DOCBASE - A DATABASE ENVIRONMENT FOR STRUCTURED DOCUMENTS

by
Arijit Sengupta

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

December, 1997

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Professor Dirk Van Gucht
(Principal Advisor)

Professor Edward Robertson

Doctoral
Committee

Professor Andrew Dillon

December 4, 1997.

Professor David Leake

To my mother and father.

Acknowledgements

I would like to thank my advisor Professor Dirk Van Gucht for his help and encouragement, without which this research would never have started. His enthusiasm always encouraged me even when things were difficult. I am grateful for all the time (sometimes even during off-hours and weekends) he spent with me in spite of his hectic schedule. He would always invite me to work with his co-researchers during some short but very useful research sessions. I would also like to thank Professor Andrew Dillon for his constant encouragement and advice. I consider my decision to take his courses at SLIS one of the most important milestones in my academic life. He was the person who showed me that my work has significant effect on the human side of system design, and without his help, the visualization part of this research would have been quite incomplete. I would also like to thank Professor Edward Robertson, who generously lended me source materials – often without my asking. The database lab was always like a home to me - with Dirk and Ed like two very close family members. Of course, this family would not be complete without the help and support of Deepa, Manoj, Munish, Ramu, Sudhir, and Vijay. Special thanks goes to Memo, who proved to be a very dear friend and was always willing to help; he never said no when I asked him to review a paper, give me a ride, or simply cheer me up during the hard days. I would also like to thank the faculty and staff of the Computer Science Department for their support.

Special goes to two students who worked with me in the development and in the testing of the Query-By-Template system. I was fortunate to have the help of Xiaojian Kang who helped fix and code many aspect of the interface, and Shawn Morgan who also helped build parts of the system. Many thanks to all the twenty participants

of the usability analysis of the interface. This work would not be complete without their help.

I would also like to thank my parents and my brother - without whose encouragement this would never have been possible. Although on the other side of the world, they have always been a source of great support. Special thanks to my spiritual teachers, the late Sri Swami Rama and Pandit Rajmani Tigunait, who showed me that material was not the only aim of life. I would like to thank my wife Anjali for her love and support in my work and my life. She was always supportive in spite of my late hours and hectic work schedules, and she always took the time to proofread my papers and help out in every way she could. I don't think I could have crossed this hurdle without her. Last but not the least, I thank my in-laws, Bill and Jane, who made me feel right at home and never let me realize that my real parents were thousands of miles away. Ragani was a perfect older sister; her visits were always times of fun.

I especially would like to acknowledge ArborText for their support and co-operation during the preparation of this dissertation. This document is prepared using ArborText's Adept Series products. The preparation was much simplified by these tools - and I recommend this product to any SGML author. I would also like to thank Open Text Corporation for their support with with the Pat software that was used extensively in this research.

This research was partially supported by U.S. Dept. of Education award P200A502367 and NSF Research and Infrastructure grant, NSF CDA-9303189

Abstract

Standard Generalized Markup Language (SGML) has been widely accepted as a standard for document representation. The strength of SGML lies in the fact that it embeds logical structural information in documents while preserving a human-readable form. This structural information in SGML documents allows processing of these documents using database techniques. SGML facilitates this goal by providing a conceptual modeling tool for collections of documents using a document type definition (DTD) and by allowing query processing beyond the classic keyword-based searches of traditional IR systems.

We use these observations about SGML as the design principles for developing and implementing a structured document database system. The key difference of our approach from other similar approaches is that the design and implementation remain entirely within the context of the SGML framework. We achieve this by using SGML as the modeling tool of the database instances, by generating SGML documents as outputs of the queries, and also by using SGML for expressing queries.

DocBase is a prototype research system that implements most of the querying features of a document database. We use SGML as the model for structured document databases, with the database schema represented using a DTD and SGML documents as instances of this schema. We propose an extended form of relational calculus and equivalent SQL-like and visual query languages for posing queries. DocBase implements an infrastructure for processing these queries by leaving the documents intact, and using special index structures and access methods over these structures.

Recognizing the importance of users in the design of systems for document retrieval, we propose a visual query formulation method that uses the principle of

familiarity to make the querying process easier and more satisfying for users. We show that even at the simplest level, this method is no less efficient or accurate than the traditional form-based query formulation, but is significantly more satisfying.

Chair: Dirk Van Gucht, Associate Professor, Computer Science Department

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The bulk of the useful information today comes in the form of documents. Newspapers, magazines, books, novels, technical manuals, legal documents are just a few types of documents that we use almost every day. Lexicographically, the term "document" refers to writings on some material substance such as paper. However, with the advent of computers and automation, documents are no longer restricted to paper and other "hard copy" media. Instead, documents are prepared and stored electronically, with computers used for displaying, formatting, printing, searching and editing. To facilitate these tasks, software vendors have designed a number of powerful word processing applications that can be used to format, typeset, edit, print and publish such documents. In most cases, however, the final publishing medium is still paper, and the computer software is primarily used for adding formatting information directed towards printed output. Such systems do provide limited ability to search documents for words and phrases in the entire document sequentially. However, for large document collections, these types of searches often prove to be too slow and restricted, and better retrieval techniques become necessary.

Recently, the WWW(World Wide Web) has significantly changed the concept of document preparation and distribution. Documents are now being prepared less with specific formatting information and more with structural information in the form of "tags". It is now possible for different viewers on different platforms to generate formatting information "on the fly", based on the capabilities of the platforms and the computer displays. These tags normally use the same encoding as the rest of the document, so that the document is readable without any formatting information and is easily interchangeable between platforms. The concept of these tags first arose with HTML (*HyperText Markup Language*) [BLC95] — the language for the World Wide

Web, and SGML (*Standard Generalized Markup Language*) [ISO86] — a generalized language for creating documents with arbitrary structure. The main emphasis of this dissertation is on SGML, although all the concepts will be fully applicable to HTML, since HTML can be considered to be an application of SGML [BLC95].

The primary goal of SGML was to create documents that are freely interchangeable between multiple systems and platforms. In addition to making documents portable, the SGML tags introduce structural information in the documents, information that can be used by applications for purposes other than formatting. As we will describe in Chapter 2, the tags can be used as meta-data for database functionality. One objective of this research is to use effectively this meta-data information to give document repositories the ability to query information contained in these documents in a manner that is currently possible only with standard database models.

As an interesting illustration of the problem, let us consider an electronic document collection, such as the ChadWyck-Healey English poetry database [Cha94], a collection of over 160,000 poems from the Anglo-Saxon period to the late 19th century. If this collection were on paper, at an average of one poem per page, this document will have 160,000 pages. This number does not take into account the table of contents and the index, without which the collection will be virtually useless. At about 500 pages per volume, this will mean about 80 volumes – enough to fill up two bookshelves! On the other hand, this whole information can be easily accommodated in a single CD-ROM, which is several orders smaller in physical size than the paper equivalent. The advances in storage medium technology have ensured easy and convenient storage of electronic documents, but storage is only half the problem. Efficient storage of large amounts of data is not of much use if the data cannot be efficiently retrieved from the storage. A popular method for extracting portion of information sources is using searches involving boolean combination of *search keywords*. This problem of "information retrieval" [Sal91] forms the basis for research in automated text extraction from a repository of documents. Information retrieval is virtually a sub-discipline in its own right within Information Science.

In its simplest form, primitive information retrieval techniques extract lines containing specified keywords from a document. It is often useful to restrict the searches

to specific portions of the document, such as in the titles or poet names in the case of the poetry collection. As an illustration, in the poetry database, the word "love" returns over 270,000 matches, but restricting it within poem title immediately reduces the number of matches to around 5,000. In order to successfully introduce such granularity into the document, one needs to demarcate important regions of the document with additional information. One common method for augmenting documents with such information is "tagging", and documents produced in this method are commonly known as tagged documents or structured documents. In the next section, we will discuss the process of tagging in more detail. Tagged documents can be used not only for simple keyword searches as before but also to perform searches that are impossible without the structural information. With the help of tags, one can now answer questions similar to ones commonly asked in the context of relational databases. Some common types of questions are:

- *Simple selections.* These are queries involving searches for text strings in various regions of the database (*e.g.*, find all the poems that contain the word "love" in the poem title).

- *Projections.* These are queries that involve extraction of specific components of documents. (*e.g.,* extract the poem titles and authors only of all poems in the database).

- *Quantification.* These are queries that involve quantifiers such as "all," "every," or "none" (*e.g.*, find the period in which all poems had the word "love" in their titles).

- *Joins.* These are queries in which multiple components of documents are combined based on one or more regions (*e.g.*, find the names of poets who have at least one common poem title).

- *Negation.* These are queries in which a search condition is negated (*e.g.*, find the poems that do not have the word "love" in the title).

- *Counting.* These are queries that involve computing the number of matched results, possibly based on certain conditions (*e.g.*, how many Shakespeare poems are there in the collection?)

- *Grouping and ordering.* These are queries in which the results need to be grouped and ordered based on certain conditions (*e.g.*, list the different periods of poetry in the collection, in ascending order of the number of poems in each period).

- *Nested queries.* These are queries in which a query includes another query (or a subquery) as a search condition (*e.g.*, find the names of poets who never used the word "love" in the title of any of their poems).

Queries similar to the above are common in the case of relational database systems. If we could model the above poem collection using a relational database, we could answer all the queries. Unfortunately, current information retrieval systems that only perform keyword searches well, cannot answer all the above queries. However, systems that support structured text can be used to automatically extract answers to the above queries. In a later chapter (Chapter 3), we will discuss in detail current research efforts providing support for queries like the above in text database systems. Some of these methods involve conversion of the text into a standard complex-object system. This dissertation proposes an implementation method that can provide support for all these queries without the need for such conversion.

1.1 Problem Context and Description

In this section, we briefly describe the research problem covered in this dissertation and the basic concepts related to this problem. The primary goal of this research is to provide database functionality to document repositories. In order to achieve this, additional structural information needs to be added to documents. This makes it possible to pose complex queries involving text and structure like the examples above. In order for novice users to be able to easily formulate their searches in

the system, we need to use Human-Computer Interaction (HCI) techniques to make searching easy and effective. In this section, we introduce some of the developments in database systems, document processing, and HCI that are used as the basis of the current research.

1.1.1 Database Systems

Databases emerged as a major research area when the necessity of taking a disciplined approach for storage and retrieval of information became obvious. The evolution of database systems is marked by three generations of database systems and models. [Ull88]

First generation database systems included the hierarchical and network data models. These models were strongly influenced by the physical implementation of the data and used pointers and links for storage and retrieval. The drawback of this method was that one needed to know the internal representation of the link structure in order to pose queries on the data. Moreover, changes to the organization of the data needed major changes to the processing applications. The early IMS (Information Management System) [McG77] and its DL/1 language fall in this category.

Second generation database systems included the relational model [Cod70], which first introduced the concept of data independence. This makes the conceptual organization of the data independent of the way the data is internally stored and processed. In the relational model (discussed in detail in Section 2.2.1.1), the physical storage and index structures can be completely changed without effecting the conceptual data model and queries for data retrieval. The second generation also witnessed better theoretical foundations in database models and query languages and better visual query formulation using the QBE (Query By Example) query language [Zlo77]. The *Entity-Relationship (ER) Model* [Che76], also introduced during this generation, better supported conceptual modeling of data, from which the database schema could be conveniently generated. Although relational databases became the standard in database systems, the simplicity of the flat table model was often proving to be too restrictive to model complex structures without causing excessive fragmentation in

the data. [AHV95, Chapter 20]

Third generation database systems, consisting of Object-Oriented and Object-Relational database systems, accommodate complex structures in the data model, thus improving the expressive power of the model. However, the increased expressive power also implied an increased complexity of the query languages.

In addition to these prominent generations of database systems, some *specialized database systems* have been proposed to model text, multi-media, spatial and temporal data during the recent years. Although technically they can be categorized in the third generation of database systems, the use of these database systems in their specialized domains make them distinguishable from the various generations.

1.1.2 Document Processing

This dissertation focuses on processing of documents containing large amounts of text. We hinted earlier that meaningful queries can be performed on documents if they contain some structural information in addition to the actual text content. In this section, we discuss the basic concept of "structured documents" as well as information retrieval for both structured and non-structured documents.

1.1.2.1 Structured Documents

In this dissertation, when we refer to documents, we primarily mean documents in electronic form. The simplest type of electronic documents is plain text which contains only the natural language text of the document, with much restricted formatting and structural information. In addition to the text, these documents may only contain spacing and positioning constraints on the text to convey specialized meanings to the text. The main advantage of plain text documents is that they can be created on any platform without the use of any special software , and hence they are highly interchangeable. However, because of the lack of presentation and layout capabilities, plain text documents have limited use.

The advent of word processing and text formatting systems introduced "tagged" documents to substitute for plain text documents. Generically speaking, a tag is

simply some extra information embedded in the document using either a text editor or a word-processor. Word processors, such as Microsoft Word^{TM}, primarily use tags specific to the system, using encoding that only particular word processors can decipher. Text processing systems, such as roff [Oss76] and TEX [Knu86], use special codes that can be entered using a computer keyboard. After a document is created, it can be processed by software programs to replace the codes with presentation information for viewing on screen or printing on paper. Roughly speaking, we term documents with additional embedded information as *tagged or structured documents*. Tagged documents can be classified in the following two major classes based on the type of tagging involved:

- *Specific tagging.* In this type of tagging, the tags primarily have font, size and other formatting information and do not necessarily define any logical regions in the document. Examples of this type of document include word-processor documents and documents in roff, TEX and other related document preparation formats.

- *Generic tagging.* In this type of tagging, the tags do not specify any font or size information but are more general in nature. The primary purpose of these tags is to define logically distinct regions in the document such as chapters, sections, headings. These tags can be translated into formatting information by applications, based on the capabilities of the platform and the screen. Documents tagged using HTML and SGML are examples of this type of tagging.

Other than the two types of tags described above, there can be a mixed type of tagging, where both generic and specific tagging are involved. Tags can also be procedural, indicating some action to be performed where used. Documents in LATEX [Lam94] format contain logical tags such as sections and chapters as well as font, size and spacing tags. A few of the HTML tags can also be seen under this mixed category. In addition, tags can also *procedural*, used primarily for giving instructions to the processing application.

In the context of this dissertation, however, we will use the term *structured documents* to denote generically tagged documents – SGML documents in particular. We

will discuss the details of generic tagging in SGML in a later chapter (Chapter 2).

1.1.2.2 Information Retrieval

As mentioned earlier, efficient storage of large amounts of text does not solve the problem of effectively extracting information from them. Fortunately, research on the issue of extracting information from large volumes of text has uncovered techniques for "information retrieval" – we look at a few of these techniques in this section.

The most common method for searching information in a document repository is by using *boolean searches* [Sal91]. In this type of search, a number of keywords combined with boolean operators (such as "and", "or", "not") are specified, and the result consists of the documents that satisfy the given boolean expression. The problem with this type of search is that all matching documents in the resulting document set are given the same importance. To avoid this, one can use the *weighted keyword search*, in which the search items are assigned weights based on their importance, and the retrieved documents can be ordered by the most relevant to the least relevant based on the number of matches and the weights of the matched terms.

Although the complexity of simple keyword search in a document is only linear to the size of the document, for very large documents this complexity turns out to be considerably expensive. For example, a simple "grep" search [GNU92] for the word "tyger" in the Chadwyck-Healey poetry database mentioned earlier takes about 6 minutes (62.9 seconds system CPU time) running on a Sun Ultra Sparc 2 with 124MB of main memory. This seemingly abysmal performance is partially due to the fact that grep does not utilize the system resources intelligently, but scans the files one line at a time. Although operating systems are usually intelligent enough to cache one or more blocks of data even for single-line reads, most of the time is still in processing I/O from secondary storage and network.

The most common approach to avoid accessing the whole document repository for every search is to create indices based on the documents. The searching applications can use the indices first to determine exact location of the file in which matches could potentially be found, and retrieve data by directly accessing the document in that location. As an illustration, the same search as above using *Glimpse* [MW93], with

only a small index file on the same machine takes about 67 seconds (4.19 second system CPU time), an improvement of around a factor of 15 on CPU usage.

Note that this improvement is even more apparent if the search keyword appears less often. In the last example, the word "tyger" has about 400 occurrences in the database. If the same experiment is performed with the word "Casabianca" (which occurs only twice in the database), the sequential search takes about the same time as before, while the index search takes only about 2 seconds (0.19 second system CPU time). On the other hand, although a sequential search on the word "love" (which appears over 200,000 times) takes about the same time as before with the sequential search, it takes about 4 minutes (22.3 second system CPU time) with the index method. The reason for this is again because of the time spent in retrieving the results from the individual files, which is I/O intensive. Although this process involves building the indices which takes about 2 hours and uses about 7–8% of the size of the database, indices are only built once and can be used for all subsequent queries.

Creation of indices also requires special considerations based on the type of data to be indexed and types of queries to be supported. In any language, there are words that are used very frequently, but very rarely searched for (such as like "and", "of", "or", "but", "the" in English). While creating indices, these words make the size of the auxiliary index structures larger, thus affecting the search time. Such words, commonly referred to as *"stop" words*, are often ignored by indexers. While indexing, it is often useful not to create separate index entries for all forms of the same word (such as various verb forms, tenses and numbers) and only include the root word in the indices. Some advanced indexing mechanisms use various forms of linguistic analysis [SR90] and thesauruses to determine the important words for indexing.

In summary, most of these techniques use the full text of the documents to build smaller auxiliary structure that can be searched faster than the actual documents. Worldwide availability of these documents are now possible using the WWW, which in turn is developing the concept of *digital libraries* [Sch97] containing not only text, but also images, sound and other multimedia objects. In Chapter 3, we discuss in more detail recent research on information retrieval techniques for document searches

with embedded structural information.

1.1.3 Human-Computer Interaction

Although efficiency and functionality are two very important considerations for any system to succeed, the "user issues" are frequently ignored. A system must be usable and appealing to the users in order to be successful. It is not trivial to determine whether an interface or visualization method is user-friendly. This task is nearly impossible without the involvement of potential users of the system, preferably in a similar environment in which the system is expected to be used. Designing for usability is another very important factor in any system design. In this research, we make use of HCI (Human Computer Interaction) tools and principles to design a visual interface for performing the task of querying document databases. To design a usable interface, importance needs to be given to cognitive considerations (*e.g.*, familiarity, visibility, perception) and social considerations (*e.g.*, context, surroundings). Some of the primary HCI concepts that we use in the subsequent chapters include the following:

Cognitive artifact A cognitive tool or a cognitive artifact is a replacement for human deficiency [Hel88, Chapter 1]. If humans could perform all the necessary tasks rapidly, we would not require additional tools. The primary reason of using a tool is that it enhances human ability. When a goal is identified, it is necessary to decide whether it is within the limits of normal human capabilities, and a tool becomes necessary if it is either impossible or inefficient to perform the task by a human.

Mental models A mental model is the *users' mental architecture* [Hel88, Chapter 2]. At the time of performing a task, a user may already have some knowledge regarding the actual method by which the task is performed. This knowledge can be in the form of (i) rules performed in sequence that govern the process; (ii) methods that generally achieve the goal, and (iii) knowledge of the components of the system and their interaction. In order to design friendly user interfaces,

the designer needs to appropriately utilize this mental model of the users, taking into account what expectations of the system they have and designing the interface accordingly.

Interface metaphors From a linguistic point of view, a metaphor is a word or phrase describing an object or idea in place of another to suggest a likeness between them. In the design of user interfaces, metaphors are used to control the complexity of the interface by exploiting the user's prior knowledge of domains comparable to the domain of the system. This approach increases the initial familiarity of the actions, procedures and concepts of a system by making them similar to those already known to the user [Hel88, Chapter 3]. The "desktop metaphor" of graphical operating systems is an example of metaphors commonly used in interface design.

Direct manipulation Direct manipulation is a technique utilized in user interfaces in which the user has a continuous representation of the object of interest, and the actions involve *physical* movement of objects rather than textual commands. In addition, the interface provides continuous feedback to the user on the status of the system [Shn87].

Individual differences One very important consideration, which is often ignored during the design of user-interfaces, is the difference between the users of the interface. It needs to be kept in mind that every user is an individual, and everyone differs in their perception of the concepts necessary to use the target system. Systems designed for usability need to be properly checked with users with varying levels of knowledge and experience if they are to be used by such users [Hel88, Chapter 6].

1.2 Research Issues

In Section 1.1.2.2, we looked at techniques for information retrieval from text documents without any structural information. This type of search, often called "full-text

search" has its limitations, and Sembok [SR90] argue that the efficiency of keyword searching has reached its theoretical limit. Thus, adding structural information in documents and using this extra information for restricting searches provides an interesting alternative to full-text keyword searches. These searches that integrate the embedded structural information (meta-data) with the actual text (data) are frequently called "queries". We provided a few examples of queries earlier in this chapter.

One natural approach for processing queries on structured text databases is to first convert the documents into a standard database format, and then use the capabilities of the database to process the queries. The problem with this approach is that the structure of documents cannot be easily modeled using standard database techniques. It is extremely difficult to model hierarchically structured documents using relational databases since the flat structure of the relational model causes excessive fragmentation in the document structure. Complex object and Object-oriented databases seem to better match document structures, and a significant number of efforts [CACS94, Zha95, Hol95, D'A95] have been devoted towards mapping SGML documents in an Object-oriented or Object-Relational database and using the database for processing the queries. The main problem with this approach is that some document structures still do not fit completely in a standard database model, and these systems resort to heuristics to get around this problem. This often results in loss of information, and in most cases requires the documents to be created and processed only by the particular database, affecting the interchangeability of the documents.

The primary research issue here is to consider SGML itself as a modeling tool and to use external indices to solve queries without the need for mapping documents into a different database format. This makes such a system "closed" within the SGML domain, just as relational database systems are closed within tabular structures. The primary advantage of this property of closure is the ability to reuse and nest the queries and their results. Current database systems that support SGML attempt to achieve this in a convoluted manner, by converting to another format and if necessary, converting back to SGML. Hence, there is a need for research to investigate whether this double-conversion can be avoided.

1.2.1 Goals of this Dissertation

The primary goal of this dissertation is to design and implement a database system for documents using a single format that provides modeling, efficient processing as well as user interfaces. In addition, we describe a prototype system that implements most of the features required of such a system, prominent among them are the following:

- *Broad range of queries.* Most index-based approaches to information retrieval systems are usually restricted to a small and often ad-hoc set of queries. The proposed system should be able to process the select-project-join queries like SQL.

- *Closure.* Closure is the property by which the input and output of a process are "closed" within the same domain, or in other words, the input and output are in the same form. For example, the input and output of a query in a relational database are both in the form of tables. The prototype system should ensure that the input and output of the queries are both valid SGML documents.

- *Efficiency.* Although the range of queries is important, the query processing should be as efficient as possible, by incorporating (or suggesting the incorporation of) fast index structures, caching and other speedup techniques commonly used in database systems.

- *Ease.* The goal of the query language portion of the dissertation is simplicity. We recognize that the primary users of document query processing systems are from humanities disciplines, and requiring users to learn new programming languages for the purpose of searching can be too imposing on the users.

Other properties of standard database systems, such as concurrency control, recovery, and views are also desirable.

1.2.2 Contributions

The primary contribution of this dissertation is the proposal, design and implementation of a database system specifically designed for structured documents. The

significant contributions are as follows:

1. *Design of a polynomial-time query language.* The central idea in this dissertation is the existence of a first-order query language which is within polynomial-time complexity. Although this language is not capable of expressing all polynomial time queries, most queries commonly used for such databases can be expressed in this language. Chapter 5 discusses the details on four equivalent versions of this language.

2. *Proposal for a standard query language for SGML databases.* Although SGML has been in existence for approximately the same time as SQL (the standard query language for relational databases), there is still no standard method for querying databases supporting SGML. Vendors of SGML databases create their own method for posing queries and hence, cause much damage to the property of portability, which was the original aim of SGML. This dissertation proposes a language which is familiar to the SGML community while retaining all the power and properties of SQL.

3. *Design of a generalized visual language for query formulation.* In spite of all the advances in graphics and visualization, interfaces for querying databases are still limited to forms. This dissertation proposes a query interface based on QBE (Query By Example) [Zlo77] that simplifies the querying process and, at the same time, incorporates most of the power in the query language referred to above.

4. *Design of a query processing infrastructure for document databases:.* This dissertation introduces structures and access methods, quite similar to those in relational database systems, that are adapted for processing queries on hierarchically structured documents.

5. *Design of a prototype system with most of the desired features:.* This dissertation describes **DocBase**, a prototype system for posing queries in a document database. The queries can be posed using either SQL or the visual interface described above. Instead of starting from scratch, this prototype uses the Open

Text software [Ope94] for simple searches and uses special indices we designed for joins and other complex searches.

6. *A generalized method for current SGML systems to support SQL-like queries.* The prototype system demonstrates how a current commercial system can be given the capability of querying using the proposed query language. The basic properties necessary are primarily for traversal of the document hierarchy – something that all current products can perform fairly well. This demonstrates that it would be possible for most current products to incorporate this functionality.

1.3 Outline of this Dissertation

This rest of this dissertation is organized as follows. Chapter 2 provides the context of this work, including the concepts of structured documents, databases, HCI principles and IR techniques. Chapter 3 reviews the current approaches in this direction and derives the feasibility and necessity of this work. Chapter 4 describes the requirements of a structured document database system. Chapter 5 describes the design of the system, including the modeling, query language and internal data structure. Chapter 6 explains architecture of DocBase describing its implementation in detail. Chapter 7 describes the visual query processing techniques and the user-centric approach in this work. Finally, Chapter 8 summarizes the research and provides directions for future research on database systems for structured documents.

1.4 About this Dissertation

To demonstrate the power and applicability of SGML in document representation and processing, this dissertation was written completely in SGML. The printed version of the dissertation was obtained from a LaTeX document which was generated dynamically from the SGML source using a style-sheet based conversion program. Moreover,

the thesis was indexed using the prototype implementation of DocBase for the purpose of posing queries on the dissertation. Additional details on the applications and code used for creating this dissertation are presented in Appendix D.

Chapter 2

Context

This chapter describes the context of the current research. Our goal is to provide database support for fully structured documents in SGML. Here, we introduce SGML and its key concepts and features, describe the current trend in standard database systems with respect to modeling and query formulation, and discuss relevant areas of Human-Computer Interaction (HCI).

2.1 SGML and Structured Documents

SGML (Standard Generalized Markup Language: [ISO86]) is an international standard for document representation. The original purpose of SGML was to standardize and thereby facilitate the encoding of documents in a platform and system independent manner by embedding a textual representation of the logical structure information in the documents. SGML incorporates structure in a document by (1) first defining the structure and (2) then representing valid document instances conforming to this structure. In this section, we describe the basic concepts used in SGML and show how documents are created, structured and validated using SGML. The rest of this dissertation uses the generic term "structured document" to describe a document encoded in SGML.

2.1.1 Key Concepts in SGML

SGML is a language for describing and encoding the structure of documents. It is a *meta-language* in the sense that SGML can be used to define languages which in turn describe valid document instances. Documents encoded in SGML use a method for

marking up textual documents to make them conform to the structure defined using a DTD (Document Type Definition). The rest of this section describes the concepts of markup and DTD, including components of a DTD and their uses.

2.1.1.1 Markup

The primary concept behind SGML is the term "markup." In the traditional sense of the word, "marking up" refers to the insertion of special symbols in paper manuscripts, primarily as instructions to an author, typist, or compositor. In the SGML context, a "markup" is a sequence of characters designated to indicate the start or end of certain regions in a document, reference to previously defined symbols and calls to external processes. The existence of markup symbols is interpreted by applications to perform some procedure for handling that area of the document. This added information serves two purposes: [Gol90]

a) separating the logical elements of the document;
b) specifying the processing functions performed on these elements.

Although SGML is the standard in markup languages, many other document preparation and typesetting systems and languages (such as nroff, LaTeX) share this same idea.

Textual markup used in these languages are often referred to as "tags." As described in Chapter 1, tags can be either (1) specific (referring to specific formatting or layout instruction) (2) generic (referring to only the logical structure of the document) or (3) a mixture of these two types. SGML documents use generic markup by enclosing particular regions of the document between "start tags" and "end tags" that denote the start and end of these regions. The symbols used for this purpose are defined in the Document Type Definition (DTD) which will be described next.

In addition to tags, SGML uses other markup sequences to define external procedure calls (processing instructions) and macro definition and substitution (entities and entity references).

2.1.1.2 Document Type Definition (DTD)

The Document Type Definition (DTD) is an essential component of any SGML application. Before preparing a document in SGML format, the structure of the document needs to be defined. The DTD defines a language by defining a grammar to which the document instances conform. The non-terminals in this grammar are referred to as *generic identifiers*(GI), and the terminal symbols are usually character data. In the following example (Figure 1), we define a DTD corresponding to a document set containing information on books and publishers (adapted from the *pubs2* database that comes with the Sybase^{TM} relational database system [Syb94, Appendix C]).

```
<!ELEMENT pubs2       O O  (publisher+, author+)>
<!ELEMENT publisher   - O  (pubname, city, state, book+)+>
<!ATTLIST publisher   pubid ID #REQUIRED>
<!ELEMENT (pubname | city | state)
                      - O  (#PCDATA)>
<!ELEMENT book        - O  (title, type, price, advance,
                             totalsales, notes, pubdate, contract, authors)>
<!ATTLIST book       titleid ID #REQUIRED>
<!ELEMENT (title| type| price| advance| totalsales |
            notes|pubdate|contract)  - O (#PCDATA)>
<!ELEMENT authors     - O  (refid)*>
<!ELEMENT author      - O  (aulname, aufname, phone, address,
                             city, state, country, postalcode, copy)>
<!ATTLIST author      auid ID #REQUIRED>
<!ELEMENT (aulname | aufname | phone |address |
                country | postalcode | copy) - O (#PCDATA)>
<!ELEMENT refid       - O  (#PCDATA)>
<!ATTLIST refid       who  IDREF  #CONREF>
```

Figure 1: The pubs2 DTD

The DTD consists primarily of a number of production lines. Each production defines an element using a *generic identifier*(GI) and describes the contents and omission rules for the markup of the element. Omission rules are important because omitting tags reduces the size of the documents and makes them easier to read. In most cases,

the parsers can insert missing tags based on the context.

