 2.8** A view type is any concrete class of views that implements the constraints imposed by the abstract class EmbeddedView (Definition 2.9) or PageView (Definition 2.13). The basic requirements for a view type are:

1. It must have one or more constructor functions that take the necessary attributes from a given result-set as input for creating view object instances of that view type.
2. It must have a method `materialize` which maps result-set instances to their hypertext representations that satisfy the requirements of its superclass, EmbeddedView or PageView, and additional properties specific to the view type.

### 2.4.3 Embedded Views

The objective of an embedded view is to present a given result-set as HTML objects within a Webpage to facilitate the user's understanding of the information captured while allowing him to selectively access any particular documents referred to by the URLs in the result-set that are of interest to him. We define constraints for an embedded view as follows:

**Definition 2.9** A well-formed embedded view $EV$ of a result-set $RS(x_1.a_1, ..., x_i.a_j, ..., x_n.a_m)$ is a region $R$ in a Webpage $P$, whose representation is specified by a string $S$ of HTML primitives/constructs within the body of $P$ such that
1. Every tuple of $RS$ is uniquely represented by a subregion within $R$ which is specified by a string of HTML primitives/constructs, as a sub-string of $S$.

2. If $RS.ordering \neq \text{undefined}$, then the relative ordering of tuples in $RS$ is preserved in their representations in $R$.

3. All tuples in $RS$ are consistently represented by subregions with the same composition and layout of HTML objects for representing the content of a tuple.

4. For every tuple $t_m$ in $RS$ represented by subregion $r_m$ in $R$ and specified by a substring $s_m$ of $S$,

   - every attribute value $x_i.a_j$ in tuple $t_m$ whose type is a URL is represented by a subregion $r_i$ in $r_m$, where $r_i$ comprises a layout of HTML objects designed for URLs in $EV$: it must either be a hyperlink $L$ with its destination URL $= x_i.a_j$ or a layout of HTML objects that contains $L$, and $r_i$.materialize() returns a substring of $s_m$.

   - every attribute value $x_i.a_k$ in tuple $t_m$ whose type is not URL (e.g., title, date) is represented by a subregion $r_j$ in $r_m$, where $r_j$ comprises a layout of HTML objects designed for the type of attribute $x_i.a_k$ in $EV$, and $r_j$.materialize() returns a substring of $s_m$ reflecting the value $x_i.a_k$.

5. Each hyperlink in $R$ is either an outward link or an interior link.

6. If the instance of $RS$ is empty, $R$ takes the value of a special EMPTY_SET region $E$ whose representation is specified by an HTML primitive/construct that indicates an empty result-set to the user, and $E$.materialize() returns the string $S$.

We will present examples of embedded views in the next chapter. Interested readers can refer to TupleListViews in Section 3.1.1, Section 3.1.2, and Section 3.1.3.

2.4.4 Page Views

Page views provide the additional expressiveness for presentation of result-sets as hypertext network of Webpages that embedded views cannot provide at the within-Webpage
level. By mapping the information captured to interconnected Webpages and frames, more illustrative and elaborate view types can be created. For example, an entity-collection which captures the binary relationship between the results returned by an index server and their neighboring documents within a given path distance may be represented by an index of tuples, with each tuple containing a link that refers to a nested-frame node of two parallel frames. With this configuration, the pair of documents referenced in a tuple can be displayed side-by-side in the browser window for convenient cross-reference when the tuple is selected for browsing. Another example of improvement is the presentation of a complex attribute: the path attribute that captures multiple URLs may be represented by a combination of interconnected Webpages and frames that is tailored for facilitating access to Webpages in a path. But above all, application architects and programmers can now construct more complex hypertext objects like index nodes and navigational structures at the Webpage-network level to support the document-collection perspective of a result-set. Our goal is to enable VD designers to gather document collections of references to the actual Webpages and to equip these collections with hypertext facilities that help a user unfamiliar with a collection to systematically browse the document elements, provide a context for him to further explore documents referenced by the documents in the view, and support subsequent references by him. In what follows, we first define some conceptual constructs: view patterns, representative nodes, and navigational structures, that can be built from primitives outlined in Section 2.4.1. We then use them to define the constraints for a page view. A rough description is given at the beginning of each definition. Readers may skip the formal definitions if not interested in the exact details.

**View Patterns**

A view pattern is a directed, rooted graph of Webpages. The type of each Webpage must be one of the supported node types, whereas the type of each link must be one of the supported link types. The base case is a simple Webpage with links, if any, defined in that page. The view pattern returns the set of strings of HTML contents for all Webpages in the graph when materialized.
Definition 2.10  Given a set of node primitives NT and a set of hyperlink primitives LT, a view pattern VP is a hypertext network of Webpages \( P = \{P_1 : NT_1, ..., P_i : NT_i, ..., P_n : NT_n\} \), \( NT_i \in NT \), connected by hyperlinks and framelinks, \( L = \{L_{P_1,1} : LT_1, ..., L_{P_i,j} : LT_j, ..., L_{P_n,m} : LT_m\} \), \( LT_j \in LT \), whose representation is specified by strings \( S = \{S_{P_1}, ..., S_{P_i}, ..., S_{P_n}\} \) of HTML contents of \( P \) such that the following conditions hold.

1. The base case of \( VP \) is a simple Webpage \( P_1 : NT_1 \) containing either hyperlinks or framelinks \( L_{P_1} = \{L_{P_1,1} : LT_1, ..., L_{P_1,k} : LT_k\} \).

2. If \( P \) has more than one Webpage, the network has the structure of a directed, rooted graph \( G = (P, L) \), where \( P \) is the set of nodes and \( L \) is the set of directed edges. The graph \( G \) has a unique root \( P_{root} \in P \), and every other node is reachable from \( P_{root} \) by following some path of links in \( L \).

3. \( VP.materialize() \) returns the set of strings \( S \).

Representative Nodes

Given a set of source Webpages referred to by a set of URLs captured in a result-set, a representative node is a view pattern such that either (1) for every given source Webpage, there is a link in the view pattern that refers to the page, or (2) there is a view pattern whose nodes are the function nodes generated by applying a supported transformation function to the given source Webpages, and they are the only function nodes found in the representative node.

Definition 2.11  Given a set of Webpages \( P_{web} \) referred to by a set of URLs \( U_{web} \) captured in a result-set and a set of transformation function primitives \( T \), a representative node \( RN \) for \( P_{web} \) is a view pattern \( VP_{rep} \) of Webpages \( P_{rep} \) containing hyperlinks and framelinks \( L_{rep} \) such that one of the following conditions is satisfied:

- For every \( U_i \in U_{web} \), there is a link in \( L_{rep} \) whose destination URL is \( U_i \).

- There exists a view pattern \( VP_{func} \) of Webpages \( P_{func} \subseteq P_{rep} \) which contains all the function nodes in \( P_{rep} \), and the contents of \( P_{func} \) are defined by \( S_{func} = T_i(S_{web}) \),
where

(i) $T_i \in T$ is some transformation function chosen for representing $P_{web}$.

(ii) $S_{web}$ and $S_{func}$ are the sets of string contents of $P_{web}$ and $P_{func}$ respectively.

The first condition applies when actual references to the original Webpages are provided, whereas the second condition holds when processed Webpages in the representative node replace the original Webpages.

Navigational Structures

We give an example of navigational structure extracted from a DocumentListView (Figure A.5) as depicted in Figure 2.8.

![Figure 2.8: Navigational Structure in a DocumentListView](image)

In this example, we present a “double-linked list” navigational structure for a given set of Webpages in the source Web. Each navigational node in the structure consists of two Webpages: a frame-composition node and a context node. Each context node contains (1) a link referring to a Webpage in the given set and (2) structural links (previous, next) that refer to neighboring navigational nodes in the structure. The leftmost frame-composition node is the root of this navigational structure. In general, a navigational structure consists of a set of navigational nodes which are linked together by the structural links of the context node in each navigational node. In this example, the structure is
designed for a set of source Webpages, and therefore each context node has a link referring to a source Webpage. However, the structure can also be designed for a set of nodes representing sets of source Webpages (Definition 2.11).

Before we proceed to define a navigational structure, we define the two types of objects that can be its elements, and the URL that identifies each type of object.

Let \( \text{OrigDoc} \) denote the type of source Webpages referred to by a set of URLs captured in a result-set. Let \( \text{RepNode} \) denote the type of representative nodes for sets of URLs captured in a result-set. Furthermore, let an OrigDoc Webpage be identified by its URL, and a RepNode set of Webpages be identified by the URL of its root.

**Definition 2.12** Given a set of objects \( O = \{O_1 : OT_1, ..., O_i : OT_i, ..., O_n : OT_n\} \) captured in a result-set, where \( OT_i \in \{\text{OrigDoc}, \text{RepNode}\} \), a navigational structure \( NS \) for \( O \) is a view pattern \( VP_{nav} \) of Webpages \( P_{nav} \) containing hyperlinks and framelinks \( L_{nav} \) such that the following conditions hold.

1. For every object \( O_i \in O \), there is a view pattern \( VP_i \) of Webpages \( P_i \subseteq P_{nav} \) with links \( L_i \subseteq L_{nav} \), where

   (a) \( \bigcup P_i = P_{nav} \);

   (b) \( P_{i,k} \in P_i \) implies \( P_{i,k} \notin P_{j,k;\neq i} \) for another object \( O_j \in O \);

   (c) there is a unique context node \( P_{i,c} \in P_i \) containing some link \( L_{i,c} \in L_i \) whose source URL is the URL of \( P_{i,c} \), and destination URL is the URL identifying \( O_i \);

   (d) all structural links \( L_{i,k} \in L_i \) have a source URL equal to the URL of the context node \( P_{i,c} \) (structural links are links that refer to a Webpage not in \( P_i \) but in \( P_{nav} \)).

   We refer to this view pattern \( VP_i \) as **navigational node** for \( O_i \).

2. There exists an \( O_j \in O \) with the root \( P_{j,root} \) of its navigational node \( VP_j \) taken as the unique root \( P_{nav,root} \) of \( VP_{nav} \).
3. For every navigational node $VP_i$ for $O_i \in O$, where $P_i.root \neq P_{\text{nav.root}}$, there exists some $VP_{j,j\neq i}$ for $O_j \in O$ which contains a context node $P_{j,c}$ and a structural link $L_{j,c}$ whose destination URL is the URL of $P_{i.root}$.

**Page Views**

Page views are designed for either the document-collection perspective or the entity-collection perspective of a result-set. The former is a special case of the latter; we isolate this simpler case for clarity. We first briefly explain the former case, i.e. the constraints for a well-formed page view of a document-collection. Such a page view must satisfy one of the following two conditions: (1) it is a representative node for all source Webpages referred to by the URLs in the document-collection by applying a supported transformation function to those source Webpages, or (2) it contains a unique index node that contains a region representing all tuples in the document-collection. Each tuple is represented by a layout of HTML object(s), which either is a simple hyperlink or includes a hyperlink. The destination of the link can be (a) the URL of the source Webpage associated with the tuple, (b) the URL of a representative node for that Webpage, or (c) the URL of a navigational node for that Webpage or the representative node of the Webpage, where the navigational node is in some navigational structure in the page view. In both cases (1) and (2), if the given set of exit points for the page view is non-empty, there is a unique exit node in the view which contains an interconnection link for every exit point; otherwise, such a node does not exist. Lastly, the set of entry points defined for the page view must include its root.

As for the entity-collection perspective, similar constraints can be imposed on each entity in each tuple of the result-set. Readers may refer to the formal definition. Examples used in the following chapters are all based on the document-collection perspective, and transformation functions are not used. We give the full definitions here for completeness. If interested in illustrative examples, readers may refer to DocumentViews (Section 3.1.4 and Section 3.1.5), DocumentListViews (Section 3.1.6 and Section 3.1.7), and DocumentTreeViews (Section 3.1.8). Their physical node-and-link configurations are depicted in Figure A.4, Figure A.5, and Figure A.6 respectively.
We can now define a page view as follows:

**Definition 2.13** Given a perspective $RS_p$ of $RS$ and a set of exit points of URLs $U_{exit}$, a well-formed page view $PV$ of $RS_p$ is a view pattern $VP_{view}$ of Webpages $P_{view}$ with links $L_{view}$ and a set of entry points of URLs $U_{entry}$ such that

1. If $RS_p$ is a document-collection $RS_{Doc}(x_1.a_1, ..., x_1.a_i, ..., x_1.a_n)$, where some attribute $x_1.a_j \in Attr(RS_{Doc})$ is the URL identifying the Webpage referred to by each tuple in $RS_{Doc}$, either one of the following conditions is satisfied:

   - $VP_{view}$ is a representative node for all Webpages referred to by values of the URL attribute $x_1.a_j$ with a transformation function $T_i \in T$ applied on the contents of the Webpages.
   - there exists a unique index node $P_{index} \in P_{view}$ containing a region $R$ specified by string $S$ which satisfies all constraints given for an embedded view of $RS$ as stated in Definition 2.9, except that the rule for tuple representation (rule 4) and the one for hyperlink (rule 5) are replaced by the following:

     (a) every tuple $t_m$ in $RS_{Doc}$ is represented by a subregion $r_m$ in $R$ specified by a substring $s_m$ of $S$, where

        i. $r_m$ comprises a layout of HTML objects designed for representing every $x_1.a_i \in Attr(RS_{Doc})$;

        ii. the object(s) must be a simple hyperlink $L \in L_{view}$ or include $L$, whose destination URL can be one of the following:

        - the URL attribute $x_1.a_j$,

        - the URL of the root $P_{rep.root} \in P_{rep}$ of a representative node ($P_{rep} \subseteq P_{view}$, $L_{rep} \subseteq L_{view}$) for the Webpage $x_1.a_j$, or

        - the URL of the root $P_{Oi.root} \in P_{Oi}$ of a navigational node ($P_{Oi} \subseteq P_{nav}$, $L_{Oi} \subseteq L_{nav}$) for the Webpage $x_1.a_j$ or its representative node, where ($P_{nav} \subseteq P_{view}$, $L_{nav} \subseteq L_{view}$) is some navigational structure in $VP_{view}$;

     (b) each hyperlink in $R$ is either an internal link, an outward link, or an interior link.
(NOTE that document-collection is a special case of entity-collection given below. We have isolated this special case for clarity.)

2. If $RS_p$ is an entity-collection $RS_{entity}(E_1, ..., E_g, ..., E_n)$, where for every $E_g \in \text{Attr}(RS_{entity})$, $P_{\text{web}.E_g}$ denotes the set of Webpages referred to by some URL $e_k$ of $E_g = (e_1, ..., e_k, ..., e_m)$, then one of the following two conditions is satisfied:

- For every $E_g \in \text{Attr}(RS_{entity})$, there exists a representative node $(P_{\text{rep}.E_g} \subseteq P_{\text{view}}, L_{\text{rep}.E_g} \subseteq L_{\text{view}})$ for the Webpages $P_{\text{web}.E_g}$ with some transformation function $T_{E_g} \in T$ applied on the contents of $P_{\text{web}.E_g}$.

- There exists a unique index node $P_{\text{index}} \in P_{\text{view}}$ containing a region $R$ specified by string $S$ which satisfies all constraints given for an embedded view of $RS$ as stated in Definition 2.9, except that the rule for tuple representation (rule 4) and the one for hyperlink (rule 5) are replaced by the following: every tuple $t_m$ in $RS_{entity}$ is represented by a subregion $r_m$ in $R$ specified by a substring $s_m$ of $S$, where

  (a) there exists a subregion $r_{m,E_g}$ in $r_m$ for every $E_g \in \text{Attr}(RS_{entity})$ which comprises a layout of HTML objects designed for representing every $e_k \in \text{Attr}(E_g)$;

  (b) if $P_{\text{web}.E_g} \neq \emptyset$, then the object(s) must be a simple hyperlink $L \in L_{\text{view}}$ or include $L$, whose destination URL can be one of the following:

    - the URL of the Webpage in $P_{\text{web}.E_g}$, if $P_{\text{web}.E_g}$ has a single element,

    - the URL of the root $P_{\text{rep}.E_g.root} \in P_{\text{rep}.E_g}$ of a representative node $(P_{\text{rep}.E_g} \subseteq P_{\text{view}}, L_{\text{rep}.E_g} \subseteq L_{\text{view}})$ for the Webpages $P_{E_g}$, or

    - the URL of the root $P_{\text{Oi}.E_g.root} \in P_{\text{Oi}.E_g}$ of a navigational node $(P_{\text{Oi}.E_g} \subseteq P_{\text{nav}.E_g}, L_{\text{Oi}.E_g} \subseteq L_{\text{nav}.E_g})$ for the Webpages $P_{E_g}$ or their representative node, where $(P_{\text{nav}.E_g} \subseteq P_{\text{view}}, L_{\text{nav}.E_g} \subseteq L_{\text{view}})$ is some navigational structure in $V_{P_{\text{view}}}$;

  (c) each hyperlink in $R$ is either an internal link, an outward link, or an interior link.
3. $U_{exit} \neq \phi$ implies there exists a unique exit node $P_{exit} \in P_{view}$, and for every $U_i \in U_{exit}$, there exists a unique interconnection link $L_k \in L_{view}$ whose destination URL is $U_i$, and whose source URL is the URL of $P_{exit}$.

4. $U_{exit} = \phi$ implies that there does not exist any exit node or interconnection link in $VP_{view}$.

5. Every $U_i \in U_{entry}$ is the URL of some $P_i \in P_{view}$.

6. There exists $U_j \in U_{entry}$ where $U_j$ is the URL of $P_{view.root}$.

2.4.5 The Design of View Types

The design of concrete view types of embedded views and page views involves the expression of the mapping of result-sets into hypertext structures in terms of the primitives and the constructs defined in the view sub-model. In the process, the application architect/programmer needs to consider the various aspects of the result-set he is going to represent: (1) perspective, (2) ordering, (3) web structure, (4) internal organization. If a tuple-collection perspective is taken, he needs to design an embedded view type; otherwise, he should consider a page view type. He then has to determine how the ordering, web-structure and internal organization, if any, are to be presented in an embedded view or index node, and what navigational structure(s) are to be provided to support them. The constituent context nodes and structural links and their configuration have to be identified in the design of each navigational structure. Regarding the presentation of the contents of the Webpages captured in the result-set, the application architect/programmer needs to choose between providing links to the actual Webpages and applying some transformation function to generate alternate representations for them in the view type. He also has to think about how to make use of frame-composition to organize simple Webpages into “composite Webpages” to provide the desired user interface. For applications with new, specific functional needs that are not covered in the framework, additional primitives and constructs can be created. They can then be applied in the view type construction process like the standard building blocks.
2.5 VD Graphs

We have used the analogy of the relationship between software architecture and program modules to introduce the relationship between a VD Graph and views in Section 2.4.1. In a nutshell, VD Graph is the extra layer of abstraction for organizing views and connecting related views. We provide this extra layer because a personal Web space should not be limited to a single view and as the number of views in the space increases, some means are required to organize them into a logical structure. Cases where supporting multiple views are useful include the following:

1. For a given result-set, a set of views with different presentations or different navigational structures can be provided to reflect various aspects of a result-set. Users who share the same Virtual Document can then choose their own preferences. Preferences can vary for a single user when he employs the result-set for different functions, or for multiple users when they examine the result-set from different views simultaneously.

2. Views for multiple related result-sets can be put together in one Virtual Document. For example, it can be subject-oriented, where information related to a given subject is collected from index servers and/or Web sites; it can be task-oriented, where documents required for the same task, which may cover several subjects, can be gathered into a Virtual Document; or it can be departmental-structure-oriented, with selected collections of Webpages from personnel of different departments or teams being grouped into different result-sets.

3. Combination of (1) and (2).

Our purpose is to provide a simple, easy-to-maintain structuring mechanism to assemble views into an integrated hypertext network that can add value to the usage of individual views. The proposed structure can then be utilized by the VD designer to represent a coherent, organized information architecture which serves as a personal Web space. In the next subsection, we present the VD Graph sub-model used by a VD designer to compose a hypertext network from a construction set of views and other building blocks.
2.5.1 Overview of the VD Graph Structure

The VD Graph sub-model features mechanisms for structuring a personal Web space (PWS) in two aspects: (1) aggregating related views into composite components that represent sub-PWSs, where sub-PWSs can contain other sub-PWSs to form a containment hierarchy, and (2) connecting hypertext units (views or new hypertext units defined in sub-PWS) based on some relationship between them, e.g., cross-reference. Our approach is based on a simple structure that we call VD Graph (see Figure 2.9). This structure is inspired by a general mathematical entity called the tube graph that is used for modeling architectural design styles of software systems [15]. A tube graph is used in their work as a semantics data structure of a design theory, whereas a VD Graph is a hypertext structure with many features added for our purpose. A VD Graph consists of a tree and an extra set of edges, called tubes, between the nodes. The hierarchical structure of the tree represents sub-PWS containment. We can use a tree to model a PWS, and each sub-tree corresponds to a sub-PWS contained in it. The root and internal nodes are aggregate nodes, which are new hypertext units introduced for gathering related views and sub-PWSs into a higher-level PWS (or sub-PWS). Each aggregate node incorporates a constituent embedded view as a segment of the node, and has a containment edge

![Figure 2.9: The VD Graph Structure](image)
linked to every constituent page view and every aggregate node of its child sub-PWS. On the other hand, the leaf nodes can either be page views or aggregate nodes with only embedded views. The extra set of edges, called tubes, are used to represent interconnections between the nodes of the tree based on some relations. We support the tube abstraction to allow new interconnections to be defined for application-specific relations, and to allow the system implementing the data model to provide different representations in the user interface for different types of tubes. For simplicity, we will assume there is just one type of tube for the cross-reference relation in this thesis.

In the following subsection, we will explain further the new conceptual primitives for building the hypertext network represented by a VD Graph. We will then define a well-formed VD Graph Configuration in terms of the components in the graph, their relations, and the constraints imposed upon them.

### 2.5.2 Conceptual Hypertext Primitives

A VD Graph can be simple or composite. A **Simple VD Graph** consists of just a single view. If it is a page view, it will be the only component of the VD Graph; otherwise, since an embedded view cannot exist on its own, we introduce a primitive, called a simple node, to incorporate the view into a segment of a Webpage. A simple node is a Webpage with a single section embodying the only embedded view in a Simple VD Graph. On the other hand, a **Composite VD Graph** consists of more than one view. The overview of the VD Graph structure given in the previous subsection is focused on the structure of a Composite VD Graph. Constituent views can be assembled into this structure by using the primitives, namely aggregate node, containment edge, and cross-reference edge, as follows:

An aggregate node can either be a simple Webpage or a “composite Webpage” formed by frame-composition (Section 2.4.1). The mandatory Webpage in an aggregate node is called a sections node. A sections node is made up of a preamble and a main body which contains a sequence of sections, each with its own set of attributes. For example, a preamble may have a title attribute which describes the purpose of the PWS represented by this aggregate node, and a table of contents for the sections in the node.
Each section in the main body either incorporates a constituent embedded view of the aggregate node as a segment of the Webpage, or it has a containment edge linked to a constituent page view or an aggregate node of its child sub-PWS; and it may have a heading describing its contents. A containment edge is composed of one or more interconnection links (Section 2.4.1), each having the sections node as its source and the entry point of a constituent page view or an aggregate node as its destination. It may consist of multiple interconnection links when its destination hypertext unit has multiple entry points. On the other hand, cross-reference edges can be used to connect any two hypertext units (page views or aggregate-nodes). A cross-reference edge is represented by an interconnection link; and as in page views, all interconnection links connecting from an aggregate node to another hypertext unit are gathered in an exit node (Section 2.4.1). If an aggregate node is not specified to be the source of any cross-reference edges, the sections node will be its only Webpage; otherwise, an aggregate node will be represented by a "composite Webpage" with one frame for the sections node and another frame for the exit node.

We now proceed with defining a well-formed VD Graph. Arbitrary cross-reference between any aggregate node and page view is supported in the configuration.

2.5.3 Configurations

Definition 2.14 A VD Graph Configuration is a tuple
\[ C = (N, V_e, V_p, Embed, Contain, CrossRef) \]
where

- \( N \) is a finite set of simple or aggregate nodes,
- \( V_e \) is a finite set of embedded views,
- \( V_p \) is a finite set of page views,
- \( Embed \subseteq N \times V_e \) is the embedment relation,
- \( Contain \subseteq N \times (N \cup V_p) \) is the containment relation, and
- \( CrossRef \subseteq (N \cup V_p) \times (N \cup V_p) \) is the cross-reference relation.
Let $TypeOfNode : N \to \{SimpleNode, AggregateNode\}$ denote a function which determines the type of a node.

We then define the constraints for the well-formedness of a Simple VD Graph and a Composite VD Graph as follows:

**Definition 2.15** A well-formed **Simple VD Graph** is a VD Graph Configuration $C = (N, V_e, V_p, Embed, Contain, CrossRef)$ such that one of the following conditions holds:

- There is only one embedded view: $(|V_e| = 1) \land (e \in V_e)$, and there is only one simple node embedding it: $(|N| = 1) \land (n \in N) \land (TypeOfNode(n) = SimpleNode) \land Embed(n, e)$; the rest are empty: $V_p = Contain = CrossRef = \emptyset$.

- There is only one page view: $|V_p| = 1$, and the rest are empty: $N = V_e = Embed = Contain = CrossRef = \emptyset$.

**Definition 2.16** A well-formed **Composite VD Graph** is a VD Graph Configuration $C = (N, V_e, V_p, Embed, Contain, CrossRef)$ such that the following conditions hold.

1. $|V_e \cup V_p| > 1$.

2. For all $n \in N$, $TypeOfNode(n) = AggregateNode$.

3. For every $e \in V_e$, there exists a unique $n \in N$ such that $Embed(n, e)$.

4. For every $p \in V_p$, there exists a unique $n \in N$ such that $Contain(n, p)$.

5. $(N, Contain)$ is a tree, i.e. it has a unique root, each node has a single parent, and there are no cycles.

**Definition 2.17** A well-formed **VD Graph** is either a well-formed Simple VD Graph or a well-formed Composite VD Graph.
2.6 Summary

We have presented the Virtual Document data model in this chapter. The proposed model satisfies the design principles in the following ways:

- Result-sets provide a high-level abstraction for VD designers (or users) to gather information of interest from the WWW by using a database query-like approach.

- Views provide a high-level abstraction for VD designers to specify a hypertext representation of a result-set for presenting and browsing the retrieved information in the set.

- A VD Graph provides a high-level abstraction for VD designers to structure the collected information by organizing and connecting the views into an integrated hypertext network, which serves as a personal Web space.

- It is tailoriable and extensible by allowing application architects and programmers to define new application-specific components, e.g., result-set operations, view types.

- It is simple and easy to maintain.

- The Virtual Document generated for representing a personal Web space is a network of HTML documents, and thus adheres to an open industry standard.

In the next chapter, we will present a VD Specification Language designed for this model.
Chapter 3

Examples of Virtual Documents and their Specifications

This chapter presents, by example, the VD Specification Language. The syntax and semantics of the language are further described in Appendix A.

We start with some SimpleVDGraphs (Section 3.1) to illustrate how different view types can be used to present a collection of documents returned by a WebSQL query in different formats of HTML pages. Each view type (Section 3.1.1 to Section 3.1.8) has its own unique features for facilitating understanding of the query results and supporting navigational aids for browsing. We then show an example of CompositeVDGraph (Section 3.2), which demonstrates how various types of views can be assembled into a complex VD hypertext structure by embedding views into nodes and by linking between nodes and views with the two edge types implemented. All the result-set operations currently supported are employed in this example.

There is a structural pattern for readers to follow in this chapter, as each example is organized into subsections which describe the three aspects of a Virtual Document respectively:

1. the VD Specification,

2. the structure and the presentation of the information content, and

3. the navigational aids for browsing and exploration.
3.1 Simple VD Graph

A Simple VD Graph consists of just one view. Users can choose a view type for it in the VIEWS section of the VD Specification. There are altogether four view types supported in the current implementation, namely *TupleListView*, *DocumentView*, *DocumentList-View*, and *DocumentTreeView*. Each of them takes in a result-set as input, and generates as output one or more HTML pages, which represent a hypertext view of the documents in the set. Some view types accept several types of result-sets and exhibit a different behavior for each result-set type. We will look at one example for every possible combination of view type and result-set type in a Simple VD Graph in the following subsections.

To make it easier for readers to compare the different view types, all the examples of views in this section are created from the same result-set of documents, though the result-set type varies or an additional result-set is used when some operation is applied onto the original result-set before forming the view. This base result-set, whose elements are defined by a WebSQL query, is given in Figure 3.1. In what follows, we first explain what the result-set is.

Suppose we want to collect documents that are related to the database course 2531 offered by the Computer Science Department at the University of Toronto, and we have partial knowledge on how the documents are arranged in the CS Department. We know the URL of the CSLab HomePage, and that the HomePage of the course lecturer, Alberto Mendelzon, can be reached from the CSLab HomePage through a link with a label containing “Alberto Mendelzon”. In addition, we have the following information on how documents under his HomePage are organized: (1) database-related documents relevant to the course can be reached from his HomePage by following the links with a label containing “Database Group”, “Papers”, or “CSC 2531” and (2) after we navigate one level down via these links, other course-related documents may be reached via links with a label containing “Course” or “Queries”. The WebSQL query (line 2 to line 7) in Figure 3.1 expresses these.

We first define three link types [*is2531Lecturer*] (line 2), [*is_db_related*] (line 3), and
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS

RESULT_SET r_course MAX_SIZE 30 {
  DEFINE LINK [is2531Lecturer] AS
    label CONTAINS "Alberto Mendelzon";
  DEFINE LINK [is_db_related] AS
    label CONTAINS "Database Group" or
    label CONTAINS "Papers" or
    label CONTAINS "CSC 2531";
  DEFINE LINK [is_course_related] AS
    label CONTAINS "Course" or
    label CONTAINS "Queries";
  SELECT CourseDocument.url, CourseDocument.title, CourseDocument.type
  FROM DOCUMENT LecturersWebPage SUCH THAT
    "http://www.cs.toronto.edu/homepages.html" [is2531Lecturer] LecturersWebPage,
    DOCUMENT CourseDocument SUCH THAT
    LecturersWebPage = | [is_db_related] | [is_db_related][is_course_related] CourseDocument;
};

Figure 3.1: The Base Result Set Used in All the Examples of Simple VD Graph

[is_course_related] (line 4). A link will match the link type [is2531Lecturer] only if its label contains the string "Alberto Mendelzon". Similarly, a link will match either of the other two link types only if it satisfies the predicate defined for the link type. Once the link types are defined, we use them in the path regular expressions as shown in line 6 and line 7. The former regular expression restricts the path starting from the CSLab HomePage to the link that matches the link type [is2531Lecturer], whereas the latter restricts the path starting from the Lecturer’s HomePage to have zero links (denoted by =), to have a link that matches [is_db_related], or to have a link that matches [is_db_related] followed by a link that matches [is_course_related].

The url, title, and type attributes of the documents found (line 5) are collected into the result-set r_course. The number of query answers to be stored in the result-set is restricted to a maximum of 30 (line 1).

With this base result-set ready, we can now start touring through the different examples of views of it.
3.1.1 Presenting Collected Information as a Table

Suppose we want to present the collected documents in \textit{r\_course} as a table to facilitate understanding of their attributes and to provide access to the documents for browsing. We can specify a Virtual Document as shown in the example \textit{tlist1\_simple}. First, we specify the result-set \textit{r\_course} to gather the course-related documents from the Web, and then we choose the view type, TupleListView, to represent the result-set as a table. This view is the only view in the Virtual Document.

The VD Specification

We now explain the listing of the VD Specification for \textit{tlist1\_simple} in Figure 3.2. A VD Specification starts with the reserved word VD followed immediately by the VD name, and ends with the reserved word END-VD. Basically, three sections can be found in every VD Specification of a Simple VD Graph: the SEARCH section, the VIEWS section, and the HYPERTEXT section (in line 1, line 4, line 6 respectively). Information of interest is specified as result-sets to be retrieved from the Web in the SEARCH section, which contains just one result-set, \textit{r\_course} (line 2), in this example. To declare the Virtual Document as a Simple VD Graph, we use the construct SIMPLE in line 7 in the HYPERTEXT section. By definition, only a single view is allowed in a Simple VD Graph, and its name must be the same as the VD name (\textit{tlist1\_simple}, in this case). The type of this view is declared in the VIEWS section, with the reserved word TUPLE_LIST used for choosing the view type TupleListView, and the result set \textit{r\_course} supplied as input for constructing the view (see line 5). No operation that could change a result-set's structure is applied to \textit{r\_course}; such a result-set is a SimpleResultSet. We specify the schema to be \textit{r\_course} (CourseDocument.url, CourseDocument.title, CourseDocument.type) in the attribute list of the SELECT clause as shown in line 3 of this specification.