A document type definition specifies the following:

1. The *generic identifiers* (GIs) of elements that are permissible in the document type.

2. For each GI, the possible *attributes*, their range of values and default values.

3. The *content model* for each GI, which includes the sub-elements of the GI and permissible characters within that GI.

Elements and Generic Identifiers Generic Identifiers (GIs) in the SGML context are names given to the non-terminals in the grammar specified by the DTD. In a DTD, a GI is declared using the `ELEMENT` specifier. Declaration of a generic identifier defines two tags — the *start tag* and the *end tag* of the GI. These tags and the text enclosed by them constitute logical elements defined by the GI. In the DTD, the definition of each element includes omission rules for its tags, attribute definitions and the content model of the element. The omission rules determine whether or not either the start tag, end tag or both can be excluded from the document. Usually exclusion of tags is feasible if it is possible to infer the start or end of the element from the context in which it appears. For example, in the DTD in Figure 1, the omission rules for almost all the elements are specified as "- O" — indicating the start tag of these elements cannot be omitted, but the end tags can be omitted if the end tag can be inferred from the context in which the tag appears.

Content Models Content models describe the contents of composite elements. The SGML DTD specifies the grammar using an *Extended Context-Free Grammar* [MK76] (context-free grammar where the right side of a production can have regular expressions). The expansion of an element, referred to as "content models" in SGML, may consist of only character data (data content), only constituent elements (structure content) or both (mixed content). A content model may be empty, indicating that the particular element does not have any content, but its simple presence indicates some special processing at that position in the document.

Data Content. SGML is primarily an untyped language, in the sense that it is not possible to declare the data types of elements. For example, in the above DTD, there is no way to directly specify that the element representing the date is actually of type "date/time." This is primarily because SGML is a structuring language and gives no semantics to the data. SGML only supports data in the form of character sequences. However, SGML does provide a few variations of character data for use in different contexts, primarily as a means for parsing support. The two main character data variants are `PCDATA` (Parsed Character Data) and `RCDATA` (Replaceable Character Data). `PCDATA` contents are parsed by the parser as usual, but `RCDATA` contents are left unparsed - only the entity references are replaced. SGML also supports a limited number of data types for the attributes which we describe later in this chapter.

Structure Content. SGML DTDs can specify the structure of an element using an Extended Context-Free Grammar notation. The structure content may contain regular expressions consisting of other GIs. Regular expressions in SGML content models may be defined formally as follows (in order to demarcate the regular expressions from the rest of the text, we enclose them within double quotation marks, but they are not part of the regular expressions):

- For every GI A, "A" is a valid regular expression, indicating one and only one occurrence of A.
- If R_1 and R_2 are regular expressions, so are the following:
  - "R_1, R_2" - indicating a single occurrence of R_1 followed by a single occurrence of R_2, in that order. This model is often called the "sequence" model.
  - "R_1 & R_2" - indicating an occurrence of R_1 and an occurrence of R_2 - without any particular order, but both need to be present.
  - "$R_1|R_2$" - indicating an occurrence of either R_1 or R_2 but not both. This model is often referred to as the "option" model.
  - "R_1*" - indicating zero or more occurrences of R_1.
  - "R_1+" - indicating one or more occurrences of R_1.

- "R_1?" - indicating zero or one occurrence of R_1 (*i.e.*, the expression R_1 is optional).
- "(R_1)" - indicating a single occurrence of R_1.

For example, the second line in the DTD in Figure 1 indicates that the content model of publisher contains one or more instances of publisher information, which includes the publisher name (pubname), city, state, and one or more books.

Mixed Content. A mixture of data and structure content is also allowed in content models, usually in a sequence or option model along with another structure model. If present in a sequence model, the presence of the `#PCDATA` indicates the only area of the content model where character data can appear. Even white space characters cannot appear in any other position in the content model. If `#PCDATA` appears in an option group, either character data or the content model may appear. The following example illustrates the two types of mixed content. The third line illustrates the use of a repetition to indicate interspersed data and structure content.

```
<!ELEMENT both    - - (#PCDATA, (A, B))>
<!ELEMENT either  - - (#PCDATA | (A, B))>
<!ELEMENT mixed   - - (#PCDATA | (A,B))*>
```

Other Content Models. In addition to the data, structure and mixed contents, a content model can be `EMPTY`, indicating that it may not contain any other element; or `ANY`, indicating that it may contain any other valid element in the DTD.

Attributes Some properties of elements do not belong directly to the content of the document. For instance, a document may be a draft of a paper and may have version number information. This information is useful to the author but it cannot be characterized as the content of the document. According to Goldfarb [Gol90],

> The GI is normally a noun; the attributes are nouns or adjectives that describe significant characteristics of the GI.

Attributes are specified using the `ATTLIST` specifier in the DTD. Each element may have only one attribute list specifier, containing unlimited number of attributes.

For each attribute, the following information needs to be specified (see Figure 2 for illustrations):

```
<!DOCTYPE atts [
<!NOTATION TeX SYSTEM    -- TeX Notation -->
<!NOTATION Roff SYSTEM   -- Roff Notation -->
<!ENTITY alpha SYSTEM "alpha.txt" NDATA Roff -- Entity declarations -->
<!ENTITY beta  SYSTEM "alpha.txt" NDATA Roff>
<!ELEMENT atts - - (eg1 | eg2)*>
<!ELEMENT eg1  - - (#PCDATA)>
<!ATTLIST eg1  first CDATA #IMPLIED    -- optional attribute --
               second ENTITY #REQUIRED -- required attribute --
               third  ENTITIES #IMPLIED
               ident ID #REQUIRED      -- ID value  --
               value  (type1 | type2 | type3) "type1"
                           -- enumerated type with default value --
               type NOTATION (TeX | Roff) "TeX">
<!ELEMENT eg2 - - (#PCDATA)>
<!ATTLIST eg2 ref IDREF #CONREF         -- content reference --
              refs IDREFS #IMPLIED      -- multiple references --
              quant NUMBER #CURRENT     -- numeric attribute -->
]>
<atts>
  <eg1 first="john" second="alpha" third="alpha beta" ident="a101"
value="type2" type="Roff">This is eg1</eg1>
  <eg1 first="jane" second="beta" ident="a102" type="TeX">
This is a second instance of eg1</eg1>
  <eg2 ref="a101" refs="a101 a102" quant="200">
  <eg2 ref="a102"><!-- quant=200 even though not specified -->
</atts>
```

Figure 2: Illustration of the different types of attributes

1. *Attribute name.* This is the name of the attribute which is unique in a particular DTD.

2. *Attribute type.* This is the *type* of the attribute. SGML allows attributes to be of a number of types, including `CDATA` (character data), `ENTITY` (reference to a declared entity), `ENTITIES`, `ID` (an identifier value for cross-referencing), `IDREF(S)` (references to identifier values), `NAME(S)` (a name), `NMTOKEN(S)` (name tokens), `NOTATION` (notation name), `NUMBER(S)` (numeric values) and `NUTOKEN(S)` (number tokens). Attribute types also include listed values (similar to enumerated

data types in programming languages).

3. *Attribute value.* In the case of a listed attribute type, one of the specified values can be indicated to be the default. In the case that no values are specified for this attribute, the default value is assumed by the parser. For the rest, the value can be specified to be (i) `#IMPLIED`, indicating that the attribute value can be omitted and will be implied by the application; (ii) `#CURRENT`, indicating that if the attribute value is omitted, the application will use the most recently used value for this attribute; and (iii) `#REQUIRED`, indicating that the attribute value cannot be omitted. For `IDREF` attributes, the attribute value may be `#CONREF`, indicating that the content of the current element is to be referenced from another element whose ID is being referred.

Entities In text documents, it is often necessary to repeat sequences of character or markup. SGML applications use the `ENTITY` feature to accomplish this. There are four primary types of entities:

1. *Character Entity.* Entities representing special characters. Characters that fall in this category include characters that cannot be keyed in using a regular keyboard (such as ©, represented as "`©`"), characters that have special meanings in SGML (such as <, represented as "`<`"), and foreign characters (such as Ω, represented as "`Ω`"). As the examples suggest, the entity references use the ampersand (&) character in front and a semicolon at the end.

2. *General Entity.* General Entities are very similar to macros in programming languages. They are similar in representation to character entities, but usually expand to one or more characters.

3. *File/Document Entity.* A file entity refers to a document, usually specified by a `SYSTEM` identifier (file name) or a `PUBLIC` identifier (a specially formatted name which can be mapped to a file). When referenced, the entity is replaced by the file it represents.

4. *Parameter Entity.* Parameter entities are only used in DTDs and may contain markup declaration. They are referenced using the % symbol instead of & as in the other entities.

2.1.1.3 The SGML Documents

Valid SGML documents are instances of the schema defined by the DTD. These documents are usually standard text documents tagged using the markup syntax defined in the DTD. An SGML document corresponding to the DTD defined in Figure 1 is shown in Figure 3.

2.1.2 SGML Applications

At bare minimum, an SGML application consists of an SGML DTD and document instances conforming to the DTD. A complete application designed for a specific task may have other modules to manage the documents and associated components. Since SGML documents do not specify any semantics, other documents often accompany SGML source documents. Such components include style-sheet specifications to associate layout information to logical regions; translation scripts to translate SGML documents into other formats for printing or displaying purposes; or indexing information for searching the documents. To use the SGML documents, applications may also require the SGML parser to check the validity of the SGML documents as well as entity management components to ensure that all the cross-referenced components are valid.

2.2 Database Background

2.2.1 Standard Database Models

Database systems are widely used for the purposes of efficient processing and management of large volumes of structured data. Relational databases, in particular, are extremely useful for many such applications. Here, we review the standard database

```
<!DOCTYPE PUBS2 system "pubs2.dtd">
<PUBS2>
  <PUBLISHER PUBID="A0736">
    <PUBNAME>New Age Books</PUBNAME>
    <CITY>Boston</CITY>
    <STATE>MA</STATE>
    <BOOK TITLEID="BU2075">
      <TITLE>You Can Combat Computer Stress!</TITLE>
      <TYPE>business</TYPE>
      <PRICE>$2.99</PRICE>
      <ADVANCE>$10,125.00</ADVANCE>
      <TOTALSALES>18722</TOTALSALES>
      <NOTES>The latest medical and psychological techniques for
      living with the electronic office.Easy-to-understand
      explanations.</NOTES>
      <PUBDATE>6/30/85</PUBDATE>
      <CONTRACT>-1</CONTRACT>
      <AUTHORS>
        <REFID WHO="A213-46-8915">
      </AUTHORS>
    </BOOK>
</PUBLISHER>
<AUTHOR AUID="A213-46-8915">
    <AULNAME>Green</AULNAME>
    <AUFNAME>Marjorie</AUFNAME>
    <PHONE>415 986-7020</PHONE>
    <ADDRESS>309 63rd St. #411</ADDRESS>
    <CITY>Oakland</CITY>
    <STATE>CA</STATE>
    <COUNTRY>USA</COUNTRY>
    <POSTALCODE>94618</POSTALCODE>
    <COPY>Mr. Green</COPY>
  </AUTHOR>
</PUBS2>
```

Figure 3: An instance of the Pubs2 DTD

models including the relational model and observe the properties that are useful and essential in the development of database systems. We also note that the relational model is not powerful enough to capture properties of complex data, and hence, describe efforts on generalization of the relational model to represent complex data structures and properties. In particular, we briefly describe the nested relational model and the object-oriented model for representing complex structures.

2.2.1.1 The Relational Model

Among the current database models, the relational model is the most extensively used model for describing database systems. One of the primary reasons behind the success of this model is its simplicity, which is also the reason behind its limitations. The relational data model is based on a strong theoretical foundation proposed by Codd [Cod70].

The primary idea in relational model is that the data is represented in tables with a fixed number of columns, and each row represents a single record. A relational database consists of a set of tables, or *relations*; the relations consist of a set of tuples within a domain. These tuples form the rows of the relational tables, and the columns represent properties of the tuples (*attributes*). Each attribute is of an atomic data type (*e.g.*, character strings, numeric values), and the type of all the attributes combined forms the type of the relation. The relational schema, which governs the structure of the relational database, consists of the relation names and the attribute names with their types. The relations are the only composite types allowed in the relational model, and attributes of relations may only be of atomic types (*e.g.,* character, numeric). However, attribute values in a relation can refer to attribute values in other relations.

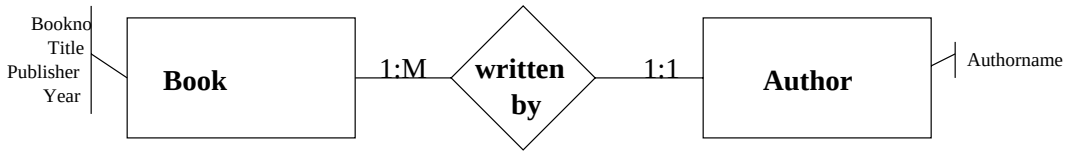


Figure 4: A simple Entity-Relationship diagram

Because of the simplicity of the model and the fact that most scenarios cannot be directly modeled using a flat tabular structure, it is often necessary to "flatten" a conceptual structure so that it can be represented by a relational database. However, flattening a complex structure introduces redundancy in the data and also introduces the possibility of anomalies (such as insertion, deletion and update anomalies) resulting from database operations [Ull88, Chapter 7]. To avoid these anomalies, the flat schema is fragmented into a set of smaller relations. This process of breaking a schema into multiple flat relations is called *normalization*. During query processing, it is often necessary to bring these fragments together, which is achieved by a special operation called *join*. As an example, we can consider a simple library catalog, in which information on books is stored. Consider a simple "book written by author" relationship as shown in the Entity-Relationship (ER) diagram[1] in Figure 4. For simplicity, assume a book can be written by multiple authors, but an author writes at most one book. The equivalent relational schema with a simple instance is shown in Table 1.

Book			
Book Number	**Title**	**Publisher**	**Year**
0198537379	The SGML Handbook	Oxford	1990
079230635X	Practical SGML	Kluwer	1990
0133098818	Developing SGML DTDs	Prentice Hall	1996

Auth_info	
Book number	**Author**
0198537379	Charles Goldfarb
079230635X	Eric Van Herwijnen
0133098818	Eve Maler
0133098818	Jeanne El Andaloussi

Table 1: A sample relational database instance

In the example schema (Table 1), the information on books include a book number,

[1]The Entity Relationship (ER) model [Che76] is a method for viewing data conceptually using distinct objects (entities) and the relationships between them. The ER diagram is a method for visually describing an ER model, and is a great tool for specifying a conceptual data model.

a title, a publisher, a year, but may include multiple authors. In order to model this multiplicity using a relational schema, the author information needs to be kept in a separate relation linked, using a common field, to the main book relation. This situation arose because of the one-to-many relationship "written-by" as shown in Figure 4.

2.2.1.2 Complex-object and OO Models

The simplicity of the relational model sometimes becomes a problem when the data is modeled as a set of flat relations. In many cases, normalization needs to be performed multiple times, making the resulting schema overly fragmented. This not only makes subsequent query processing difficult, but also makes the schema vastly different from the natural representation of the structure. To get around this problem, various forms of "complex-object" models have been proposed. One of the most prominent complex-object models is the *nested relational model* [TF86, AB95], in which the attributes in a relation are allowed to be of composite type (*e.g.*, sets, lists or other relations), thus making the actual representation of the data in this model closer to its conceptual structure.

The schema in Figure 4, using a nested relational model, will have an instance as shown in Table 2. In the nested relational model, an attribute can be another relation (such as "bibliographic info" in the example) or a set type (such as "authors" in the example).

Book No.	**Title**	**Bibliographic Info**		**Authors**
		Publisher	**Year**	
0198537	The SGML Handbook	Oxford	1990	Charles Goldfarb
079230635X	Practical SGML	Kluwer	1990	Eric Van Herwijnen
0133098818	Developing SGML DTDs	Prentice Hall	1996	Eve Maler
				Jeanne El Andaloussi

Table 2: An instance of a complex-object schema

In addition to the different types of complex-object models, object-oriented database

models have also been proposed [BBB+88]. One of the primary aspects of the object-oriented model is the use of abstract data types and inheritance. In the object-oriented model, the schema may include user–defined complex types (also known as Abstract Data Type or ADT) having both data and procedural attributes (or methods). Some abstract data types may be defined as subtypes of other types inheriting properties from supertypes and defining new properties of their own.

2.2.2 Database Query Languages

Modeling and storage form only part of the problem in database systems. Database systems need to provide methods for efficiently retrieving the data from the database. To retrieve data from the database based on certain conditions, one uses different query languages for the particular database model. Codd proposed two query languages - *relational algebra* and *relational calculus* for the relational model. These two languages form the theoretical basis for other languages for relational databases. In this section, we discuss the formal languages and SQL (Structured Query Language), the standard language for relational databases.

2.2.2.1 Formal languages

The two primary formal languages for the relational model are relational calculus and relational algebra. Relational calculus is a declarative query language, in which queries are expressed with formulas in first order logic to describe the resulting relation. Relational algebra is a procedural counterpart of the relational calculus which formulates queries with relational expressions. In both these languages, input to a query consists of a set of relations, and the output of a query is also a relation. Observe that this guarantees the possibility of composing queries in either language.

Relational Calculus In relational calculus, a query is expressed using a logical formula, called relational formula, that hold true for a set of tuples that form the result of the query. The relational formula $\{x_1, x_2, \ldots, x_k | \psi(x_1, x_2, \ldots, x_k)\}$ describes relational tuples of the form $(x_1, x_2, \ldots, x_k)$ that satisfy a formula $\psi(x_1, x_2, \ldots, x_k)$

using a query , where $x_1, x_2, \ldots, x_k$ are variables. Formulas may contain variables which can be either free or bound by quantification operators. The formulas may also contain predicates corresponding to relations in the database schema. A formal description of the relational calculus is based on the definitions of *terms*, *atomic formulas* and *well-formed formulas (wff)*, defined formally as follows:

- *Terms.*
  - Every variable x is a term.
  - A constant c is a term.
- *Atomic Formula.*
  - Every literal $P(x_1, x_2, \ldots, x_k)$ is an atomic formula, where P is a k-ary predicate symbol representing a relation and $x_1, x_2, \ldots, x_k$ are terms.
  - Every arithmetic comparison of the form $x\theta y$ is an atomic formula, where x and y are terms and θ is in $\{<, \leq, =, >, \geq, \neq\}$.
- *Well-formed Formula (wff).*
  - All atomic formulas are wffs, and all the variables appearing in the formulas are said to be free in these formulas.
  - If F_1 and F_2 are wffs, then so are formulas constructed by combining them with logical operators, such as $F_1 \wedge F_2$, read as "F_1 and F_2;" $F_1 \vee F_2$, read as "F_1 or F_2;" and $\neg F_1$, read as "not F_1." The free variables of the resulting formula is the union of the free variables in F_1 and F_2. There is no influence of a variable bound in one formula if it occurs free in the other.
  - If F is a wff and x is a variable, then $(\exists x)F$, read as "there exists an x such that F is true," and $(\forall x)F$, read as "for all x F is true," are wffs. The variable x in this case is said to be bound by the "$\exists$ or $\forall$" in F. All other variables in F are free.
  - If F is a wff, so is (F).

The relational formula represents a relation with a set of free variables $x_1, x_2, \ldots, x_k$ that constitute the result relation.

Relational Algebra Relational algebra uses relation variables and provides a specialized operators on these relation variables. The main operators provided by relational algebra are *select*(σ), *project*(π) and *cartesian product*($\times$). Other important operations include set operations like *union* ($\cup$) and *difference*($-$) and specialization of other operators such as *join*($\bowtie$) and *division*($\div$). Queries in relational algebra are expressions that map one or more relations combined using the above operators into a resulting relation.

Equivalence of Relational Query Languages One important property of the two relational languages is that they have the same expressive power. It can be formally proved that any query expressed in relational calculus[2] has an equivalent relational algebra query and vice-versa [Ull88]. Other query languages such as SQL, which we describe later (in Section 2.2.2.2), are also designed to implement all queries of relational calculus with carefully controlled extensions. The significance of this equivalence is that, during processing, queries formulated in the more natural calculus-like languages can be converted to the procedural languages for optimization and evaluation.

Query Languages For Complex-object and OO Models Although the primary languages described in this section are designed for the relational model, these languages can be easily adapted for the complex-object and object-oriented models discussed above. Most of the query languages for the complex-object models have been designed as extensions to forms of relational algebra and relational calculus. The extensions are primarily intended to handle complex types — both the construction of complex types from atomic types and extraction of the constituent atomic types from the complex objects. One example of a complex-object query language is the

[2]In the standard relational calculus, it is possible to write queries that produce results of infinite size. The equivalence only holds when we consider a "safe" version of the calculus, in which all variables need to be properly bounded before they can be used in a query.

nested relational algebra [TF86] that uses operations such as set construction (nest) and set decomposition (unnest) to handle the complex-valued attributes.

2.2.2.2 Structured Query Language (SQL)

SQL [SQL86a] has been the standard query language for relational databases for over ten years. SQL is based on tuple relational calculus, and the syntactic nature of the language has its roots in the original SEQUEL language [AC75]. SQL is more expressive than the core relational calculus since it supports extra functionality such as grouping and ordering mechanisms, aggregate functions (*e.g.*, count, sum and average), and arithmetic operations. In spite of being more expressive than the formal languages, it uses a very simple syntax closely related to English, and all queries are based upon a simple SELECT-FROM-WHERE combination as described below. Another advantage of SQL is its widespread use in most commonly used commercial as well as research-based database systems. SQL has been revised multiple times in order to enhance the functionality using various programming language constructs. However, the core part of SQL is a natural-language equivalent of the tuple relational calculus.

```
SELECT item-list
FROM   relation-list
WHERE  condition-expression
```

Figure 5: Core SQL syntax

The syntax for the core SQL language is summarized in Figure 5. In the SQL syntax shown here, "*relation-list*" refers to a list of relations in the database, possibly repeated and augmented with tuple variables; "*item-list*" refers to a list of attributes from the relations, possibly with aggregate functions applied to them, and "*condition-expression*" refers to a list of predicates combined with logical connectives.

2.2.2.3 Query By Example (QBE)

Query By Example [Zlo77] is a high-level visual language that provides the user with a unified interface to query and update relational databases. This language has a simple interface composed of tabular skeletons representing tables in the database. Users specify queries by entering sample values in appropriate areas of the table skeleton. These values can be either constants (usually search strings) or variables (also called "examples" in the context of QBE). The purpose of the variables is mainly to perform "join" operations, but they are also used to specify output attributes.

Book	**Bookno**	**Title**	**Publisher**	**Year**
	P.	P.		P.1996

Table 3: A QBE implementation of the query: "Print the book numbers and titles of the books published in 1996"

Book	**Bookno**	**Title**	**Publisher**	**Year**
	P.<u>1234</u>	P.		

Auth_info	**Bookno**	**Author**
	<u>1234</u>	Charles Goldfarb

Table 4: A QBE implementation of the query: "Print the book numbers and titles of the books written by Charles Goldfarb."

QBE can also handle complex boolean combinations of such search expressions using a special section of the screen termed *condition box*. Aggregation operations such as sum, count and average can also be performed by indicating the respective operation in the tabular skeleton. Table 4 displays a join query. The underlined words in the above example are variables that indicate that the "Book ID" attribute of *Book* as well as the "Book ID" attribute of *Written_By* should be the same. Using this method, the user provides an example of outputs that she expects from her query, and the query engine looks in the database for data that matches the given example.

This works nicely for relational databases, primarily because the tabular structure of the database fits quite well with tabular skeletons used in the interface. Some of the properties of QBE that are of importance to this research are:

Simplicity The core QBE has a simple visual appearance and does not require users' knowledge of the database schema.

Equivalence The presentation of the query interface is conceptually equivalent to the internal tabular structure of relational databases.

Closure The query is constructed using table skeletons, and the results are displayed using similar tabular structures.

Completeness The core QBE, combined with some additional constructs such as condition boxes, can construct all the queries specifiable using relational algebra or relational calculus.

In a later section, we will show how our approach keeps all these properties in a generalized interface designed primarily for documents, yet applicable to any complex structured data. There have been other attempts at generalizing QBE for complex structures. Notable among them is the Generalized Query By Example (GQBE) [JW83] which uses an interface very similar to QBE for application for databases with complex hierarchical structures. This method uses nested tables, similar to the one shown in Figure 2, for composing queries as well as specifying insertion, deletion and update commands to the database. There are a few other attempts at generalizing and extending QBE — a survey on these methods can be found in [OW93].

2.2.2.4 Fill-out Forms to Express Queries

Although QBE is a formally accepted visual language for relational databases, few relational database systems fully implement QBE, possibly because of the complexity of implementation. Some commonly used relational database systems implement variations of QBE. However, application developers designing query interfaces seldom use the QBE method directly in their implementations since usually a simpler and

non-general method is more suited for specific applications. In these cases, developers use form-based interfaces.

L597 User-centered Database Class People

First Name	Sengupta
Last Name	
Program	Ph.D
email	
phone	
age	
sex	
interests	
languages	

Figure 6: An example of a form interface for formulating queries.

In form-based interfaces, the user is presented with a list of searchable fields, each with an entry area that can be used to indicate a search expression. Searches are restricted to only the fields listed, and the types of searches are restricted to simple boolean combinations of these search conditions. To pose a query, the user needs to enter keywords in the relevant places. This provides users with a quick and easy way to specify some searches on the databases and proves to be adequate for many applications. However, these form-based searches are not scalable and do not adapt well to changes in the database schema. An example of the use of forms for formulating queries is shown in Figure 6.

Although forms usually require ad-hoc application-specific design, there have been attempts to formalize the querying process using forms. The Natural Forms Query Language (NFQL) [Emb89] is a language for specifying queries using forms. It allows queries to be specified using a "universal relation" from which the database relations are derived. A filled-in form in this language can be thought of as a "view". It allows

both queries and updates to the relational database without the necessity from the users of knowing the internal schema of the database.

2.3 HCI Background

HCI (Human-Computer Interaction) plays a very important role in this research. Although the primary goal of this research is to propose methodologies to build database systems for structured documents, we try to ensure that these systems are well within the limits of the targeted users. The primary users of text resources (such as novels, poetic works, journals) typically have limited computing experience. Requirements of mastering a query language or the internal database schema are thus often inappropriate. One objective in this work is to retain all the necessary properties of database systems without sacrificing the usability of the system. This section introduces the important concepts of HCI that play an important role in this research. The design of *Query By Templates* (described in Chapter 7) will demonstrate how we use these concepts in our design process.

2.3.1 Principles for Usable Interface Design

Usability is a primary concern of any user-interface design. It is futile to expend resources and efforts in developing systems that can only be used by a handful of people. Some of the most important concepts necessary for designing usable interfaces are [Nor90, Chap. 1]:

- *Provision of a useful Conceptual Model.* It is important for interfaces to provide a good conceptual model to the users. A good conceptual model allows the users to predict the effect of the actions. Components of an interface need to have a direct mapping with the functionality of the interface. Without this mapping, the task is often made more difficult by the presence of functional components that have no apparent relationship with the rest of the system. Without this kind of information, users may be able to perform tasks based

on prior instructions given to them, but they will need to have much better understanding of the system in order to recover if something goes wrong.

- *Principle of Visibility.* Functional components of interfaces need to be visible to the users. In particular, tasks performed often should not be "hidden" within the interface. The user interface components that perform such tasks should be directly visible to the user and should be able to serve as a conceptual model of the task that they perform (as described above).

- *Principle of Mapping.* "Mapping" indicates the relationship between two or more things. In the context of user interfaces, "mapping" refers to the interaction between physical manipulation of the controls and the resulting effect on the system.

- *Principle of Feedback.* Users need to see changes when they perform an action that produces changes. In many cases, these changes are visual and can be easily communicated to the user by expressing the change in the visual appearance of the interface (*e.g.*, marking a section of a line bold using a word-processor should result in the section displayed in boldface). However, when such visual equivalence does not exist for a certain action, the user needs to be reassured that her action was completed using some type of feedback mechanism. Such feedback could be in form of a message (*e.g.*, a message saying "the file has been saved" when the user performs the save action). Other types of feedback include events such as audio alerts and easy-to-detect visual changes on the screen.

Although most interaction principles are quite flexible, proper incorporation of such concepts in user interfaces makes a significant difference between an intuitive and a non-intuitive cumbersome interface.

2.3.2 Ensuring Usability

Simply implementing the HCI principles indicated above does not guarantee the usability of an interface. According to Shackel [Sha84], there is no manual for effectively

incorporating human factors in computer systems. System designers usually have different capabilities and strengths compared to the users of the systems. Designers may also have different notions of intuitiveness. This often results in systems that are usable by designers but not the target users. Thus it is imperative that users be part of the design process as early as possible and continue throughout the process of design and development. Rubin [Rub94] argues that the three principles of user-centered design are: (i) an early focus on users and tasks, (ii) empirical measurement of product usage, and (iii) iterative design whereby a product is designed, modified and tested repeatedly.

In the early phase of the design process, a designer should (i) identify the target users, (ii) have direct communication with the users, (iii) visit user locations and (iv) observe the users working, and if possible, record their actions [Gou95]. The knowledge of users' abilities and behavior is an essential component of the design process, and the earlier this knowledge is acquired the better. During the process of design and development, designers need to continue interactions with the users, and if necessary, use an iterative (design, test; design, test...) method during the development cycle. During this cycle, users need to be involved in testing the interface (or possibly prototypes of the interface), and lessons learned from these tests need to be incorporated in the next cycle of development. Such tests on the usability of systems (or prototypes) is often termed as *usability testing*.

2.3.2.1 Usability Testing

Interface design principles by themselves cannot ensure the usability of systems. Designers frequently need to "augment these intuitions with evidence obtained from observing [our] artifacts being utilized by real users, which is known as *usability testing* and *user testing*." [BGBG95] Usability testing primarily involves letting members of the targeted users use the system for realistic tasks and collecting information based on feedback from these users for the purpose of validating the design or redesigning problem areas. Nielsen [NP93] terms the process of usability testing by setting performance goals or metrics as *usability engineering*. Testing for usability should be, in fact, a prominent milestone by itself in the system design process.

The usability testing phase usually includes a phase for evaluating the testing method itself. This phase is often called a "pilot test" [DR93, Chapter 17]. Although, technically, this is just a prototype of the usability analysis itself, it can assist immensely in determining the possible problems with the testing method. A pilot test can also determine if the results obtained from the testing process can actually be used in determining the whether or not the target interface is usable. A secondary objective of pilot tests is to gain some practice with the actual testing process.

The actual testing process involves getting potential users of the system to perform realistic tasks using the system and recording their behavior during the process. Performing tasks in the target environment is usually more effective. This is achieved by performing the tasks in the environment the system will be used. In the case the target environment is not available during testing, a mock-up [Gou95] of this environment can be used.

Recording users' behavior can be done using several different techniques. Some of the most common techniques are:

- *Videotaping.* Videotaping the users in action is very useful for the purposes of measuring time, errors, and user attitudes [Gou95]. Watching videotapes of users unable to use the system for apparently simple tasks often serves as evidence of unusability.

- *Thinking aloud.* In this method, the participants of the usability test talk out loud as they try to perform certain tasks. This method enables designers to get an idea of users' mental states and helps in discovering ways to match their mental models. This method is usually useful in finding out missing or misleading visual cues in the interface. However, requiring users to think aloud may affect precision of time and performance measures.