The Structure and the Presentation of the Information Content

Figure 3.3 shows the Virtual Document \textit{tlist1\_simple} in a Netscape Navigator browsing session.
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS 58

VD tlist1_simple

1 SEARCH
2 RESULT_SET r_course MAX_SIZE 30 {
   DEFINE LINK [is2531Lecturer] AS
   label CONTAINS "Alberto Mendelzon" ;
   DEFINE LINK [is_db_related] AS
   label CONTAINS "Database Group" or
   label CONTAINS "Papers" or
   label CONTAINS "CSC 2531" ;
   DEFINE LINK [is_course_related] AS
   label CONTAINS "Course" or
   label CONTAINS "Queries" ;
3 SELECT CourseDocument.url, CourseDocument.title, CourseDocument.type
   FROM DOCUMENT LecturersWebPage SUCH THAT
   "http://www.cs.toronto.edu/homepages.html" [is2531Lecturer] LecturersWebPage,
   DOCUMENT CourseDocument SUCH THAT
   LecturersWebPage = [ is_db_related ] [ is_db_related ] [ is_course_related ] CourseDocument ;
};
END_SEARCH

4 VIEWS
5 VIEW tlist1_simple TUPLE_LIST r_course ;
END_VIEWS

6 HYPERTEXT
7 SIMPLE
tlist1_simple
END_SIMPLE
END_HYPERTEXT

END_VD

Figure 3.2: tlist1_simple - The VD Specification
### SECTION 1: Tuple Table for tlist1_simple

<table>
<thead>
<tr>
<th>Course Document.url</th>
<th>Course Document.title</th>
<th>Course Document.type</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.cs.toronto.edu/~esin/tok.html">http://www.cs.toronto.edu/~esin/tok.html</a></td>
<td>Database And Information Systems Faculty</td>
<td>text/html</td>
</tr>
<tr>
<td><a href="http://www.cs.toronto.edu/~esin/tok/teslam2.html">http://www.cs.toronto.edu/~esin/tok/teslam2.html</a></td>
<td>(null)</td>
<td>(null)</td>
</tr>
<tr>
<td><a href="http://www.cs.toronto.edu/~esin/tok/teslam5.html">http://www.cs.toronto.edu/~esin/tok/teslam5.html</a></td>
<td>(null)</td>
<td>(null)</td>
</tr>
</tbody>
</table>

Figure 3.3: tlist1.simple - Presenting Collected Information as a Table

### SECTION 1: Tuple Table for tlist2_simple

<table>
<thead>
<tr>
<th>Group 1 where CourseDocument.type=text/html</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course Document.url</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td><a href="http://www.cs.toronto.edu/~esin/tok.html">http://www.cs.toronto.edu/~esin/tok.html</a></td>
</tr>
<tr>
<td><a href="http://www.cs.toronto.edu/~esin/tok/teslam2.html">http://www.cs.toronto.edu/~esin/tok/teslam2.html</a></td>
</tr>
<tr>
<td><a href="http://www.cs.toronto.edu/~esin/tok/teslam5.html">http://www.cs.toronto.edu/~esin/tok/teslam5.html</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group 2 where CourseDocument.type=(null)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course Document.url</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td><a href="http://www.cs.toronto.edu/~esin/tok.html">http://www.cs.toronto.edu/~esin/tok.html</a></td>
</tr>
<tr>
<td><a href="http://www.cs.toronto.edu/~esin/tok/teslam.html">http://www.cs.toronto.edu/~esin/tok/teslam.html</a></td>
</tr>
<tr>
<td><a href="http://www.cs.toronto.edu/~esin/tok/teslam2.html">http://www.cs.toronto.edu/~esin/tok/teslam2.html</a></td>
</tr>
<tr>
<td><a href="http://www.cs.toronto.edu/~esin/tok/teslam5.html">http://www.cs.toronto.edu/~esin/tok/teslam5.html</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group 3 where CourseDocument.type=application/postscript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course Document.url</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td><a href="http://www.cs.toronto.edu/~esin/tok/teslam7.html">http://www.cs.toronto.edu/~esin/tok/teslam7.html</a></td>
</tr>
</tbody>
</table>

| | | |
| | | |

Figure 3.4: tlist2.simple - Grouping Collected Information into Subtables
We call this view type TupleListView because the ordering of the tuples in the result-set is preserved, though the semantics and relevance of the ordering depends on individual result-sets. In this example, the ordering does not have specific meaning, but we will give an example to show how it is applied later.

To represent the logical table as a hypertext, a TupleListView is formatted into an HTML table. There is only one HTML page in a Simple VD Graph of TupleListView: this page serves as a simple hypertext node for embedding the table as the only section in the document. The format of the section shown in Figure 3.3 — a symbol image, a heading, and a body — is the standard presentation format for a section in a node. We will see more complicated nodes with multiple sections later in the examples of Composite VD Graph.

The Navigational Aids for Browsing and Exploration

For users to browse the document with a url given by an attribute value in the table, a hyperlink is created for each url attribute as shown in the first column of Figure 3.3. The value of the url attribute is used as both the label and the destination url of the HTML anchor corresponding to each hyperlink. When a user clicks on a link, instead of displaying the selected document in the same browser window that shows the table, the target frame of all anchors is set to a new frame so that a separate browser window is opened for displaying the chosen document. Every subsequent click on a url hyperlink in the table will cause the document of that url to be shown in this separate browser window. Moreover, when the user finds some interesting link in the selected document during browsing, he can click on the link to bring in the new document which may not be included in the table to explore outside the view. With two browser windows working in parallel, users can perform the browsing and exploration functions in the new window while keeping the TupleListView as the navigation context in the original window.

1Icon Copyright by the WWW Virtual Library project (http://www.w3.org/), by kind permission.
3.1.2 Grouping Collected Information into Subtables

The information collected in a result-set can very often be classified into categories to improve organization and facilitate analysis. As in the previous example, we may want to group the collected documents into subtables according to the types of the documents. The Virtual Document \textit{tlist2.simple} is specified for this purpose.

The VD Specification

The VD Specification for \textit{tlist2.simple} shares the same basic structure as that for \textit{tlist1.simple} listed in Figure 3.2. Only the significant differences are pointed out here for comparison purpose.

The core change is the division of the tuple elements in \textit{r.course} into subgroups according to the value of its \texttt{CourseDocument.type} attribute: tuples sharing the same value of the attribute are organized into a subgroup. This is achieved by adding the following lines into the new VD Specification, just before the VIEWS section at line 4 of Figure 3.2:

\begin{verbatim}
OPERATIONS
    r.grp.course = GROUP_BY_FIELD( r.course, CourseDocument.type );
END_OPERATIONS
\end{verbatim}

The above lines specify an OPERATIONS section that we have not used in the previous example, where a GROUP\_BY\_FIELD operation is applied to the query result-set \textit{r.course}, taking the \texttt{CourseDocument.type} attribute as an argument. The operation returns a \textit{GroupedResultSet} named as \textit{r.grp.course}.

We then create a TupleListView from this new result-set by replacing the view declaration at line 5 of the original specification with the declaration as follows:
In the new declaration, we supply \texttt{r\_grp\_course} in place of \texttt{r\_course} for constructing \texttt{tlist2\_simple}.

Readers can also refer to the complete listing in Figure 3.5.
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS

The Structure and the Presentation of the Information Content

Figure 3.4, which displays a TupleListView of a GroupedResultSet in a Netscape Browser, is shown just below the one for SimpleResultSet (Figure 3.3). Readers can compare the two on the same page.

The Navigational Aids for Browsing and Exploration

A TupleListView for a GroupedResultSet differs from the one for a SimpleResultSet only in the way it structures and presents its information content. It supports the same standard navigational aids for TupleListView introduced in the previous subsection.

3.1.3 Comparing Two Result Sets

When we have two result-sets defined by similar queries, it is always convenient to have a view that allows us to compare the two result-sets side by side and find out the common answers shared by both as well as the answers that are unique in each set. This view also helps us to decide which document in the result-sets to choose to browse. To provide such a view, a Virtual Document can be specified to collect the two sets of information from the Web into two result-sets, apply the COMPARISON operation to them, and then choose the view type TupleListView to present the derived result-set as a table. In what follows, the Virtual Document tlist3.simple is given as an example. We introduce another result-set to be compared with r-course and specify a view for this purpose.

The VD Specification

An additional result-set r-query is used in this example, as given below:

Suppose we also want to collect the course documents related to database queries that can be reached from the HomePage of the lecturer of course 2531. We know that we might find them either by following the links with a label containing "Queries" or "Query" from the lecturer’s HomePage or, by first navigating one level down via a local link (a link that points to a document on the same server), and then following the links with a label containing "Queries" or "Query", as we do at the upper level. The query
RESULT_SET \textit{r\_query} MAX\_SIZE 30 
\{  
  \text{DEFINE} LINK [is\_2531\_Lecturer]  
  \text{AS}  
  \text{labeled CONTAINS "Alberto Mendelzon" ;}  
\}  
\text{DEFINE} LINK [is\_query\_related]  
\text{AS}  
\text{labeled CONTAINS "Queries" or  
\text{labeled CONTAINS "Query" ;}  
\}  
\text{SELECT} \text{CourseDocument.url, CourseDocument.title}  
\text{FROM}  
\text{DOCUMENT LecturersWebPage SUCH THAT}  
\text{"http://www.cs.toronto.edu/homepages.html" [is\_2531\_Lecturer] LecturersWebPage,}  
\text{DOCUMENT CourseDocument SUCH THAT}  
\text{LecturersWebPage} = | [is\_query\_related] | (\Rightarrow [is\_query\_related]) CourseDocument;  
\}

(line 1 to line 5) expresses these. A new link type \text{[is\_query\_related]} is defined in line 2,  
and we use the path regular expression in line 5 to restrict the path from the lecturer's  
HomePage to have zero links (denoted by =), to have a link that matches the link type  
\text{[is\_query\_related]}, or to have a local link (denoted by a right arrow) followed by a link  
that matches \text{[is\_query\_related]}.  

The \text{url} and \text{title} attributes\(^2\) of the documents found (line 3) are collected into the  
result-set \textit{r\_query}.  

This result-set declaration is added into the SEARCH section of the specification in  
Figure 3.2 to specify that both \textit{r\_course} and \textit{r\_query} are to be retrieved for building the  
Virtual Document.  

We then insert the OPERATIONS section before the VIEWS section, applying a  
COMPARISON operation to the two result-sets as follows:

\(^2\)Note that the schema is slightly modified to give a more concise presentation, so only the \text{url} and \text{title} attributes are selected. The same change on the schema is done on the SELECT clause for \textit{r\_course} in this particular example; otherwise the base result-set is kept constant throughout all examples in this section.
The COMPARISON operation takes in the two result-sets and a label string for each of them as arguments, and returns a *ComparisonResultSet* called *r_compare*. Again, we use *r_compare* in place of *r_course* for constructing *tlist3_simple* in a new view declaration, as we did in the previous subsection.

Readers can refer to Figure 3.6 for the complete listing.

The Structure and the Presentation of the Information Content

Figure 3.7 shows the Virtual Document *tlist3_simple* in a browsing session.

A TupleListView of a *ComparisonResultSet* is structured as a nested table (the outer table) of two attributes, each being a table (the inner table) itself. The outer table heading is formed by the two labels supplied for the result-sets in the COMPARISION operation which serve as the attribute names of the new schema. Their implicit type is result-set. For both of its inner tables, the schema is given by that of each result-set as in *SimpleResultSet*. A tuple in the outer table is the concatenation of one tuple element from each inner table. Empty columns are filled with null values if one inner table has more elements than the other. In this case, there are ten course-related documents but only eight query-related documents, so the two attributes of the *r_query* table are padded with null entries in the last two rows.

To highlight the comparison function in the presentation as an HTML table, color-ball images are displayed with the first attribute of each of a matching tuple pair. Ten different color-ball images are supported for first ten matching tuples in the current implementation. For example, five matching entries are identified when comparing the Course-Related result-set with the Query-Related result-set: we use navy-blue-ball image
FIGURE 3.6: tlist3.simple - The VD Specification
### SECTION 1: Tuple Table for tlist3_simple

<table>
<thead>
<tr>
<th>Course-Related Documents</th>
<th>Course Document.title</th>
<th>Query-Related Documents</th>
<th>Course Document.title</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="http://www.cs.toronto.edu/~mendel/" alt="Course Document.url" /></td>
<td>Alberto Mendelzon's Home Page</td>
<td><img src="http://www.cs.toronto.edu/~mendel/" alt="Course Document.url" /></td>
<td>Alberto Mendelzon's Home Page</td>
</tr>
<tr>
<td><img src="http://www.cs.toronto.edu/~mendel/database.html" alt="Course Document.url" /></td>
<td>Database And Information Systems Faculty</td>
<td><img src="http://www.cs.toronto.edu/~mendel/database.html" alt="Course Document.url" /></td>
<td>Database And Information Systems Faculty</td>
</tr>
<tr>
<td><img src="http://www.cs.toronto.edu/~mendel/top.ps.html" alt="Course Document.url" /></td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
</tr>
</tbody>
</table>

Figure 3.7: tlist3_simple - Comparing Two Result Sets
to highlight the first tuple in Course-Related Documents which matches the first tuple in Query-Related Documents, red-ball image to highlight the matching pair of the fourth r.course tuple and the fifth r.query tuple, and a different color-ball image for each of the remaining three matching tuples. Like all TupleListViews, the outer table is embedded as the only section in an HTML page.

The Navigational Aids for Browsing and Exploration

The only difference from TupleListViews of other result-set types is that there are two inner tables in a TupleListView for ComparsionResultSet, and hyperlinks are created for all url attributes for both tables regardless.

3.1.4 Creating a Compact View for a Document Collection

Very often, we specify queries to retrieve documents from the Web for building topical document collections. In these cases, each tuple of such a query is mapped to a single document in the Web (see the document-collection perspective in Section 2.3.3). It is desirable to create a compact view that enables us to refer to an index of all documents in the collection when we are browsing, in the same window, either a specific document selected from the collection or some document reachable from a document in the collection by following links. The result-set r.course is an example of a document-collection. We specify the Virtual Document d1.simple to choose the view type DocumentView for representing r.course for this purpose.

The VD Specification

Based on the VD Specification for TupleListView given in Figure 3.2, only one significant modification needs to be made to create d1.simple — the view declaration at line 5 is replaced by the following:

```
VIEW d1.simple DOCUMENT r.course (CourseDocument.url, CourseDocument.title);
```
In the new declaration, the reserved word DOCUMENT is used for choosing the view type DocumentView, and \texttt{r.course} is supplied as the input result-set for constructing the view. Unlike a TupleListView, which takes in every attribute (even spanning multiple documents) of a general result-set with any number and any type of attributes supported in the WebSQL query language, a DocumentView only uses two attributes of a result-set: a mandatory url attribute and an optional title attribute (with the same Table Variable as that of the url attribute). In this example, the two attributes of \texttt{r.course}, \texttt{CourseDocument.url} and \texttt{CourseDocument.title}, are specified as input for the DocumentView.

Complete listing of the specification is shown in Figure 3.8.

\begin{verbatim}
VD d1_simple
SEARCH
RESULT_SET r.course MAX_SIZE 30 {
  DEFINE LINK [is253-lecturer]
  AS
    label CONTAINS "Alberto Mendelzon";
  DEFINE LINK [is.db.related]
  AS
    label CONTAINS "Database Group" or
    label CONTAINS "Papers" or
    label CONTAINS "CSC 2531";
  DEFINE LINK [is.course.related]
  AS
    label CONTAINS "Course" or
    label CONTAINS "Queries";
  SELECT CourseDocument.url,
       CourseDocument.title,
       CourseDocument.type
  FROM DOCUMENT LecturersWebPage SUCH THAT
      "http://www.cs.toronto.edu/homepages.html" [is253-lecturer] LecturersWebPage,
     DOCUMENT CourseDocument SUCH THAT
      LecturersWebPage = | [is.db.related] | [is.db.related][is.course.related] CourseDocument;
}
END/Search

VIEWS
VIEW d1_simple DOCUMENT r.course (CourseDocument.url, CourseDocument.title);
END/VIEWS

HYPERTEXT
SIMPLE
  d1_simple
END_SIMPLE
END_HYPERTEXT

END_VD

Figure 3.8: d1_simple - The VD Specification
\end{verbatim}
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS

The Structure and the Presentation of the Information Content

Intuitively, a DocumentView is a set of documents uniquely identified by every value of the url attribute in the input result-set. The optional title attribute provides more descriptive information about each document. Each element in the view must represent a document in the Web, whereas such a restriction does not exist in a TupleListView, whose tuple elements may be formed from attributes of multiple documents in some cases.

In the presentation as shown in Figure 3.9, we take advantage of the HTML frame feature to create a compact DocumentView by horizontally dividing the top (root) frame of the browser window into two frames. The upper frame is the Context Frame for showing an index of the documents in the view, and the lower one is the Doc(ument)-Display Frame for showing the current selected document from the document-set. Figure 3.10 gives the full contents of the index page of d1_simple in the Context Frame, listing the attributes, CourseDocument.url and CourseDocument.title of the ten document elements from r_course. By default, the first element returned by the WebSQL query is chosen to be the active page to be displayed in the Doc-Display Frame when the Virtual Document is first loaded (the course lecturer’s home page is the default active page in this case). Users can activate one document in the view at any one time.

The Navigational Aids for Browsing and Exploration

Similar to TupleListView, DocumentView supports one hyperlink for each value of url attribute in the result-set for users to choose the active document to browse. One minor difference is that we use the bullet image and the document number as the label of the anchor rather than the url value. The document pointed to by an activated hyperlink in the Context Frame is displayed in the Doc-Display Frame, which is the target frame of all hyperlinks in the DocumentView. Figure 3.11 shows the result when a user clicks on the eighth document hyperlink, CSC2531 Home Page. This provides an alternative to using two browser windows as in TupleListView for users to keep a navigation context of a document-set and select a document to browse and explore. Suppose the user wants to know more about the References in the course after reading the details of the
Figure 3.9: *d1_simple* - Creating a Compact View for a Document Collection

Figure 3.10: *d1_simple* - Contents of the Index Page in the Context Frame
Figure 3.11: d1_simple - Choosing a Document to be Browsed in the Doc-Display Frame

Figure 3.12: d1_simple - Navigating Outside the View Context in the Doc-Display Frame
CSC2531 Home Page. He can activate the ‘References’ hyperlink in the page and navigate further down the hypertext structure afterwards (see Figure 3.12). At any point of his exploration, he can return to the basic result-set of primary interest by clicking on any hyperlink in the upper Context Frame.

3.1.5 Grouping the Documents in a Compact View of a Collection

As described in Section 3.1.2, we can group the collected information of a result-set into subtables. If a result-set is a document-collection and the view type DocumentView introduced in the previous section is chosen to represent it, we can also group the documents of the view into subsets and reflect this structure in the index. We can then refer to this structured index when we are browsing a document in the same window. The Virtual Document \( d_{2\text{..simple}} \) is used to illustrate this.

The VD Specification

As in Section 3.1.2, we insert exactly the same OPERATIONS section with the GROUPBYFIELD operation into the specification in Figure 3.2 to form the result-set \( r_{-\text{grp\_course}} \). We then replace the view declaration by the following:

```
VIEW d2_simple DOCUMENT r-grp-course (CourseDocument.url, CourseDocument.title);
```

Readers can refer to Figure 3.13 for the full VD Specification listing.

The Structure and Presentation of Information Content

The document-set of the view is grouped into subsets according to the three distinct values of CourseDocument.type found in it, giving three result-sets of documents, namely text/html, (null), and application/postscript. This structure is reflected in the index


VD *d2_simple*

SEARCH
RESULT_SET r.course MAX_SIZE 30 {
    DEFINE LINK [is2531Lecturer] AS
    label CONTAINS "Alberto Mendelson" ;
    DEFINE LINK [is.db.related] AS
    label CONTAINS "Database Group" or
    label CONTAINS "Papers" or
    label CONTAINS "CSC 2531" ;
    DEFINE LINK [is.course.related] AS
    label CONTAINS "Course" or
    label CONTAINS "Queries" ;
    SELECT CourseDocument.url, CourseDocument.title, CourseDocument.type
    FROM
    DOCUMENT LecturersWebPage SUCH THAT
    "http://www.cs.toronto.edu/homepages.html" [is2531Lecturer] LecturersWebPage,
    DOCUMENT CourseDocument SUCH THAT
    LecturersWebPage = [is.db.related] [is.db.related][is.course.related] CourseDocument ;
};
END SEARCH

OPERATIONS
    r_grp.course = GROUP_BY_FIELD( r.course, CourseDocument.type ) ;
END OPERATIONS

VIEWS
    VIEW d2_simple DOCUMENT r_grp.course (CourseDocument.url, CourseDocument.title);;
END VIEWS

HYPERTEXT
    SIMPLE
        d2_simple
    END SIMPLEX
END HYPERTEXT

END VD

Figure 3.13: *d2_simple* - The VD Specification
Figure 3.14: \textit{d2\_simple} - Grouping the Documents in a Compact View of a Collection

Figure 3.15: \textit{d2\_simple} - Contents of the Index Page in the Context Frame
of the DocumentView (see Figure 3.15). There is no other difference between a DocumentView of *SimpleResultSet* and a DocumentView of *GroupedResultSet*. The same presentation format using Context Frame and Doc-Display Frame introduced in the previous subsection is adopted here, as shown in Figure 3.14.

**The Navigational Aids for Browsing and Exploration**

DocumentViews for both *SimpleResultSet* and *GroupedResultSet* support the same navigational aids.

### 3.1.6 Structuring Documents into a Navigational List

To further facilitate navigation through the documents in a collection, we can structure them into a navigational list. This list structure can be imposed on the documents in a collection regardless of their actual underlying hypertext structure in the source Web: the result-set may capture a graph of documents interconnected by hyperlinks, or it may consist of any arbitrary collection of unrelated documents. This is useful because many users, who are more accustomed to reading books, still prefer paging through a linear structure. In the example *r_course* that we have been using, we can structure the course documents collected into a navigational list. The list can be viewed as a sequential guided tour through the specific region of the hypertext structure of the lecturer’s Webpages reachable from his HomePage. The Virtual Document *dlist1_simple* is specified for this purpose.

**The VD Specification**

The VD specification is modified in a way similar to the modification of DocumentView specification in Section 3.1.4. Basically, we only need to substitute a new DocumentListView declaration (see below) for the one at line 5 of Figure 3.2:

```
VIEW dlist1_simple DOCUMENT_LIST r_course (CourseDocument.url, CourseDocument.title);
```
The new VD Specification is shown in Figure 3.16.

```plaintext
VD dlist1_simple
SEARCH
RESULT_SET r_course MAX_SIZE 30 {
  DEFINE LINK [is2531Lecturer]
    AS
      label CONTAINS "Alberto Mendelzon" ;
  DEFINE LINK [is.db.related]
    AS
      label CONTAINS "Database Group" or
      label CONTAINS "Papers" or
      label CONTAINS "CSC 2531" ;
  DEFINE LINK [is.course.related]
    AS
      label CONTAINS "Course" or
      label CONTAINS "Queries" ;
  SELECT CourseDocument.url, CourseDocument.title, CourseDocument.type
  FROM DOCUMENT LecturersWebPage SUCH THAT
    "http://www.cs.toronto.edu/homepages.html" [is2531Lecturer] LecturersWebPage,
    DOCUMENT CourseDocument SUCH THAT
    LecturersWebPage = | [is.db.related] | [is.db.related][is.course.related] CourseDocument ;
};
END/Search
VIEWS
  VIEW dlist1_simple DOCUMENT_LIST r_course (CourseDocument.url, CourseDocument.title);
END_VIEWS
HYPERTEXT
  SIMPLE
dlist1_simple
END_SIMPLE
END_HYPERTEXT
END_VD

Figure 3.16: dlist1_simple - The VD Specification
```

The Structure and Presentation of the Information Content

The concept of document-set used in DocumentView was further developed to impose an ordering on the set to give a list structure in DocumentListView. We first mentioned ordering when we looked at TupleListView in Section 3.1.1; now we give an example of how to apply it. Like TupleListView, DocumentListView preserves the ordering of the elements in the input result-set, though the semantics and relevance depends on individual result-sets. In this example, the ordering is determined by the WebSQL search engine in simulating automatic navigation for collecting documents into r_course. The range
expression we used in the query effectively generates a search tree through the hypertext
graph with the Lecturer’s WebPage as root. WebSQL iterates over the anchors in each
document it encounters and adopts a depth-first search, and the result-set \( r\text{-}course \) is
a flattened spanning tree of the subset of documents in the graph that satisfy the path
regular expression. As we expect, the documents in \( r\text{-}course \) are returned in the natural
depth-first traversal order as if a user systematically explored the hypertext. Thus, a
document-list formed from \( r\text{-}course \) can be viewed as a sequential guided tour through
the specific region of the hypertext structure, which gives the relevance of this logical
ordering. As mentioned earlier, DocumentListView is useful because many users are not
used to navigating a graph of documents and would rather page through a linear struc-
ture. In addition, a guided tour helps reduce the “lost in hyperspace” problem when
users are exploring an unfamiliar territory.

We do not consider the ordering in \( r\text{-}course \) to be relevant in TupleListView since
the table structure of TupleListView usually suggests random access. Ordering in Tu-
pleListView is most appropriate for reflecting the ranking of answers returned by querying
an index server. In those cases, users can select from the table which entries they want
to browse based on the ranking in the table and the attribute values of the documents.

Note that it is not necessary to have a relevant ordering in the result-set when spec-
ifying a DocumentListView. Even in cases where the documents are unordered in the
set, we can create a DocumentListView just to provide a list structure for facilitating
navigation.

To represent the DocumentListView as hypertext, we generate a “double-linked list”
of nodes, and an index of all documents in the view as the root of the resultant hypertext
graph. One node is generated for every document in the view such that the forward links
of the nodes follow the ordering explained above. Backward links are complementary
support for navigation in the reverse direction. We will describe how the nodes are
structured and presented in detail later in this subsection.

Two browser windows are used: one for the index, and the other for the current active
node selected by the user. The index page is displayed in the original browser window
where the Virtual Document is loaded. As shown in Figure 3.17, it contains a list of
attributes, CourseDocument.url and CourseDocument.title, of the document elements from r-course, and functions as a table of contents or a menu. Users can click on any entry in the index to load the associated node of the document in the second browser window. We call the top frame of this new window the Content Frame. Figure 3.18 shows the contents of the two browser windows after the first entry is activated to start a sequential navigation from the head of the list. Observe that the Content Frame is horizontally divided into two frames, namely the Context Frame (the upper frame) and the Doc-Display Frame (the lower frame). This is how a node is presented. Every time a node is activated, a pair of HTML pages will be displayed in the two child frames respectively: the document associated with the node in the Doc-Display Frame, and the Context Page generated for that document in the Context Frame. The main object in a Context Page is a navigation bar which contains the hyperlinks for forming the "double-linked list" structure. As shown in Figure 3.18, two button hyperlinks can be found in the Context Page of the head of the list. The light-bulb button ('current' button) represents the hyperlink to the document associated with this node, and the right-arrow button represents the forward link that points to the next node in the list. The Lecturer's WebPage, which is the first document found in the list, is displayed in the Doc-Display Frame. Users can click on the forward link in every Context Page of a node to navigate down the list.

Apart from the two end nodes, all internal nodes have both a forward link (right-arrow button) to the next node in the list and a backward link (left-arrow button) to the previous node. Figure 3.19 gives the full contents of the Context Page of the eighth node as an example. Note that a page number that indicates the current position in the list is given above the navigation bar. This node is shown in Figure 3.20, and finally, the last node in the sequence is shown in Figure 3.21.

The Navigational Aids for Browsing and Exploration

We have already presented how the "double-linked list" structure is used in sequential navigation. Now we give an overall picture of how a DocumentListView facilitates browsing and exploration. In supporting both an index of all documents in the view and a
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS

Figure 3.17: dist1_simple - The Index of a Document List

Figure 3.18: dist1_simple - Starting a Sequential Navigation of the Document List
Figure 3.19: dlist1_simple - Contents of the Navigation Bar in the Context Frame

Figure 3.20: dlist1_simple - Navigating along the Document List

Figure 3.21: dlist1_simple - Reaching the End of the Document List
navigation bar for each document, we aim to provide two levels of navigational context. The index serves as the global context of the view, whereas the navigation bar can be regarded as the local context, i.e., a sliding window along the list. At any one point of a sequential navigation, a user may choose to explore links outside the view in the Doc-Display Frame, and he may return to the local context afterwards, either to resume navigation along the list by clicking on a forward/backward button, or to jump back to the original document he was browsing by clicking on the 'current' button. On the other hand, the index supports random access to any document in the view. Regardless of where the user is located, he can select a document from the index to be browsed in the second window. A new local context will be established in the Context Frame, and he can start sequential navigation from that position.

3.1.7 Grouping Documents into Navigational Lists

We use the Virtual Document $dlist2\_simple$ to illustrate how we can organize a result-set into groups of lists. In the example, documents sharing the same MIME type are structured into a separate navigational list. This enables us to systematically browse each group of documents. It is a slight variation of DocumentListView introduced in the previous subsection, so we will not repeat the details here. The changes made on the original specification are similar to those described for DocumentView of $GroupedResultSet$ in Section 3.1.5, but we use the view type DocumentListView instead. Readers can refer to the complete VD Specification listing in Figure 3.22. Figure 3.23 and Figure 3.24 show how the three sublists formed from $r\_course$ are presented in the Virtual Document.

Note that when we choose to create a DocumentListView of a $GroupedResultSet$, we are no longer interested in a relevant ordering of WebSQL answers in the result-set. We would rather combine the advantages of grouping and a list structure: the index gives us a global context of different groups of documents in the view, while the sublists support sequential navigation in each group separately. For example, the user may want to just scan through pages of type text/html, as shown in Figure 3.24, and use the index to pick some postscript file to read.
Chapter 3. Examples of Virtual Documents and their Specifications

VD \textit{dlist2\_simple}

SEARCH
RESULT\_SET \texttt{r\_course} MAX\_SIZE 30 {
\hspace{1em}DEFINE LINK \texttt{[is2531Lecturer]} AS \\
\hspace{2em}label CONTAINS "Alberto Mendelzon" ; \\
\hspace{1em}DEFINE LINK \texttt{[is\_db\_related]} AS \\
\hspace{2em}label CONTAINS "Database Group" or \\
\hspace{2em}label CONTAINS "Papers" or \\
\hspace{2em}label CONTAINS "CSC 2531" ; \\
\hspace{1em}DEFINE LINK \texttt{[is\_course\_related]} AS \\
\hspace{2em}label CONTAINS "Course" or \\
\hspace{2em}label CONTAINS "Queries" ; \\
\hspace{1em}SELECT CourseDocument.url, CourseDocument.title, CourseDocument.type \\
\hspace{1em}FROM \\
\hspace{2em}DOCUMENT LecturersWebPage SUCH THAT \\
\hspace{3em}"http://www.cs.toronto.edu/homepages.html" \texttt{[is2531Lecturer]} LecturersWebPage, \\
\hspace{2em}DOCUMENT CourseDocument SUCH THAT \\
\hspace{3em}LecturersWebPage = \texttt{[is\_db\_related]} \texttt{[is\_course\_related]} CourseDocument ; \\
}\;
END\_SEARCH

OPERATIONS \\
\hspace{1em}\texttt{r\_grp\_course} = \texttt{GROUP\_BY\_FIELD( r\_course, CourseDocument.type )} ;
END\_OPERATIONS

VIEWS \\
\hspace{1em}VIEW \texttt{dlist2\_simple} DOCUMENT\_LIST \texttt{r\_grp\_course} \texttt{CourseDocument.url, CourseDocument.title};
END\_VIEWS

HYPERTEXT \\
\hspace{1em}SIMPLE \\
\hspace{2em}\texttt{dlist2\_simple} \\
\hspace{1em}END\_SIMPLE \\
END\_HYPERTEXT

END\_VD

Figure 3.22: \textit{dlist2\_simple} - The VD Specification
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS 84

Figure 3.23: dist2_simple - The Index of Groups of Navigational Lists

Figure 3.24: dist2_simple - Navigating along a Document Sublist
3.1.8 Structuring a Graph of Documents into a Navigational Tree

Suppose we want to make use of the hypertext structure of the documents captured in \texttt{r\_course} to build a navigational structure for browsing these documents in the course-related collection. We also want to use the hypertext structure as a map of the specific portion of the lecturer's Webpage network that is of interest to us. The Virtual Document \texttt{dtree\_simple} is specified for these purposes.

The VD Specification

In this example, we introduce a new type DocumentTreeView in the view declaration as follows:

```sql
VIEW dtree_simple DOCUMENT_TREE r_course (CourseDocument.url, CourseDocument.title, CourseDocument.path);
```

Note that we add a new attribute CourseDocument.path in both the SELECT clause and the view declaration. This attribute is mandatory for forming a DocumentTreeView. The complete listing is shown in Figure 3.25.

The Structure and Presentation of the Information Content

One of the strengths of WebSQL is its ability to discover the structure of a subgraph of the Web, given a known URL as the root. We have explained in Section 3.1.6 how a WebSQL query is used this way and generates the result-set \texttt{r\_course} as a flattened spanning tree of documents in the depth-first traversal order. In DocumentTreeView, we reconstruct the spanning tree from a result-set of this kind, using the path attribute. The document-tree structure so formed serves as a map of that part of the Web.
VD dtree_simple

SEARCH
RESULT_SET r_course MAX_SIZE 30 {
DEFINE LINK [is2531Lecturer]
AS
    label CONTAINS "Alberto Mendelzon" ;
DEFINE LINK [is_db_related]
AS
    label CONTAINS "Database Group" or
    label CONTAINS "Papers" or
    label CONTAINS "CSC 2531" ;
DEFINE LINK [is_course_related]
AS
    label CONTAINS "Course" or
    label CONTAINS "Queries" ;
SELECT CourseDocument.url, CourseDocument.title, CourseDocument.path
FROM
    DOCUMENT LecturersWebPage SUCH THAT
    "http://www.cs.toronto.edu/homepages.html" [is2531Lecturer] LecturersWebPage,
    DOCUMENT CourseDocument SUCH THAT
    LecturersWebPage = | [is_db_related] | [is_db_related][is_course_related] CourseDocument ;
};
END.SEARCH

VIEWS
    VIEW dtree_simple DOCUMENT_TREE r_course (CourseDocument.url, CourseDocument.title, CourseDocument.path);
END.VIEWS

HYPERTEXT
SIMPLE
dtree_simple
END_SIMPLE
END_HYPERTEXT
END_VD

Figure 3.25: dtree_simple - The VD Specification
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS 87

The hypertext of a DocumentTreeView is similar to the one generated for a DocumentListView in Section 3.1.6, except a “double-linked tree” of nodes replaces a “double-linked list”. As before, we create an index of all documents to be the root of the resultant hypertext graph, and we also use two browser windows and frame structures to present the index and the current active node. These are shown in Figure 3.26 and Figure 3.27. The document-tree is represented as a hierarchy in the index, with a number reflecting the position of the document in the hierarchy. In this example, the root document, i.e., the Lecturer’s WebPage, has four children documents directly linked to it (numbered as 1.1 to 1.4). That is why we have four down-arrow buttons in the navigation bar of the root’s Context Page; each down-arrow button represents a downward link that points to a child node in the tree (see Figure 3.27).

Apart from the root and the leaf nodes, all internal nodes have both an upward link (up-arrow button) to the parent node in the tree and one or more downward links (down-arrow buttons) to the child nodes. In this example, suppose a user clicks on the third down-arrow button to navigate from the root to document 1.3 (Course2531 Home Page). He will reach an internal node with the root as parent and document 1.3.1 (a postscript file of the course outline) as its only child in the view (Figure 3.28 and Figure 3.29). Navigating further down will bring him to the leaf as shown in Figure 3.30.