- *Surveys.* The most common method of collecting user feedback is by using written surveys. Surveys let the designers get a feel for users' opinions, attitudes, preferences and behavior. However, although surveys are often deployed in the usability analysis process because of the low cost and overhead, they are not

suitable for observing and recording what users actually do while using the system.

2.3.2.2 Testing strategies

The goal of a usability analysis process is to determine the presence or absence of features that affect usability of a system under design. For example, a simple usability factor would be the time taken by a user of a word processor to complete the task of writing a letter. Based on this factor, one may be interested in whether users can perform the task faster in a word-processor than on a typewriter. The factors that are tested are the *dependent variables* of the testing process. The dependent variables are "dependent" on certain conditions – referred to as the *independent variables* of the test process. The primary independent variable is usually the type of interface used – where comparison is drawn between the interface under test and other similar but alternative systems. There could be other independent variables in the testing strategy, such as gender or experience of the users or type of computer terminal, depending on the objective of the analysis. In order to select a subset of independent variables from the other factors that affect the dependent variables, the rest of the factors should be kept unchanged throughout the testing process to avoid the effect of unwanted factors.

Usability tests are usually carried out using one of the following two strategies [Ebe94, Chapter 5]:

- *Within-users tests.* In a "within-users" analysis, all users are exposed to all the independent variables. For example, if two different systems are being compared, all users are asked to use both systems, and the results are collected based on this use.

- *Between-users tests.* In a between-users test, users are distributed among the dependent variables. For example, if two different systems are being compared, half the users may be given the target interface, and the other half may be given another interface.

2.3.2.3 Usability Analysis

After usability testing is performed, the collected results need to be analyzed to determine the extent of usability of the system. In most testing schemes, the target system is compared with common alternatives based on several properties. The properties that need to be tested are the dependent variables in the test, and the factors affecting these properties (such as the system used, knowledge and experience of users) serve as the independent variables. The goal of the analysis process is to determine if any of the independent variables affect the dependent variables to a statistically significant extent.

The test of significance is usually measured using several statistical methods, the most common among them being the ANOVA (Analysis of Variance) technique [Ebe94, Chapter 5]. The type of analysis varies depending on the type of the experiment and the number of independent variables. Common variations of ANOVA measures are one-way ANOVA, two-way ANOVA, repeated measures and randomized blocks. In all of these cases, a measure of significance (also referred to as the F-ratio) is computed, and the usability result is inferred by determining whether or not the F-ratio is below or above a pre-determined threshold. However, statistical significance is not always warranted if the number of users involved in a usability analysis is too few [NP93].

Chapter 3

Related Work

The main goal of this dissertation is to provide database support for text databases, and in particular, to investigate and develop methods for better query formulation and processing on databases that primarily contain textual data. The practicality for storing large text documents has certainly increased with the development of fast but inexpensive storage media. The concomitant increase in the sizes of these documents, however, has rendered the prevalent search and processing techniques unsuitable. Recent research has produced a number of relevant methods for efficient storage and retrieval of textual documents, but has, as yet, failed to exhibit standards and properties for search and management similar to database systems. Analogously, searches in document repositories, referred to as "information retrieval," have not been able to reach the popularity level of database languages such as SQL.

The primary difference between the conventional information retrieval (IR) techniques and database querying techniques lies in the use of meta-data or schema information in databases. Conventional IR methods use documents without structural information. In these methods, documents are treated as sequences of keywords, possibly intermixed with stop-words. These techniques retrieve document components using keywords, usually combined with boolean operators. The most common techniques for this kind of retrieval is the creation of indices of keywords based on the documents. Briefly, a retrieval operation begins with a preliminary search on the indices and subsequent extraction of document components based on the results of the initial search. The initial search performed on the index structures is usually very efficient. Use of indices also increase search efficiency by not accessing the complete documents stored in slower secondary storage.

Recent research has been directed towards achieving database-like properties for

document search systems by introducing meta-data or structural information in documents. This is achieved by "tagging" — a process of embedding additional code in the text that represents the meta-data information. Tags were originally introduced as a means for embedding special layout instructions in documents, but are more commonly being used for representing structural information. Document representation standards such as SGML and HTML use such tags to embed structural information in the documents. As described in Chapter 1, tagging can be either generic or specific. Since generic tagging is conceptually closer to the logical structure of documents, this form of tagging is more appropriate for the purpose of enhancing document searching capabilities.

In this section, we first discuss different techniques adopted for information retrieval from unstructured documents. We then observe the recent trend in using fully structured documents for more advanced query processing and discuss the recent research in this area. We also describe recent research efforts on using semistructured data in information management and retrieval.

3.1 Unstructured Information Retrieval

Unstructured text documents (plain text or ASCII text documents without any form of tagging) form the bulk of the electronic information today. Although plain text cannot represent all the various forms of documents of the modern information age, plain text was the primary format of these documents created years ago. The main purpose of unstructured information retrieval is to search for the position(s) of keywords in unstructured text documents. Other schemes of information retrieval that attempt to reduce the problems with keyword searches have also been proposed. In this section, we discuss the some of the conventional keyword-based retrieval methods as well as some alternative retrieval strategies. We describe the concept of indexing and the factors that determine the effectiveness of the indexing techniques. We also discuss retrieval strategies based on the indexing techniques.

3.1.1 Conventional Retrieval Methods

In conventional information retrieval methods, documents are stored as a sequence of words or phrases - often referred to as *terms* [Sal91]. Searches usually consist of boolean combinations of keywords, (*i.e.*, keywords combined with the operators *and, or, not*). The retrieval system is designed to extract, from the repository, fragments of documents that match the request. The *granularity* of the results, or the extent to which the resulting documents are fragmented usually depends on the underlying storage structure. For instance, in the case of documents represented as separate files in a file system, retrieval systems may retrieve complete files or lines from files that match the given conditions. In most cases, the granularity for search and retrieval is the individual lines of the document. We will see later that the presence of structural information enables a database system to allow the users to control the granularity of their results.

The actual search method to retrieve document components based on keywords depends on the underlying application that performs the search. This may simply consist of a sequential scan of the documents as performed in the popular Unix^{TM} utility "grep" [GNU92]. More often, manual and automatic indexing techniques are used to create indices on keywords found in the documents, and these index structures are used to quickly find the positions of matched keywords in the documents. During the indexing process, some systems can filter out words that are variations of the same root or words that are primarily used as "stop words" which are rarely used as search keywords. More advanced indexing techniques are also quite common, and we will discuss them presently.

The boolean model for information retrieval using boolean combinations of keywords has a number of limitations. Using anything more than simple boolean combinations often requires knowledge and training in logic. Also, all the terms in boolean expressions are treated as equally important, and therefore, retrieved documents are ordered arbitrarily. Moreover, boolean operators can only have either true or false values. This property often proves to be too rigid as the presence of one term in a disjunctive (OR) group results in the acceptance of the whole group and the absence of a single term in a conjunctive (AND) group results in the rejection of the whole

group. Boolean logic also does not have any way to group, order, or constrain the retrieved documents based on specific criteria in addition to the search keywords.

3.1.2 Alternative Retrieval Methods

One common alternative to boolean retrieval is the use of term weights to discriminate the search terms. This allows retrieved documents to be ordered according to the "extent of match," which reduces the rigidity of boolean retrieval systems. However, this method still depends on boolean logic and has similar limitations of boolean logic. Extending this weighted model with "strictness indicators" alleviates the rigidity of boolean logic.

Another alternative approach is to use *vector spaces* for the purpose of information retrieval [Sal91]. In the vector space method, documents are identified by sets of terms like the boolean method. In addition, term weights indicate the importance of terms. Thus a document is conceptually represented by a multi-dimensional vector of $\langle term, weight \rangle$ pairs. Queries are also represented using weighted term sets similar to the document representation. Retrieval is performed using mathematical measures of similarity between the document vector and the query vector. This method provides for simple and parallel treatments for queries and documents.

Probabilistic methods for performing information retrieval [Sal91, p.975] have also been proposed. In these methods, a quantity for relevance of a query Q_j for a document D_i is determined. The result of queries can now be presented in descending order of the relevance probability.

Another technique for information retrieval has been proposed using "rough sets." [Sri89] This method uses the rough set concept by Pawlak [Paw82]. Index terms are used to create equivalence relations, each equivalence class containing semantically identical (or synonymous) terms. Searches using this type of indices can use the rough equivalence to intelligently find matches for keywords that are approximately equivalent to the search keywords and, hence, increase the probability of recall. Boolean combination of keywords can be mapped to set unions and intersections in this method.

3.1.3 Indexing and Text Analysis

Most of the retrieval techniques described above depend on the indexing strategy (the terms used in the indices) and the index structure (the actual data structure used in the indices). Indexing of documents and the choice of terms can be done in either or both of the following two ways:

1. *Manual indexing.* In this method, terms to be selected for indexing are manually specified, typically by associating a set of keywords with documents or document components. In this case, the indexing system associates the documents with the given keywords irrespective of whether the keywords actually appear in the particular document component. This term selection is usually performed by trained personnel who are very familiar with the content of the documents.

2. *Automatic indexing.* In automatic indexing, a computer system uses special selection criteria to extract words from the document to be included in the index. There are a number of techniques for selecting the terms to be indexed, some of which are described below.

3.1.3.1 Automatic Indexing techniques

The simplest type of automatic indexing consists of assigning single-term indexing units to represent text content [Sal91]. This is frequently performed by identifying the individual words that occur in the documents. The index size can be reduced by omitting words from a set of common function words (such as "and", "of", "the", "but"), known as stop words. Words that are variations of the same base are detected by identifying a set of commonly used suffixes (such as "ing", "ed"), stripping these suffixes from the end of the words, and indexing only the stripped result. Suffix removal enables the indexing of words like "searches," "searching," "searcher" by using a common form like "search" and, hence, reduces the size of the index. Term weights for the indices are calculated from the various factors such as frequency of occurrence of the terms (term frequency) and the inverse document frequency (a measure of the probability of a term occurring in a document) [SYY75, SB88].

Other ways of automatically selecting terms for indexing purposes includes linguistic and knowledge-based approaches. Common practice includes the use of a thesaurus and phrases. A thesaurus combines the words closely related in meaning into groups. Although this method works well for single words, use of thesaurus with phrases is more difficult because of the uncertainty of combining groups of words. A number of methods involving simple techniques (*e.g.*, word frequencies and co-occurrence characteristics) and complex techniques (*e.g.,* automatic syntactic analysis) have been proposed to perform phrase grouping [SYY75]. In more advanced linguistic approaches (such as in [SR90]), semantic translation of natural language is used in document retrieval systems. In this technique, document and query texts are translated into sets of first order predicates which are used as their content indicators or indices. Actual semantics of words are considered for their grouping, using a grammar as a semantic translational aid. For retrieval purposes, queries are translated in the same way documents are translated. The result is obtained by the similarity of the query with the indexed items and by combining individual results using boolean combinations. The retrieval process uses a measure of similarity between the query index and the document index using statistical means.

Other statistical techniques have been used for the process of selecting words and phrases for indexing and for later use in retrieval by keywords. These methods are based on the observation that a good index term is distributed differently than a poor index term. The Poisson Distribution is usually considered to be a good method for identifying how a term is distributed over one or more documents. Commonly used models in this method include the Two–Poisson model (see [Sri90a] for a comparison of this model with the inverse document frequency method described above). In this method, the distribution of a term is described by two Poisson distributions, thereby specifying the manner in which an index term differs from a non-index term. The intuition behind this model is that a good index term will have a different distribution than a poor index term. Moreover, in the Two–Poisson model, the index terms will divide the documents into two components: (i) a component in which the index term is relevant and (ii) a component in which the index term is not relevant. Attempts at generalizing these models to more than two Poisson distributions have also been

made [Sri90b].

3.2 Structured Document Databases

This section describes approaches taken for query formulation and processing with document collections. As described in an earlier section (Section 1.1.2.2), traditional information retrieval techniques are directed primarily towards unstructured text using indexing techniques involving various linguistic and statistical analysis. These techniques do allow fast searches in large document repositories, but the lack of structures in the documents make it difficult to properly classify and organize search results. The existence of internal logical structure in documents makes it possible to use the structural information as schema for posing queries involving structure. This section discusses the current trends in research on structured document databases and, in particular, query processing with structured documents.

Current research approaches towards databases or query processing systems for structured documents can be classified in two categories based on the design process: (1) top-down and (2) bottom-up.

3.2.1 Top-down Approaches

In this approach, the design starts at the conceptual level, where a model of the database is formed. This stage leads to the design of operations based on the model and eventually leads to the low-level implementation of the model. The ongoing work using this approach can be further divided into two broad categories based on the underlying theory behind the work: (1) complex-object approach and (2) grammar-based approach.

3.2.1.1 Complex-object Approach

Desai et al. [DGS86] and Guting et al. [GZC89] used an algebraic approach with constructs for specifying queries in algebraic form, and with complex-object constructs like *set, unnest, project* and *group-by*. The algebraic approach gave the possibility of

optimization and rewriting or rephrasing queries. This method reflects the complex-object nature of documents, in which there are various instances of list, set, and bag-like structures.

Pistor et al. [PT86] used a declarative approach in which they extended the *Structured Query Language (SQL)* with complex sorts to formulate queries involving complex objects. In this work, the complex nature of the database was handled using different complex-object operators, embedded clauses, grouping objects, and other similar query constructs.

The problem with the complex-object approach lies in the fact that the query languages admit quantification over complex sorts. Quantification is a major problem since it results in increased expressiveness of the language, resulting in increased complexity of queries. Normally in databases, efforts are made to restrict the expressiveness query languages so that all queries can be solved in polynomial time (PTIME) and by using logarithmic amount of temporary space (LOGSPACE). However, by letting a query language quantify over complex sorts inherently introduces the possibility of explosive complexity.

In another variation of the complex-object approach, Abiteboul et al. [ACM93] and Christophides et al. [CACS94] used an object-oriented database to model textual data, particularly data encoded in the SGML [Gol90, ISO86] format. They used a mapping procedure to map the Data Type Definition (DTD) for the document into an object-oriented class definition in the language CO_2, which is the programming language in the object-oriented database environment O_2 [BBB+88]. In this mapping procedure, the document written in SGML was mapped to an instance of the class schema declared from the DTD. The query language associated with O_2 is then used to query the data.

The significant drawback of this system is its dependence on the capabilities of O_2 for both storing and querying documents. Documents are not used in their native form but mapped into instances of an object-oriented database. Since the mapping procedure was not straightforward, the authors had to alter the schema to fit the conversion procedure. This procedure is prone to loss of information contained in the original SGML documents, when any advanced SGML feature (*e.g.*, marked sections,

CONCUR, SUBDOC) is used.

A recent work by Zhang [Zha95] for building a dictionary system using an object-oriented database system can also be classified under this category.

3.2.1.2 Grammar-based Approach

This approach involves describing the database schema using Context Free Grammars (CFGs) or Attribute Grammars [Knu68]. Gonnet et al. [GT87] view the data model as a limited context-free grammar, and any database based on the model is formulated as a parse-tree of the grammar. The data model built on the grammar involves the use of ordered tuples and sets, lists, and union sorts. In this method, queries can easily be formulated using regular expressions satisfying the grammar. Gonnet et al. focus on dictionaries (*e.g.,* the New Oxford English Dictionary database), news clippings, legal documents, and other documents whose high degree of structuring makes them very difficult to represent in table format.

In another similar work, Gyssens et al. [GPG89] elaborate on the mathematical fundamentals of grammar-based models. They define algebra and calculus based on such grammars for various operations on the p-strings and the parse trees.

3.2.2 Bottom-up Approaches

In this approach, the design starts from the bottom with an implementation strategy (in terms of data models and data structures) laid out first, and the capabilities of the system depend heavily on this strategy. A data structure for modeling, storing and indexing the data is first decided upon, and operations are then provided that are efficient in utilizing the special data structure. In this section, we describe two prominent data structures that fall under this type of approach.

3.2.2.1 Patricia Trees

Patricia trees are based on the semi-infinite string (sistring) model of textual data [GBY91, BYG89]. In this model, textual data (whether structured or unstructured) is viewed as a string starting at each position of the text and continuing indefinitely

to the right. This model is primarily targeted towards unstructured textual data but can be adapted to text with structure. Patricia trees (often referred to as Pat trees) are digital trees in which the individual bits of the keys are used to decide on the branching. Pat trees are constructed over all the possible sistrings of a text collection. Thus, for a text of size n, there are n external nodes (or leaves) in the Pat tree and $n-1$ internal nodes, thus making the tree $O(n)$ in size. For example, the Patricia tree for the string 01100100010111 when the first eight sistrings have been inserted looks like Figure 7 (Figure taken from [BYG89]).

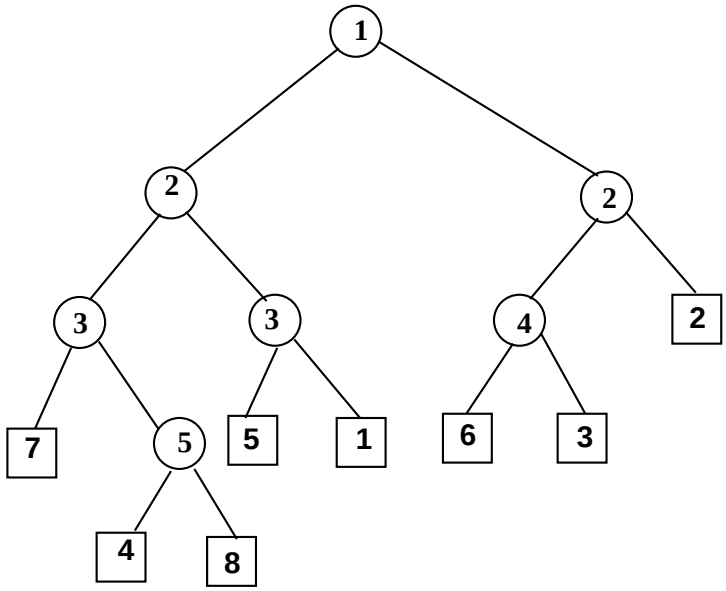


Figure 7: The Pat Tree for the string 0110010001011 after the insertion of the first eight sistrings

In Figure 7, the numbers in the leaves represent the position of the sistring at that node (*e.g.,* the leftmost leaf represents the seventh sistring 00010111 in the input starting at bit 7). The numbers in the non-terminals represent the actual bit position of the strings at that node (a skipped number indicates the skipped bit has no effect in the traversal). For every node, a bit value of 0 causes a traversal to the left branch, and to the right branch for a bit value of 1. The intuition behind the creation of the Patricia tree in this figure lies in the fact that the leaf nodes represent the complete sistring starting at the corresponding position, not simply the

position itself. For example, a prefix search for the string 010 causes a left, right and a left traversal in sequence to reach the leaf node marked 5, which represents the fifth sistring 0100010111.

Gonnet proposed a query language that includes the standard regular expressions defined by the operations of concatenation (.), union (+) and Kleene closure ($*$). In addition, the query language uses the operation *fby* to denote interposition of Σ^*.. Thus, $a\ fby\ b$ is the same as $a.\Sigma^*.b$. [BYG89]

Using a Patricia tree to encode all sistrings in text, Gonnet showed that many types of queries can be performed very efficiently. The most natural query using Pat trees is a prefix search, in which a query consisting of a single search string returns the positions of the document where the given string is a prefix of a possibly larger word. For a document of length n, an arbitrary prefix search can be performed in $O(log\ n)$ time, independent of the size of the answer. In practice, the length of the query is usually less than $O(log\ n)$, so the search time is proportional to the query length.

Patricia trees thus present a very efficient means for string searching in text documents. In addition to prefix searches, Pat trees can be used for proximity searching, range searching, longest repetition searching, "most frequent" searching as well as regular expression searching [GBY91]. Open Text Corporation uses this structure in their commercial structured search product for very efficient document searches [Ope94].

3.2.2.2 Concordance Lists

Burkowski [Bur92] proposed a concordance list structure for modeling hierarchically organized textual data. A concordance list is a special structure to keep track of the position and nesting properties of the various static contiguous extents, such as words and text elements. In formal terms, a text collection in this method is a finite sequence of n words $w_0, w_1, w_2, \ldots w_{n-1}$ of length n. A contiguous extent e is specified by two integers $\alpha(e)$ and $\omega(e)$ such that $0 \leq \alpha(e) \leq \omega(e) \leq n$. A concordance list $G = \{[\alpha(e_i), \omega(e_i)]\}_{i=1}^{m}$ is defined to be a set of bounds specifying disjoint contiguous extents. For example, consider the document fragment in Figure 8. Assuming the

words shown in this example (without the tags) number from 1, a fragment of the concordance list for the words "macbeth" and "witch" and the tags "<sp>" and "<v>" are shown in Table 5.

"macbeth"	**"witch"**	**<sp>**	**<v>**
[1,2]	[7,8]	[6,8]	[8,20]
[58,59]	[21,22]	[20,22]	[22,32]
[205,206]	[33,34]	[32,34]	[34,42]
[335, 336]	[43,44]	[42,44]	[44,47]
...	...	...	...

Table 5: A sample of the concordance list for the example document

Burkowski [Bur92] proposed an algebra based on this concordance list structure. The algebra consists of functions that take one or two concordance lists as operands and produce a new concordance list as a result. The language included operations for union, intersection and negation of the concordance lists. Clarke, Cormack and Burkowski [CCB95] generalized the concordance list structure to include nested and overlapping extents. The freely available *Sgrep* (Structured grep) system implemented by Jaakkola and Kilpeläinen [JK96] includes an extension of this GC-list structure and the associated algebra.

3.3 Semistructured Data

With the advent of the World Wide Web, the necessity of text document repositories has greatly increased. The documents on the web usually conform to a well-established structure defined by the HyperText Markup Language (HTML). However, HTML was designed as a simple language to introduce structure involving some logical document divisions (using heading tags) and some cross-referencing mechanisms (hyperlinks). HTML does not provide any way to define logical components of documents such as authors and affiliations. The structure of documents is more hidden in the physical formatting specifications such as indentations and typeface differentiations.

```
<p>
  <tp>MACBETH</tp>
  ...
  <ac>
     <ta>ACT I</ta>
     <sc>
        <ts>SCENE I</ts>
        <sv>
           <sp>First Witch</sp>
           <v>When shall we three meet again
              In thunder, lightning, or in rain?</v>
        </sv>
        <sv>
           <sp>Second Witch</sp>
           <v>When the hurlyburly's done,
               When the battle's lost and won.</v>
        </sv>
        <sv>
        ...
     </sc>
     <sc>
        <ts>SCENE II</ts>
        <sv>
           <sp>Duncan</sp>
           <v>What bloody man is that? He can ...
            ... state.</v>
        </sv>
        ...
     </sc>
   </ac>
   <ac>
   ...
</p>
```

Figure 8: A sample tagged document

Formally, the term semistructured data refers to data in which there is no separate mechanism for specifying the type (or structure) of the data. In most cases, the structure can be inferred from the manner in which documents are presented. In most cases, the best way to treat such a structure is as a labeled graph, while using languages associated with graphs to formulate queries. One of the most common formats for representing such data is the OEM model [PGMW95]. Recent work on extracting structure from such semistructured data, on designing query languages and on processing techniques has primarily been inspired by the recent growth of the WWW and by the presence of enormous quantities of data with little structure. A collection of such recent work can be obtained from [Suc97].

Although the object of the current work is to give database support for documents for which the structure is already well-established, the research on semistructured data provides us with the support necessary when we encounter documents that do not follow a well-defined structure. In this case, one of the structure inferring techniques described above can be used, first, to infer the document structure and, subsequently, to provide support for advanced query processing by using the inferred structure.

Chapter 4

Objectives and Requirements

The goal of this research is to demonstrate how the power of database techniques can be successfully applied to structured document repositories. This chapter focuses on the objectives and requirements for the database system component used primarily for the purpose of storing and managing structured documents. The goal here is to develop methodologies to effectively store such documents and to provide easy query formulation and efficient query evaluation mechanisms. Other database functionality such as concurrency control and recovery, inserts and updates are left as future work. The requirements consist of two parts: (1) functional requirements including data modeling, query language, and system requirements and (2) non-functional requirements including advanced database functionality and usability.

4.1 Functional Requirements

4.1.1 System Properties

As part of the functional requirements, a number of important properties of the system need to be satisfied. First, the design process has to be top-down, so that the capabilities of the system are not dependent upon any particular data representation. Second, the system needs to follow the traditional three-level architecture in traditional database systems described shortly. Third, the system needs to be able to use SGML documents directly without the necessity of conversion into another format. Here we present details on each of these requirements.

4.1.1.1 Top-down Design

The method for top-down design has already been presented in Chapter 3. In keeping with the top-down design method, we first decide on the conceptual nature of the data and its querying requirements; we then design physical data structure and access methods to satisfy these requirements. In Section 4.1.2, we present our basic data model based on the SGML document model. In Section 4.1.3, we describe the intended querying capabilities. The system specifics such as data structures and access methods are then selected to satisfy this design. In some cases, multiple alternative designs may need to be considered based on various trade-offs in efficiency and suitability.

4.1.1.2 Three-level Abstraction

One important motivation behind the use of database systems is to hide the actual handling of physical data from the user. To achieve this *information hiding*, the design of any database system involves multiple levels of abstraction. A standard way of leveled design includes three levels of abstraction: (1) physical, (2) conceptual and (3) external (views), as shown in Figure 9 [Ull88, Chapter 1].

At the topmost external or view level, users interact with different "views" of the data presented to them by the system. This level provides the possibility of creating views or subschemes. A view is an abstraction of a portion of the conceptual database and provides the user with a natural representation of the data. Different users may have different views of the same data or may be presented with only the relevant information pertaining to their interests. For example, students using a course registration system only need to see courses that they are registered for, while teachers of classes may need to see all students enrolled in their classes. In this case, the student and teacher will interact with two different views of the same data from the registration database. Actions on the views get mapped to equivalent actions to the conceptual database.

The conceptual level of the database provides a middle ground between the views and the physical data representation. This level provides a conceptual model of

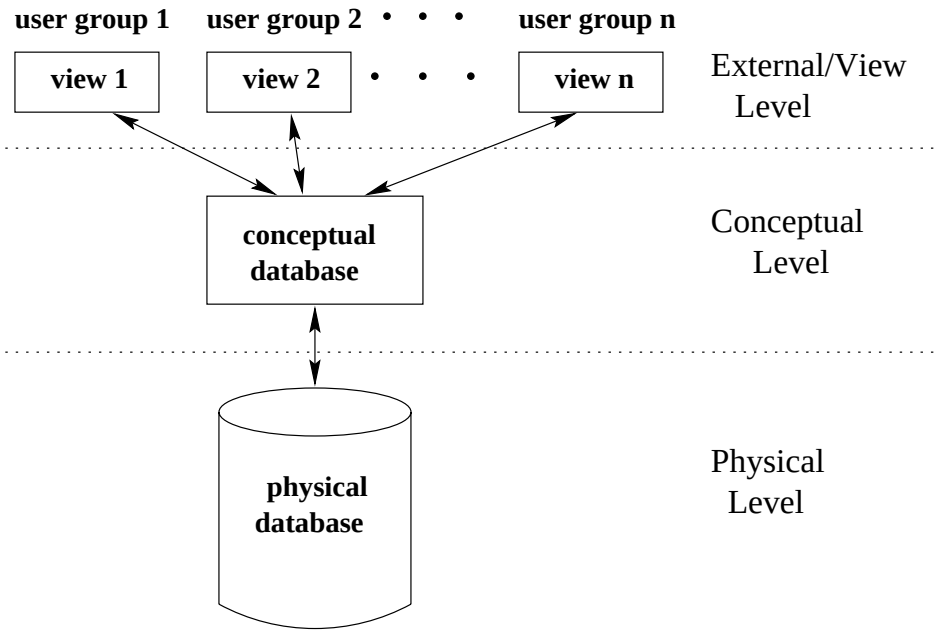


Figure 9: Levels of abstraction in a database system

the data which is independent of the physical data structures (enabling information hiding), but includes all the details of the managed data. The primary components in this level include a conceptual data model and query languages (declarative or procedural) that allow operations on the data in its conceptual form. The database system usually provides a manipulation language based on the conceptual model to add, update and search for data from the system. This level and the languages related to it support the data model for the underlying database system (such as the relational model for relational database systems and object-oriented model for object-oriented database systems).

The physical level is the lowest level in a database system. This is the level closest to the operating system, and it deals directly with physical resources and data representation on physical storage devices. This level includes the data structures and access methods for efficient querying and update of the physical data, in support of the query languages in the conceptual level.

A document database system needs to have a similar multi-level abstraction to keep the internal details of the system hidden from the users. At the physical level,

	Relational databases	**Document databases**
View Level	Relational views	Views of document components and the visual presentation of these components
Conceptual level	Relational model, relational algebra and calculus	SGML document model, extended relational algebra and calculus
Physical level	physical data structures to support relational operations	SGML documents, parse structures and indices

Table 6: Comparison of the levels of abstraction for relational and document databases

documents are stored in their standard SGML format, but parsed structures and indices are built to augment the documents. At the conceptual level, we incorporate SGML in the data model (Section 4.1.2) and provide query languages similar to those in the relational domain to manipulate the documents. Query results are presented as SGML documents and can be expressed as document fragments in a similar manner as relational views. Table 6 displays the equivalent levels in relational databases and DocBase.

4.1.1.3 Native Data Representation Format

SGML documents stored as regular text files on a filesystem do not provide efficient means for processing because of the sequential nature of such files. However, it is practically the only way information can be interchanged between systems and platforms without modification. Although other document formats such as plain text, word processor formats, postscript [Sys85], PDF (portable document format) [BCM96] have been used for document publication and distribution, none of them has been able to deliver the full capability of electronic documents. As discussed in Chapter 1, although plain text is possibly the most portable method of interchanging documents, its applicability is quite limited. Word processor files are useful only if the corresponding software is available in a particular platform. Postscript is a language primarily for describing the visual image of documents and, hence, not ideal for a

uniform and generic document description language.

PDF has some layout as well as structural information and is often used as a means for document interchange. It uses a page-oriented view of documents and includes information pertaining to the visual appearance of each page of the document. Recent revisions of PDF also allows inclusion of "thumbnails," "hyperlinks" and limited amount of structural information such as table of content levels. The applicability of PDF as a document database format, however, is highly questionable because of the overhead of processing PDF documents and the lack of schema information to validate database instances.

SGML uses a plain text format to encode the document. Characters and markup that cannot be represented by a text character are encoded by "entities" (see Chapter 2) and by character set references, which only use plain text character representations. Thus, SGML documents can be exchanged between platforms and systems without any modifications. In the recent years, there has been an increased use of SGML for the production and interchange of documents such as technical manuals, electronic texts and journals, and electronic theses and dissertations. Usually these document collections are distributed in read-only media such as CD-ROMs and can be used directly from these CDs.