The Navigation Aids for Browsing and Exploration

The way we use the index as a global context and the navigation bar as a local context has been explained in Section 3.1.6. In addition to these supports, the hierarchical structure given in the index can function as a map of the documents within the view, providing orientation for a user in navigation.
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS

Figure 3.26: dtree_simple - The Index of a Navigational Tree

Figure 3.27: dtree_simple - Starting Navigation from the Root of the Document Tree
Figure 3.28: *dtree_simple* - Contents of the Navigation Bar in the Context Frame

Figure 3.29: *dtree_simple* - Reaching an Internal Node of the Document Tree

Figure 3.30: *dtree_simple* - Reaching a Leaf of the Document Tree
3.2 Composite VD Graph - Integrating Views

Unlike a Simple VD Graph which consists of just one view, a Composite VD Graph is a more complex hypertext structure assembled from more than one view. In this section, we present the Virtual Document *compose* as an illustrative example. This example is specially designed to demonstrate all important features rather than targeted for a practical use. Practical examples are provided in the next chapter when we discuss the applications of Virtual Documents. In the example *compose*, we specify different views for *r_course* to enable us to view it from different perspectives and to provide different choices of navigational structures for browsing. The views are organized and connected into an integrated hypertext network which facilitates systematic access to a view and enables us to conveniently navigate from one view to another.

The VD Specification

Due to the length of the original file, the VD Specification is separated into two parts, which can be found in Figure 3.31 and Figure 3.32. Moreover, only portions of it are shown. The full specification can be found in Virtual Document Manager HomePage [42].

First of all, we specify the information to be retrieved from the Web. We use the two result-sets defined earlier in this chapter, namely *r_course* and *r_query* (line 1 and line 2 respectively). By applying various operations on them (line 3), new result-sets are then constructed. We then create five views out of these result-sets: (1) *compose_compare_tab* at line 4 is a TupleListView for comparing *r_course* and *r_query*; (2) *compose_common_docs* at line 5 is a DocumentView of the common documents found in the two sets; (3) *compose_course_excl* at line 6 is a DocumentListView of the Course-Related Documents that are not Query-Related; (4) *compose_course_all_docs* at line 7 is a DocumentListView formed from the union of the two sets, which is then grouped by type; (5) *compose_course_dtree* at line 8 is for representing the Course-Related Documents as a DocumentTreeView.

We then design how these views are assembled into a Composite VD Graph in the sec-
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS

second half of the specification (see Figure 3.32). The overview of the graph structure is given in the HYPERTEXT section, which begins with the construct COMPOSITE at line 1. For clarity, we first specify how the views are integrated into a primary containment tree before adding in other secondary edges to give the final graph. In this case, the tree consists of a ROOT node compose, another NODE compose_childNode, a view EMBEDDED IN the root compose, and EDGES with keyword CONTAINMENT (line 2-line 5). There are two types of ContainmentEdges (e.g., the edge root_to_childNode at line 3 is a simple ContainmentEdge with the root compose as source and node compose_childNode as destination, whereas the edge child_to_course_excl with the keyword INDEX at line 5 is an index ContainmentEdge that links the node compose_childNode to the view compose_course_excl). Declarations for edges of type CrossRefEdge (specified with keyword CROSS_REF) then follow to complete the graph overview specification.

Next, we specify the internal structures for every node in the graph using NODETEMPLATES (see line 7 and line 11). As shown for compose, a node template consists of a TITLE and a sequence of SECTIONs. Each section has a HEADING, a heading type (H1), and a body, which can either be a TUPLE_TABLE for an embedded view (line 8), or an ANCHOR for a ContainmentEdge (e.g., line 9 and line 10). For simple ContainmentEdges, a string can be defined as the LABEL of an ANCHOR (e.g., line 12); otherwise, for index ContainmentEdges, labels are system-generated by specifying SYSGEN_LABEL (see line 13).

The Structure and Presentation of the Information Content

The basic components of a Composite VD Graph are the views of result-sets. They are assembled first into a tree structure by using two classes of objects: AggregateNodes (or nodes for short) and ContainmentEdges (edges). A node can contain embedded views and/or edges. An edge links a source node to another node or a multi-page view. Among the view types covered in Section 3.1, TupleListView is the only type which is an embedded view, the rest are all multi-page views. CrossRefEdges are then added to link a node to another node, a node to a view or vice versa to give the final graph structure.

As explained in Section 3.1, a hypertext structure is generated for a view according
CHAPTER 3.  EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS

VD compose

SEARCH
1  RESULT_SET r_couse MAX_SIZE 30 {
   ... 
   SELECT CourseDocument.uri, CourseDocument.title, CourseDocument.type, CourseDocument.path 
   FROM 
   ... 
   DOCUMENT CourseDocument SUCH THAT 
   LecturersWebPage = | [is_db_related] | [is_db_related][is_course_related] CourseDocument ; 
};
2  RESULT_SET r_query MAX_SIZE 30 {
   ... 
   SELECT CourseDocument.uri, CourseDocument.title, CourseDocument.type 
   FROM 
   ... 
   DOCUMENT CourseDocument SUCH THAT 
   LecturersWebPage = | [is_query_related] | -> [is_query_related]) CourseDocument; 
};
END-SEARCH

OPERATIONS
3  r.couse_summary = PROJECTION( r_couse, CourseDocument.url, CourseDocument.title );
   r.query_summary = PROJECTION( r_query, CourseDocument.url, CourseDocument.title );
   r.compare = COMPARISON( r_course_summary, "Course-Related Documents", 
                              r_query_summary, "Query-Related Documents" );
   r.course_typed = PROJECTION( r_couse, CourseDocument.url, 
                                CourseDocument.title, CourseDocument.type );
   r.intersect = INTERSECTION( r_course_typed, r_query );
   r.union = UNION ( r_course_typed, r_query );
   r.diff = DIFFERENCE ( r_course_typed, r_query );
   r.grpall = GROUP_BY_FIELD ( r_union, CourseDocument.type );
END-OPERATIONS

VIEWS
4  VIEW compose_compare.tab
   TUPLELIST r.compare ;
5  VIEW compose_common_docs
   DOCUMENT r.intersect (CourseDocument.url, CourseDocument.title);
6  VIEW compose_course.xml
   DOCUMENT_LIST r.diff( CourseDocument.url, CourseDocument.title );
7  VIEW compose_all.docs
   DOCUMENT_LIST r.grp.all( CourseDocument.url, CourseDocument.title );
8  VIEW compose.dtree
   DOCUMENT_TREE r.couse( CourseDocument.url, CourseDocument.title, CourseDocument.path );
END_VIEWS

Figure 3.31: compose - The VD Specification (Page 1 of 2)
Chapter 3. Examples of Virtual Documents and their Specifications 93

HYPERTEXT

1

COMPOSITE

ROOT compose;

NODE compose_childNode;

EMBED compose_compare_tab IN compose;

2

EDGE root_to_common_docs

CONTAINMENT NODE.VIEW( compose, compose_common_docs );

3

EDGE root_to_childNode

CONTAINMENT NODE.NODE( compose, compose_childNode );

4

EDGE child_to_course_excl.s

CONTAINMENT NODE.VIEW( compose_childNode, compose_course_excl );

5

EDGE child_to_course_excl.i

CONTAINMENT INDEX NODE.VIEW( compose_childNode, compose_course_excl );

...

6

EDGE all_docs_to_root

CROSS_REF “Back to Root” VIEW.NODE( compose_all_docs, compose );

END_COMPOSITE

END_HYPERTEXT

7

NODE.TEMPLATE compose

TITLE “Root Node - An Example of A Composite Graph of Nodes and Different Types of Views”;

SECTION

HEADING H1

“Embedded View - Comparing Result Sets Returned by Two Queries that Use Different Choice Criteria”;

8

TUPLE_TABLE compose_compare_tab;

END_SECTION

SECTION

HEADING H1

“Containment Edge - linked to a Document PageView of the Common Documents in the 2 Query Results”;

9

ANCHOR root_to_common_docs LABEL “Click Here”;

END_SECTION

SECTION

HEADING H1

“Containment Edge - linked to another Node of Other Navigation Structures”;

10

ANCHOR root_to_childNode LABEL “Click Here”;

END_SECTION

END_NODE.TEMPLATE

11

NODE.TEMPLATE compose_childNode

...

12

ANCHOR child_to_course_excl.s LABEL “Click Here”;

END_SECTION

SECTION

...

13

ANCHOR child_to_course_excl.i SYSGEN_LABEL;

END_SECTION

...

END_NODE.TEMPLATE

END_VD

Figure 3.32: compose - The VD Specification (Page 2 of 2)
to the view type chosen. Now we show how these sub-hypertexts are composed into a more complex hypertext graph, putting emphasis on the representations of nodes, edges, and CrossRefEdges.

When a Virtual Document is first loaded, its root node will be displayed as the entry-point to the hypertext graph (see Figure 3.33). A node that is specified to be the source of some CrossRefEdges is represented by a “composite HTML page” with two frames: one contains a Sections Page, and the other contains an Exit Page. The Sections Page in the upper frame begins with a preamble that comprises a symbol image\(^3\) for a node, a title, and a list of headings for all the sections in this node. This preamble is followed by the sequence of sections, each being used to hold an embedded view or an edge. Figure 3.34 presents section 1 of the root node *compose* with a section heading and an HTML table generated for the TupleListView. The rest of the node can be found in Figure 3.35, which shows the sections for the two edges in this node: the first edge’s destination is the DocumentView *compose.common_docs* (Figure 3.36), and the second edge points to the node *compose.childNode* in Figure 3.37 where more examples of edges are presented.

We use three pairs of edges in *compose.childNode* to illustrate the difference in the two types of ContainmentEdges currently supported: simple and index. The first example in Figure 3.38 gives the pair of edges that both point to the DocumentListView *compose.course.excl*. Note that the simple edge at the top of the figure is represented by a single anchor labeled with the string supplied in the specification, and the anchor points to the index page in the sub-hypertext generated for the view. On the other hand, the index edge at the bottom is represented by multiple anchors, and one anchor is automatically generated for each document node in the list. In fact, an index edge exactly corresponds to an index page in Section 3.1; however, the index is now embedded into a node as an edge and points directly to the document-list within the destination view. Two more examples are shown in Figure 3.39 using another variation of DocumentListView and in Figure 3.40 using a DocumentTreeView.

CrossRefEdges are handled separately. They are presented as anchors in a frame

\(^3\)Logo image Copyright 1996 by Shay Barsabe, "Simple GIFs" (http://www.cruzio.com/ djj/calvary/sg.html), by kind permission
containing the Exit Page at the bottom of the browser window, as shown in Figure 3.41 of the DocumentListView *compose_all_docs*. There are four CrossRefEdges with this view as their source: one points to the root node, another points back to its parent node which shows other navigational structures, the third links this view to a view of common documents, and the last points to another view of course documents. Readers can also refer to other examples in the figures already presented. Observe that CrossRefEdges can start from any node/multi-page view and point to another node/multi-page view, independent of the hierarchy imposed by the tree structure. A node (or a view) which is not specified to be the source of a CrossRefEdge is represented by a simple Sections Page (see Figure 3.40).

**The Navigational Aids for Browsing and Exploration**

The hierarchical containment tree provides the information organization that is customiz- able in the VD Specification to serve the needs of the targeted user. This makes it easier for him to navigate through the hypertext structure and find a view of his interest. A specification designer can choose between the two ContainmentEdge types currently supported when he designs the containment tree. A simple edge is the only allowable type for edges with a DocumentView as destination because this view-type provides only one entry point. On the other hand, both simple and index edges are valid options for linking a node to a multi-page view of other types (e.g. DocumentListView, DocumentTreeView) that have separate index pages as roots of the generated sub-hypertexts and that provide multiple entry points for all indexed navigational nodes within the view. An index edge, which embeds an index page into a node, is good for reducing the number of traversed links to reach a document in the view by one. A simple edge, represented by a single anchor in a node that points to an index page instead of embedding the index, is better for grouping more views into one node without putting too much information into the node and making it too large to be easily browsed/printed.

To facilitate navigation to a section within a node, an interior link is generated for every section in the preamble. Thus, a user can jump directly to a section of choice without using the scroll bar of the browser window to search for its location.
CrossRefEdges support flexible navigation on top of the hierarchical navigation structure built from ContainmentEdges. Backward-linking from a lower level node/view to its ancestor, shortcut-linking from a higher level node in the tree to a frequently accessed descendant node/view, and linking between two related views are some examples commonly used. In general, CrossRefEdges allow arbitrary linking from a node to another node, from a node to a view, or vice versa.
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS

Figure 3.33: An Example of a Composite VD Graph

Figure 3.34: An EmbeddedView in a Node
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS

SECTION 2: Containment Edge – linked to a Document PageView of the Common Documents in the Quay Query Results

- Click Next

SECTION 3: Containment Edge – linked to another Node of Other Navigation Structures

- Click Next

Figure 3.35: Containment Edges in a Node

Figure 3.36: A DocumentView as the Destination of an Edge from the Root Node
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS

SECTION 1: Simple Edge – linked to a DocumentList PageView of Course Documents that are not Specifically Query–Related

SECTION 2: Index Edge – linked to a DocumentList PageView of Course Documents that are not Specifically Query–Related

SECTION 3: Simple Edge – linked to a DocumentList PageView of All Results of the 2 Queries Grouped by Type

SECTION 4: Index Edge – linked to a DocumentList PageView of All Results of the 2 Queries Grouped by Type

SECTION 5: Simple Edge – linked to a DocumentTree PageView of Course–Related Query Results in a Hierarchy

SECTION 6: Index Edge – linked to a DocumentTree PageView of Course–Related Query Results in a Hierarchy

Figure 3.37: A Node as the Destination of an Edge from the Root Node

Figure 3.38: Edges with a DocumentList View of SimpleResultSet as Destination
Figure 3.39: Edges with a DocumentListView of GroupedResultSet as Destination

Figure 3.40: Edges with a DocumentTreeView as Destination
CHAPTER 3. EXAMPLES OF VIRTUAL DOCUMENTS AND THEIR SPECIFICATIONS

Figure 3.41: A PageView with Cross-Reference Edges

Figure 3.42: A PageView without Cross-Reference Edges
Chapter 4

Applications

After going through the basic examples in the previous chapter, readers should be more familiar with the concepts of the objects used in the VD Specification Language and the interactive behaviors of them in a VD browsing session. In this chapter, we proceed to identify some potential areas of applications for Virtual Documents (Section 4.1-4.4). Two practical examples in two different areas are given in Section 4.5 to further illustrate how Virtual Documents can be utilized to model personal Web spaces.

4.1 Querying Multiple Index Servers

When a user starts research in an area and only has vague ideas of what to look for, index servers available on the Web like AltaVista, Excite, HotBot, Yahoo, WebCrawler offer him some good starting points. There is usually an experimental stage in the beginning of the search which may involve activities like: (1) interactively trying out queries using different keywords related to the area, (2) selectively browsing some results of a query, or sequentially browsing the top N results, (3) exploring the neighborhood of the results to look for related topics, (4) repeating the fore-mentioned activities with another index server to see if there are interesting documents not previously discovered.

Instead of interactively trying query after query and search engine after search engine, we can design a Virtual Document to facilitate the process. Multiple WebSQL queries using related keywords can be specified to simulate the experimental sequence of keyword
search, and they can be combined with path regular expressions to simulate automated navigation in exploring the neighborhood of the search results. Answers are gathered into result-sets to be presented as an integrated hypertext view, which can be used as a personal Web for the user to browse and explore. Batch-querying can also be applied to meta-search over multiple index servers by defining result-sets with the same query but a different index server definition. The meta-search approach provides the flexibility in choice of index servers and broadens the scope of search to overcome the limitations of any one index server.

In contrast to other multi-service search approaches, such as MetaCrawler [37] and RAW [8], which employ some sorting function or rank integration function that merges the results from different index servers into an ordered list, the Virtual Document prototype does not provide any new integrated ordering methods. Instead, we currently support a set of operations, namely union, intersection, difference, projection, group_by_field, and comparison. These operations can be freely applied to result-sets of different queries to derive other useful result-sets which facilitates analysis. We believe this approach is more flexible, and it allows the preservation of the original rankings computed by individual index servers. Moreover, our query capability is more powerful as WebSQL integrates both keyword search and automatic navigation. The designer of a Virtual Document can customize a combination of queries with different index server definitions, keywords, path regular expressions, and test conditions that serves the purpose of a search plan. In addition, more than one result-sets can be included in the same Virtual Document as identifiable units either by using the comparison operation or assembling multiple views into a Composite VD Graph. This eliminates the need to bring up parallel index server sessions for comparing different query results or for cross-query browsing. Instead, users can refer to different query results in one or more views in a Virtual Document and freely jump from an answer in one query to another query. It also relieves the restriction caused by compulsory integration of query answers with existing meta-search services, where users have to try one keyword after another if they want to browse the results for each query separately, and there is no way to divide the merged answers for individual index servers. Note that the lack of rank integration function is not a limitation of the
underlying model itself: it can be added as the implementation of a new result-set operation or even multiple operations, one for each new algorithm, and the model always gives the VD designer full control of what operations to use.

Virtual Document provides a richer user interface model than other front-ends to multiple search engines. The designer can select a view type (or more than one if multiple perspectives are desired) for each result-set to provide an appropriate hypertext structure for browsing and exploration of the results, and he can organize multiple views into a Composite VD Graph to facilitate systematic navigation and flexible cross-referencing.

Designing a VD specification may seem to take extra effort at first sight, but in view of the inconvenience and the "lost-in-hyperspace" problem that may involve in trying out everything interactively across search-engine sessions, this initial planning step is worthwhile. Moreover, once a Virtual Document is materialized, it can be reloaded into any browser sessions; otherwise, a user either has to bookmark the query results or retry some queries if he wants to distribute a lengthy search process into multiple sessions and recall previous answers. It is particularly useful for search process that is repeated periodically to collect newest information from the Web; we simply re-materialize the Virtual Document as required.

4.2 Building Customized Global Views in a Web site

Very often, a global view of all documents in a Web site is too complicated and confusing to use. It is desirable for Webmasters to be able to customize views for different kinds of users that are expected to browse/analyze the site by focusing on the aspects of particular interest to them and hiding others. In this regard, Virtual Documents are useful in two aspects of Web-site management: (1) generating Web-site maintenance reports, (2) pre-building subject-oriented views of the Web-site for specific groups of users or communities of interests to support structured browsing, or as a basis for pursuing further exploration.

For Web-site maintenance, we have demonstrated how WebSQL queries can be formed to discover broken links, find references to documents in other Web sites, and find external references from other Web sites, etc. in [3, 41]. Some examples quoted in them can
be easily modified for other maintenance functions like finding local in- and out-links for documents on the Web-site, discovering images referenced by certain documents. Webmasters can specify queries for different functions on either the entire site or portions of the site defined by the range expressions in the FROM clause and the filtering conditions in the WHERE clause of the queries. To generate an integrated maintenance report, Virtual Document views can be defined for result-sets of interested queries, and these views can be organized into a Composite VD Graph for review and analysis.

For building subject-oriented Virtual Documents in general, a Webmaster may be more interested in gathering document collections related to a subject from his site for structured browsing by users. More emphasis is put on providing navigational structures for individual collections by assigning appropriate view types to them, whereas a simple TupleListView as summary of answers is sufficient for most queries used in Web-site maintenance. To facilitate targeted users' understanding of the subject, special attention will be given to the design of a Composite VD Graph, which serves as an organizational structure that guides them through the various collections.

Several commercial products which generate views of Web-site are discussed in the paper on WebCutter [25], a system for dynamic and tailorable site mapping. This class of tools focus on dynamic site mapping and visualization. Basically, the navigational views these products build are structural maps of a Web-site or part of a site. Structural map, however, is only one choice among all view types supported in Virtual Documents: we use the DocumentTreeView to represent a spanning tree map formed by depth-first traversing a subgraph of documents from a starting URL or the actual subtree of documents if hierarchical organization is imposed in the Web-site; whereas other view types can be used to generate alternate navigational structures for that portion of the site, e.g. DocumentListView imposes a flattened list structure on top of a subtree; furthermore, we can choose views types that creates navigational structures for a collection of documents originally not linked together in the site.

Another difference is the customization support. Most site mapping tools do not support generation of customized map for users' interests, but provide exclusively structural map of the whole site. Some tools are primarily for Web-site management/maintenance
purpose, e.g. WebAnalyzer [16], and the customization filters are designed for reporting specific information like broken links, file types, file sizes and for identifying specific areas of the site for close-up analysis. Those available for end-users’ browsing as well do not allow customization of the site map. WebCutter is the only one among them that provides customizable maps that take into account users’ interests by integrating navigation and content-based search. It employs text-analysis technology which evaluates and ranks the relevance of documents while a crawler performs “fish search”. Relevance information is used in the map construction stage to shape the map according to users’ interests by exploring more thoroughly in directions where relevant document is found and neglecting others. While our approach does not do any relevance evaluation, WebSQL supports regular expression matching of text content and flexible boolean combination of test conditions to perform content-based filtering. Our strength lies in the support of path regular expressions with user-defined link predicates to guide the search and simulate controlled navigation; WebCutter does not make use of any link information as selection criteria. Moreover, WebCutter is better for incremental exploration to generate a customized map interactively, whereas Virtual Document’s specification approach is more suitable for batch querying with navigation control and well-defined filtering criteria.

Our support of Composite VD Graph for integrating views on possibly different portions of a site into a single browsing structure is also what existing products lack.

4.3 Sharing Views in a WorkGroups Environment

There is a growing trend for building distributed Web information systems (or Web applications) that consist of several sites. We will consider one possible application here: documentation in a software development environment. Usually, a software development project produces a set of technical documents in different phases of the process like requirements documents, design documents, object interface definition documents, and also other documentations for users. Peoplewise, the project may involve collaborative workgroups. Each of them is responsible for specific tasks in the project and produces its own set of documents for the tasks. In addition, there are various support groups
in charge of areas like hardware, networking, or programming standard, which provide underlying support information usable by all projects. By storing documents of all phases and all levels as hypertext on the Web, Virtual Documents tailored for different tasks can be created as integrated views of document collections selected from this domain of hypertexts, which may be distributed across multiple Web sites.

For example, in a project which consists of several workgroups of developers (one on server side and one on client side, say), each workgroup may be only interested in the portion of information from other groups that are relevant and helpful to their specific task, e.g. the subset of object interface definition documents as specifications for the task, user manuals of developed products, and technical support documentations related to the technology employed in their scope. A VD Specification can be designed to define the queries for gathering the documents of interest, choose appropriate view types as navigational structures, and organize the views into a browsing graph structure that facilitates reference by a workgroup in their task. In effect, the generated hypertext structure forms a coherent Web space of useful documents for specific problem-solving on top of the documentation Web space. Each Virtual Document is shared by all members of the group, and it can be re-materialized periodically to reflect the newest state of documentations during the project lifetime.

We are not aware of other work directly comparable to Virtual Document in this aspect. Instead, we will use two examples to show the diversity of management approaches currently adopted in distributed Web applications, the lack of support for customizable views, and how Virtual Document can provide the support independently.

WebArchitect with PilotBoat [39] and WebComposition [12] are systems which focus on the construction and maintenance of distributed Web application during the Web engineering cycle, but both do not provide any support for deriving customizable views for users' interests. The former approach models each Webpage in a Web application as a resource with meta-links of different semantic types in the HTTP header and additional attributes defined for maintenance purposes. These Webpages are stored in file systems. The HTTP servers are enhanced for handling meta-links, and IP multicast notification agents are used to support distributed collaborative authoring and maintenance. Like
some site mapping tools mentioned in the previous section, the system provides structural map of the whole application meta-link architecture as navigational aid, but filtering of views are just based on maintenance attributes such as access rights. On the other hand, WebComposition adopts a fine-grained object-oriented model to construct a Web application as a hierarchy of components. All components are stored in a central component store (DBMS) accessible by developers and maintenance people through a component server, and the push approach is used by potentially distributed resource generators to map location-related part of the component model to the local file system at each site as cache. One major interest of their project is in incremental automatic generation of affected file-based resources by considering the dependencies of components when a component is modified. Both systems, though very different in their management approaches, can use Virtual Documents as customized views for users' browsing on top of the Webpages constructed by them.

4.4 Organizing Information into Personal Navigation Spaces

In the previous sections, we have looked at applications that gather information on the Web from index servers, from a local Web site, or from multiple Web sites respectively. In general, Virtual Document can be used to collect information from any combinations of computable Web domains, which are defined by using keyword-matching at index servers or through controlled navigation starting from Web sites' root URLs or any known URLs. This makes it a feasible tool for users to build their own personal views of the Web flexibly. Currently, the most common way for users to compensate for the lack of global view of the Web is the use of bookmark mechanisms provided by browsers for incrementally building their own personal information spaces, e.g. Netscape allows users to arrange bookmarks into a hierarchy of folders. This is a convenient method of collecting individual documents of interest when they are discovered serendipitously during exploration and accumulatively in any browser sessions. We see Virtual Documents as an alternative method that can be used to complement the bookmark and folder support. Our approach
is particularly suitable for dynamically gathering documents into pre-determined result-sets that can be defined by queries which simulate users’ navigation and filtering criteria, and also for construction of an integrated navigational structure specified for these result-sets, whereas a user can insert a bookmark with an exact URL of a document into any folders as he likes. Moreover, we support views of result-sets which are more expressive than simple folders, e.g. DocumentListView can facilitate sequential navigation from document to document in the collection, and DocumentTreeView can provide a structural map for a subtree of documents collected in a result-set. In addition, our VD Graph model which supports cross-reference edges on top of the hierarchical organization facilitates more convenient access from a view to other related views.

4.5 Practical Examples

In this section, we show how we can design a personal Web for a specific purpose by using the Virtual Document model in two practical examples. A simple three-step design process is used to outline the high-level design of a Virtual Document:

1. Identify the base result-sets to be collected from the Web

2. Sketch the ResultSet-View (RSV) Relationship Diagram
   (The operators are listed in Figure 4.1.)

3. Sketch the VD Graph Diagram
   (The primitives are listed in Figure 4.3.)

Details of the two Virtual Documents are not covered. Readers can refer to the Virtual Document Manager HomePage [42] for the complete specifications and the materialized VD Contents.

4.5.1 Query Report on Data Warehousing

In this example, we present a case when a user wants to query multiple index servers on the subject “data warehousing”. The index servers he uses most often are AltaVista,
HotBot, and Excite. The one he trusts most is AltaVista. He is not sure which keyword he should use to get the most useful information for him: data warehousing, warehousing, or data warehouse. Yet, he came across the subject when he read about an article which mentions the term data warehousing, so he decides to base his search primarily on this particular keyword. In addition, he is interested in exploring neighboring Web pages referenced by the answers through links which are related to OLAP or Decision Support. To solve this problem, he wants to specify a simple search plan of queries and create a Web space to help him review and explore the results at different times and for different purposes.

First of all, the following base result-sets are identified:

1. r_altavista
   attributes: Doc.url, Doc.title, Doc.type, Doc.length, Doc.modif
description: answers returned when using ‘data warehousing’ as keyword from AltaVista.
2. r_hotbot
   attributes: Doc.url, Doc.title, Doc.type, Doc.length, Doc.modif
description: answers returned when using ‘data warehousing’ as keyword from HotBot.
3. r_excite
   attributes: Doc.url, Doc.title, Doc.type, Doc.length, Doc.modif
description: answers returned when using ‘data warehousing’ as keyword from Excite.
4. r_testcond_1
   attributes: RelevantDoc.url, RelevantDoc.title
description: answers returned from AltaVista when using ‘data warehousing’ as keyword.
5. r_testcond_2
   attributes: RelevantDoc.url, RelevantDoc.title
description: answers returned from AltaVista when using ‘warehousing’ as keyword.
6. r_testcond_3
   attributes: RelevantDoc.url, RelevantDoc.title
description: answers returned from AltaVista when using ‘data warehouse’ as keyword.
7. r_neighbourhood
description: answers returned from AltaVista when using 'data warehousing' as keyword (HomePage), and documents referenced by these answers through links with a label containing 'OLAP' or 'Decision Support' together with the answers themselves (ReferencedDoc).

Views to be constructed from the base result-sets are then shown in the RSV Relationship Diagram (Figure 4.2). Six TupleListViews are provided as overview tables for quick review and selective access, e.g. the user can immediately tell if there are common documents found in all three servers in dwh_common_docs, and he can have combined pictures of different query results as given by the comparison tables dwh_compare_servers (different index servers), dwh_compare_conds and dwh_compare_conds_next (different keywords). To help him systematically browse specific result-sets and facilitate his exploration, a DocumentView is created for the answers of AltaVista, and two DocumentListViews with different grouping criteria (one on document type, the other on the starting page of search) are constructed from the result-set r_neighbourhood.

The integrated Web space for these views is specified in the VD Graph Diagram in Figure 4.4.
Figure 4.1: ResultSet Manipulation and View Creation Operators

Figure 4.2: dwh_report - RSV Relationship Diagram
Figure 4.3: VD Graph Primitives

Figure 4.4: dwh_report - VD Graph Diagram
4.5.2 Web-Team Library

The second example is to build a customized Virtual Document for introducing the DB web-team's pages of the CS Web site at the University of Toronto. We assume the search starts from the Webpage in the site that contains links to all personal homepages of the CS lab, and ignore documents unreachable from this starting point.

The base result-sets for constructing the Virtual Document are as follows:

1. r.team_leader
   attributes: MembersWebPage.url, MembersWebPage.title
   description: the team leader's homepage referenced by homepages.html of the CS Web site.

2. r.team.others
   attributes: MembersWebPage.url, MembersWebPage.title
   description: the team members' homepages referenced by homepages.html of the CS Web site.

3. r.documents
   attributes: Member.url, Member.title, MembersDoc.url, MembersDoc.title
   description: the documents referenced by the whole team's homepages, through links whose label contains a DB or Web-related keyword (a small set of keyword choices are selected).

4. r.alberto
   description: the subtree of documents referenced by the team leader's homepage, through links that satisfy a path regular expression.

5. r.websql
   attributes: WebSQLDocument.url, WebSQLDocument.title
   description: the subtree of documents on the WebSQL project.

6. r.websemantics
   attributes: WebSemanticsDoc.url, WebSemanticsDoc.title
   description: the subtree of documents on the WebSemantics project.
Views are then defined from these result-sets in the RSV Relationship Diagram as shown in Figure 4.5. We provide two primary views for quick reference by users: a TupleListView serves as an overview of all web-team’s homepages, and a DocumentListView web_team_grp_docs presents an individual list of documents referenced by each member. In addition, special navigation structures for touring through the team leader’s collection and through the projects, and other alternate navigation structures for the web-team’s homepages are provided.

Finally, the overall structure of the web-team library Web space is given by the VD Graph Diagram (Figure 4.6).

This ends our illustration of applications of Virtual Document. Readers can view all the examples used in the previous and this chapter at the Virtual Document Manager HomePage [42], and you can try out writing your own VD specifications to generate personal Web space at the CGI interface supported in the prototype.

We will explain the implementation of the prototype in the next chapter.
Figure 4.5: *web_team.lib* - RSV Relationship Diagram

Figure 4.6: *web_team.lib* - VD Graph Diagram
Chapter 5

Implementation

We have given an overview of the system architecture in the introduction of the thesis. As outlined in Section 1.4.3, the system consists of four components: the VD Specification Translator, the VD Construction Driver, the VD Construction Framework, and the WebSQL System. The VD Construction Translator takes a VD Specification file as input and translates it to a Java source program, which is then compiled into a VD Construction Driver class. This Driver class together with other system-supplied Java classes in the Construction Framework and the WebSQL System constitute the VD Construction application for materializing a Virtual Document. In this chapter, we proceed to briefly explain how the two processing phases of translating a VD Specification to Java and materializing a Virtual Document are implemented in our prototype. They are described in Section 5.1 and Section 5.2 respectively.

As for interface with the external world, we have developed two different interfaces that allow a user to submit a VD Specification and issue a materialization request to generate a Virtual Document. One is a simple CGI Interface accessible from a VD Manager HomePage (Section 5.3) and the other is a stand-alone application (Section 5.4).

5.1 Translation to Java

The VD Construction Translator was built using Unix's YACC and written in C. The grammar given in Section A.1 was rewritten as grammar rules in YACC format. We
introduced error recovery rules and logic to detect as many errors as possible in a pass, and the complete list of errors found are reported in a file with a line number locating each error and a message explaining it. Internal data structures are constructed during parsing, and once no syntax and semantics errors are detected, they are used to generate Java source.

The generation mechanism is straightforward. A set of import statements for referencing other Java classes is first output and then the declaration of a public class named after the specified Virtual Document is generated. The class just has a single method main indicating that this is the main program of a Java application. The core of the method is a try statement which contains all the statements for constructing the Virtual Document and has a catch exception part to report any error detected by the construction programming logic or by the Java interpreter. The statements for constructing the Virtual Document can be divided into nine parts: (1) statements for creating instances of the internal system objects whose methods are invoked in the subsequent statements to create the constituent objects of a VD Graph, (2) statements for adding query result-sets, (3) statements for executing the result-set operations to create derived result-sets, (4) statements for creating a VD Graph, (5) statements for adding views, (6) statements for adding nodes, (7) statements for adding sections and for adding containment edges, (8) statements for adding cross-reference edges, and finally (9) the statement for materializing the VD Graph.

One minor point is that we implemented extra YACC grammar rules for extracting the schema attributes in a WebSQL query to perform more semantics checking at translation time. To simplify prototype implementation, both the auxiliary WebSQL define statements before the SELECT statement and the part of the SELECT statement after the reserved word FROM need to be bracketed with \{ \}. By using the JavaCC compiler compiler [38] instead of YACC and implementing the Translator in Java, we could make use of the method getSchema available in latest version of WebSQL API.
5.2 Materialization

A VD Construction application is composed of a Driver class generated for a specific Virtual Document as the main program of the Java application, a library of system-supplied Java classes for supporting all VD Construction applications, and a WebSQL class library that serves as an application programming interface (API) to the WebSQL query engine for access to the Web.