One of the main consideration in this research is to leave the original documents intact and build only secondary data structures on top of these documents to facilitate processing. The use of such indices enables the referred documents to stay in their original locations (such as CDs, disks and other secondary or tertiary storage media) and only be used in the final stage of query processing to extract fragments of the original documents. Also, conversion or mapping of documents into a database format and replicating the information in the database becomes unnecessary if external indices are present. Moreover, if documents are replicated in other database formats, updates to the original documents can be expensive, since the replicated data will also need to be updated. On the other hand, an updated document will only necessitate the updating of the index structures, if only external indices are used.

4.1.2 Data Model

Conceptual modeling of data is always one of the primary steps in a database design process. Relational database systems follow the relational model in which the conceptual structure is represented as a set of tables. In this research, the data model is based on SGML [ISO86]. In SGML, the schema of the data is initially defined in terms of a DTD (Document Type Definition), and documents are then created based on the defined schema. Every valid SGML document has two components: (i) the first component indicates the DTD (often shared across many documents) which describes its document structure and (ii) the rest describing the actual structure of the document instance. Hence, all documents based on a given DTD belong to a language specified by the grammar defined in the DTD.

As described in an earlier chapter (Chapter 2), SGML was designed primarily as a system-independent and platform-independent document description language for the purpose of interchange. The design was motivated primarily from a linguistic and publishing point of view, and hence, the language does not directly conform to a single known formal model. One close match is the extended context-free grammar by Madsen [MK76], but it fails to model all the features and properties of SGML. However, because of the standardized nature of SGML and the years of research based on SGML, the absence of a formal model for SGML does not cause problems in building formal models based on SGML. The recent resurgence of the use of SGML in serious document production and delivery applications, the growth of the World Wide Web, and the upcoming XML (Extensible Markup Language) standard for the web all add to the importance of having database support for SGML. Our formalism is based on SGML as a database model.

4.1.2.1 Structured Document Databases

To define the notion of structured document databases, we first define a few sets: **gi** is a countably infinite set of generic identifiers (GIs); **doc** $\subseteq$ **gi** is the set of document types (we describe later the reason for making this distinction), and **att** is a countably infinite set of SGML attributes. We also define **dom**, which is a countably infinite

set of constant character strings, and **var**, a countably infinite set of variables.

Types We only consider two types:

1. *Basic type β.* The base type comprises of character strings, with **dom** being the range of values. SGML only supports characters as element contents, and although it supports limited types in attribute values, we do not consider attributes as an integral part of our language at present.

2. *Complex type τ.* The complex types form the set **gi**. Within this set, the subset **doc** defines document types which are special complex types in the database. Document types are considered to be special complex types because they define the type of a complete document for a given DTD. However, any GI in a DTD defines a complex type, and new DTDs may be constructed rooted at that particular GI, thus making it a document type.

Documents and Databases Documents form the core component in a document database system. We define documents and databases consisting of documents as follows:

- *Document.* Intuitively, a document is an SGML instance of a DTD. A DTD can be modeled as a grammar represented by a quintuple $d = (\tau, \mathcal{G}, \mathcal{A}, \mathcal{C}, \mathcal{P})$ where $\tau \in$ **doc** is a document type, $\mathcal{G} \subset$ **gi** is a set of generic identifiers, $\mathcal{A} \subset$ **att** is a set of SGML attributes and $\mathcal{C} \subset$ **dom** is a set of constants. $\mathcal{P}$ is a set of production rules describing the structure of conforming document instances. Analogous to the relational database model, the DTD serves as the schema for the managed data, and documents conforming to the DTD serve as instances of the schema.

- *Database. A database, in this setting, is a finite set of SGML documents conforming to one of the document type definitions in* **doc**.

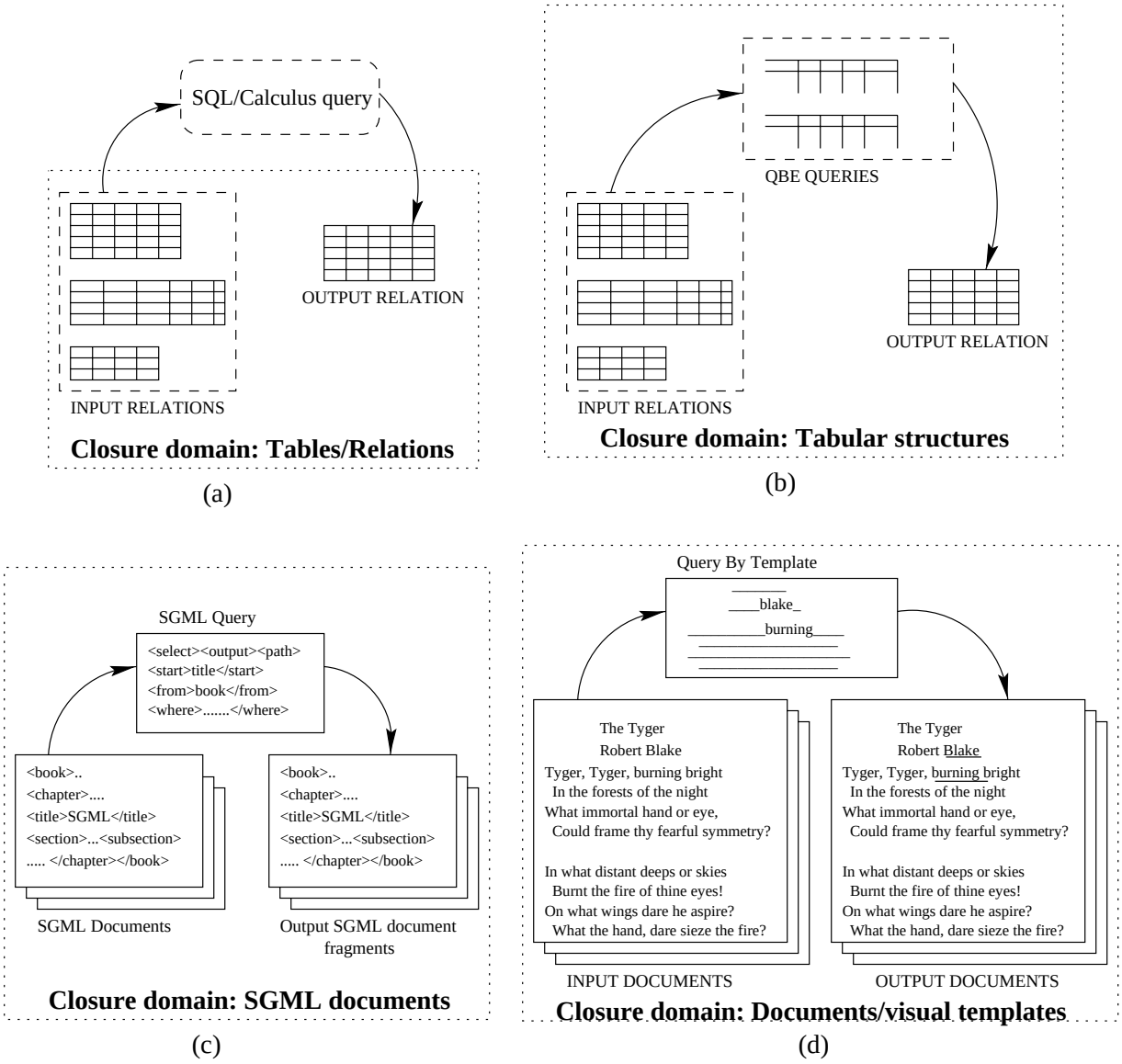


Figure 10: Examples of closure: (a)Relational databases with SQL or relational calculus; (b) Relational databases with QBE; (c) SGML documents with an SGML query language; and (d) SGML databases with a template-based query language

4.1.2.2 Closure

In addition to using SGML for the storage and modeling of the data, one primary issue in this research is to achieve closure in the domain of SGML. Simply put, this means that SGML is to be used as the input as well as the output format for queries. Closure is prominent in the relational database model as tables are both inputs to queries and outputs from queries, as shown in Figure 10(a). In addition, using the QBE query language, queries are formulated also using tables, as in Figure 10(b). The primary reasons for the use of a closed-domain system are the following:

- Closure enables the possibility of using *views* in the same manner as the data itself, thus allowing views to be used as inputs to further query processing.
- Closure allows the input and output to be handled in the same way; no separate mechanisms need to be devised for presentation and storage of query results, as they can inherit such properties from the input data itself.

In addition to the above primary properties, the effectiveness of closure can be heightened by having queries in the same format as the input and output data, and hence enable the possibility of storing, presenting and managing queries in the same way as the data. Queries on such stored queries can be used for future performance tuning. Moreover, the capability of treating queries as data gives a means for adding reflection properties to a query language [JMG95].

Traditional information retrieval systems deal with unstructured text. In this case, the idea of closure does not have much importance, since the query results do not need to follow any structure of the input documents. However, this also restricts the presentation of the results, since different ways of presenting the results individually or collectively need to be designed. Some current document database systems [Inf95, Hol95] translate structured documents into a standard database model. Queries are performed against this database, and the results are presented in the format of the host database system, thus violating closure. Some similar systems (such as in [CACS94]) can translate the query results back to the original format by performing a reverse mapping method. Although this method achieves closure, in many

cases the resulting documents do not retain the complete structural information in the original documents.

Many of the properties of closure can be achieved in these systems by restricting updates to the host database format only, but the method is still expensive and requires discarding the original documents — a requirement that is often not feasible for very large document repositories. In the current research, we intend to achieve closure by using the document structure in query languages as well as input and output presentations. We can use the document structure in the form of document representation (as in Figure 10c) and in the form of document layout (as in Figure 10d).

4.1.3 Query Languages

In Chapter 1, we described two primary approaches to document databases: top-down and bottom-up approaches. Our approach falls in the former category. To remain consistent with this approach, our design starts from a conceptual view of the database and the class of queries on the data, followed by a query language for these queries, and finally low-level processing methodologies for supporting these queries. The actual query language will be described in the next chapter (Chapter 5). In this section, we describe the types of queries we intend to solve and the basic properties of query languages that we can use.

In Chapter 3, we described some current systems and techniques applied for providing database querying functionality for text documents. Among these, the research at INRIA [ACM93, CACS94] provides the most complete method of querying, owing to the use of an object-oriented query language for data modeled and stored using an object-oriented database. The methods that require the conversion of the textual data into a standard database format usually have sufficiently complete query languages because of their use of the query language provided by the host database system. The bottom-up approaches, on the other hand, suffer from the fact that the query language they support is usually restricted to the operations that the physical data structure provides. For this reason, the Patricia tree system can evaluate certain types of queries very efficiently, but generalizing its capability to pose standard

database queries common in the relational world can be quite difficult.

Types of queries that are usually provided by standard database systems and are quite relevant in document database settings include simple selections, projections, joins, negations, counting, grouping and ordering, and nested queries (see Chapter 1 for examples of such queries). All these queries can be expressed in query languages using first order logic. The query language requirements for this research, therefore, involve the following:

- The query language should be expressible in first order logic, similar to the relational query languages.
- The query language should be within PTIME, and possibly within LOGSPACE. In other words, any query expressed in the query languages can be evaluated in an amount of time proportional to a polynomial of the size of the input, and using an amount of temporary space proportional to a logarithm of the size of the input.
- The query language should be powerful enough to express queries commonly used with document databases, such as the ones described in Chapter 1.
- The query language should have at least one visual equivalent so that users can express their queries without knowing the specific syntax constraints of a language and without knowing the complete database schema.

4.2 Non-functional Requirements

4.2.1 Usability Requirements

In Chapter 2, we introduced the concept of "designing for usability." Although functionality is perhaps more important in terms of requirements, we put a great deal of importance on making the system significantly easy to use. In order to provide a good measure for usability of the system, the following factors need to be considered:

- *Efficiency.* How fast can the users pose a query and retrieve desired results?

- *Accuracy.* How accurately can the users pose queries, given an English statement of the query?
- *Satisfaction.* How satisfied are the users with the interface and its performance and functionality?

The usability analysis should be performed by comparing the proposed system interface with a commonly used interface for similar purposes. We selected forms to be an obvious choice for this comparison. We would like our system to be at least equal, if not exceed, the performance and usability of forms.

4.2.2 Advanced Database Requirements

In addition to the modeling, language and system requirements described in the previous section, a number of other properties are also desired for a database system for documents. Most of these properties have already been researched and systems exist for such purposes. The most prominent database features include:

- *Concurrency control and recovery.* When a document repository is not fixed and is constantly being changed by authors and maintainers, it is necessary to control concurrent access to the documents so that data is not lost from conflicting operations [BHG87]. Moreover, in case of a system crash during an update operation to the database, the system needs to gracefully recover from the crash and retain consistency of the data.
- *Version control and collaborative authoring.* One essential feature of a document repository is the ability to track versions of different documents. This is even more important when multiple users are given the task of authoring the same document. Revision tracking mechanisms for program code and plain text authoring systems (such as RCS [Tic85]) are often not powerful enough to track revisions on structured documents. Systems like SGML Editor from Grif S.A. (http://www.grif.fr) have been designed to track revisions between the actual document structures, not just the text.

- *Integrated authoring and database functionality.* Document database systems that use a standard database system for storing documents (such as in [CACS94, Hol95, Inf95]) need to provide a mechanism for authoring the documents in a transparent manner so that users do not realize that the documents are stored in a database in fragments and only presented to them as a whole. Since our approach is to keep documents as is in the database, we do not need this functionality; users can use existing authoring mechanisms to author and update the documents. However, a connection between these authoring tools and the database needs to be maintained so that the database system can update the necessary structures and indices when an update is made to the documents. A number of methods have been proposed for interfaces between editors and databases that allow such database connections between authoring tools and backend databases. Premier among these are Informix^{TM} datablades from ArborText (http://www.arbortext.com) and GATE (Grif Application Toolkit Environment) from Grif S.A. (http://www.grif.fr).

Chapter 5

Conceptual Design

This chapter describes the design of the DocBase document database management system. As described in Chapter 4, a database system usually includes three layers of abstraction. We described the conceptual data model supported by a database system for structured documents. Here we describe the design of formal as well as practical query languages based on this model and highlight the relationships between these languages.

5.1 Formal Query Languages

The design of query languages in this research is based on extensions torelational query languages . In Chapters 1 and 2, we have identified the problems that make the relational model unsuitable for describing hierarchical document structures. We also noted that the relational query languages have many useful properties that are necessary in document query languages. Thus, a document query language designed as an extension of relational query languages can include all the functionality of the relational languages and add the features necessary to accommodate the hierarchical data format. In this section, we describe two equivalent formal query languages: a calculus and an algebraic language. The first language is Document Calculus (DC): a declarative query language for documents developed as an extended form of the relational calculus allowing complex path terms. The second language is Document Algebra (DA): a procedural language which uses specialized operators on sets of documents to specify queries. While the calculus provides a logical description of the language, the algebra is useful for implementing the language, describing the complexity and providing a good understanding of the operations available in the

language.

5.1.1 A Document Calculus (DC)

Here we describe a calculus for documents. The notion of document databases has been formally introduced earlier (Chapter 4). As stated before, this calculus is an extended form of relational calculus supporting path expressions and operations on document structures. To reduce the complexity of the query language, we use a simplified path expression construct and compare the proposed path expression constructs with traditional notions of path expressions. We then define the language by defining the terms, operators, predicates, formulas, and finally, queries in the language.

5.1.1.1 Path Expressions

The notion of path expressions (PEs) came from two different areas: (i)graph query language s and (ii) object-oriented query languages. For graph query languages (*e.g.*, [MW95]), a path expression defines a path from one node in the graph to another in terms of intermediate node and edge labels. For object-oriented query languages, a path expression defines a path from one object to another using membership and inheritance relationships. We describe both of these methods and propose a simplified path expression construct specifically for structured documents.

Traditional PEs in Graphs Path expressions in graphs are typically used in navigation-oriented queries. This technique has been applied in hypertext data as well as object-oriented languages. Typically, such queries are expressed using regular expressions denoting paths in the graph representing the data [AV97]. In this setting, the data is organized in the form of nodes linked by labeled edges. A notion of path queries can be built using a similar approach for defining regular expressions [HU79]. Abiteboul and Viannu [AV97] describe their path queries in terms of *regular path queries*, and then generalize them to *general path queries* as follows:

Regular path queries A regular path query is defined as a regular expression over some finite alphabet Σ consisting of the set of labels in the graph. This notion

uses the same semantics for regular expression symbols, such as "+" representing union and "*" representing the Kleene closure. Examples of such path queries are:

```
section(paragraph + figure)caption
engine (subpart)* name
```

The first instance refers to a path from a section to captions of paragraphs or figures in the section, where the second path refers to the names of any subpart within an engine.

General path queries In regular path expressions, all labels need to be included, and there is no easy way to express a "don't care" expression. An extension to the above regular path queries can be expressed by allowing string regular expressions *inside* the labels in the path. Abiteboul and Vianu [AV97] term such path expressions as *general path queries*. Examples of such path queries include:

```
"doc" ("[sS]ections?" "text" + "[pP]aragraph")
"section" (".*")* "caption"
```

General path expressions allow *perl*-style regular expressions *inside* the labels of the path. In the above examples, the items within the quotes indicate labels in the path expression. In the first example, the path refers to section text or paragraphs inside a document (ignoring the starting case of section and paragraph). The more interesting example is the second one, in which the path expression denotes paths from *section* nodes to *caption* nodes, without specifying the intermediate labels in these paths.

PEs in Object-Oriented Databases The concept of path expressions in object-oriented query languages started with the necessity for abbreviating expressions involving long chains of membership or inheritance relationships [KKS92, dBV93]. Traditionally, a path expression over a given OODB schema is an expression of the form

$x.A_1.A_2. \ldots .A_n, n \geq 1$, where x is an instance of class C_1, A_1 is an attribute of the class C_1, and for $1 < i \leq n$, the type of the attribute A_{i-1} in the class C_{i-1} is the class C_i, and A_i is an attribute of that class C_i. This definition of path expressions leads to several unnecessarily long expressions describing paths, even between object pairs for which only one unique path exists.

Generalizations of the above path expressions have also been proposed [KKS92]. In one such approach, Van den Bussche and Vossen [dBV93] simplified the fully expanded path expressions to partial path expressions without any change in the syntax, extending the syntax of the "." operator to incorporate an unspecified path. During computation, the paths are expanded to a minimal path between the object instances. Another approach by Kifer et al. [KKS92] uses the path labels as lists and allows the selection of particular instances of labels among a set of such labels. (*e.g.,* the expression `Book.Chapter[1].title` refers to the title of the first chapter of a book.)

PEs in Document Structure Path expression for document structures have also been considered by Christophides et al. [CACS94, CCM96]. These approaches allow variables on paths, which instantiate over paths in the current domain. In this scenario, it is possible to formulate such expressions as $node1.X = node2.Y$ where $node1$ and $node2$ represent tree nodes, and X, Y represent path variables. This expression evaluates to true if any instance of the paths X and Y results in an equality of the two sides. In the cases where the two path instances result in incompatible types, an error condition is generated, and the corresponding instances are ignored. Christophides et al. also propose a syntactic sugared path expression of the form $my\_article..title(t)$ when the actual values of path variables are not important. Formally, the above expression evaluates to:

$$\exists p \ \mathrm{PATH}(p) \wedge my\_article.p.title(t)$$

Here, the expression indicates that there is a path between $my\_article$ and $title(t)$, but the actual path itself is not of significance.

Simple Path Expressions (SPE) We are interested primarily in posing queries in document databases. Although completely generalized path expressions with path variables and selectors give a lot of power to a language, they are often not necessary in a document context, especially for SGML documents. We make the following simplifying observations:

- Document instances of all SGML DTDs are strictly hierarchical. Although there is a feature in SGML called CONCUR that allows multiple simultaneous hierarchical structures, any particular document instance only conforms to one of these structures. In general, thus it can be assumed that a particular instance of a document has a strictly hierarchical structure (*i.e.,* an element cannot be a direct child of more than one other element, and an element cannot be an ancestor of a parent element). This property is, in general, not true for object-oriented databases.

- Although SGML allows the use of IDREF attributes to link to other elements, they are not part of the document structure. In this work, we are using path expressions only for structural navigation and not for pointer/link chasing. In other words, path expressions in this language would be used as a means for describing an abstract link between two nodes in the structure itself, rather than an explicit path following pointers in an instance of that structure. A method for following links will be discussed later.

- Document structures defined in SGML are usually non-recursive. SGML does allow recursive structures, but in documents, it is usually a common practice to use descriptive tags rather than recursive structures. For example, a document with a chapter-section-subsection-subsubsection hierarchy can be represented in a recursive section (see Figure 11), with the semantics given to the depth of recursion, although it is seldom structured that way.

Given the above assumptions, we propose the notion of simple path expressions (SPE) using the operators "." and "..", formally defined shortly with the semantics as described above. Note that we use the context of documents defined earlier, with **gi**,

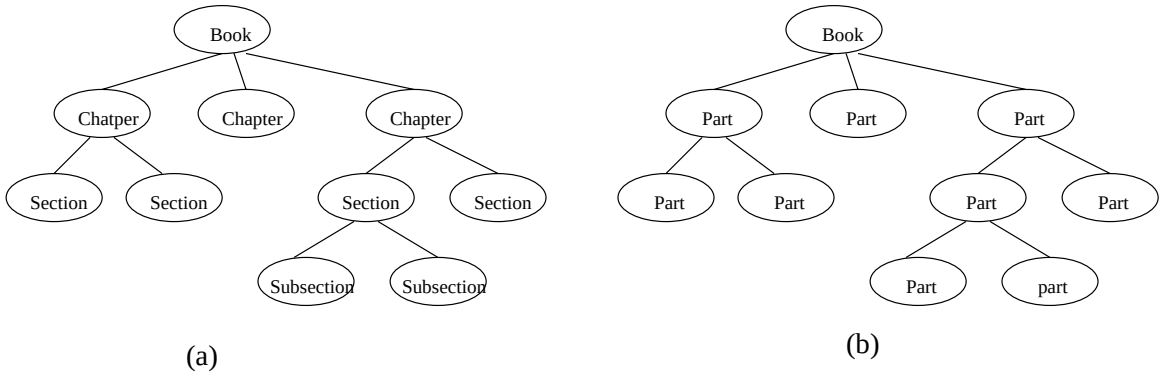


Figure 11: Two ways of structuring a book — (a) without using recursion, and (b) using recursion.

the set of generic identifiers; **doc**, the set of document types; **att**, a set of SGML attributes; and **dom**, which is a set of constants. Note that in general, all paths need to be specified relative to a particular document type definition (DTD) and should use only the generic identifiers provided by the DTD. We now define the notion of simple path expressions (relative to the DTD D) and two useful functions $first(\mathcal{P})$ and $last(\mathcal{P})$ ($first(\mathcal{P}), last(\mathcal{P}) : \mathbf{SPE} \rightarrow \mathbf{gi} \cup \{\epsilon\}$). The definition is given inductively as follows:

- *Null path.* ϵ, the null path, is an SPE. $first(\epsilon) = last(\epsilon) = \epsilon$
- *Basic path.* Every $x \in \mathbf{gi}$ is an SPE, and we call these paths "basic paths." $first(x) = last(x) = x$.
- *Listed path*: A listed path $\mathcal{P}$ is a fully expanded SPE of the form $\mathcal{A}_1.\mathcal{A}_2. \cdots .\mathcal{A}_n$ where $n \geq 1$, and for every i, $\mathcal{A}_i \in \mathbf{gi}$, and $\mathcal{A}_{i+1}$ is in the *content group*[1] of $\mathcal{A}_i$. Here, $first(\mathcal{P}) = \mathcal{A}_1$ and $last(\mathcal{P}) = \mathcal{A}_n$.
- *Abbreviated path.* An abbreviated path $\mathcal{P}$ of the form $\mathcal{P}_1..\mathcal{P}_2.. \cdots ..\mathcal{P}_k$ is an SPE, where $k \geq 1$, and for every i, $\mathcal{P}_i$ is a basic or listed path, and $first(\mathcal{P}_{i+1})$ is a descendant of $last(\mathcal{P}_i)$ in the context of the current DTD. This notion is

[1] In SGML, the declaration of a generic identifier is written as: `<!ELEMENT thisgi - - (G1, G2, G3...)>`. The GIs within parenthesis are the children of thisgi, and are said to be in thisgi's content group.

similar to the partial path specification in [dBV93] and the ".." operator in [CACS94], described above. Note that the operation ".." can be defined in terms of the operation "." as follows:

$$\mathcal{P}_1..\mathcal{P}_2 \triangleq \exists \mathcal{A}_1, \ldots \exists \mathcal{A}_k \; \mathcal{A}_1 \in \mathbf{gi} \wedge \ldots \wedge \mathcal{A}_k \in \mathbf{gi} \wedge \mathcal{P}_1.\mathcal{A}_1.\mathcal{A}_2.\cdots.\mathcal{A}_k.\mathcal{P}_2 \text{ for some } k \geq 0$$

Note that in case of an abbreviated path $\mathcal{P}$ of the above form, $first(\mathcal{P}) = first(\mathcal{P}_1)$ and $last(\mathcal{P}) = last(\mathcal{P}_k)$.

The above definition of SPEs can also be demonstrated using an equivalent BNF notation for the sake of clarity, as follows:

$$\begin{aligned} SPE &::= AbbrPath \mid \epsilon \\ AbbrPath &::= ListedPath \; \{.. \; Listedpath\}^* \\ Listedpath &::= BasicPath \; \{. \; BasicPath\}^* \\ BasicPath &::= \mathbf{gi} \end{aligned}$$

Based on the above definitions of path expressions, and the predicates $first$ and $last$, we define a few special path expressions as follows:

- *Rooted SPE.* A rooted SPE $\mathcal{P}$ is an SPE where $first(\mathcal{P}) \in \mathbf{doc}$.
- *Terminal SPE.* A terminal SPE $\mathcal{P}$ is an SPE where $last(\mathcal{P})$ has a data content (*i.e.,* one of the children of $last(\mathcal{P})$ is `#PCDATA` or any other possible character data types in SGML).
- *Complete SPE.* A complete SPE is an SPE which is both *Rooted* and *Terminal* (*i.e.,* $first(\mathcal{P}) \in \mathbf{doc}$ and $last(\mathcal{P})$ has a data content).

Semantics of SPEs As mentioned above, path expressions are always in the context of a DTD. Posit an interpretation $\mathbb{M}$ of a database for a given DTD D and an environment (assignment of values to all variables) η. The DTD D represents a set of documents conforming to the DTD and is hence similar to a complex relation. In

this interpretation, $\mathbb{M}(D)$ is a set of documents conforming to the DTD D. A path expression $\mathcal{P}$ applied to a set of documents is a function from a set of documents to another set of documents rooted at $last(\mathcal{P})$. We use the following notation to describe fragments of trees[2] which will be used in the definition of the semantics:

$$\begin{array}{c} A_1 \\ A_2 \\ \vdots \\ A_k \\ \triangle \end{array} \quad \text{matches any tree rooted at } A_1\text{, that it has a path } A_1.A_2 \cdots .A_k.$$

Formally, the interpretation of a path expression E in the context of a DTD D can be defined inductively as follows, assuming the existence of an interpretation $\mathbb{M}$ of the database which is essentially a finite set of documents conforming to the DTD D.

- $[\![\epsilon]\!]^{\mathbb{M}} = \mathbb{M}(D)$
- $[\![A]\!]^{\mathbb{M}} = \left\{ \begin{array}{c} A \\ \triangle \end{array} \mid \exists A_1, A_2, \ldots A_k \in \mathbf{gi} \begin{array}{c} A_1 \\ A_2 \\ \vdots \\ A_k \\ A \\ \triangle \end{array} \in \mathbb{M}(D), k \geq 0 \right\}$
- $[\![\mathcal{P}.A]\!]^{\mathbb{M}} = \left\{ \begin{array}{c} A \\ \triangle \end{array} \mid \exists B \in \mathbf{gi} \begin{array}{c} B \\ A \\ \triangle \end{array} \in [\![\mathcal{P}]\!]^{\mathbb{M}} \right\}$
- $[\![\mathcal{P}..A]\!]^{\mathbb{M}} = \left\{ \begin{array}{c} A \\ \triangle \end{array} \mid \exists A_1, A_2, \ldots A_k \in \mathbf{gi} \begin{array}{c} A_1 \\ \vdots \\ A_k \\ A \\ \triangle \end{array} \in [\![\mathcal{P}]\!]^{\mathbb{M}}, k > 0 \right\}.$

[2]Notice that in this tree fragment, the path is a listed path (*i.e.,* all nodes in the path are specified), and that A_2 is not the only child of A_1 (A_1 may have other children, but we are not interested in them).

Comparison of SPEs with general PEs Since SPEs only include direct child relationships and descendant relationships, it is not as powerful as the regular path expressions described earlier. For example, the simple path expression construct does not permit paths with arbitrary Kleene closures (such as `A.(B.C)*.D`). Clearly, SPEs describe a subset of the general path queries, since any SPE can be expressed using an general path query. With the simplifying notion of strict hierarchical documents with infrequent recursive structures, it is still possible to pose many interesting queries using these simplified path expressions. In the next section, we will examine the properties of a calculus language using this notion of path expression and see the types of queries that can be formulated using this language.

5.1.1.2 A Formal Specification of DC

Based on the above discussion of path expressions, we now discuss the document calculus (DC) as an extension of the relational calculus that can use SPEs as terms. We first describe the language by defining the accepted terms in the language and the operators that are supported in the language. We next define the predicates and formulas in the language, and finally the queries in the calculus specification. In this specification, we use the same formalism of document databases (as presented in Chapter 4) as sets of documents conforming to the schema represented by a DTD. As before, we use the quintuple $d = (\tau, \mathcal{G}, \mathcal{A}, \mathcal{C}, \mathcal{P})$ with the usual significance of the symbols. Recall also that the types of terms can only be one of two types: simple types (character strings) and complex types (governed by one of the generic identifiers). We will use the symbol τ for types, and the symbol $\circ$ to represent one of the path expression operators **.** and **..** in the following discussion.

Terms Terms in DC comprise of the following:

- *Constant.* A constant $c \in$ **dom** is a term.
- *Variable.* A variable $x_\tau \in$ **var** is a term, representing a tree of a given type $\tau \in$ **gi**. If the type is implied, the suffix of x_τ may be dropped.

- *Path term.* An expression of the form $x \circ \mathcal{P}$ where $\circ \in \{. \ , \ ..\}$, referred to as a *path term*, is a term in the language representing a set of trees obtained by traversing the path $\mathcal{P}$ starting from the root of the tree denoted by x. Semantics of this operation is given shortly. The type of a term $x \circ \mathcal{P}$ is given by $last(\mathcal{P})$ if $\mathcal{P} \neq \epsilon$ and is given by the type of x otherwise.