Upon a materialization request, the Driver class activates methods in the underlying Java classes to create internal objects in a Simple or Composite VD Graph. Base result-set objects are instantiated by executing the queries associated with the result-sets; derived result-set objects are instantiated by evaluating the operation expressions that define them; view objects are instantiated by building the internal data structures for the associated result-set instances; and node objects, edge objects and tube objects are instantiated with the parameters supplied in the VD Graph structure.

To generate the physical pages of a Virtual Document with URLs automatically assigned, the first step is to read a configuration file dir.setup residing in the current directory of the Construction classes and instantiate the target directories variables in the node objects and the page view objects accordingly. This file is for users to pre-define target directories for generation of different sets of pages in a Virtual Document. Its format is as follows:

```plaintext
name = default
local.dir = local_dir.name
http.dir = http_dir.name
VIEW
name = view_name
local.dir = local_dir.name
http.dir = http_dir.name
...
NODE
name = node_name
local.dir = local_dir.name
http.dir = http_dir.name
...
```

where `local_dir.name` is the full path name of a directory in local file system to be used as the prefix of the filenames to be generated, `http_dir.name` is the full http address of that directory to be used as prefix of the URLs to be generated, `view_name` is the
name of a page view defined in the VD Specification, and \texttt{node.name} is the name of an aggregate node defined in the specification.

As shown in the format outline, the configuration file consists of three sections: the default section is mandatory for both Simple and Composite VD Graph whereas both the VIEW and NODE sections are optional for Composite VD Graph and not required for Simple VD Graph. The default section is for defining default target directories for all nodes and page views in the Virtual Document. To overwrite them, the VIEW and/or NODE section can be used to define a record of \texttt{[name, local.dir, http.dir]} for each page view or node to be assigned special directories.

URL generation is done by concatenating the prefix string supplied in the configuration file and the filename string internally assigned by the page view or node for each of its pages by following some naming convention. For example, for a DocumentList View named \texttt{web.team.list}, it assigns filenames to each of its pages in the following way: every filename has the view name \texttt{web.team.list} as its prefix; the root filename has the postfix \texttt{-entrydoc.html} and similarly, the filename for every node primitive in the view has its own postfix; multiple occurrences of navigational nodes have an instance number for each occurrence as part of its filename. And the URL for the root entry point of this view could be something like \texttt{http://www.cs.toronto.edu/wong/web.team.list.entrydoc.html}.

In general, to materialize an object in a VD Graph, the method \texttt{materialize} of the object is invoked which in turn invokes the method \texttt{materialize} of each of its constituent objects.

Thus, to materialize a Simple VD Graph, if the only view in it is a page view, the method \texttt{materialize} of that page view is directly invoked; otherwise, if it consists of an embedded view, it activates the method \texttt{materialize} of the simple node which invokes the method \texttt{materialize} of its only section and the latter method in turn materializes the embedded view.

To materialize a Composite VD Graph, all page views in the graph are materialized and then all aggregate nodes are materialized. For each page view, if it contains any exit point corresponding to a tube with this page view as its source, the exit points are materialized while the view generates its own set of pages. For each aggregate node,
every constituent section is materialized and the node’s exit points are materialized; and for each section, if it contains an embedded view, the embedded view is materialized; otherwise, the containment edge is materialized.

The methods materialize of the implemented view types are based on the view types’ hypertext structures as illustrated by the examples given in Chapter 3. The structures are also described in the section on semantics in Appendix A. If readers are interested in the physical node-and-link configurations of the materialized page views, please refer to the figures given in that appendix.

Developers can also design and implement view types for generating new hypertext representations of a result-set. Currently, we have not implemented the hypertext primitives proposed in the view sub-model to facilitate development of view types because the prototype was built before we further generalized and improved the ideas. The supporting Java class library can be extended to develop a more mature framework.

5.3 CGI Interface

Figure 5.1: The VD Manager HomePage
The VD Manager HomePage [42] shown in Figure 5.1 supports the four Virtual Document operations:

1. edit and compile a VD Specification
2. materialize a Virtual Document for a stored specification
3. view a previously submitted VD Specification
4. view the content of a materialized Virtual Document

The menu frame of the Homepage contains a hyperlink for each Virtual Document operation. Each of these links is either connected to a static HTML form or a CGI program for dynamically constructing an HTML form or menu page for supporting the corresponding operation. All CGI programs in the prototype were written in C.

![Figure 5.2: The Form for Editing and Compiling a VD Specification](image)

Figure 5.2 shows the static form for editing and compiling a VD Specification. This form is connected to a CGI program that invokes a stand-alone VD Specification Translator program running on our server machine. The form contains three fields for input: apart from the mandatory Virtual Document name, either the file name of a prepared
Virtual Document Specification is entered or a specification can be input at the corresponding textarea field. In the former case, the specification file must be directly readable by the CGI program running on the same machine as our CS Department Web server hosting the form. When the 'Compile' button is pressed, the CGI program first checks for valid inputs and report any errors. It then invokes the Translator to parse the VD Specification once the form inputs are error-free. If the Translator finds errors in parsing the VD Specification, the error file it produces will be formatted by the CGI program as an HTML file and returned to the browser. Otherwise, the CGI program will invoke the Java Compiler to compile the Java source code produced by the Translator. Any compilation errors are reported by the CGI program as bugs for inspection by programming and system staff. On the other hand, successful compilation results in a \textit{vd\_name.class} being generated where \textit{vd\_name} is the Virtual Document name as defined in the specification. This Java class is stored in an assigned system directory for later use. Similarly, the input VD Specification is stored in another assigned system directory either by copying from the source file or saving the specification text typed in.

![Figure 5.3: The Form for Materializing a Virtual Document](image)

When the hyperlink associated with the 'Materialize VD Content' operation is activated, a CGI program is invoked to generate on-the-fly a form for selecting a Virtual Document to materialize (see Figure 5.3). The names of all currently stored Java classes of VD Construction Driver are collected to construct a list of select options in an HTML form. When one of the VD names is selected and the 'Materialize' button is pressed, the
CGI program invokes a stand-alone VD Construction application with the selected Driver class as the main program running on our server machine. The application materializes a Virtual Document as HTML files in the system directory specified in the configuration file dir.setup. Any errors detected in executing the application are reported by the CGI program in HTML format.

![Figure 5.4: The Form for Viewing a VD Specification](image)

Similarly, when the hyperlink associated with the 'View VD Specification' operation is activated, a CGI program is invoked to dynamically generate a form for selecting a VD Specification file to view (see Figure 5.4). The names of all currently stored VD Specifications are collected to build a list of select options in an HTML form. When one of the VD names is selected and the 'View' button is pressed, the CGI program formats the chosen file into HTML format and then sends it to the the browser.

Lastly, to view the Content of a materialized Virtual Document, the associated hyperlink is activated to invoke a CGI program. The program generates on-the-fly an HTML menu page (Figure 5.5) with a list of hyperlinks pointing to the root entry point of each Virtual Document in the system. Users can then click on any of these links to start browsing its content.
5.4 Stand-alone Application

The stand-alone application consists of the two programs introduced when we described the CGI interface, namely the VD Specification Translator and the VD Construction application. These programs run locally once they are installed on the user’s machine. To use the VD Specification Translator, the following command can be issued:

```
vd.exe vdspec-file javasource-dir [err_file]
```

where `vdspec-file` is the name of the source VD specification file to be translated, `javasource-dir` is the target directory of the Java source code to be generated, and `err_file` is the name of an optional file for the output of error messages produced in parsing the VD Specification (stderr is used if this argument is omitted).

If translation succeeds, the Java source file `vd_name.java` for the Virtual Document `vd_name` is generated in the given directory. It can then be compiled with other static classes of the VD Construction application to produce the Driver class `vd_name.class` for this particular Virtual Document.

To materialize the Virtual Document, the Java interpreter can be invoked with the Driver class supplied as the main program and the classpath set to include all classes in the VD Construction application. The Virtual Document is materialized as a set of HTML pages in the directory or directories specified in the configuration file `dir.setup`. They can then be viewed by starting a Web browser.
Chapter 6

Summary, Conclusions, and Future Extensions

6.1 Summary

In this thesis we proposed a Virtual Document model of personal Web Space for dynamically gathering information of interest from the Web and building customized, integrated views over the collected information for personal use.

We first presented the three high-level abstractions used in the model: result-sets, views, and VD Graphs. Result-sets are used to model the sets of information to be retrieved from the WWW by using a database query-like approach; views, which are further classified into embedded views and page views, are used to model hypertext structures for presentation and navigation of result-sets; and a VD Graph is used to model the structure for organizing and connecting views into an integrated hypertext network that serves as the user's personal Web space. We described WebSQL's relational model of the WWW upon which our result-set sub-model is built, and the support of result-set operations for deriving new result-sets. Moreover, we explained the conceptual framework defined by the view sub-model for construction of new view types. We showed the Virtual Document model is extensible and tailorable, as it allows developers to define application-specific result-set operations and view types.

We then presented the VD Specification Language designed for the proposed model.
The result-set operations and the view types supported in our prototype were explained. To illustrate the specification language, we presented a comprehensive list of examples of Virtual Documents and their specifications.

In addition, we demonstrated the reusability of the model by showing the four applications: (1) querying multiple index servers, (2) building customized global views in a Web site, (3) sharing views in a workgroups environment, and (4) organizing information into personal navigation spaces. We also used two practical examples, namely a query report on data warehousing and a web-team library, to further illustrate how Virtual Documents can be utilized to model personal Web spaces.

Furthermore, we explained the translation of a VD Specification to Java and the materialization of a Virtual Document in the form of HTML pages, as implemented in the system components VD Specification Translator and VD Construction application respectively. We described the two interfaces supported in our prototype: a CGI interface connected to the VD Manager HomePage and a stand-alone application.

### 6.2 Conclusions

We find that the Virtual Document model provides a simple, flexible mechanism to build customized, integrated views of dynamic Web-based information collections. One challenging issue in supporting personal Web spaces is the problem of providing a high level of personalization while at the same time reducing the complexity of specifying user's preferences. Our strategy is to separate the high-level specification of a customized Virtual Document from the design and programming of the underlying components that support all Virtual Document constructions in a particular implementation of the model. Developers build application-specific components that satisfy the requirements for an application while VD designers author specifications based on an easy-to-use model using these specially designed, pre-built components. The trade-off in this decision is that not every hypertext structure can be used in specifying a customized Virtual Document — the choice of a presentation and navigation structure for a result-set is limited by the available options of view types supported by the underlying implementation, and the
choice of an organization structure for integrating views is restricted by the VD Graph structure imposed. Nevertheless, we believe this is a necessary compromise to simplify the process and to enforce a standard, easy-to-maintain structure in supporting personal Web space applications. Hypertextual view or document generation languages that support more general hypertext representations of Web-based information usually are much more difficult to use. Those languages, if they can interface with Java, are more appropriately used as a tool in the underlying component programming layer to facilitate building of new view types from scratch. In addition, despite the expressiveness of those languages, they cannot replace Java in supporting view construction, as they are basically special-purpose languages, lacking the full power of general-purpose programming languages. An object-oriented language like Java is useful in building personal Web space applications when mappings of information to hypertext representations require more demanding processing logic, and the support of reuse enables fast development of new representations.

Moreover, the approach we adopted aims to give the users control over where in the Web to look for information of interest, what criteria are used to select or filter information, how the collected results are combined for analysis, and what hypertext representations are used to facilitate navigation and organization of the collected information. This differs from approaches that attempt to avoid user involvement and automate as much as possible. For example, some systems guess what information is most suitable for the user, mechanically apply a built-in method of integrating the information, and/or automatically organize the information into a system-determined structure, which could be the result of some built-in categorization algorithm. The problem with these approaches is that the machine often does not produce what the users want. Clearly, different applications may have different requirements, and no general algorithm can address all diverse needs. In this regard, the Virtual Document model is especially suitable for building customized personal Web space applications. Further research is required to determine possible methods of integrating the two approaches to make the optimal use of both.

Furthermore, the Virtual Document model presented in the previous chapters can be seen as a skeletal design of personal Web spaces that provides useful support for
dynamically collecting information of interest from the Web and building customized, integrated views over them. There is much room for research and development to further enhance its functionalities and user-friendliness. In the following section, we explore some possibilities in supporting personal Web spaces and suggest future extensions in certain areas.

6.3 Future Extensions

Suggestions for future work are listed and discussed below.

1. One area requiring further research concerns the support of transformation functions, mentioned in Section 2.4.1 where we introduce conceptual primitives for constructing view types. A transformation function is a function which processes the contents of the Webpages referred to by a given subset of URLs in a result-set to generate a new set of Webpages for representing them in the view. We refer to the new Webpages formed as function nodes. Transformation functions and function nodes are used in defining representative nodes (Definition 2.11) for supporting information reuse. A representative node can either contain actual URL references to the original Webpages or function nodes that replace the original Webpages.

To facilitate implementation of transformation functions, existing tools can be used. One possibility is WebMethods’ WIDL and the Web Interface Toolkit [46], which relies on XML-compliant tags [43] for building Java objects that can be directly used from applications. It has a document object library and built-in HTML/XML parser which superimposes a Document Object Model over HTML/XML documents. For example, a specific element in an HTML table can be referenced by the expression doc.mytable.tr[1].td[0]. Document files are effectively converted to self-describing data containers for automated access. Moreover, the product supports very flexible ways (e.g., indexing, pattern matching) of accessing objects, object attributes, and data structures in an object repository built from Web data. The other possible option in this regard is WebOQL [2], a new Web data manipulation language developed by our group. In WebOQL, documents are represented by two
data structures: hypertrees and webs; both of these structures can be manipulated using WebOQL and created as a result of a query. A hypertree can be built from an HTML file using a general HTML wrapper, whereas sets of related hypertrees are collected into webs. The language offers many features for querying internal structures of documents and restructuring documents. Though more complex, WebOQL is more powerful than WIDL in areas like automatic navigation via links and direct generation of new pages.

2. Indexing and searching can also be supported as an optional service. In generating a Virtual Document, an index can be built over the Virtual Document pages and the referenced Webpages in the source Web. A special view type can be designed to provide an interface for accessing a search program that consults the index. Another possibility is to support domain search over a set of personal Web spaces. We can treat each personal Web space as a subject domain and provide an external search tool that enables users to specify a (VD name, keyword) pair as a query, and the index for the corresponding Virtual Document will be consulted to return answers in that subject domain.

3. To effectively support "stand-alone" Web sites, we can provide an option for users to cache source Webpages referenced by the URLs captured in a Virtual Document.

4. Sharing of VD Specifications is another useful facility. We can allow users to specify a Virtual Document as a view in a VD Specification. Effectively, users can access the information collections in the Virtual Document from the point of view of the author of the Document. Furthermore, IMPORT and EXPORT support in programming languages can be used for sharing result-set and view definitions, but these will involve more complicated management and maintenance issues.

5. Currently, we assume that updates of Virtual Documents are initiated manually upon users' materialization requests. Scheduler programs can easily be built to trigger automatic, periodic updates and regularly generate snapshots. It may be impractical in many situations to support automatic updates that reflect all underlying changes in the Web affecting a Virtual Document to ensure view consistency.
Further study is required to see if incremental update is feasible. Another possible service is to generate a report on changes since the last version of a Virtual Document, and to let users review upon access.

6. There are several other facilities that can be supported as special view types or new primitives in view types. For example, a sitemap can be generated for a Virtual Document as a type of meta view. If indexing is supported, the index built over Webpages in a Virtual Document can be generated as a Keyword-Index view. In addition, as mentioned earlier when we talked about indexing and searching, a special Search-Interface view can be created for accessing the search engine that consults the index built. On the other hand, we can allow users to annotate the source Webpages referenced by a Virtual Document; annotations for a Webpage can be supported as an annotation node, a new primitive for construction of view types.

7. New view types can be designed for supporting the entity-collection perspective of result-sets that we introduced in the result-set sub-model. Readers can refer to the examples of entity-collections given in Section 2.4.4.

8. To further facilitate specification of a Virtual Document, a GUI tool can be designed so that users can easily specify the parameters for the components in a Virtual Document and assemble their own personal Web space.

9. Another problem that needs to be addressed is the maintenance of the WebSQL queries as the Web evolves and the changes are outside our control.

In particular, (1) supporting information reuse and (2) adding search facility are most important for improving services supplied to end-users. Another direction is to provide better support for developers in the design and implementation of application-specific components.
Appendix A

VD Specification Language

In this appendix, we describe the syntax and semantics of the VD Specification Language designed for the Virtual Document model. They are outlined in Section A.1 and Section A.2 respectively.

A.1 Syntax

One special feature of our design is that we chose to provide a section in the VD specification for explicitly specifying the high-level architecture of the hypertext network, leaving the details of the contents and internal structures of individual nodes to be specified in subsequent sections. This is different from hypertext-generation languages such as Penelope that specify a network of pages on a pure page template basis, where users have to read through different page templates or refer to external diagrams or documentations to gather the interconnections and overall structure.

Figure A.1- A.3 show the grammar for the VD Specification Language. The strings enclosed in single quotes represent terminals, the ordinary strings without quotes are non-terminals, the words in italics represent special lexical tokens whose structures are described after the grammar, the vertical bar symbol | denotes disjunction, braces { } mean that an item can occur zero or more times, brackets [ ] mean that an item is optional, and parentheses ( ) mean grouping. The start symbol is vd_specification.
Figure A.1: Syntax of VD Specification Language (part 1 of 3)
hypertext_section ::= 'HYPERTEXT'
                graph
                'END_HYPERTEXT'

graph ::= simple_graph
       | composite_graph

simple_graph ::= 'SIMPLE'
                view_name
                'END_SIMPLE'

composite_graph ::= 'COMPOSITE'
                   root_stat
                   [other_nodes_stat]
                   [embedded_view_stats]
                   [edge_stats]
                'END_COMPOSITE'

root_stat ::= 'ROOT' root_name ';'
root_name ::= node_template_name
node_template_name ::= ID
other_nodes_stat ::= 'NODE' node_name {', 'node_name} ';'
node_name ::= node_template_name
embedded_view_stats ::= 'EMBED' view_name {', 'view_name} 'IN' node_template_name ';' 
edge_stats ::= edge_stat {edge_stat}
edge_stat ::= 'EDGE' edge_name edge_spec ';' 
edge_name ::= ID
edge_spec ::= containment_edge_spec
            | cross_ref_edge_spec
containment_edge_spec ::= 'CONTAINMENT' [edge_property] source_dest_spec
edge_property ::= 'SINGLE_ANCHOR'
                 | 'INDEX'
cross_ref_edge_spec ::= 'CROSS_REF' ref_name source_dest_spec
ref_name ::= STRING
source_dest_spec ::= 
                 | 'NODE_NODE' '(' source_node ',' dest_node ')' 
                 | 'NODE_VIEW' '(' source_node ',' dest_view ')' 
                 | 'VIEW_NODE' '(' source_view ',' dest_node ')' 
                 | 'VIEW_VIEW' '(' source_view ',' dest_view ')' 
source_node ::= node_template_name
dest_node ::= node_template_name
source_view ::= view_name
dest_view ::= view_name

Figure A.2: Syntax of VD Specification Language (part 2 of 3)
APPENDIX A. VD SPECIFICATION LANGUAGE

node_template_sections ::= node_template_section { node_template_section }
node_template_section ::= 
  'NODE TEMPLATE' node_template_name
  node_template_header
  node_template_body
  'END_NODE_TEMPLATE'
node_template_header ::= 'TITLE' title_string ';
title_string ::= STRING
node_template_body ::= document_section { document_section }
document_section ::= 
  'SECTION'
  section_heading
  section_content
  'END_SECTION'
section_heading ::= 'HEADING' heading_type heading_text ';
heading_type ::= 
  'H1'
  | 'H2'
  | 'H3'
  | 'H4'
  | 'H5'
  | 'H6'
heading_text ::= STRING
section_content ::= 
  embedded_view_spec
  | anchor_for_edge_spec
embedded_view_spec ::= 'TUPLE TABLE' view_name ';
anchor_for_edge_spec ::= 'ANCHOR' anchor_name label_spec ';
anchor_name ::= ID
label_spec ::= 
  'LABEL' label_string
  | 'SYSGEN LABEL'
label_string ::= STRING

Special Lexical Tokens:
1. ID denotes the set of character sequences consisting of a letter followed by zero or more letters, digits or '‐'.
2. STRING denotes the set of sequences of zero or more characters enclosed in double quotes.
3. INTEGER denotes the set of sequences of one or more digits.

Figure A.3: Syntax of VD Specification Language (part 3 of 3)
A VD specification consists of a sequence of five parts: (1) a search section, (2) an optional operations section, (3) a views section, (4) a hypertext section, followed by (5) an optional sequence of node template sections. In what follows, we describe the main points of the syntax for each constituent section.

The search section contains a sequence of base result-set declarations. Each base result-set is declared with a name, a maximum size, and a Web query string.

The optional operations section is made up of a sequence of operation statements. Each statement defines a derived result-set by assigning the result of an operation or operations performed on result-sets to this new set. The current prototype supports the following operations:

1. UNION('result_set_name', 'result_set_name')
2. DIFFERENCE('result_set_name', 'result_set_name')
3. PROJECTION('result_set_name', attr_name {', attr_name}')
4. INTERSECTION('result_set_name', 'result_set_name')
5. COMPARISON('result_set_name', 'new_attr_str', 'result_set_name', 'new_attr_str')
6. GROUP_BY_FIELD('result_set_name', 'attr_name')

where result_set_name and attr_name are as defined in Figure A.1 and new_attr_str is a STRING.

In the views section, each view is declared with a name, a view type chosen for representing a result-set, the name of that result-set, and an optional list of attributes in the result-set that are the required parameters of the view type. 'NULL' is supplied in place of an attribute value when the attribute is not used. We have implemented four view types:

1. TUPLE_LIST result_set_name
2. DOCUMENT result_set_name('url_type_attr', title_attr')
3. DOCUMENT_LIST result_set_name('url_type_attr', title_attr')
4. DOCUMENT-TREE result.set_name('url.attr', 'title.attr', 'path.attr')

where

- url_type_attr ::= table_var."url" | table_var."base" | table_var."href"
- url_attr ::= table_var."url"
- title_attr ::= table_var."title" | 'NULL'
- path_attr ::= table_var."path"
- table_var ::= ID as defined in Figure A.3.

This is followed by the hypertext section, in which the high-level hypertext network structure of the Virtual Document is specified as a VD Graph. A VD Graph can be simple or composite. To specify a Simple VD Graph, the name of the only view in the graph is given within a simple_graph construct. On the other hand, a Composite VD Graph is specified as a composite_graph construct which consists of a sequence of four parts: (1) a root statement, (2) other nodes statement (optional), (3) embedded view statements (optional), and (4) edge statements (optional). Both the embedded view statements and edge statements are optional because a Composite VD Graph may contain only embedded views or page views. The names of the node templates for the root and all other nodes (aggregate nodes) in the graph are specified in the root statement and the other nodes statement respectively. Embedded views in each node template are then specified by giving a list of view names and a node template name in an embedded view statement. These are followed by an optional sequence of edge statements, where a name and an edge specifier are specified for each edge. Two types of edges, namely containment edge and cross-reference edge, are supported in the current prototype, and a different edge specifier is supported for each edge type. To specify a containment edge, one of its two properties can be selected: SINGLE_ANCHOR for a simple edge or INDEX for an index edge; and the node/view names of the edge's source and destination are supplied. As for a cross_reference edge, a reference string is specified as the hyperlink label for the edge, and then the node/view names of the edge's source and destination are specified.
The last part of a VD specification is an optional sequence of node template sections. Each node template is given a name and is divided into two parts: a header and a body. A title string is supplied for instantiating the title in the header. On the other hand, the body consists of a sequence of sections, which are composed of two parts: a heading and content. A heading is defined with a type that is equivalent to the HTML heading style (H1 to H6) and also a text string. The content part can contain an embedded view specifier or an anchor specifier for an edge. An embedded view name is supplied in the former whereas an anchor name and a label specifier is given in the latter. Labels can be specified in either of the two ways: a label string followed by the reserved word 'LABEL' or, just the reserved word SYSGEN_LABEL.

A.2 Semantics

We have already described the semantics of the Virtual Document model in Chapter 2. In this section, we further explain the semantics of details of the concrete objects and show which objects in a VD specification corresponds to which objects in the semantics of the Virtual Document model.

The search section is for specifying the base of a Virtual Document (Definition 2.4). Each constituent base result-set (Definition 2.3) is defined by a Web query supplied (Definition 2.2) and given a maximum size to restrict the number of query answers to be included. The latter is for practical performance concern, as a Web query may be costly if the number of Web accesses made to retrieve documents is large.

The operations section is for specifying the additional result-sets to be derived from the base that are required for the construction of the Virtual Document. Each derived result-set is the result of applying an operation or operations on result-sets. The four operations union, difference, projection, and intersection have the standard semantics in relational algebra whereas the semantics of comparison and group_by_field are given as follows;

- COMPARISON( rs1, str1, rs2, str2) :-
  this function takes as inputs two result-sets rs1 and rs2, and two descriptive label
strings str1 and str2 for rs1 and rs2 respectively; its pre-condition is that rs1 and rs2 must be schema-compatible. And it returns a new complex result-set with two attributes: str1:rs1 and str2:rs2. The returned instance contains a tuple of two attributes: one consists of the set of tuples in rs1 and the other consists of the set of tuples in rs2, with the ordering of tuples in the two result-sets preserved. Common tuples in rs1 and rs2 can be located. We call the returned result-set a comparison result-set.

- GROUP_BY_FIELD( rs, attr ) :-
  this function takes as inputs a result-set rs and the name of an attribute of rs; its pre-condition is that the supplied attribute name must be one of the attributes defined in the schema of rs. And it partitions rs according to the value of attr such that tuples in rs sharing the same value of attr are grouped into a sub-result-set. The schema of such sub-result-set equals that of rs. A new result-set which consists of all sub-result-set partitions of rs is then returned. We call the returned result-set a grouped result-set.

All views (Definition 2.7) for representing base or derived result-sets in the Virtual Document are specified in the views section. The semantics of the four view types (Definition 2.8) supported in the prototype are given below. View types can belong to either the class of embedded views (Definition 2.9) or the class of page views (Definition 2.13).

1. TUPLE_LIST rs :-
  this is a view type that belongs to the class of embedded views. The given result-set rs is represented by an HTML table with the original ordering of tuples preserved. If rs is a grouped result-set, the table is divided into sub-tables, one for each sub-result-set. If rs is a comparison result-set, the table is nested with two inner tables as its attributes.

2. DOCUMENT rs( url, title ) :-
  this is a view type that belongs to the class of page views. Only simple result-sets and grouped result-sets are accepted. The given result-set rs is viewed as a set of documents uniquely identified by the given url attribute of rs. An optional title
The attribute of rs is taken as input to provide more descriptive information about each document. In addition, a non-empty set of exit points may be supplied in cross-reference edges specifications for constructing the view. This view is represented as hypertext by horizontally dividing the top frame of the browser window into two child frames. The upper frame is the context frame for showing an index of the documents in the view, and the lower one is the document-display frame for showing the current, active document chosen from the document-set. If rs is a grouped result-set, the index is organized into sub-groups. The top frame-composition node is the entry point of the view. If a non-empty set of exit points is given, the frame hierarchy will be extended to include an extra frame that displays the exit node containing all exit points. The hypertext structure of the view is depicted in Figure A.4.

Figure A.4: The Hypertext Structure for a DocumentView

3. DOCUMENT_LIST rs( url, title) :-
   this is a view type that belongs to the class of page views. Only simple result-sets and grouped result-sets are accepted. The given result-set rs is viewed as a list of documents uniquely identified by the given url attribute of rs. An optional title attribute of rs is taken as input to provide more descriptive information about each document. In addition, a non-empty set of exit points may be supplied in cross-reference edges specifications for constructing the view. A “double-linked list” navigational structure is imposed where one navigational node is generated in
the view for every document such that the forward links of the navigational nodes follow the ordering of the documents in rs. Backward links support navigation in the reverse direction. This view is represented as hypertext by using two browser windows: one for displaying an index of all documents whose ordering in rs is preserved and the other for displaying the current, active navigational node. The index node is displayed in the original browser window where the view is loaded. Each index item in this node contains a hyperlink whose destination is the navigational node of the document corresponding to the item; and the target frame is the top frame of a new browser window. This new top frame is horizontally divided into two child frames. The upper frame is for showing the context node that contains all structural links of this navigational node, and the lower one is for showing the current document. If rs is a grouped result-set, a separate “double-linked-list” is formed for each sub-result-set of documents and the index is organized into sub-groups. All the navigational nodes are made entry points of the view. In addition, the index node is the root entry point if there is no exit point; otherwise a frame-composition node with a frame link pointing to the index node is the root entry point of the view. The other frame link of the frame-composition node points to an exit node in the latter case. The hypertext structure of the view is depicted in Figure A.5.

Figure A.5: The Hypertext Structure for a DocumentListView
4. DOCUMENT TREE rs( url, title, path ) :-
this is a view type that belongs to the class of page views. Only simple result-sets are accepted. The given result-set rs is viewed as a tree of documents uniquely identified by the given url attribute of rs. The input path attribute is used to reconstruct the underlying tree structure for the documents, whereas an optional title attribute of rs is taken as input to provide more descriptive information about each document. In addition, a non-empty set of exit points may be supplied in cross-reference edges specifications for constructing the view. If the given path attribute values do not represent a tree, exception is thrown and error is reported. A “double-linked tree” navigational structure is created where one navigational node is generated in the view for every document such that there is a corresponding downward link between navigational nodes for every directed edge between documents in the tree. Upward links support navigation in the reverse direction. Like DOCUMENT_LIST, this view is represented as hypertext by using two browser windows: one for displaying an index of all documents whose ordering in rs is preserved and the other for displaying the current, active navigational node. The index node is displayed in the original browser window where the view is loaded, and it is designed to represent the document tree as a hierarchy of items. Each index item in this node contains a hyperlink whose destination is the navigational node of the document corresponding to the item; and the target frame is the top frame of a new browser window. Frame composition in this view is also similar to that in DOCUMENT_LIST. The top frame of the new browser window for displaying a navigational node is horizontally divided into two child frames. The upper frame is for showing the context node that contains all structural links of this navigational node, and the lower one is for showing the current document. All the navigational nodes are made entry points of the view. In addition, the index node is the root entry point of the view if there is no exit point; otherwise a frame-composition node with a frame link pointing to the index node is the root entry point of the view. The other frame link of the frame-composition node points to an exit node in the latter case. The hypertext structure of the view is depicted in Figure A.6.
The hypertext section allows designers to specify the high-level hypertext network structure of the Virtual Document as a VD Graph (Definition 2.17), which can either be simple or composite. For Simple VD Graph (Definition 2.15), only the view needs to be given: if it is an embedded view, a simple node will be automatically generated by the system to incorporate it into the only page of the graph; otherwise, if it is a page view, this view will be the graph. As for Composite VD Graph (Definition 2.16), the syntax of its structure in terms of root, other nodes, embedded views and edges as described in the previous section is self-explanatory. In what follows, the semantics of the details of edges and node templates which have not been covered is described.

We support two types of edges in the prototype: containment edges and cross-reference edges, where the latter is used as an example of special edges in a VD Graph called tubes (Section 2.5.1). A containment edge can be a simple edge or an index edge. A simple edge is represented by a single anchor whose label is specified in the node template of the edge’s source node, and its destination is either another node or the root entry point of a page view. On the other hand, an index edge is represented by multiple
anchors which are automatically generated, one for each entry point to directly access a
document element in the destination page view. All cross-reference edges are represented
by single anchors whose labels are specified as reference strings in the edge statements.

Node template sections are used for specifying the contents and internal structures of
the sections nodes in aggregate nodes (Section 2.5.2). The syntax of each sections node is
self-explanatory. Each section body either contains an embedded view or a containment
edge. In the latter case, either an anchor with a given label string represents a simple
edge or the system generates a list of anchors for an index edge.

Having explained the semantics of the concrete objects in the various
sections of a VD Specification, we summarize by showing which objects in the speci-
fication corresponds to which objects in the semantics of the Virtual Document model
(Definition 2.1).

Given

1. the Web query language WebSQL,

2. the set of result-set operations $\mathcal{OP}_s =$

   \{$\text{UNION, INTERSECTION, DIFFERENCE, PROJECTION, COMPARISON, GROUP BY, FIELD}$\},

3. the set of embedded view types $\mathcal{VT}_e =$ \{$\text{TUPLELIST}$\}, and

4. the set of page view types $\mathcal{VT}_p =$

   \{$\text{DOCUMENT, DOCUMENTLIST, DOCUMENTTREE}$\},

the mapping of the syntax of the VD Specification Language to the the semantics of
the Virtual Document model is as follows:
SYNTAX

VD vd
SEARCH
RESULT_SET r₁ ... qᵢ
END.SEARCH

OPERATIONS
rⱼ = opⱼ(...
...
END.OPERATIONS

VIEWS
VIEW vₖ vtₖ rsₖ ... 
VIEW vᵢ vtᵢ rsᵢ ...
...
END.VIEWS

For a Simple VD Graph:

HYPERTEXT
vₘ
END.HYPERTEXT

For a Composite VD Graph:

HYPERTEXT
ROOT nᵣoot ; 
NODE n₁, ..., nᵢ, ..., nₙ ; 
EMBED v₁, ..., vᵢ IN nⱼ ;
...
EDGE eₖ CONTAINMENT ... (sₖ, dₖ),
...
EDGE eᵢ CROSS-REF ... (sᵢ, dᵢ),
...
END.HYPERTEXT

NODE TEMPLATE nᵢ
...
END.NODE TEMPLATE
...
END.VD

SEMANTICS

vd = (RSₜ, RSₜ, N, Vₜ, Vₚ, Embed, Contain, CrossRef)

rᵢ ∈ RSₜ where qᵢ in WebSQL

rⱼ ∈ RSₜ where opⱼ ∈ OPₜ,

vₖ ∈ Vₜ where vtₖ ∈ VTₜ and rsₖ ∈ RSₜ ∪ RSₜ

vᵢ ∈ Vₚ where vtᵢ ∈ VTₚ and rsᵢ ∈ RSₜ ∪ RSₜ

This ends our brief description of the VD Specification Language. For illustrative examples, readers can refer to Chapter 3.
Bibliography