Operators Basic comparison operators and logical operations are supported in this language. The following operations are supported in particular:

- *Comparison operators.* All comparison operators are binary operators, and are functions that return a boolean value (`true` or `false`). Two types of comparison operators are used in DC:
  - *Comparison between sets and atoms.* The operators $\ni$ and $\not\ni$ can be used to perform comparisons between a set and an atom. The set to be compared must have the type of a generic identifier that has a data content (*i.e.,* theSPE denoting the set must be a terminal SPE).
  - *Comparison between sets.* The set comparison operators in DC are $\{=, \neq, \subseteq, \subsetneq, \cap, \not\cap\}$. The first four operators are the standard set equality, inequality, subset and non-subset operators. The operations $\cap$ and $\not\cap$ are defined by:

$$\begin{aligned} A \cap B &\triangleq A \cap B \neq \emptyset \\ A \not\cap B &\triangleq A \cap B = \emptyset \end{aligned}$$

- *Logical operators.* The logical operators supported by this language are $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT).

Predicates The predicates supported are document and path predicates (which can be thought of as complex relational predicates), defined as follows:

- *Document predicates.* $D \in$ **doc** is a document predicate and represents a set of documents conforming the document type D.

- *Path predicates.* A path predicate is of the form $D \circ \mathcal{P}$ where $D \in \mathbf{doc}$, and $\mathcal{P}$ is an SPE. The path predicates represent sets of documents rooted at $last(\mathcal{P})$ if $\mathcal{P}$ is non-null and rooted at the root generic identifier of D if $\mathcal{P}$ is ϵ.

- *Path term predicates.* Since path terms described above represent sets of documents, they can also be treated as predicates. A path term predicate is of the form $x \circ \mathcal{P}$, as above.

Formulas Formulas are functions from a set of variables to the boolean values `true` and `false`. A formula is a function from a valuation of a set of *free* variables to one of the two boolean outcomes. The formulas in DC include the following:

1. *Atomic formulas.* $\mathcal{R}(x)$ is an atomic DC formula, where $\mathcal{R}$ is a predicate, with the following forms:

   (a) $D(x)$, where x is the only free variable, and $D \in \mathbf{doc}$. In this formula, x must be a variable of type D, where D is the root generic identifier of the DTD D.

   (b) $D \circ \mathcal{P}(x)$, where x is the only free variable, and $\mathcal{P}$ is an SPE. Here the variable x must be of type $last(\mathcal{P})$ if $\mathcal{P}$ is non-null. If $\mathcal{P} = \epsilon$, this formula reduces to the formula above.

   (c) $x \circ \mathcal{P}(y)$ where x and y are the two free variables. As before, the variable y needs to be of type $last(\mathcal{P})$ if $\mathcal{P}$ is non-null, and the same type of x otherwise.

2. $x \circ \mathcal{P}\theta c$ is a DC formula, where $\theta \in \{\ni, \not\ni\}$, $x \circ \mathcal{P}$ is a path term and $c \in \mathbf{dom}$ is a constant. In order for this comparison to make sense, $last(P)$ needs to have a data group, and the semantics of this formula is to compare the data in the data group of the term with the constant. In this formula, x is the only free variable.

3. $t_1 \theta t_2$ is a DC formula, where $\theta \in \{=, \neq, \subseteq, \subsetneq, \cap, \not\cap\}$ and $t_1 = x_1 \circ \mathcal{P}_1$ and $t_2 = x_2 \circ \mathcal{P}_2$ are two path terms. Although in practice, terms could refer to any

complex type, a comparison between two complex terms involves comparisons between trees. For the purpose of this formalism, we will consider all terms to be path terms involving complete SPEs. In other words, for every term t, $first(t) \in$ **doc** and $last(t)$ has a data group (the predicates $first$ and $last$ for such term can be defined in the same manner they were defined for path expressions). In this formula, x_1 and x_2 are both free variables.

4. If φ and ψ are formulas, so are the following:

   - $\varphi \vee \psi$
   - $\varphi \wedge \psi$
   - $\neg\varphi$

   In the above, the set of free variables is the union of the sets of free variables in φ and ψ.

5. If $\varphi(x, x_1, x_2, \ldots, x_n)$ is a DC formula with $n+1$ free variables $x, x_1, x_2, \ldots x_n (n \geq 1)$ then the following are DC formulas:

   - $\exists x\ \varphi(x, x_1, x_2, \ldots, x_n)$ (existential quantifier).
   - $\forall x\ \varphi(x, x_1, x_2, \ldots, x_n)$ (universal quantifier

   In each of the above two forms, the free variables are $x_1, x_2, \ldots, x_n$. The variable x is said to be *bound* by the corresponding quantification operation.

6. If φ is a formula, so is (φ). The set of free variables remains unchanged.

Formulas are the primary means for expressing queries in any calculus language. A formula intuitively represents the values given to the free variables that "satisfies" the formula (*i.e.,* results in a truth value). In a normal database application, the database consists of only a finite amount of data. Hence, ideally formulas are useful if only a finite number of such sets of values satisfies the formula.

However, in the above setting of formulas, it is not possible to guarantee that only a finite combination of the free variables satisfy the formula. For example, the query

"all documents not in the database" can be represented by the formula $\neg D(x)$ and can be satisfied by an infinite number of values of x. Such formulas are formally called unsafe formulas, because queries that include such formulas can never be computed in a finite time. To avoid this problem, we define the notion of *safe formulas* next.

Safe DC Formulas Safe DC formulas (or, in short,SDC formulas) are the formulas which can be satisfied by only a finite set of values for the free variables. This is achieved by ensuring that the values of all free variables are always restricted to finite sets and ensuring that potentially unsafe operations (such as negation, as in the example above) always occur along with another formula that restricts the selection of values of the free variables. We can define the notion of safe formulas inductively as before, by starting with formulas that are intuitively safe and building up the formulas ensuring safety at every step. Here we give the intuition behind the safety of the formulas. A rigorous proof will follow the discussion of the algebraic language. Safe formulas can be defined as follows:

1. *Safe atomic formulas.* The following are safe atomic formulas:

   (a) $D(x)$ is safe, since it represents the finite set of documents that satisfy the DTD D.

   (b) $D \circ \mathcal{P}(x)$ is a safe formula, since the path expression $D \circ \mathcal{P}$ represents a finite set of documents.

   (c) If φ is a safe formula with a single free variable, so is $\varphi(x) \wedge x \circ \mathcal{P}(y)$. In this formula, the variable x can be thought of as being bound to a finite set of possible values by the safe formula φ. The rest of the formula is safe as before. In the subsequent discussions, we will use the notation $x^Q \circ \mathcal{P}(y)$ to represent formulas of this form, where $Q = \{z|\varphi(z)\}$ is the set of values that make φ true.

2. $x^Q \circ \mathcal{P}\theta c$ is a safe DC formula, where $\theta \in \{\ni, \not\ni\}$, $x \circ \mathcal{P}$ is a path term and $c \in \mathbf{dom}$ is a constant. It is trivial to see that this formula is safe.

3. $t_1 \theta t_2$ is a safe DC formula, where $\theta \in \{=, \neq, \subseteq, \subsetneq, \cap, \not\cap\}$ and $t_1 = x_1^{Q_1} \circ \mathcal{P}_1$ and $t_2 = x_2^{Q_2} \circ \mathcal{P}_2$ are two path terms. This formula is safe since Q_1 and Q_2 both represent safe sets.

4. If $\varphi(x_1, x_2, \ldots, x_n)$ and $\psi(x_1, x_2, \ldots, x_n)$ are safe formulas with the same set of free variables $x_1, x_2, \ldots, x_n$, then the following are also safe formulas:

   (a) $\varphi(x_1, x_2, \ldots, x_n) \vee \psi(x_1, x_2, \ldots, x_n)$

   (b) $\varphi(x_1, x_2, \ldots, x_n) \wedge \neg\psi(x_1, x_2, \ldots, x_n)$

   The first formula is safe because it is intuitively a union of two finite sets. In the second formula, the first clause provides a finite number of possible values for the free variables, thus making the negation safe.

5. If $\varphi(x_1, x_2, \ldots, x_n)$ and $\psi(y_1, y_2, \ldots, y_n)$ are safe formulas with possibly overlapping set of free variables $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$ respectively, then $\varphi(x_1, x_2, \ldots, x_n) \wedge \psi(y_1, y_2, \ldots, y_n)$ is also a safe formula. The intuition for the safety of this formula comes from the fact that the formula represents a set intersection of two finite sets.

6. If $\varphi(x, x_1, x_2, \ldots, x_n)$ is a safe formula with free variables $x, x_1, x_2, \ldots, x_n, n \geq 1$, then $\exists x \varphi(x, x_1, x_2, \ldots, x_n)$ is a safe formula.

In this safe version of the calculus, we build the formulas inductively from other safe formulas in such a way that every formula is satisfied by a finite number of values of the free variables assuming the database has only a finite number of documents. Another way of viewing a safe variant of a calculus language is to assume that all free variables in a formula are range restricted. We can show that such a language, where all variables have a finite range of values, is equivalent to the safe calculus proposed here. This can be proved by structural induction on the formulas.

Tuple Construction The path expressions provide a means for extracting components of a composite object. DC supports dynamic creation of composite types by

creating new generic identifiers with already existing generic identifiers as children. This is achieved by providing a tuple construction expression of the form:

$$z = R\langle x_1, x_2, \ldots, x_n \rangle \quad n \geq 1$$

The tuple construction operation can be treated as a formula that always returns a truth value, and thus can be combined with other formulas using a conjunction.

Queries A query is an expression denoting a set of documents described by a safe DC formula φ. All queries in DC are of the form

$$\{x \mid \varphi(x)\}$$

Here we only consider formulas with a single free variable, since such a formula can always be constructed from any formula with multiple free variables using the tuple construction mechanism described above, as follows:

$$\varphi_1(z) \triangleq (z = R\langle x_1, x_2, \ldots, x_k \rangle) \wedge \varphi(x_1, x_2, \ldots, x_k)$$

5.1.1.3 Semantics of DC

We now present the semantics of DC. Consider an interpretation $\mathbb{M}$ of the database and a valuation η of all variables. We define the semantics of the DC by defining the semantics of the terms and the formulas in the language. The semantics of path expressions defined in Section 5.1.1.1 is used in this definition.

Terms We consider three types of terms as described above:

- *Constants.* $[\![c]\!]_\eta^{\mathbb{M}} = c$
- *Variables.* $[\![x]\!]_\eta^{\mathbb{M}} = \eta(x)$
- *Path terms.* $[\![x \circ \mathcal{P}]\!]_\eta^{\mathbb{M}} = [\![\mathcal{P}]\!]^{\eta(x)}$

Formula A formula can only have one of the two boolean values: "true" and "false."

Given an interpretation $\mathbb{M}$ and a valuation of variables η, we say a formula φ

holds when it results in a true value for the given interpretation $\mathbb{M}$ and valuation η, and is written as $(\mathbb{M}, \eta) \models \varphi$. Since interpretation of formulas does not rely on safety, we will relax the safety issues and only describe the interpretations formulas in the unsafe form as described in Section 5.1.1.2. Depending on the type of formula, the interpretation is defined as follows:

- *Atomic formulas.*
  - $(\mathbb{M}, \eta) \models D(x)$ iff $\eta(x) \in \mathbb{M}(D)$
  - $(\mathbb{M}, \eta) \models D \circ \mathcal{P}(x)$ iff $\eta(x) \in [\![D \circ \mathcal{P}]\!]_\eta^{\mathbb{M}}$
  - $(\mathbb{M}, \eta) \models x \circ \mathcal{P}(y)$ iff $\eta(y) \in [\![\mathcal{P}]\!]^{\eta(x)}$
- *Formulas of the form $x \circ \mathcal{P}\theta c$.* The interpretation is defined as follows depending on θ:
  - $(\mathbb{M}, \eta) \models x \circ \mathcal{P} \ni c$ iff $[\![c]\!]_\eta^{\mathbb{M}} \in [\![\mathcal{P}]\!]^{\eta(x)}$
  - $(\mathbb{M}, \eta) \models x \circ \mathcal{P} \not\ni c$ iff $[\![c]\!]_\eta^{\mathbb{M}} \notin [\![\mathcal{P}]\!]^{\eta(x)}$
- *Formulas of the form $x_1 \circ \mathcal{P}_1 \theta x_2 \circ \mathcal{P}_2$.* The interpretation is defined as follows depending on θ:
  - $(\mathbb{M}, \eta) \models x_1 \circ \mathcal{P}_1 = x_2 \circ \mathcal{P}_2$ iff $[\![\mathcal{P}_1]\!]^{\eta(x_1)} = [\![\mathcal{P}_2]\!]^{\eta(x_2)}$
  - $(\mathbb{M}, \eta) \models x_1 \circ \mathcal{P}_1 \neq x_2 \circ \mathcal{P}_2$ iff $[\![\mathcal{P}_1]\!]^{\eta(x_1)} \neq [\![\mathcal{P}_2]\!]^{\eta(x_2)}$
  - $(\mathbb{M}, \eta) \models x_1 \circ \mathcal{P}_1 \subseteq x_2 \circ \mathcal{P}_2$ iff $[\![\mathcal{P}_1]\!]^{\eta(x_1)} \subseteq [\![\mathcal{P}_2]\!]^{\eta(x_2)}$
  - $(\mathbb{M}, \eta) \models x_1 \circ \mathcal{P}_1 \subsetneq x_2 \circ \mathcal{P}_2$ iff $[\![\mathcal{P}_1]\!]^{\eta(x_1)} \subsetneq [\![\mathcal{P}_2]\!]^{\eta(x_2)}$
  - $(\mathbb{M}, \eta) \models x_1 \circ \mathcal{P}_1 \cap x_2 \circ \mathcal{P}_2$ iff $[\![\mathcal{P}_1]\!]^{\eta(x_1)} \cap [\![\mathcal{P}_2]\!]^{\eta(x_2)} \neq \emptyset$
  - $(\mathbb{M}, \eta) \models x_1 \circ \mathcal{P}_1 \not\cap x_2 \circ \mathcal{P}_2$ iff $[\![\mathcal{P}_1]\!]^{\eta(x_1)} \cap [\![\mathcal{P}_2]\!]^{\eta(x_2)} = \emptyset$
- *Formulas with logical operators.*
  - $(\mathbb{M}, \eta) \models \varphi \wedge \psi$ iff $(\mathbb{M}, \eta) \models \varphi$ and $(\mathbb{M}, \eta) \models \psi$
  - $(\mathbb{M}, \eta) \models \varphi \vee \psi$ iff $(\mathbb{M}, \eta) \models \varphi$ or $(\mathbb{M}, \eta) \models \psi$
  - $(\mathbb{M}, \eta) \models \neg\varphi$ iff $(\mathbb{M}, \eta) \not\models \varphi$
- *Formulas with quantification.* To define this, we consider the notion of substitution in valuations. We say $[a/x^\tau]\eta$ is a valuation in which the

value $a \in \tau$ is substituted for the variable x where τ is the type of x. We also denote all possible such values in the interpretation $\mathbb{M}$ as $|\mathbb{M}(\tau)|$.

- $(\mathbb{M}, \eta) \models \exists x^{\tau} \varphi$ iff $(\mathbb{M}, [a/x^{\tau}]\eta) \models \varphi$ for some $a \in |\mathbb{M}(\tau)|$

5.1.1.4 Examples

To illustrate the language, consider some of the queries we discussed in Chapter 1. For the examples, consider the schema in Figure 12. The DC queries corresponding

```
<!DOCTYPE POEM [
<!ELEMENT poem       - - (head, body)>
<!ELEMENT head       - - (period, poet, title)>
<!ELEMENT body       - - (stanza)+>
<!ELEMENT stanza     - - (line)+>
<!ELEMENT (period | poet | title | line) - O (#PCDATA)>
]>
```

Figure 12: A simple poem database schema

to some of the queries mentioned in Chapter 1 are given below.

1. Find all poems that contain the word "love" in the poem title.

$$\{x | x^{\{z|poem(z)\}}..title \ni \text{“love”}\}$$

2. Extract titles and authors of all poems in the database.

$$\left\{w | \exists x(w = R\langle y, z\rangle) \wedge (x^{\{z|poem(z)\}}..title(y)) \wedge (x^{\{z|poem(z)\}}..poet(z))\right\}$$

3. Find the period in which all poems had the word "love" in their titles. In this query, let $\psi = \{z|poem(z)\}$. The query is intuitively written using first order logic as:

$$\left\{x | \forall y \; poem(y) \Rightarrow \left(y^{\psi}..period(x) \Rightarrow y^{\psi}..title \ni \text{“love”}\right)\right\}$$

Note that this formulation is not explicitly safe. In order to ensure safety using the notion of safe formulas that we describe above, we first need to replace the implication with the equivalent logical expression ($A \Rightarrow B \equiv \neg A \vee B$) and use DeMorgan's laws to reduce the universal quantifier to an existentialquantifier . The query is then reformulated as follows:

$$\{x \mid poem..period(x) \wedge \neg\exists y\,(poem(y) \wedge (y^{\psi}..period(x) \wedge (poem(y) \wedge \neg y^{\psi}..title \ni \text{“love”})))\}$$

4. Find the pairs of names for poets who have at least one common poem title. Again, let $\psi = \{z|poem(z)\}$. The query is then represented as:

$$\{v \mid (v = R\langle w, z\rangle) \wedge (\exists x, y\ (x^{\psi}..title \cap y^{\psi}..title \wedge x^{\psi}.\epsilon \not\cap y^{\psi}.\epsilon \wedge x^{\psi}..poet(w) \wedge y^{\psi}..poet(z))\}$$

5. Find the poems that do not have the word “love” in the title.

$$\left\{x \mid poem(x) \wedge \neg x^{\{z|poem(z)\}}..title \ni \text{“love”}\right\}$$

5.1.2 The Document Algebra (DA)

The document calculus language specified above describes a first-order language for expressing queries on documents. The document algebra (DA) being described here is predictably an extension of the relational algebra. We use essentially the same operators of relational algebra (with modified semantics) and a few new operators.

Although the calculus language described in the previous section can sufficiently describe the properties and powers of the language, there are a number of motivations for describing an algebraic language which has the same expressive powers as the calculus language. Prominent among them are:

1. The algebraic language is procedural, and hence provides an easy means for implementation of the language using a procedural programming language.

2. Since the algebra consists of operations that map sets of documents to sets of documents, it is convenient to prove the safety of the language by showing that

the sets generated are finite if the inputs are finite sets.

3. Because of the procedural nature of the operations in the language, it is convenient to describe the complexity of the language in terms of the complexity of the individual operations.

In this section, we present Document Algebra (DA), an algebraic language for manipulating and querying documents. In a subsequent section we will prove the equivalence of the calculus and algebraic languages and demonstrate their properties. As in the calculus, we will use the symbol $\circ$ to represent one of the path expression operators **.** and **..** in the following discussion.

5.1.2.1 Primary DA Operations

All DA operations are described as functions from one or more sets of documents to another set of documents. Every DA expressionE^{τ} represents a set of documents of a particular type τ. We define DA expressions inductively by first defining the basic document expression and then defining the operations *cross product* ($\times$), *selection* (σ), *projection* (π), *union* ($\cup$), *intersection* ($\cap$) *and set difference* ($-$). We will also define the operations *join* ($\bowtie$), generalized product ($\prod$) and root addition (ρ), as a combination of the primitive operations. All the operators generate documents of specific types, as shown in Table 7.

Expression	**Type**	**New productions**
D	D	
$E \circ \mathcal{P}$	$last(\mathcal{P})$	
$E_1^{\tau_1} \cup_R E_2^{\tau_2}$	R	$R \rightarrow \tau_1 \mid \tau_2$
$E_1^{\tau} - E_2^{\tau}$	τ	
$E_1^{\tau_1} \times_R E_2^{\tau_2}$	R	$R \rightarrow \tau_1, \tau_2$
$\sigma_{\gamma} E^{\tau}$	τ	

Table 7: Types of Document Algebra operations and new created types.

A DA expression E^{τ} and its semantics are defined inductively as follows (assume the usual notation $\mathbb{M}$ for a database with a finite set of documents in the context of

a DTD D):

Document The expression D represents the set of all documents in the database. Thus, $[\![D]\!]^{\mathbb{M}} = \mathbb{M}(D)$.

Path selection ($\circ$) Given a DA expression E^{τ} and a SPE $\mathcal{P}$, $E^{\tau} \circ \mathcal{P}$ is a DA expression that returns the set of documents rooted at $last(\mathcal{P})$ obtained by traversing the path $\mathcal{P}$ from each of the documents in E^{τ}. So, $[\![E^{\tau} \circ \mathcal{P}]\!]^{\mathbb{M}} = [\![\mathcal{P}]\!]^{[\![E^{\tau}]\!]^{\mathbb{M}}}$.

Union ($\cup$) Union is the usual set union operation, without the restriction that both operands of the union be of the same type. Given two DA expressions $E_1^{\tau_1}$ and $E_2^{\tau_2}$, the result of the union $E_1^{\tau_1} \cup_R E_2^{\tau_2}$ is a set of documents of a new type R which is created by adding a production for R with τ_1 and τ_2 in an option group. To explain this operation, we use the notation $R\langle\mathcal{S}\rangle$ to denote an expression that includes the documents in the set $\mathcal{S}$, each augmented with a special root generic identifier R. With this notation,

$$[\![E_1^{\tau_1} \cup_R E_2^{\tau_2}]\!]^{\mathbb{M}} = R\langle[\![E_1^{\tau_1}]\!]^{\mathbb{M}}\rangle \cup R\langle[\![E_2^{\tau_2}]\!]^{\mathbb{M}}\rangle$$

Here the operation $\cup$ is the regular set union operation.

Intersection ($\cap$) The intersection operation is the usual set intersection operation $E_1^{\tau} \cap E_2^{\tau}$, containing the set of documents that are both in E_1^{τ} and in E_2^{τ}. Hence, $[\![E_1^{\tau} \cap E_2^{\tau}]\!]^{\mathbb{M}} = [\![E_1^{\tau}]\!]^{\mathbb{M}} \cap [\![E_2^{\tau}]\!]^{\mathbb{M}}$

Set difference ($-$) The set difference is the usual set difference operation $E_1^{\tau} - E_2^{\tau}$, containing the set of documents in E_1^{τ} that are not in E_2^{τ}. Hence, $[\![E_1^{\tau} - E_2^{\tau}]\!]^{\mathbb{M}} = [\![E_1^{\tau}]\!]^{\mathbb{M}} - [\![E_2^{\tau}]\!]^{\mathbb{M}}$

Cross Product ($\times$) Given two DA expressions $E_1^{\tau_1}$ and $E_2^{\tau_2}$, the expression $E_1^{\tau_1} \times_R E_2^{\tau_2}$ is a DA expression, and it represents a set of documents with a new type R the members of contain two subcomponents: one from the set $E_1^{\tau_1}$ and the other from $E_2^{\tau_2}$. In the resulting set, each member of the set $[\![E_1^{\tau_1}]\!]^{\mathbb{M}}$ is combined with each member of the set $[\![E_2^{\tau_2}]\!]^{\mathbb{M}}$. Hence,

$$[\![E_1^{\tau_1} \times_R E_2^{\tau_2}]\!]^{\mathbb{M}} = \{R\langle x, y\rangle \mid x \in [\![E_1^{\tau_1}]\!]^{\mathbb{M}} \wedge y \in [\![E_2^{\tau_2}]\!]^{\mathbb{M}}\}$$

Here $R\langle x, y\rangle$ represents a document with two components x and y.

Selection (σ) The selection operation $\sigma_\gamma E^\tau$ extracts a subset of documents from the input set E^τ that satisfy a selection condition γ. γ can be of one of two forms: (i) $\mathcal{P}\theta c$ where $\mathcal{P}$ is a path expression, θ is in $\{\ni, \not\ni\}$ and $c \in$ **dom**, and (ii)$\mathcal{P}_1\theta\mathcal{P}_2$ where $\mathcal{P}_1$ and $\mathcal{P}_2$ are path expressions and $\theta \in \{=, \neq, \subseteq, \subsetneq, \cap, \not\cap\}$. Mathematically, the semantics can be represented by:

i.. $[\![\sigma_{(\mathcal{P}\theta c)} E^\tau]\!]^{\mathbb{M}} = \left\{x | x \in [\![E^\tau]\!]^{\mathbb{M}} \wedge [\![x \circ \mathcal{P}]\!]^{\mathbb{M}} \theta c\right\}$

ii.. $[\![\sigma_{(\mathcal{P}_1\theta\mathcal{P}_2)} E^\tau]\!]^{\mathbb{M}} = \left\{x | x \in [\![E^\tau]\!]^{\mathbb{M}} \wedge [\![x \circ \mathcal{P}_1]\!]^{\mathbb{M}} \theta [\![x \circ \mathcal{P}_2]\!]^{\mathbb{M}}\right\}$

5.1.2.2 Derived DA Operations

In addition to the primary operations described above, some additional operations can also be observed to be useful. These are composite operations that can be derived from one or more of the above primary operations. The types and productions created by the derived operations are shown in Table 8.

Expression	**Type**	**New productions**
$E_1^\tau \cup E_2^\tau$	τ	
$\pi^R_{\mathcal{P}_1, \mathcal{P}_2, \ldots \mathcal{P}_k} E^\tau$	R	$R \rightarrow first(\mathcal{P}_1), first(\mathcal{P}_2), \ldots, first(\mathcal{P}_k)$
$\prod^R_{E_1^{\tau_1}, \ldots, E_n^{\tau_n}}$	R	$R \rightarrow \tau_1, \tau_2, \ldots, \tau_n$
$\rho_R E^\tau$	R	$R \rightarrow \tau$
$E_1^{\tau_1} \bowtie^R_{p_1 \theta p_2} E_2^{\tau_2}$	R	$R \rightarrow \tau_1, \tau_2$

Table 8: Derived DA operations and new created types.

Ordinary Union The union operation defined above is a more general operation in which it is not necessary that the operands be of the same type. The normal

union operation can be defined by composing the general union with a path selection, as follows:

$$E_1^\tau \cup E_2^\tau \equiv (E_1^\tau \cup_R E_2^\tau) \circ R.\tau$$

Projection (π) The projection operation extracts subtrees from document trees. The projection expression $\pi^R_{\mathcal{P}_1,\mathcal{P}_2,\ldots\mathcal{P}_k} E^\tau$ creates a new type R containing the projected types in sequence from the expression E^τ. This operation can be defined as a composition of cross product, selection and path selection operations. The intuition behind this property lies in the projection-like capability of the path selection operation $\circ$. In fact, path selection is like a single-item projection, and to obtain multiple items, multiple path selections followed by cross product and a final reconstruction needs to be performed, as the following projection of two components illustrates:

$$\pi^R_{\mathcal{P}_1,\mathcal{P}_2}(E^\tau) \equiv \left(\sigma_{S.R.last(\mathcal{P}_1)\cap S.\mathcal{P}_1}\left(\sigma_{S.R.last(\mathcal{P}_2)\cap S.\mathcal{P}_2}\left((E^\tau \circ \mathcal{P}_1 \times_R E^\tau \circ \mathcal{P}_2) \times_S E^\tau\right)\right)\right) \circ S.R$$

Generalized Product ($\prod$) The product described above uses only two operands. We can also define a stronger generalized product operation with an arbitrary number of operands $n (n > 1)$ as: $\prod^R_{E_1^{\tau_1},\ldots,E_n^{\tau_n}}$, which represents a set of documents that contain n subcomponents from the respective operands. In terms of the primitive operations defined above, this operation could be written as:

$$\prod^R_{E_1^{\tau_1},\ldots,E_n^{\tau_n}} = \pi^R_{R_1.\tau_1,R_1.R_2.\tau_2,\ldots,R_1.R_2.\cdots.R_{n-1}.\tau_{n-1},R_1.R_2.\cdots.R_{n-1}.\tau_n}\left(E_1^{\tau_1} \times_{R_1} \left(E_2^{\tau_2} \times_{R_2} \cdots \left(E_{n-1} \times_{R_{n-1}} E_n\right)\ldots\right)\right)$$

Add Root (ρ) The add root operation $\rho_R E^\tau$ is a simple operation that takes documents from the expression E^τ and adds the root R to them. It creates a new type R containing only the type of E^τ. It is trivial to observe that $\rho_R E^\tau = E^\tau \cup_R E^\tau$.

Join ($\bowtie$) Join can be defined as a combination of cross product and selection, as follows:

$$E_1^{\tau_1} \bowtie^R_{p_1\theta p_2} E_2^{\tau_2} \equiv \sigma_{p_1\theta p_2}\left(E_1^{\tau_1} \times_R E_2^{\tau_2}\right)$$

5.1.2.3 Examples of DA Expressions

We now demonstrate how the algebraic expressions for the queries described earlier in Section 5.1.1.4 can be formed.

1. Find all poems that contain the word "love" in the poem title.

$$\sigma_{poem..title \ni \text{"love"}} Poem$$

2. Extract titles and authors of all poems in the database.

$$\pi^{AT}_{poem..title, poem..poet} Poem$$

3. Find the period in which all poems had the word "love" in their titles. We solve this query using stages.

$$\begin{aligned} Periods &= Poem \circ poem..period \\ PT &= \pi^{P}T_{poem..period, poem..title} Poem \\ Result &= Periods - \left(PT - \sigma_{PT.title \ni \text{"love"}} PT\right) \circ PT.period \end{aligned}$$

4. Find the pairs of names for poets who have at least one common poem title.

$$\pi_{P_1..poet, P_2..poet} \left(\sigma_{P_1..poet \not\ni P_2..poet} \left(\rho_{P_1} Poem \bowtie^{R}_{P_1..title \cap P_2..title} \rho_{P_2} Poem\right)\right)$$

5. Find the poems that do not have the word "love" in the title.

$$Poem - \sigma_{poem..title \ni \text{"love"}} Poem$$

5.1.3 Properties of the Query Languages

5.1.3.1 Equivalence of DC and DA

We now show that the two languages (DC, DA) defined above are semantically equivalent (*i.e.*, any query written in the document calculus is equivalent to some document

algebra expression, and vice versa). We prove this in two steps. First, we show that any DA expression is equivalent to some DC query, and then we show the reverse.

Theorem If E is a document algebra expression, then there is an expression E^C in document calculus equivalent to E.

Proof. We prove this by strong induction on the number of operators in the algebraic expression:

- *Induction hypothesis.* For any DA expression E with fewer than n operators ($n \geq 1$), we can construct a safe DC formula E^C with one free variable such that $E \equiv \{x \mid E^C(x)\}$.

- *Base case.* (number of operators = 0) The only possible expression is D. The equivalent calculus expression is: $\{x|D(x)\}$.

- *Induction step.* Denote a DA expression with n operators by E_n. We prove the theorem by considering all the cases of the definition of DA expressions using only the primitive algebraic operators.

  1. $E_n = E^\tau \circ \mathcal{P}$. By the induction hypothesis, $E^\tau \equiv \{x \mid E^C(x)\}$ for some safe DC formula E^C, since E^τ has $n-1$ operators. So we have

  $$E_n \equiv \left\{ y | \exists x \; x^{\{w|E^C(w)\}} \circ \mathcal{P}(y) \right\}$$

  2. $E_n = E_1^{\tau_1} \cup_R E_2^{\tau_2}$. By the induction hypothesis, since the expressions $E_1^{\tau_1}$ and $E_2^{\tau_2}$ both have fewer than n operators, there are safe DC formulas E_1^C and E_2^C such that $E_1^{\tau_1} \equiv \{x \mid E_1^C(x)\}$ and $E_2^{\tau_2} \equiv \{x \mid E_2^C(x)\}$. Hence,

  $$E_n \equiv \left\{ z \mid (z = R\langle x \rangle) \wedge (E_1^C(x) \vee E_2^C(x)) \right\}$$

  3. $E_n = E_1^\tau \cap E_2^\tau$. By the induction hypothesis, since the expressions E_1^τ and E_2^τ both have fewer than n operators, there are safe DC formulas E_1^C and

E_2^C such that $E_1^\tau \equiv \{x \mid E_1^C(x)\}$ and $E_2^\tau \equiv \{x \mid E_2^C(x)\}$. Hence,

$$E_n \equiv \left\{x \mid E_1^C(x) \wedge E_2^C(x)\right\}$$

4. $E_n = E_1^\tau - E_2^\tau$. By the induction hypothesis, since the expressions E_1^τ and E_2^τ both have fewer than n operators, there are safe DC formulas E_1^C and E_2^C such that $E_1^\tau \equiv \{x \mid E_1^C(x)\}$ and $E_2^\tau \equiv \{x \mid E_2^C(x)\}$. Hence,

$$E_n \equiv \left\{x \mid E_1^C(x) \wedge \neg E_2^C(x)\right\}$$

5. $E_n = E_1^{\tau_1} \times_R E_2^{\tau_2}$. By the induction hypothesis, since the expressions $E_1^{\tau_1}$ and $E_2^{\tau_2}$ both have fewer than n operators, there are safe DC formulas E_1^C and E_2^C such that $E_1^{\tau_1} \equiv \{x \mid E_1^C(x)\}$ and $E_2^{\tau_2} \equiv \{x \mid E_2^C(x)\}$. Hence,

$$E_n \equiv \left\{z \mid (z = R\langle x_1, x_2\rangle) \wedge E_1^C(x_1) \wedge E_2^C(x_2)\right\}$$

6. $E_n = \sigma_\gamma E^\tau$. Since E^τ has $n-1$ operators, by the induction hypothesis, there is a safe DC formula E^C such that $E^\tau \equiv \{x \mid E^C(x)\}$. Now, depending on the form of γ, one of the following two cases may arise:
   (a) $\gamma = \mathcal{P}\theta c$. $E_n \equiv \{x \mid x^{\{z|E^C(z)\}} \circ \mathcal{P}\theta c\}$
   (b) $\gamma = \mathcal{P}_1\theta\mathcal{P}_2$. $E_n \equiv \left\{x \mid \left(x^{\{y|E^C(y)\}} \circ \mathcal{P}_1\right) \theta \left(x^{\{z|E^C(z)\}} \circ \mathcal{P}_2\right)\right\}$

The proof follows by strong induction on n. ■

Calculus to Algebra We now show that for every safe DC query, there is an equivalent DA expression. To prove this, we make use of a *cannonical form* of DC queries. We define a *distinguished query* to be a query of the following form:

$$\begin{aligned}\mathcal{Q}_D^\varphi \quad = \quad & \{z|(z = R_0\langle z_1, z_2, \ldots z_k\rangle) \wedge (z_1 = R_{x_1}\langle x_1\rangle) \wedge (z_2 = R_{x_2}\langle x_2\rangle) \wedge \ldots \wedge (z_k = R_{x_k}\langle x_k\rangle) \\ & \wedge\varphi(x_1, x_2, \ldots, x_k)\}\end{aligned}$$

Here all the names $R_{x_0}, R_{x_1}, \ldots R_{x_k}$ are distinct. The names follow the intuition that all the variables in the formula are distinguished and can be individually projected out from the query via a projection. This is, in general, not possible if two free variables in a formula are of the same type. In the conversion theorem, we are going to generate algebraic expressions corresponding to distinguished queries, and we use the following lemma to get our final result:

Lemma 1. If Q_D^{φ} is the distinguished query corresponding to any SDC formula $\varphi(x_1, x_2, \ldots, x_n)$, then there exists a DA expression E_D^{φ} which is equivalent to Q_D^{φ}.

Proof. We present the proof by structural induction on the queries in DC:

- *Induction hypothesis.* For every safe DC formula with k free variables ($k \geq 1$), of the form $\varphi(x_1, x_2, \ldots x_k)$, there is an algebraic expression $E_D^{\varphi} \equiv Q_D^{\varphi}$, where Q_D^{φ} is the distinguished query corresponding to $\varphi(x_1, x_2, \ldots x_k)$.

- *Inductive proof.* We will describe this proof by structural induction on the calculus formula.

  - *Base case.* The base case is given by $\varphi = D(x)$. We have, $E_D^{\varphi} = \rho_{R_0}(\rho_{R_x} D)$

  - *Atomic formulas.* We consider all three variants of atomic formulas:

    1. if $\varphi = D(x)$, this case is the same as the base case.
    2. $\varphi = D \circ \mathcal{P}(x)$ We have, $E_D^{\varphi} = \rho_{R_0}(\rho_{R_x}(D \circ \mathcal{P}))$
    3. $\varphi = (\psi(x)) \wedge (x \circ \mathcal{P}(y))$. By the induction hypothesis, we have $E_D^{\psi} \equiv Q_D^{\psi}$. To avoid confusion, let us assume that the root generic identifier of E_D^{ψ} is R_0 (i.e., $E_D^{\psi} \equiv \{z | (z = R_0\langle z_1 \rangle) \wedge (z_1 = R_x\langle x \rangle) \wedge \psi(x)\}$). Then we have

    $$E_D^{\varphi} = \pi^R_{S.R_0.R_x, S.Ry}(\sigma_{S.R_0.R_x.\mathcal{P} \cap S.R_y.last(\mathcal{P})}(E_D^{\psi} \times_S (\rho_{R_y}(E_D^{\psi} \circ R_0.R_x.\mathcal{P}))))$$

  - $\varphi = t\theta c$. Here, t is a path term and is of the form $x^{Q^{\psi}} \circ \mathcal{P}$ (recall that this is an abbreviation of the expression $\psi(x) \wedge x \circ \mathcal{P}$ and $Q^{\psi} = z | \psi(z)$. By the

induction hypothesis,we have $E_D^{\psi} \equiv Q_D^{\psi}$. Hence,

$$E_D^{\varphi} = \sigma_{R.R_x.\mathcal{P}\theta c} E_D^{\psi}$$

– $\varphi = t_1 \theta t_2$. Here, $t_1 = x_1^{Q_1^{\psi_1}} \circ \mathcal{P}_1$ and $t_2 = x_2^{Q_2^{\psi_2}} \circ \mathcal{P}_2$. By the induction hypothesis, we have the expressions

$$\begin{aligned} E_D^{\psi_1} &\equiv \{z | (z = R_1\langle z_1 \rangle) \wedge (z_1 = R_{x_1}\langle x_1 \rangle) \wedge \psi_1(x_1)\} \\ E_D^{\psi_2} &\equiv \{z | (z = R_2\langle z_1 \rangle) \wedge (z_1 = R_{x_2}\langle x_2 \rangle) \wedge \psi_2(x_2)\} \end{aligned}$$

Hence,

$$\begin{aligned} E_D^{\varphi} &= \{z | (z = R_0\langle z_1, z_2 \rangle) \wedge (z_1 = R_{x_1}\langle x_1 \rangle) \wedge (z_2 = R_{x_2}\langle x_2 \rangle) \wedge t_1 \theta t_2\} \\ &= \pi^{R_0}_{S.R_1.R_{x_1}, S.R_2.R_{x_2}} \left(\sigma_{S.R_1.R_{x_1}.\mathcal{P}_1 \theta S.R_2.R_{x_2}.\mathcal{P}_2} \left(Q_1^E \times_S Q_2^E\right)\right) \end{aligned}$$

– $\varphi = \psi_1(x_1, x_2, \ldots x_m) \vee \psi_2(x_1, x_2, \ldots x_m)$. By the induction hypothesis, we have

$$\begin{aligned} E_D^{\psi_1} &\equiv \{z | (z = R_0\langle z_1, z_2, \ldots, z_m \rangle) \wedge (z_1 = R_{x_1}\langle x_1 \rangle) \wedge (z_2 = R_{x_2}\langle x_2 \rangle) \wedge \\ &\quad \ldots \wedge (z_k = R_{x_k}\langle x_k \rangle) \wedge \psi_1(x_1, x_2, \ldots, x_k)\} \\ E_D^{\psi_2} &\equiv \{z | (z = R_0\langle z_1, z_2, \ldots, z_m \rangle) \wedge (z_1 = R_{x_1}\langle x_1 \rangle) \wedge (z_2 = R_{x_2}\langle x_2 \rangle) \wedge \\ &\quad \ldots \wedge (z_k = R_{x_k}\langle x_k \rangle) \wedge \psi_2(x_1, x_2, \ldots, x_k)\} \end{aligned}$$

Notice that we have assumed that both the above expressions are rooted at R_0 to simplify matters. If they are not rooted at the same GI, we can always make them rooted at the same GI using a projection followed by an add-root operation. Hence,

$$E_D^{\varphi} = E_D^{\psi_1} \cup E_D^{\psi_2}$$

- $\varphi = \psi_1(x_1, x_2, \ldots x_m) \wedge \neg\psi_2(x_1, x_2, \ldots x_m)$. By the induction hypothesis, we have $E_D^{\psi_1}$ and $E_D^{\psi_2}$ as in the previous case. So,

$$E_D^{\varphi} = E_D^{\psi_1} - E_D^{\psi_2}$$

- $\varphi = \psi_1(x_1, x_2, \ldots x_m) \wedge \psi_2(y_1, y_2, \ldots y_n)$. In this case, notice that ψ_1 and ψ_2 do not have the same free variables. They may have overlapping or completely exclusive variable sets. To construct the equivalent algebraic expression, it is necessary to create expressions in which the corresponding documents for variables are properly matched and aligned. This is usually accomplished by performing a series of product operations in the proper order of the variables. Consider the following cases:

  1. If ψ_1 and ψ_2 have the same set of free variables, the transformation is easy. From the induction hypothesis, we know that we have expressions $E_D^{\psi_1}$ and $E_D^{\psi_2}$ corresponding to these formulas. Furthermore, we can also assume that they are of the same type, since they have the same variables, and if they have different types, we can make them the same type using π and ρ. Hence, we can now write: $\varphi^E \equiv \psi_1^E \cap \psi_2^E$.
  2. If ψ_1 and ψ_2 have overlapping set of variables, the construction needs to be performed in several steps. A general mathematical expression for these steps is overly complex. We take one example to show how in general this is done. Suppose $\varphi(x, y, z) = \psi_1(x, z) \wedge \psi_2(y, z)$. The following stages are necessary in the construction of E_D^{φ}:
     (a) *Using the induction hypothesis.* By the induction hypothesis, we have the following:

$$\begin{aligned} E_D^{\psi_1} &\equiv \{u|(u = R_1\langle u_1, u_2\rangle) \wedge (u_1 = R_x\langle x\rangle) \wedge (u_2 = R_z\langle z\rangle) \wedge \psi_1(x, z)\} \\ E_D^{\psi_2} &\equiv \{v|(v = R_2\langle v_1, v_2\rangle) \wedge (v_1 = R_y\langle y\rangle) \wedge (v_2 = R_z\langle z\rangle) \wedge \psi_2(y, z)\} \end{aligned}$$

     (b) *Padding and Reorganizing.* In this stage, each of the component expressions need to be padded to include all the variables in the

final expression, and the variables need to be reorganized so that they have the same order in all the components, in the following way:

$$\begin{aligned} E_1 &= \pi^R_{R_0.R_1.R_x, R_0.R_y, R_0.R_1.R_z}\left(E_D^{\psi_1} \times_{R_0} \left(E_D^{\psi_2} \circ R_2.R_y\right)\right) \\ E_2 &= \pi^R_{R_0.R_x, R_0.R_2.R_y, R_0.R_2.R_z}\left(E_D^{\psi_2} \times_{R_0} \left(E_D^{\psi_1} \circ R_1.R_x\right)\right) \end{aligned}$$

(c) *Final intersection.* Now the expressions have the same type and all the components are organized in the same way. So, we can perform an intersection to obtain $E_D^{\varphi} = E_1 \cap E_2$

– $\varphi(y_1, y_2, \ldots y_n) = \exists x \psi(x, y_1, y_2, \ldots y_n)$. From the induction hypothesis, we have

$$\begin{aligned} E_D^{\psi} \equiv\ & \{z | (z = R_0\langle z_0, z_1, z_2, \ldots z_n\rangle) \wedge (z_0 = R_x\langle x\rangle) \wedge (z_1 = R_{y_1}\langle y_1\rangle) \wedge \\ & (z_2 = R_{y_2}\langle y_2\rangle) \ldots (z_n = R_{y_n}\langle y_n\rangle) \wedge \psi(x, y_1, y_2, \ldots, y_n)\} \end{aligned}$$

Hence, we can write:

$$E_D^{\varphi} = \pi^R_{R_0.R_{y_1}, R_0.R_{y_2}, \cdots, R_0.R_{y_n}} E_D^{\psi}$$

■

Lemma 2. An algebraic expression equivalent to a distinguished query can be converted to an algebraic expression that is equivalent to the corresponding non-distinguished query $\{z | (z = R\langle x_1, x_2, \ldots, x_k\rangle) \wedge \varphi(x_1, x_2, \ldots, x_k)\}$

Proof. The proof is simple. Since all the variables are distinguished by distinct non-terminal symbols, we only need a projection for each variable. So, if the DA expression equivalent to the non-distinguished version of the query is denoted by E^{φ} and the distinguished version is denoted by E_D^{φ}, we have:

$$E^{\varphi} \equiv \pi^R_{R_0.R_{x_1}.\tau_1, R_0.R_{x_2}.\tau_2, \cdots, R_0.R_{x_k}.\tau_k} E_D^{\varphi}$$

where $\tau_1, \tau_2, \ldots, \tau_k$ are the types of the corresponding variables $x_1, x_2, \ldots, x_k$. ■

Lemma 3. If $\varphi(x_1, x_2, \ldots, x_n)$ is an SDC formula, then there exists a DA expression E^φ such that $E^\varphi = \{x | x = R\langle x_1, x_2, \ldots, x_n\rangle \wedge \varphi(x_1, x_2, \ldots, x_n)\}$

Proof. The proof can be presented in three stages:

1. Given the formula $\varphi(x_1, x_2, \ldots, x_n)$, we can construct a distinguished DC query Q_D^φ corresponding to it, as shown in the construction of distinguished queries.

2. Using Lemma 1 we can show that there is an algebraic expression E_D^φ which is equivalent to Q_D^φ.

3. Using Lemma 2, we then show that we can obtain E^φ from E_D^φ.

This completes the proof. ■

Theorem If $Q = \{x^\tau | \varphi(x^\tau)\}$ is a safe document calculus query, then there exists a document algebra expression E^Q is equivalent to Q.

Proof. The proof essentially follows from Lemma 3, noticing that φ has one single free variable, so $E^\varphi \equiv \{z | (z = R\langle x\rangle) \wedge \varphi(x)\}$ from Lemma 3.

Hence, $E^Q = E^\varphi \circ R.\tau$. ■

5.1.3.2 Safety Properties

We intend to show here that the algebraic language as well as the safe document calculus language are safe (*i.e.,* they map finite sets of documents to finite sets of documents). Since it is proved that the two languages are semantically equivalent, it suffices to prove the safety using only one of the two variants. As pointed out earlier, because of the procedural nature of the DA operations, safety and complexity properties are easier to analyze using the algebraic language. In this section, we demonstrate the safety of the language, and in the next section, we will discuss the complexity of the language.

Theorem The DA language is safe (*i.e.,* it maps finite sets of documents into finite sets of documents).

Proof. The proof is by structural induction on the DA operations:

- *Induction Hypothesis.* Given there are only a finite number of documents in the database, a DA expression E^τ with type τ will only return a finite number of documents as the result.

- *Base case.* The base case is provided by the expression D. The proof follows from the assumption, since there are only a finite number of documents corresponding to D.

- *Induction step.* We denote the number of documents returned by a DA expression E_n by $|E_n|$, and consider all possible cases for building E_n using only the primitive algebraic operators:

  1. $E_n = E^\tau \circ \mathcal{P}$. By the induction hypothesis, $|E^\tau|$ is finite. Since each of the documents in E^τ is of finite size, the operation $\circ$ only returns nodes in the document structure, in the worst case, $|E_n| = |E^\tau| \times m$, where m is the maximum number of nodes among the documents returned by E^τ.

  2. $E_n = E_1^{\tau_1} \cup_R E_2^{\tau_2}$. By the induction hypothesis, $|E_1^{\tau_1}|$ and $|E_2^{\tau_2}|$ are finite. Since the operation is essentially a union operation, we have in the worst case $|E_n| = |E_1^{\tau_1}| + |E_2^{\tau_2}|$, which is finite.

  3. $E_n = E_1^\tau \cap E_2^\tau$. By the induction hypothesis, $|E_1^\tau|$ and $|E_2^\tau|$ are finite. So in the worst case, $|E_n| = max(|E_1^\tau|, |E_2^\tau|)$, a finite number.

  4. $E_n = E_1^\tau - E_2^\tau$. Since this is a regular set difference operation, in the worst case, $|E_n| = |E_1^\tau|$, which is finite by the induction hypothesis.

  5. $E_n = E_1^{\tau_1} \times_R E_2^{\tau_2}$. By the induction hypothesis, $|E_1^{\tau_1}|$ and $|E_2^{\tau_2}|$ are finite. Hence, $|E_n| = |E_1^{\tau_1}| \times |E_2^{\tau_2}|$, a finite number.

  6. $E_n = \sigma_\gamma E^\tau$. By the induction hypothesis, we have we know that $|E^\tau|$ is finite. Since the selection operation returns a subset of the input, we have $|E_n| \leq |E^\tau|$, and hence, is finite, regardless of the structure of γ.

In the above, we show that for every way of constructing a DA expression, if the constituent expressions are finite, the resulting size of the expression is finite. Hence, by structural induction, DA is safe. ■

5.1.3.3 Complexity properties

The query language described above is a simple yet powerful language for hierarchical document structures. One important property of this language that we describe here is that the language is in PTIME. In other words, all operations in the language can be performed by algorithms in time proportional to a polynomial of the size of the input. In this section, we prove this statement. Since we have shown above that the document calculus language is semantically equivalent to the document algebra language, it is sufficient to show that the operations allowed in the algebra are within PTIME. Here, we first define the notion of the input size in our model and then show that it is possible to compute all algebraic operations in PTIME.

Input Size In our model for document databases, we treat a database as a set of documents conforming to a given DTD. We further note that every document has only a finite size (*i.e.,* has a finite number of nodes in its structure). Suppose the number of documents in a database is n, and the document with the maximum number of nodes in it has m nodes. Then the product $m \times n$ gives us an approximate size of the database. Notice that there are some expressions that increase the number of nodes of the documents, but since the increase is always linear and there is no looping mechanism, the complexity is restricted to a polynomial on the number of nodes of the initial trees.

Theorem. Given a DA expression E on database D with size $m \times n$ (as above), there is an algorithm $\mathcal{A}^E$ that can evaluate E in $O(f(m,n))$ time, where f is a polynomial function with parameters m and n.

Proof. We prove this by strong induction on the number of operators of E, as follows:

- *Induction hypothesis.* Given an algebraic expression E with k operators ($k > 0$), there is an algorithm $\mathcal{A}^E$ that can evaluate E using at most $f(m, n)$ operations, where f is a polynomial function. The possible operations here are (i) traversal based on node label (gi) and (ii) comparison of the leaf with a query value. Both operations are considered atomic and are assumed to take constant time.

- *Base case.* The base case is trivial. Here $E = D$ (number of operators = 0), and D can be evaluated in constant time.

- *Induction.* We consider all the possible algebraic operators discussed above and describe algorithms that can evaluate the expression. (Note that the algorithms here are essentially brute-force algorithms, and no claim on efficiency is being made at this time.)

  1. $E_n = E \circ \mathcal{P}$. Consider the following algorithm:

     - Let the number of trees in E be n_E.
     - For each GI in the path expression $\mathcal{P}$, perform breadth-first search on each of the n_E trees to select trees rooted at that particular gi, append any new matched node to a temporary list of trees, and after all the original trees have been considered, replace original list by the created temporary list. Continue this for every GI in $\mathcal{P}$.

     If the number of GIs in the path is p, then the maximum number of traversal operations is given by:

     $$\underbrace{m \times n_E + m \times (m \times n_E) + \ldots}_{k \text{ times}} = m \times f'(m, n) \times (1 + m + m^2 + \ldots + m^{k-1}) = f_{poly}(m, n)$$

     where $n_E = f'(m, n)$ is a polynomial in m, n by the induction hypothesis.

  2. $E_n = E_1^{\tau_1} \cup_R E_2^{\tau_2}$. This is trivial. By the induction hypothesis, we have E_1 and E_2 can be computed in polynomial operations. Suppose E_1 and E_2 return n_{E_1} and n_{E_2} trees respectively, and also suppose the maximum number of nodes in these trees is m. A simple algorithm to compute the union will start with one set, and for every element of the second set, check

if the element is already in the result, including it if not. This step requires comparison of two trees, and since exact matches are only considered, the number of string comparison operations required is the minimum of the nodes in the two trees (m in the worst case). The number of operations is $O(n_{E_1} \times n_{E_2} \times m)$, a polynomial from induction hypothesis. Note that this is not necessarily the most efficient way of performing this operation, but it is sufficient to show that the operation has a polynomial complexity.

3. $E_n = E_1^\tau - E_2^\tau$. Computation of this operation is also trivial and similar to the above, with the number of operations being $O(n_{E_1} \times n_{E_2} \times m)$, which is a polynomial (since by the induction hypothesis n_{E_n} and n_{E_2} are polynomials on the size of the input).
4. $E_n = E_1^\tau \cap E_2^\tau$. Computation of intersection is also trivial and has the same complexity as the above.
5. $E_n = E_1^{\tau_1} \times_R E_2^{\tau_2}$. Once again, this operation can be computed by two loops, one each for the two operands. Thus, the number of operations is $O(n_{E_1} \times n_{E_2} \times m)$, which is a polynomial (since by the induction hypothesis n_{E_n} and n_{E_2} are polynomials on the size of the input).
6. $E_n = \sigma_\gamma E^\tau$. We need to consider the following two cases, based on the form of γ:
   (a) $\gamma = \mathcal{P}\theta c$. Suppose the expression E^τ has n_E results. Consider the following algorithm:
      - For each of the trees $e^\tau \in E^\tau$, compute $e^\tau \circ \mathcal{P}$ using the method described above.
      - Compute the set membership of c on each of the results in the previous step.
      - Select the e^τ which returns non-zero members.

      Once again, the number of traversal operations in the first step is polynomial from before. Since c is a constant, the number of comparisons in the second as well as the third step is linear. Hence the total time is also polynomial.

(b) $\gamma = \mathcal{P}_1 \theta \mathcal{P}_2$. This is essentially the same as the previous method, the only difference being that, in the first step, both the operands need to be evaluated, while in the second step, the operation is a set intersection instead of membership. The combination is still a polynomial operation.

Hence, the proof follows by induction. ■

5.2 Practical Query Languages

5.2.1 DSQL - An SQL-like Language

This section describes *DSQL* (Document SQL)[3], an extended version of SQL which is a user-friendly pseudo-natural language form of DC. An informal introduction and examples of this SQL can be found in [Sen96]. The primary motivations behind having such a language is to provide users of database systems with a simple means for expressing queries using a natural language form. Also, since SQL is widely accepted as a standard query language for relational databases, it was a natural choice as a document database query language. DSQL is designed as an extension to the standard SQL-86 [SQL86b]. Conceptually DSQL supports SGML documents as the objects for constructing queries. From the language point of view, however, there are only two major differences from the standard SQL, which are the following:

1. *Path Expressions.* Path expressions are handled in the same way they are handled in the formal languages. To use path expressions, two main changes are made to SQL. The standard "**.**" operator used commonly in SQL to denote relation attributes can now be cascaded to express listed paths. In addition, a "**..**" operator is introduced, which is used to construct an abbreviated path from a listed path.

[3]Note that DSQL (or Document SQL) is different from SDQL (Standard Document Query Language, which is a part of the ISO 10179 DSSSL (Document Style Semantics and Specification Language) standard [ISO94].

2. *Complex selections.* Standard SQL deals with flat tables as primary objects, and hence specifies the output as a number of columns that constitute the output table. In DSQL, the primary objects on which queries are built are documents. To ensure closure, output of queries is also specified using document formulation constructs. To accommodate this feature, the select clause allows the creation of composite document types from constituent components. This is similar to the tuple construction operation in DC, using which, new types are created.

In this section, we present a subset of the complete DSQL language that we call core DSQL. This subset contains the core `SELECT - FROM - WHERE` construct of the language without any aggregate functions and nested queries. This core language is used primarily to demonstrate the power of the proposed extensions to SQL. The grammar for the complete language is given in Appendix A.

5.2.1.1 The Core DSQL

The core DSQL includes the basic `SELECT - FROM - WHERE` construct of SQL, without any aggregate functions and nesting. In addition, the core language does not include any grouping or ordering mechanism. In order to reduce the size of the grammar, we remove any implicit operator precedence in the logical operations. In addition, we only consider comparison predicates as in the formal language, but restrict the comparison operator to the $\ni$ operator as described earlier. To simplify the presentation, we represent the $\ni$ operator with the simple equality symbol $=$ (*i.e.,* the expression $A = c$ checks if the constant c is in the set returned by A).

The core DSQL syntax is presented below in a BNF form:

$$
\begin{array}{rcl}
\textit{query-exp} & ::= & \text{SELECT } \textit{output qry-body} \\
\textit{output} & ::= & \textit{outputname}(\textit{target}) \\
\textit{target} & ::= & \textit{scalar-exp-list} \mid * \\
\textit{scalar-exp-list} & ::= & \textit{scalar-exp}\ [, \textit{scalar-exp}]^* \\
\textit{qry-body} & ::= & \textit{from-clause}\ [\textit{where-clause}] \\
\textit{from-clause} & ::= & \text{FROM } \textit{db-list} \\
\textit{db-list} & ::= & \textit{db}\ [, \textit{db}]^*
\end{array}
$$

$$
\begin{array}{rcl}
db & ::= & path\text{-}exp\ [alias] \\
where\text{-}clause & ::= & \text{WHERE}\ search\text{-}cond \\
search\text{-}cond & ::= & [\text{NOT}]\ search\text{-}cond \mid search\text{-}cond\ \text{AND}\ search\text{-}cond \\
 & & \mid search\text{-}cond\ \text{OR}\ search\text{-}cond \mid bool\text{-}term \\
bool\text{-}term & ::= & comp\text{-}pred \mid (search\text{-}cond) \\
comp\text{-}pred & ::= & scalar\text{-}exp\ =\ \{salar\text{-}exp \\
scalar\text{-}exp & ::= & atom \mid col \\
col\text{-}list & ::= & col\ [,col]^* \\
col & ::= & path\text{-}exp \\
path\text{-}exp & ::= & path\text{-}list\ [..path\text{-}list]^* \\
path\text{-}list & ::= & gi\ [.gi]^*
\end{array}
$$

The above BNF captures the complete syntax of the core DSQL language. The basic idea is the same as the calculus language presented earlier, quantifications are the only missing operations from the calculus presented earlier. The complete language presented in Appendix A includes all the advanced features of SQL. The primary motivation for a core subset of the language is to identify the most critical features of the language and provide methods for implementation of such features. In Chapter 6, we show how this language is implemented using available systems and languages.

The core DSQL is the most important starting point in the design of a practical document query language. The most important aspect of this language is that it provides a link to the theoretical foundations introduced earlier in this chapter and allows extensions to the language to be implemented on top of the core component with a known expressive power. We show here that any DSQL query can be expressed using an equivalent DC query.

Theorem Any core DSQL query is equivalent to some DC query.

Proof. We prove this by taking a core DSQL query and showing how the same query is equivalent to a DC query. A completely general query is difficult to formulate,

so we take a query representing all the features of the core DSQL:

$$
\begin{array}{rl}
\texttt{SELECT} & R(D.P_{o0}, A_1.P_{o1}, A_2.P_{o2} \ldots, A_3.P_{ok}) \\
\texttt{FROM} & D, D.P_{a1}\ A_1, D.P_{a2}\ A_2, \ldots D.P_{ak} A_k \\
\texttt{WHERE} & D.P_{j1} = D.P_{j2} \\
\texttt{AND} & A_1.P_{c1} = a_1 \\
\texttt{OR} & A_2.P_{c2} = a_2 \\
\texttt{AND} & \texttt{NOT}\ A_k.P_{ck} = a_k
\end{array}
$$

The above query is not fully general, but it includes most of the prominent features of the core DSQL. In the above, P_{oi} represents output path expressions, A_i represents aliases, and a_i represents constants. This query is essentially a rewritten version of the following DC query:

$$
\begin{array}{rcl}
Q^{DC} & = & \{z | z = R\langle z_0, z_1, \ldots z_k\rangle \wedge \\
& & D.P_{a1}(A_1) \wedge D.P_{a2}(A_2) \wedge \ldots \wedge D.P_{ak}(A_k) \wedge \\
& & D.P_{o0}(z_0) \wedge A_1.P_o1(z_1) \wedge A_2.P_{o2}(z_2) \ldots, A_3.P_{ok}(z_k) \wedge \\
& & D.P_{j1} = D.P_{j2} \wedge A_1.P_{c1} = a_1 \vee A_2.P_{c2} = a_2 \wedge \neg(A_k.P_{ck} = a_k)\}
\end{array}
$$

The equivalent expressions for any core DSQL query can be constructed in a similar manner. ■

5.2.1.2 Examples

Consider the same queries described earlier (Section 5.1.1.4), using the same database schema as before. The following are the same queries in DSQL. All the queries cannot be solved using the core DSQL, so we use the full DSQL language. Notice that these queries are almost direct translations from the calculus queries shown in Section 5.1.1.4.

1. Find all poems that contain the word "love" in the poem title.

```
SELECT poem
FROM poem
WHERE poem..title = "love"
```

2. Extract titles and authors of all poems in the database.

```
SELECT R(poem..title, poem..poet)
FROM poem
```

3. Find the period in which all poems had the word "love" in their titles.

```
SELECT X
FROM poem..period X
WHERE NOT EXISTS (
  SELECT * FROM poem Y
  WHERE Y..period = X
  AND NOT (Y..title = "love"))
```

4. Find the pairs of names for poets who have at least one common poem title.

```
SELECT R(P1..poet,P2..poet)
FROM poem P1, poem P2
WHERE P1..title = P2..title
AND P1..poet <> P2..poet
```

5. Find the poems that do not have the word "love" in the title.

```
SELECT poem
FROM poem
WHERE NOT (poem..title = "love")
```

5.2.2 SQL in the SGML Context

During the discussion on the closure requirements in Chapter 4, we observed two main types of closure. Closure in the context of query languages primarily involves the input and output of queries. To achieve closure, query languages need to provide the result of queries in the same conceptual form as the inputs. In the relational model, relational query languages (such as relational algebra and calculus) use relations as the input and describe an output relation containing the result of the query. However, we also mentionedQBE , a query language that uses relational skeletons to specify queries. In this language, in addition to the inputs and outputs, the query language itself is "closed" under the notion of tabular representations. This stronger notion of closure can be easily achieved in a document database context by using SGML itself as a query language.

In Chapter 2, we described SGML as a meta-language, which can define languages which, in turn, define valid document instances. Thus, SGML can be conveniently used to define a query language. The DSQL syntax described in the previous section can be translated into an SGML DTD which can be used to write valid queries. There are a few distinct advantages of using SGML as a query language:

- First and foremost, this query language retains the properties of both SQL and SGML. Being an application of SGML, this language is inherently portable and is independent of the underlying system and platform. On the other hand, since it is equivalent to DSQL, the DSQL DTD defines a first order, low complexity query language.

- Since queries are in SGML, which is the same data format as the database itself, the queries can be stored and managed in the same way as the data itself. This immediately implies some interesting possibilities:

  - Queries can be stored as data and, subsequently, can be queried themselves to extract information that will be very suitable in applications such as data mining and performance tuning.

  - The capability of storing queries as data allows subsequent treatment of

data as queries. This ability is commonly known as "reflection" in programming languages, and gives a language a higher expressive power and the capability of performing meta-data queries. Many attempts of providing reflection support in query languages have been researched [JMG95], and the use of SGML as a query language for SGML databases provides a natural way to achieve this property.

- Queries formulated and stored in SGML can easily be converted into any other query language (including visual query languages) without much effort.
- Users posing queries in SGML can do so within their familiar environment of SGML editors. This capability also ensures that they do not have to learn the syntactic details of a new language, and a validating editor will ensure that all the queries are valid DSQL queries.
- SGML queries can be seamlessly integrated within other SGML documents (possibly using the SUBDOC feature of SGML) for dynamic document content. Queries embedded in a document can be replaced by the results obtained from the queries before presenting the final document. This is a natural way of dynamic document content generation for the WWW.

A Document Type Definition for the SGML implementation of DSQL and description of all the generic identifiers is presented in Appendix A.

5.2.2.1 Examples

Consider the same queries described earlier (Section 5.1.1.4), using the same database schema as before. Here, we present the same queries written in SGML using the SQL DTD. The queries are completely normalized, so they display all the necessary open and close tags, and hence have somewhat expanded size. However, in real situations, these queries will be created using an SGML editor or a translator from the regular SQL, and most of the tags as shown here will be hidden from the users.

1. Find all poems that contain the word "love" in the poem title.

```
<select>
  <output><scalar><col><pathlist><gi>poem</pathlist></col></scalar>
  </output>
<from><db><pathlist><gi>poem</pathlist></db></from>
<where><cond><predicat><compare EQUAL>
   <scalar><col><pathlist><gi>poem</pathlist>
      <pathlist><gi>title</pathlist></col></scalar>
   <scalar><atom>"love"</scalar>
</predicat></cond>
</where>
</select>
```

2. Extract titles and authors of all poems in the database.

```
<select><output name="R">
  <scalar><col><pathlist><gi>poem</pathlist>
    <pathlist><gi>title</pathlist></col></scalar>
  <scalar><col><pathlist><gi>poem</pathlist>
    <pathlist><gi>poet</pathlist></col></scalar>
  </output>
<from><db><pathlist><gi>poem</pathlist></db></from>
</select>
```

3. Find the period in which all poems had the word "love" in their titles.

```
<select>
  <output><scalar><col><pathlist><gi>X</pathlist></col></scalar>
  </output>
<from><db alias="X"><pathlist><gi>poem</pathlist>
  <pathlist><gi>period</pathlist></db></from>
<where><cond><predicat><exists NOT>
  <select>
    <output><all></output>
  <from><db alias="Y"><pathlist><gi>poem</pathlist></db></from>
  <where><cond>
    <cond>
    <predicat><compare EQUAL>
      <scalar><col><pathlist><gi>Y</pathlist>
        <pathlist><gi>period</pathlist></col></scalar>
      <scalar><col><pathlist><gi>X</pathlist></col></scalar>
    </predicat></cond>
  <logic AND>
  <cond NOT>
    <cond>
      <predicat><compare EQUAL><scalar><col><pathlist><gi>Y</pathlist>
       <pathlist><gi>title</pathlist></col></scalar>
       <scalar><atom>"love"</scalar></predicat></cond>
    </cond>
```

```
  </cond>
  </where>
</select>
</predicat></cond></where></select>
```

4. Find the pairs of names for poets who have at least one common poem title.

```
<select>
  <output><scalar><col><pathlist><gi>P1</pathlist>
    <pathlist><gi>poet</pathlist></col></scalar>
  <scalar><col><pathlist><gi>P2</pathlist>
    <pathlist><gi>poet</pathlist></col></scalar>
  </output>
<from><db alias="P1"><pathlist><gi>poem</pathlist>
  </db><db alias="P2"><pathlist><gi>poem</pathlist></db></from>
<where><cond><cond><predicat><compare EQUAL>
  <scalar><col><pathlist><gi>P1</pathlist>
    <pathlist><gi>title</pathlist></col></scalar>
  <scalar><col><pathlist><gi>P2</pathlist>
    <pathlist><gi>title</pathlist></col></scalar>
  </predicat></cond>
<logic AND>
  <cond NOT><cond><predicat><compare EQUAL>
    <scalar><col><pathlist><gi>P1</pathlist>
      <pathlist><gi>poet</pathlist></col></scalar>
    <scalar><col><pathlist><gi>P2</pathlist>
      <pathlist><gi>poet</pathlist></col></scalar></predicat></cond>
  </cond></cond>
</where></select>
```

5. Find the poems that do not have the word “love” in the title.

```
<select>
  <output><scalar><col><pathlist><gi>poem</pathlist>
    </col></scalar></output>
<from><db><pathlist><gi>poem</pathlist></db></from>
<where>
  <cond NOT><cond><predicat><compare EQUAL>
    <scalar><col><pathlist><gi>poem</pathlist>
      <pathlist><gi>title</pathlist></col></scalar>
    <scalar><atom>"love"</scalar></predicat></cond>
  </cond>
</where></select>
```

Chapter 6

Implementation

This chapter describes the architecture as well as the actual implementation of all the components of the proof-of-concept prototype of a document database system, which we call DocBase. DocBase has a client-server architecture. The server-side applications and command-line client applications are Unix-based, but the query interface clients are web-based and, hence, platform-independent. In this chapter, we first introduce the platforms, supporting applications and languages that were used to develop this prototype. We then describe the architecture and the physical data representation used in the implementation of DocBase. We then describe the query engine architecture and how queries from the user are processed. The implementation of the web client interface is described in Chapter 7 as part of the user interface development.

6.1 Languages, Platforms and Tools

C++ was the primary programming language used for the implementation of the command-line and backend clients. The Java^{TM} programming language was used for implementing the web-based query interface client. One important consideration behind the use of object-oriented languages was that they ensure easy extensibility using inheritance and overloading. Program components specific to particular platforms were kept limited to subclasses of the platform-independent generic superclasses implemented as virtual classes in C++. This type of design assures a simple design through use of features of existing applications. In the prototype implementation, we used external applications for storage management and index building.

The prototype system was designed to run on Unix. In particular, the storage

management server and SGML index management servers were Unix-based applications. Hence, all the storage and retrieval functions were limited to Unix platforms. We used a SUN Sparc-5 system as a test server for the prototype application.

The query interface client developed in Java was, however, platform-independent because of the availability of Java virtual machines on most platforms. The application was developed on a Unix workstation but tested on all the platforms that support Java, and it was found to work satisfactorily. More details on the design and implementation of this Java interface are given in Chapter 7.

The primary supporting applications in the prototype were storage management and index management applications. The function of the storage manager was to store the special indices and catalogs, and the function of the index management module was to create special indices on the SGML documents and to facilitate navigation of the hierarchical document structure using these indices. Query processing capabilities were built into clients of the storage management system. Figure 13 shows exactly where these applications are used in the architecture of DocBase. Details on these applications are presented next.

6.1.1 Storage Management Applications

The Exodus storage manager [CDF+86] was the primary storage management server used in this prototype. Exodus is a storage manager developed at the University of Wisconsin which is frequently used in the management of extremely large volumes of data. Exodus allows low-level handling of its data using a native Application Programming Interface (API) that can be used in an application to manipulate the stored object in the storage manager.

Exodus has a client-server architecture. Exodus clients are applications that use pre-defined procedures from a *client-library* provided by Exodus. These client library procedures are used to establish a connection with the server and to initiate storage and retrieval tasks.

Exodus provides three primary kinds of services to its clients:

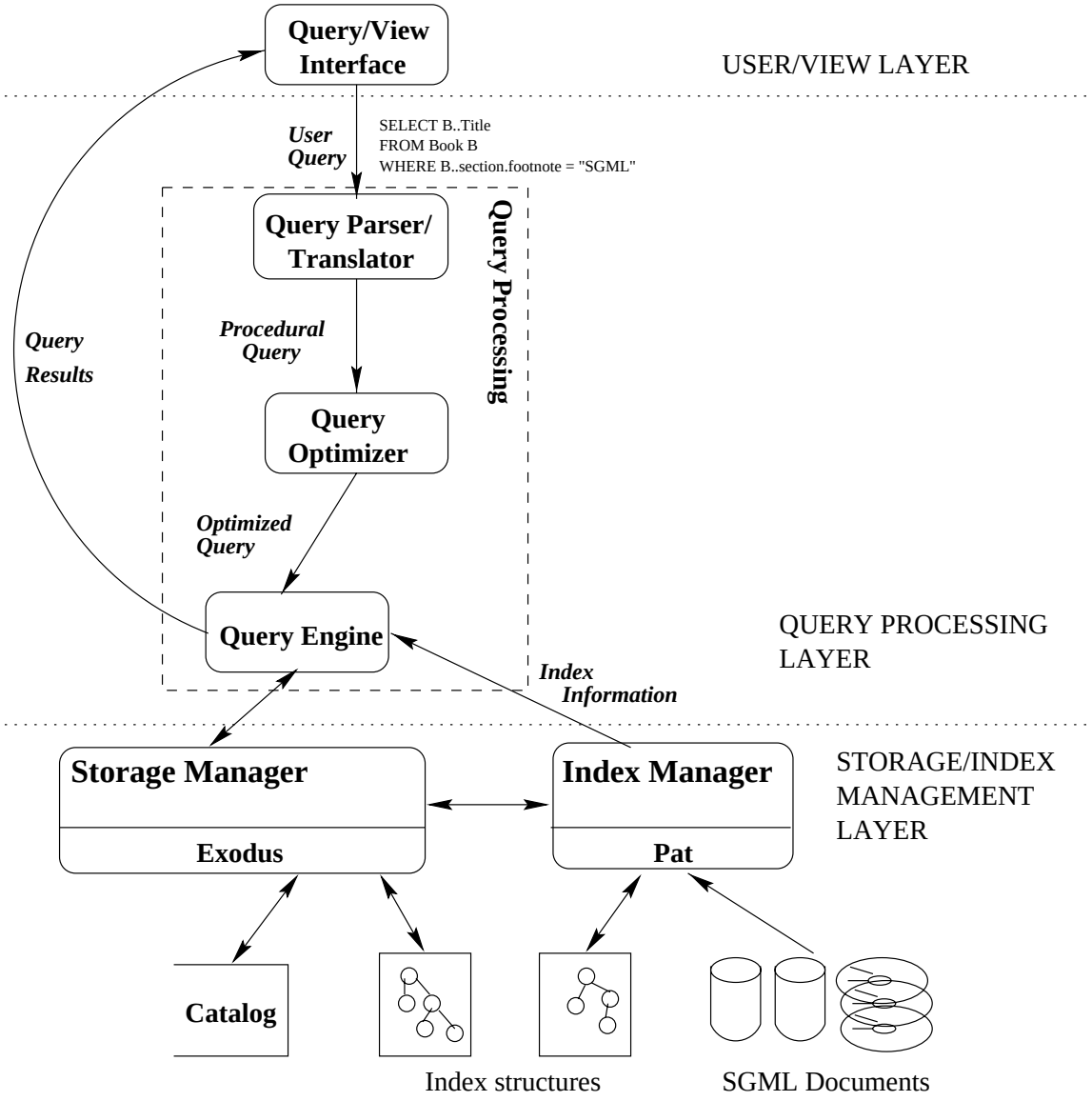


Figure 13: The architecture of DocBase

- *Storage Management.* Storage management services include storage abstractions and procedures to manipulate these abstractions. The basic storage abstraction in Exodus is an *object* of any arbitrary size. Objects are stored in pages of fixed sizes. Exodus is capable of building linear hash and B-tree indices on the objects based on a key in order to speed up the retrieval process. Storage management objects are persistent (*i.e.*, the objects are preserved in secondary storage even after the client terminates).

- *Buffer Management.* Buffer management services include the process of efficiently using the available main memory to store data temporarily to speed up the read and write operations. Buffers are volatile objects (*i.e.*, they are not saved in secondary storage unless their contents are explicitly saved by the client). Clients can have local buffers that are in the clients' memory space and are removed when a client terminates. Clients can also utilize server-side buffers that are in the server's memory space and are only removed when the server terminates.

- *Transaction Management.* Transaction management involves controlling concurrent access to the stored data for read and write operations, as well as recovery of stored data in the case of an abnormal server shutdown. Clients need to initiate transactions and commit the transactions when the operations are completed. The server keeps a log of these transactions and uses the log to recover from any unexpected failure.

Later in this chapter, we will discuss in detail the types of objects used in this prototype. The primary objects stored by clients in the prototype are simple data-offset pairs and derivatives of such objects. Exodus was chosen in this prototype for its ready availability and its flexibility on the types of objects it can handle as well as its capability of providing built-in index structures. However, the storage management features can be performed easily by any system that is able to store and retrieve simple binary records. For example, a relational database system can be used

to store and retrieve the storage objects[1].

6.1.2 Index Management Applications

The primary index management application used in this prototype is the Pat system from Open Text [Ope94]. Pat was developed at the University of Waterloo as a full-text searching and indexing system for text repositories. Pat uses the Patricia tree structure (discussed in Chapter 3) for its internal index representation. Pat has a client-server architecture, although unlike normal client-server systems, the Pat server does not run continuously waiting for connections from clients. Clients using the Pat API typically invoke Pat in a "quiet mode" as a child process and redirect the input/output operations from Pat using Unix pipes. Inputs sent to Pat use the query language provided by Pat (discussed below), and the outputs from Pat use a tagged format that can be parsed either by the client code or with the "SINSI" application programming interface provided as part of the Pat distribution.

Data is added to a Pat database using an indexing process. Pat does not have its own storage management features. Documents indexed by Pat are left in the secondary storage as standard files. Pat only creates special index files that can be used to speed up the search process. Two primary types of index structures are created by Pat when used on a document repository structured by SGML. The first index structure, called the "main index" or the "word index", is based on the Patricia tree structure [BYG89]. A short description of this structure is given in Chapter 2. The second structure, called the region index, is a similar structure created using only the meta-data information contained in the SGML tags in the document.

The Pat query language. Pat provides a query language that reflects the capabilities of the Patricia tree structure [BYG89, GBY91] (the basic building block of Pat indices) and allows efficient computation of various kinds of searches, most common among them being prefix searches. Every operation in the Pat query language returns a set of offsets (positions in the documents) where a match is found. Queries in this

[1] In fact, an alternative implementation of the storage management module was built using the Sybase relational database management system. This storage manager was used primarily for the purpose of testing the primary storage manager.

language can return the offsets of either the data that match the query or the regions (meta-data) within which the match is obtained. The types of Pat query language operations that were used most frequently in this implementation are the following:

- *Prefix search.* A string enclosed within double quotation marks constitutes a prefix search and returns document positions where strings with the given prefix are located. For a small search string, the complexity of this operation (measured by the number of traversal operations on the index structure) is proportional to the length of only the input string (see Chapter 3 for a discussion on prefix searches with Patricia trees).

- *Bounded prefix search.* A query of the form "`region A including "string"`" returns document positions rooted at the GI "A" that contain the prefix "string." The implementation of this query in Pat involves a search for the region "A" in the region index and a search for the prefix "string" in the word index, followed by an inclusion test. The first two operations are linear in complexity to the search strings (using prefix search algorithm on Patricia trees). The intersection can be performed by scanning both sets and selecting common elements, with a complexity linear to the number of elements in each set. This linear complexity is possible since Pat index operations always return results sorted by the offsets, because of the left-to-right storage and retrieval method in Patricia trees. However, because of the proprietary nature of the data structures and operations implemented in the commercial Pat software, it is not known whether Pat uses this exact strategy.

- *Traversal to ancestor nodes.* A query of the form "`region A including region B`" returns document positions rooted at the GI "A" that include document positions rooted at the GI "B" and, in effect, returns the ancestors of "B" with label "A." This operation involves selection of elements from the second set which are included within the bounds of some element of the first set, and can be performed using a linear scan operation. The complexity, as before is linear in the size of the individual components (*i.e.,* the number of elements of "region A" and the number of elements in "region B").

- *Traversal to descendant nodes.* A query of the form "`region B within region A`" returns document positions rooted at the GI "B" that fall within document components rooted at the GI "A". This operation can also be computed in linear time using a scan operation on the two sets.

- *Set union.* A query of the form "$Q_1 + Q_2$" returns a set union of the results of the two queries Q_1 and Q_2. Since in Pat, the results are always ordered in terms of the offset, the actual union operation only has linear complexity.

- *Set intersection.* A query of the form "$Q_1 \wedge Q_2$" returns a set intersection of Q_1 and Q_2, and has a linear complexity on the size of the sets, using a linear scan and merge operation to combine the two sets.

6.2 An Architectural Overview of DocBase

The architecture of DocBase closely follows the tri-level design of database systems described in Chapter 5. In this architecture, there are three distinct layers: (i) a top layer involving interaction with the user, (ii) a middle layer involving query parsing, translation and optimization, and (iii) a bottom layer involving actual processing of the query using a storage manager and an index manager (see Figure 13).

Figure 13 presents an overall view of the DocBase system and the life-cycle of a query during its processing. Details on each of the components will be presented later in this chapter. The rest of this section describes the distribution of the data and indices as well as the typical data flow process for evaluation of a query.

6.2.1 Data Distribution

In this section, we describe the distribution of the data in the prototype among the applications that process the data. In particular, we consider the data (in the form of SGML documents and document type definitions), the index structures and the meta-data or catalog information. In the current implementation of DocBase, a structured document database is physically viewed as a collection of SGML documents, each

document (or possibly a set of interlinked documents) conforming to a valid SGML document type definition (DTD). There could be multiple DTDs, but every document must conform to one of these DTDs. All of these documents are stored as standard text files in the file system. In addition to the documents, special structures for the purpose of indexing and searching are also stored. In this section, we present the details on how the data (SGML documents), indices, and catalogs are physically stored in DocBase.

Data. In the current prototype, the data is stored in the form of SGML documents in a file system. While not an ideal method for a database representation, this was necessary to allow the use of Pat for the index creation process. For this prototype, advanced storage management issues such as concurrency control and recovery of documents were not considered. In addition, in order to keep a correspondence between the documents and their physical storage, documents conforming to the same DTD were stored in the same distinct directory of the file system.

However, this distribution is not crucial for the functionality of the system. The current implementation of DocBase only has support for the core DSQL language including simple selections and joins, and hence the input SGML data is never modified by a query. The SGML documents are only accessed by Pat and its indexing applications (see Figure 13).

Indices. Two types of indices were used for processing queries. Indices of the first type are created by the Pat indexing applications. Indices of the second type are created to speed up queries that cannot be processed using the Pat query language described earlier in this chapter (Section 6.1.2). As shown in Figure 13, the Pat-specific index structures are accessed and modified by Pat, and the auxiliary index structures are managed by the storage manager.

1. *Pat Indices.* In order to support the operations provided by the Pat query language, a Pat application needs to create some specific indices based on the input documents. The Pat indices are special binary files in a Unix file system. Pat indices of many types were created and used in this prototype: (i) word indices to speed up the search for words or phrases in the database (files with an extension of `.ind`), (ii) region indices for searching for keywords delimited

by SGML elements (files with an extension of `.rgn`), and (iii) *fast find* indices - a special auxiliary index structure supported by Pat for databases spread over multiple files in a file system (files with an extension of `.ffi`). Because of the proprietary commercial nature of Pat, the internal formats of these indices were not available. However, it was known that both the word and region indices use the Patricia tree structure discussed earlier (Section 3.2.2.1).

2. *Join Indices.* In addition to the Pat indices, auxiliary index structures were created for the processing of queries not supported by Pat (typically the queries involving joins). The auxiliary indices developed for the purpose of processing queries involving joins were created and maintained using the Exodus storage manager. These auxiliary indices can be specifically built or can be dynamically created when necessary. These join indices are created by first using a Pat query to extract the proper offsets, and then indexing the result obtained from the query. Details on these index structures are given later in this chapter.

Catalog. Pat keeps track of the indices it creates in a data description file using a tagged format. This file (known as the data description file, with an extension of `.dd`) is also stored in the file system as a regular text file. The information contained in this file is primarily for use by Pat in processing its queries. The current implementation of DocBase also creates a detailed catalog of objects in the database, including a binary representation of the document structure and a list of the different types of objects (*e.g.,* SGML documents, DTDs, stored queries, auxiliary join indices and temporary structures). More details on these structures will be provided later in this chapter. As shown in Figure 13, the storage manager has full control over this catalog information.

6.2.2 The Life Cycle of a Query

This section describes the process by which a query is formulated, processed and evaluated in the current implementation of DocBase. Note that this section only describes the flow of the query as shown in Figure 13. The details on the operations and associated algorithms will be presented later.

A query is usually formulated by the user by using either (i) a command-line interface to directly specify the query in DSQL, or (ii) a graphical user interface to express the query using a simple visual template. Details on the design and implementation of the visual interface are described in Chapter 7.

Queries from the user interface are processed by a parser. In addition to determining the validity of the query, the parser also translates the query into a list of individual operations (or query fragments).

The query components generated by the parser are optimized by a query optimizer and evaluated by the query engine. Currently the query optimizer is used primarily to determine the nature of each of the query fragments and to determine an access plan for processing each fragment. Based on the type of query fragment, the target element of the query fragment when it is evaluated, and the current state of the computed result, one of three possible decisions is made:

- *Evaluate later.* The query fragment can be easily processed by Pat, and the current state of the query allows it to be evaluated using the Pat query language. In this case, the optimizer simply generates the query in the Pat query language and stores the current result of computation as the query itself.

- *Evaluate now.* The query can be evaluated by Pat but cannot be further processed using Pat itself. In this case, the appropriate query is constructed and sent to the Pat server for evaluation. The result is left in Pat's storage space.

- *Store now.* The current state of the query and the new query fragment cannot be combined using Pat operations. This is usually the case for join queries. In this case, the results are extracted from Pat and indexed in the storage manager, while the rest of the computation continues in the storage manager or, if possible, the result is written back to Pat's storage space for use with subsequent query fragments.

The primary function of the query engine is to evaluate the query components generated by the parser using one or more of the methods described above. For the most part, the query operations are translated to corresponding index operations using the

Pat query language. The results of these operations are stored as Pat queries, but the actual evaluation of the queries is delayed as much as possible. Joins are given special attention since they cannot be performed using the Pat index operations. When a join is detected, the query is processed in the following steps: (i) identifying the two components of the join operation, (ii) evaluating the components separately using Pat indices, (iii) dynamically creating storage manager indices based on the intermediate results (if such an index does not already exist), and finally, (iv) performing the join using these special index structures.

Once all the query fragments have been processed, the query engine determines the structure and format of the output and then combines the query fragments. If all the query fragments can be processed by Pat, this final stage consists of sending the resulting Pat query to the Pat server, extracting the result, and possibly rearranging it for presentation. In the case that portions of the query cannot be evaluated using Pat, the storage manager indices are used to evaluate those portions, and the result is converted back into Pat's storage space to combine the result with the rest of the query. The final result is extracted from Pat as a set of SGML document fragments. The current prototype of DocBase does not implement nested queries. However, it allows results of queries to be stored internally as a set of "virtual documents", similar to database views in relational databases. These "virtual documents" are not proper SGML documents; they are simply a set of offsets in the document from which the fragments were extracted. These virtual documents can be used later in a query to achieve the effect of nesting.

6.2.2.1 Examples of the query processing method

The query life cycle is best demonstrated using examples. Suppose we want to process the following query on the sample database for which the structure is shown in Figure 14:

```
SELECT B.author
FROM Book B
where B..chapter..head = "SGML"
```

```
and B..section..head = "optimization"
```

This is a simple query without any join conditions. When the query is parsed, a query tree is built in which the two main query components are the two conditions in the `WHERE` clause. Currently, there are no optimization methods for reordering the conditions, so they are evaluated in the order they appear in the query. Moreover, since the `FROM` clause contains only one document component, only one accumulator is sufficient to evaluate this query. The following steps are used in the evaluation of this query:

1. Evaluate the query components in the `FROM` clause. Since there is only one component, we only have one accumulator B, and it is initialized with `B = region Book`

2. Evaluate the first condition. The "evaluate later" method is used, and the query is stored as the Pat expression `q1=(region head within (region chapter within (*B))) incl "SGML"`

3. Evaluate the second condition. The "evaluate later" method is used again, and the query is stored as the Pat expression `q2=(region head within (region section within (*B))) incl "optimization"`

4. The logical connective is now found. Since two unevaluated queries need to be combined, a series of operations are performed using the "evaluate now" method. In this case, because of the conjunction, a set intersection operation is used to combine the individual selections. The following operations are performed:

   ```
   q1 = (*B) incl (*q1)
   q2 = (*B) incl (*q2)
   q3 = *q1 ^ *q2
   ```

   In the first step, the first condition is processed by evaluating it relative to the corresponding accumulator. The second condition is similarly evaluated, and then an intersection operation is performed to combine the two results.

5. Finally, the result needs to be determined. The path expression in the `SELECT` clause requires a traversal down to the *author* region. Since all the conditions have been evaluated, the accumulator is first updated with the result of the conditions, and then a traversal for the path expression is performed to obtain the final result, as follows:

```
B = *q3
final=(region author within *B)
```

In the case of a query with a join condition, the comparison is usually between two path expressions. In this case, both the path expressions are evaluated and stored using a "store now" method and evaluated in the storage manager. The details on processing of queries with join conditions will be presented shortly.

6.3 Physical Data Representation

A single storage manager can be used to handle the data as well as structures built on top of the data. A single storage manager is sufficient if the storage manager is capable of creating and processing the index structures in addition to storing and handling the data and the catalog. However, if the index structures are created and managed by an external system, it is often necessary to let this system manage its data and indices. This does affect the control that the storage manager has over the data, but it provides more flexibility in the implementation, since this enables the use of external indexing applications in managing the indices and reduces the complexity of the internal indexing process. In this section, we first describe an ideal data representation that facilitates processing of the class of queries described as "Core DSQL" in Chapter 5. We next describe a variant of this structure that was implemented and briefly compare the two methods.

6.3.1 Ideal Data Representation

The ideal implementation would have all of the data controlled by one storage manager. In this case, the storage manager would have full control over the documents,

catalog information, full-text indices, and structure indices. The advantage of a single storage manager is that the various components of the stored elements (*e.g.,* data, catalog, indices) can be kept synchronized easily, since the storage manager can easily determine the dependencies between data and indices and decide when an index needs to be rebuilt. Any external system will then need to access the data through the storage manager. The primary types of data to be managed are (i) native data (SGML documents) (ii) meta-data (catalog information) and (iii) auxiliary structures (indices). Figure 14 depicts a simplified representation of the interaction of these three types of information handled by the storage manager. Analogous to the above three types of data to be handled, three primary types of data structures are necessary for the query processing: (i) a hierarchical structure for the actual parse tree for document instances, (ii) a hierarchical structure for the catalog (representing the DTD) and (iii) optional auxiliary index structures on the meta-data for the purpose of efficient query processing (see Figure 14).

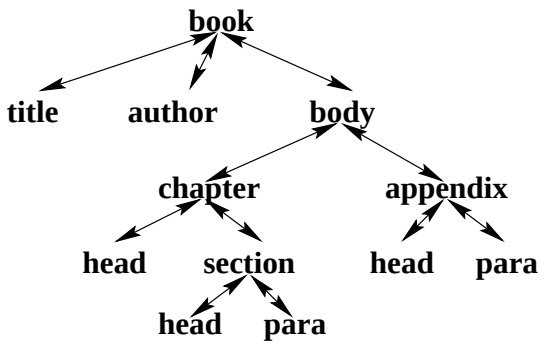
6.3.1.1 The Parse Tree

The parse tree shown in Figure 14(c) is an instance-level structure representing the hierarchical structure of the actual document instances. Prior to the incorporation into the database, every document needs to be parsed by an SGML validating parser to assure the conformance of the document to a DTD. The structure created by the parser is the parse tree generated from the particular document instance. This parse tree contains all the information about the structure of the document, but does not replicate the actual text present in the document. Instead, each node in the parse tree contains information on the *offset* in the actual document from which the data can be obtained. This not only reduces the size of the tree but also eliminates the necessity of recreating documents as query results from fragmented components. The additional overhead of performing a "seek" in the original document can be reduced by implementing the storage manager accordingly.

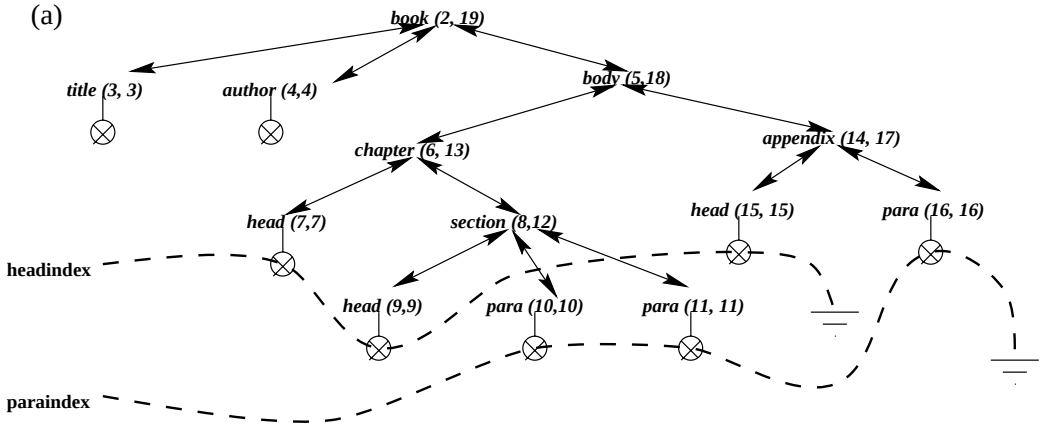
The basic structure of the parse tree is a normal tree structure with bidirectional edges between parents and children. This structure is primarily used for performing

```
1. <!DOCTYPE book SYSTEM "book.dtd">
2. <book>
3. <title>More about SGML</title>
4. <author>Jane Doe</author>
5. <body>
6.   <chapter>
7.     <head>First chapter</head>
8.     <section>
9.       <head>The SGML standard</head>
10.      <para>First para</para>
11.      <para>Second para</para>
12.     </section>
13.  </chapter>
14.  <appendix>
15.    <head>SGML DTD</head>
16.    <para>The DTD is pretty big</para>
17.  </appendix>
18. </body>
19. </book>
```

(a)



(b)



(c)

Figure 14: A simple representation of the data structures: (a) the SGML document (b) the catalog structure and (c) the parse tree and auxiliary indices

the necessary structure traversal for evaluating path expressions. The simplest approach for finding path expressions is to perform a breadth-first search, optimizing the search by pruning nodes that can never express the given path expression. The details of an algorithm for evaluating path expressions is given in Section 6.4.2.2.

6.3.1.2 The Catalog

The catalog (see Figure 14b) is a schema-level structure representing the hierarchy of the generic identifiers defined by the DTD. The catalog is essentially a simple internal representation of the DTD but only includes the structural relationship and not the additional information needed for parsing (*e.g.,* attribute types, omission rules). In addition, the catalog structure does not distinguish between the different types of content groups (*e.g.,* option groups, sequence groups), but only includes all elements in the content group of a particular GI as child nodes of the GI in the tree representing the structure. Technically, the catalog is also a tree structure with bidirectional links, created from the DTD.

The catalog is used primarily to evaluate and optimize path expression queries. Any path expression is first compared with respect to the catalog to decide if that path expression can ever be evaluated in the given DTD. In this way, the catalog can be used to prune the search paths that can never match the path expressions. Details on the algorithms will be given in a later section (Section 6.4.2.2).

6.3.1.3 Join Indices

In addition to the parse tree structure that can be considered as a special index structure, additional index structures need to be created in order to speed up the processing of join queries, which cannot be evaluated using the Pat query language. These auxiliary join indices are shown in Figure 14(c) as horizontal chains across the parse tree, connecting similar nodes in the parse tree. The simplest type of index is just a linked list of the nodes for a particular GI, although usually they are implemented using B^+-trees, hash structures, or application-specific index structures. Not all GIs need to be indexed, and the catalog contains information on whether a

particular GI is indexed.

Although termed as *join indices*, these auxiliary indices can also be used for fast processing of queries involving selection on the particular GI on which the indices are created. In the prototype, these indices are often built "on the fly" when a join operation is evaluated.

6.3.2 Implementation of the Data Structures

The current prototype of DocBase makes a trade-off between the implementation of the necessary structures from scratch and the use of available commercial and non-commercial applications that implement similar structures. Instead of using a standard SGML parser to parse the SGML documents and building the parse tree structure, the implementation uses Pat region index and word indices, since most[2] of the navigational operations on the parse tree can be performed using these indices. The Pat region index is implemented using a Patricia tree of the document tags (regions) and has approximately the same functionality as a parse tree. The Pat word index creates a Patricia tree index based on the character data in the document, thereby speeding up word searches. To ensure that the system is not completely dependent on the Pat indices, the index manager is implemented using a virtual superclass "Hier_engine," with the Pat-specific functionality in a subclass "pat_engine" (see Figure 15). This ensures that support for other index management applications can be added to DocBase merely by implementing a new subclass of "Hier_engine." For the programming-level interface of these classes, refer to Appendix B.

The catalog structure is created as described above from a DTD and an optional configuration file that includes the GIs that need to be indexed. If no configuration files are present, all the GIs are indexed. The configuration file allows the user to select a subset of GIs for the purpose of querying. If a GI present in a DTD is not included in the configuration file, no queries can use that GI in a term. However, the

[2]Not all tree traversal operations can be performed using the Pat indices. Because of the way Pat flattens the structure to perform its queries, it is not possible to obtain an immediate child or immediate parent of a node using the Pat indices. Navigation can only be performed to named ancestors and descendants.

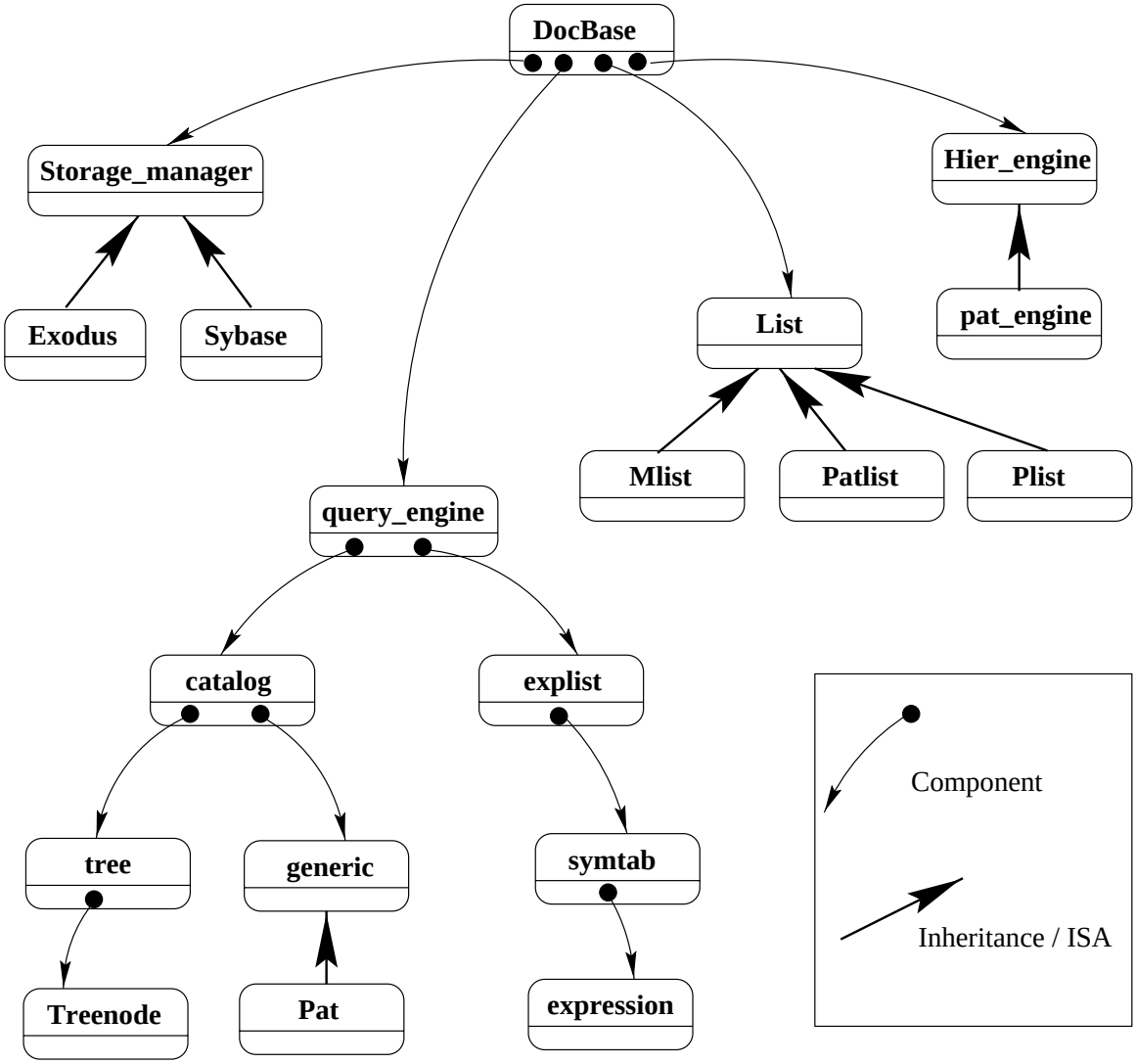


Figure 15: The class hierarchy of the DocBase query processing system.

elements in the instances corresponding to the non-indexed GIs are still available in the parse tree (implemented using the region index of Pat). The configuration file follows a very simple syntax, containing three fields in each line: (i) the first field containing the name of the GI (case insensitive), (ii) the second field containing a description of the GI separated by a '/' from the first field, and (iii) an optional third field, consisting of a single asterisk '*' for one of the GIs, indicating that the corresponding GI is to be used as the default GI for the purpose of querying. The catalog is a component of the "query_engine" class and is implemented as described above using a "tree" class consisting of nodes implemented by the "treenode" class (see Figure 15).

The catalog is created using a combination of perl and C++ code. The perl code was used primarily to utilize perlSGML, a library of perl routines for manipulating SGML documents and DTDs in particular. The perl routine reads the DTD and the configuration file, and creates an intermediate structure which is used as the input to the index-building routines written in C++, as well as the Java code in the user interface for automatically generating the structure display (see Chapter 7 for details on the user interface).

6.3.3 Storage Management Functions

Storage management is typically performed using a client-server architecture. A server having the essential functionality of a storage manager (*e.g.,* concurrency control, recovery) runs continuously and waits for connections from clients. Clients attempting to perform storage management tasks send requests to the server as necessary. In DocBase, the storage management functions are implemented using Exodus [CDF+86, Uni93], a popular storage manager developed at the University of Wisconsin. Exodus has a client-server architecture; it acts as a server for DocBase, which is an application built using the library of functions provided by the Exodus Application Programming Interface (API). The storage management module of DocBase uses this interface to communicate with the Exodus server that processes requests for storage management functionality.

The storage management functions in DocBase are fairly limited. Since all the data is stored as regular text files on the file system and are only accessed using the Pat query language, no special storage management operations are performed on the input SGML documents. However, all the persistent structures such as the catalogs and auxiliary join indices created on the data are managed by the storage manager. In addition to persistence of the structures, the storage manager also performs concurrency control on these structures. Descriptions of these structures are given in the next section under index management functions.

The storage management clients were implemented in an object-oriented fashion, allowing the interchangeability of storage managers and incorporation of other servers in the future. The most essential storage management functionality was included in the virtual superclass "Storage_manager," and the actual implementations were included in the subclass "Exodus" implementing the Exodus storage management client[3].

6.3.4 Index Management Functions

The Open Text Pat system was used for the purpose of creating indices in the prototype implementation. However, to ensure that the system is not completely dependent on the Open Text indices, the required navigation procedures were included in a virtual class "Hier_engine," and the Open Text index processing was handled by a subclass "Pat_engine" of the "Hier_engine" class. The "Hier_engine" class includes mechanisms for traversing the hierarchical structure and for extracting nodes from the tree based on string matches. Appendix B describes the details of the methods that any hierarchical structure manager must implement. The primary ones include basic structure navigation and ancestor/descendant searches.

The auxiliary join indices were implemented as lists of (offset, data) pairs and had three variations. All the generic functionality of the list class was incorporated in the class "List", and the functions specific to platforms were included in subclasses of this

[3]As Figure 15 suggests, the storage management functions were also tested using Sybase, in which the indices were stored as flat tables. This was possible because of the simple offset-data pair structure of the indices.

class (see Figure 15). In particular, three subclasses were implemented: (i) "Mlist" or memory list, that implements small main-memory lists primarily for temporary computation purposes such as sorting; (ii) "Plist" or persistent lists that implement the join indices and stored query results (views); and (iii) "Patlist" or Patricia tree lists. The last type of list refers to temporary structures in the memory space of the Pat engine - structures used primarily for the exchange of data between Pat and the external functions that use the data from Pat; and also for storing intermediate results for the processing of large queries. Information regarding the names and other properties of these structures were stored in an extension of the catalog. This information was used to update and remove these objects when necessary.

6.4 Query Engine Architecture

Queries using DocBase can be formulated using either a command-line interface or a graphical user interface. In the command-line mode, the results of the queries are displayed on the standard output. The graphical user interface is implemented as a WWW client, in which the query is formulated using the interface, and the results are processed by a DocBase running as a CGI application. When run in this mode, DocBase generates the output specifically for display on a WWW browser. In either case, the query is first parsed and translated into a sequence of operations which are then evaluated and combined by the execution system as shown in the architectural overview (Figure 13). In this section, we briefly describe the parser and the translation routines, showing the query processing algorithms in details. Appendix B gives a detailed description of the source files used for these purposes.

6.4.1 The Parser and Translator

The query parser for the DSQL language was implemented using lex and yacc [LMB92]. The parser supported the entire DSQL language described earlier in Chapter 5 and implemented all the productions shown in the BNF presented there. Three parsers based on the same grammar were created during the implementation stage. The first

parser, containing only the DSQL productions in the yacc syntax with no special handlers, was designed primarily for the purpose of validating the grammar and removing the shift-reduce and reduce-reduce conflicts in the grammar. Although the prototype restricted queries to the core DSQL language, the parser is capable of parsing the full DSQL language. The yacc source code for this parser is shown in Appendix B. This source can be used as a skeleton for other advanced applications that require parsing of the DSQL queries, similar to the ones described below. The grammar has one shift-reduce conflict which is acceptable in this situation.

The second parser was designed as a translator from DSQL queries to queries written in SGML conforming to the DSQL DTD (see Chapter 5). This parser uses the BNF rules activated during parsing of the DSQL query to generate the corresponding SGML tags in the DSQL DTD.

The above two parsers were created to test the parsing process. In the current prototype implementation, queries formulated in SGML are first translated into an equivalent DSQL query and then parsed using a DSQL parser. This apparently reverse translation was performed only because of the easy availability of lex and yacc. However, once SGML applications are readily available, using SGML parsers to do the query parsing of the SGML queries should be more feasible. DSQL queries can then be evaluated by first translating to SGML using the second parser described above and subsequently processing it within the SGML application.

The third parser invokes the query processing system. This parser is capable of parsing queries written in complete DSQL but can only process queries in the core DSQL language, exiting with a warning otherwise. This parser receives the DSQL query in the standard input and creates instances of the storage management, index management and query engine classes, invoking appropriate methods of these instances to evaluate and process the query. The unimplemented features are, however, not impossible to implement in this setting and were only dropped because of the limited resources and the proof-of-concept nature of the prototype. Since the infrastructure is very similar to the relational query implementations, techniques used in relational databases for evaluating nested queries (such as tuple substitution

[SAC+79]) can also be used in this setting. Moreover, grouping, ordering and aggregate operations can be implemented using filters on the result of the queries. We intend to show here that a reasonably self-contained subset of queries can be implemented using the proposed model and structures, and to propose the implementation methods for the rest of the queries as future work. The details on the evaluation of queries are described in the next section.

6.4.2 Query Evaluation

As described above, queries are evaluated by the DSQL parser module by initially creating instances of the storage management, index management and query engine classes, invoking appropriate methods of these classes in response to the rules processed by yacc. The actual evaluation of the queries takes place in the query engine class, and the algorithms for processing queries are described in the rest of this section. Here, we first show the basic algorithm for processing a simple select query (*i.e.,* a query without any joins and path expressions that require special processing). We then show how path expressions and joins are processed, how other DSQL operations are implemented, and how the prototype system can be augmented with some of the unimplemented operations.

The current implementation of DocBase uses an accumulator-based evaluation method. An accumulator here is simply an internal representation of a document relation used in the query. One or more accumulators may be needed depending on the number of relations used in the `FROM` clause of the query. An accumulator can be conceptualized as a list of (offset, data) pairs, possibly sorted in ascending order of either the offset or the data, depending on how it is used. The concept of offsets is specific to the Pat system - Pat calculates an offset value for specific positions in the files in its multi-file system. The accumulator denotes a list of virtual SGML documents rooted at a particular GI starting at the given offset in the SGML repository. Hence every accumulator corresponds to a GI (we will refer to this GI as *accumregn* in the following discussion). For normal evaluation, we assume that the accumulator is sorted in ascending order of the offset, which is the same order in

which the data appears in the document.

Given an accumulator, it is possible to traverse the document structures upwards or downwards from the accumulator. Given an accumulator and its corresponding GI *accumregn*, a traversal down to a target GI results in an accumulator associated with the target GI, containing a list of document components rooted at the target GI that are descendants of *accumregn*. Similarly, an upward traversal results in an accumulator with elements rooted at the target GI that are ancestors of *accumregn*. In addition, given an accumulator and a path expression, it is often necessary to only retain the elements that match a path expression P so that *accumregn* matches $last(P)$. The three algorithms are described in Figure 16.

We now consider a brief analysis of the *traverseup, traversedown* and *selectpath* algorithms in Figure 16. Notice that the algorithms here are presented in general terms, using a method that would be used if the Pat indices were not available. This is necessary to get a feel for the actual complexity of these operations. However, these operations are implemented using the Pat queries which use the Patricia tree indices. Hence, for analysis purposes, we will also mention the implementation of the algorithms with Pat operations and the complexity of the operations if Pat indices were to be used. The primary difference between the use of Pat operations and regular tree operations lies in the fact that a search for a string in a Pat index depends only on the length of the string being searched (see Chapter 2 for details). In particular, this implies that searching for a node labeled with a given generic identifier in a tree component does not require a full breadth-first search as described in the above algorithms, thus significantly improving the search performance. Another important distinction is that the operations allowed by the Pat query language are applied to a set of document positions, never individually applied to a single document position. Hence, although realistically some operations (in particular, the `selectpath` algorithm) are more naturally applied to every individual element, it is more efficient to perform the same task using set operations with Pat.

- *traverseup.* Since SGML documents are strictly hierarchical, this algorithm is quite simple. Every node can have exactly one parent. For every element in the accumulator, the worst-case cost of traversing up to the given region

```
traverseup(list accumulator, GI accumregn, GI targetgi:input)
begin
  if ((accumregn == targetgi) ||(targetgi==null)) return;
  templist = empty
  for each element e in accumulator do
  | repeat
  | | follow the parent of e upwards
  | | if parent node has GI targetgi
  | |   if parent not already in templist
  | |      append parent to templist
  | |   endif
  | | endif
  | until no more parents
  endfor
  return (templist, targetgi);
end traverseup

traversedown(list accumulator, GI accumregn, GI targetgi:input)
begin
  if ((accumregn == targetgi)||(targetgi==null)) return;
  templist = empty
  for each element e in accumulator do
     starting for e, do breadth-first search for nodes labeled targetgi
     during search, do not add nodes that can can never reach targetgi
                    using the catalog
     append matched nodes not yet visited to templist
  endfor
  return (templist, targetgi);
end traversedown

selectpath(list accumulator, GI accumregn, string pathexp)
begin
  if (first(pathexp) == root GI of the DTD)
    create a finite automaton for pathexp
  else
    rootgi = root GI of the active DTD
    create a finite automaton for rootgi..pathexp
  endif
  for each element e in accumulator do
    construct the path by traversing from e up to the root and reversing it
    if the constructed path is accepted by the FA, retain e
    else reject e
  endfor
  return (accumulator, accumregn);
end
```

Figure 16: Upward and downward traversal algorithms

is thus the maximum height of the parse tree of the document instance. For document structures without recursion, this height is constant and is governed by the DTD, since the document structures cannot be indefinitely deep. For recursive structures, however, the worst case can be the number of nodes in the tree. With Pat, this operation is simply performed by an ancestor search using the "`including`" operator of the Pat query language, which has a linear complexity proportional to the number of nodes in the accumulator and the number of nodes with the region to traverse to (see Section 6.1.2). To see that this achieves the desired result, notice that the above Pat expression selects only the nodes (of the given GI) which include (*i.e.,* have as a descendant) at least one of the accumulator nodes. This has the same effect as traversing upwards from the accumulator nodes to the given GI.

- *traversedown.* The worst-case complexity of this algorithm is the total number of nodes in the elements of the accumulator. However, in practical cases, however, it is easy to determine from the DTD if a particular GI will ever have a descendant with label g, and in a practical document structure, this will immediately prune many branches. Using the Pat client, this operation is significantly simpler, since traversal downwards simply requires a bounded search for the given GI, with the boundaries marked by the start and end tags of each of the elements in the accumulator. In the Pat query language, this operation is performed by a descendant search using the "`within`" operator in the Pat query language. Once again, to see that this Pat operation produces the correct result, note that the "`within`" operator produces regions with the given GI which lie within (*i.e.*, are descendants of) the accumulator nodes.

- *selectpath.* Any non-null path expression can easily be represented by a deterministic finite automaton (see Figure 17). The reasoning behind this lies in the fact that the all path expressions can be written as a regular expression by replacing the "**..**" operators with $\mathbf{gi}^*$, where **gi** is the set of generic identifiers in the DTD. For example, the path expression A.B..C is the same as the regular expression $AB(\mathbf{gi})^*C$). If we have a fully expanded path (which is what is

created in this algorithm), the cost of determining if the path satisfies the given path expression is simply the length of the path, which is once again, in the worst case, the maximum height of the tree structure representing the document instance. This general algorithm is used on each node, and the nodes for which the path expressions satisfy the DFA are selected. Using Pat operations, this requires evaluating the path expression and performing an intersection of the result with the original list. The intersection operation has a linear complexity since the lists are sorted on the offset values. Notice that this operation is performed on sets, by first evaluating the set of nodes satisfying the given path expression and then performing a set intersection with the accumulator.

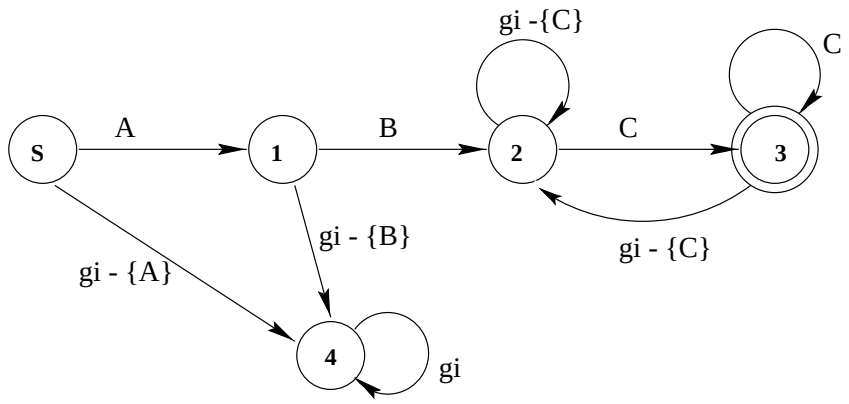


Figure 17: Example of constructing a deterministic finite automaton for the path A.B..C

To demonstrate the correctness of this operation, note that the path expression evaluation using Pat ensures that the resulting nodes satisfy the path expression. Moreover, the intersection performed also ensures that the selected nodes are from the accumulator. Hence, all selected nodes are the accumulator nodes that satisfy the given path expression. The path expression evaluation uses the *traversedown* algorithm described above, and is shown to be correct.

The algorithms described in the following sections use only the above three set-based algorithms, so that the implementation uses the Pat indices and query language. We also assume that the Pat implementations of the above algorithms terminate, and we use this fact in the analysis of the following algorithms.

6.4.2.1 Simple Select Queries

Simple select queries have a plain "`SELECT - FROM - WHERE`" structure without any nesting of the queries and without any joins. We assume that every condition is simple, and involves the comparison of a region with an atomic value (character strings). We further assume that there are no composite path expressions (path expressions with more than one label) either in the `SELECT` clause or in the `WHERE` comparisons. In other words, regions on which comparisons are to be made or regions which need to be selected are specified only using the generic identifier (GI) corresponding to that region. For example, the query "find all the titles in the book database in which a paragraph has the word 'SGML' in it" could be written as:

```
SELECT title
FROM book
WHERE para = "SGML"
```

Notice that instead of specifying path expressions such as "`book.head.title`", we simply used "`title`." Obviously, if there are multiple ways of reaching the title GI, it cannot be solved using a simple DSQL query (we will deal with such queries later). Simple select queries form the basis on which most queries are processed in the prototype engine.

The basic algorithm for processing simple select queries is somewhat different from the ones deployed in relational database systems. In relational database systems, if there are no joins or products, all the attributes are obtained from the same table. Because of the flat nature of the relational databases, this makes simple selections quite easy. However, in the case of a hierarchical structure, the target region can be deep inside the structure, which would require traversal of the structure to the specific region.

Since the underlying structure is essentially hierarchical, simple select queries often not suitable for selections that involve different branches of the tree structure. Let us clarify this with an example. Suppose we have the structure described in Figure 14. Consider the query: "Find the chapters in the books written by Goldfarb in which the chapter heading contains 'logic'. " One may be tempted to implement this query

using a simple selection query such as the following:

```
SELECT chapter
FROM book
WHERE author = "goldfarb"
AND head = "logic"
```

Suppose we ignore for the time being that the region *head* could appear in multiple places in the structure other than *chapter*. The above SQL query still does not provide the correct answer to the search problem in question. The reason is clear if we consider the equivalent DC form of the above SQL query:

$$\{x..chapter | Book(x) \wedge x..author = \text{“}goldfarb\text{”} \wedge x..head = \text{“}logic\text{”}\}$$

Clearly, the above query returns all chapters of books written by Goldfarb such that the book contains at least one chapter with 'logic' in the chapter heading. The correct query for this problem is:

```
SELECT Y
FROM book X, X.chapter Y
WHERE X.author = "goldfarb"
AND Y.head = "logic"
```

Obviously this is not a simple select query since it requires path expression evaluation and the use of multiple accumulators.

The above discussion indicates that the scope of simple select queries is quite limited. We are still interested in simple select queries because they form the core of the query engine, and give an intuition on how more complex queries are evaluated.

Simple select queries are evaluated by the use of a single accumulator. Query fragments representing each condition are evaluated relative to this accumulator, and are incrementally combined using set intersections or unions based on the logical operation performed. For every condition, a mini-selection based on the accumulator is performed into a temporary structure for that particular condition. Each mini-select involves a traversal down from the accumulator to the region on which the

comparison is being performed. This operation results in the selection of only the regions that match the given condition, and a traversal back up to the accumulator to select only the elements in the accumulator that resulted in a match. Note that the accumulator is left unchanged. The resulting subset of the accumulator is stored in a temporary structure until all the conditions are evaluated. The mini-select algorithm, in general terms, is given in Figure 18. In this algorithm, a condition has a GI, an operator, and an atom: the GI is compared with the atom using the operator.

```
miniselect(condition C, list Accumulator, GI accumregn : input;)
/* c is of the form (gi, op, searchstring) op is = or <> */
begin
   /* Make a temporary structure */
   temp_acc = Accumulator; temprgn = accumregn;
   /* traverse down to the condition target */
   traversedown(c.GI, temp_acc, temprgn);
   for each item in Accumulator
       if item does not match c according to (c.op, c.searchstring)
         remove item from temp_acc
   end for
   /* go back to the accumulator level */
   traverseup(accumregn, temp_acc, temprgn);
   return (temp_acc, temprgn);
end
```

Figure 18: Algorithm for evaluating an individual selection condition in a simple query

Analysis of miniselect. The miniselect procedure is self-explanatory. It uses the traverseup and traversedown procedures introduced before. In addition, *miniselect* has a for loop corresponding to each item in the accumulator resulting from the downward traversal. A sequential search through this, as shown here, is obviously not efficient, so the actual search may be implemented using different types of indices. In the prototype implementation, we use the Pat indices and the `including` operator to perform a bounded prefix search which has a very low complexity (linear in the length of the search string). To demonstrate its correctness, observe that the procedure works by finding all elements with the given GI under every node of the accumulator, selects only those that match the given condition, and traverses back up to the accumulator

region, in effect selecting only those accumulator elements that match the original condition.

Once all the conditions are evaluated, they can be combined using the logical operations between them. The order of evaluation is governed by the parsing mechanism. The parser creates a tree structure based on operator-precedence and the presence of parentheses. The parser generates the conditions a node at a time, with two branches and the logical operation that connects them. The complete algorithm is shown in Figure 19.

```
simplequery(SQL query:input; list accumulator, GI accumregn: output)
begin
  accumregn = GI in FROM clause;
  accumulator = all elements rooted at GI;
  parse WHERE clause into condition tree;
  (accumulator, accumregn) =
           evaluate (conditionroot, accumulator, accumregn);
           /* the evaluate procedure is shown below */
  traversedown(accumulator, accumregn, GI in select clause);
end simplequery

evaluate(Condion_node cond, list Accumulator, GI accumregn: input)
 if (cond is composite)
    /* cond is of the form condition1 logic condition 2 */
   (accumc1, c1reg) = evaluate (condition1, Accumulator, accumregn);
   (accumc2, c2reg) = evaluate (condition2, Accumulator, accumregn);
   if (logic == AND)
     accumc3 = intersection of accumc1 and accumc2
   else if (logic == OR)
     accumc3 = union of accumc1 and accumc2
   endif
   c3reg = c1reg;
 else /* base case: cond is simple */
   (accumc3, c3reg) = miniselect(cond, Accumulator, accumregn);
 endif
end evaluate
```

Figure 19: Algorithm for processing a simple query

Analysis of simple queries An example of processing a simple select query with path expressions has already been shown in Section 6.2.2.1, which uses the algorithm described in Figure 19. This algorithm described may seem to be somewhat inefficient,

since we perform individual selections and then a union and intersection based on the logical operation used ("AND" or "OR"). As mentioned earlier, this implementation strategy is influenced by the use of Pat indices, since the operations on these indices are primarily set operations. An evaluation system that does not use a Pat engine can also use the above algorithm by simply keeping track of the requested operations and performing the whole operation at the end on every individual tree.

Termination. In this algorithm, we have assumed that there are no composite path expressions. So, as shown in the the pseudocode, the algorithm uses a recursive evaluation strategy to evaluate the conditions in the `WHERE` clause. Since there can be only a finite number (say, n) of such conditions in a DSQL query, the *evaluate* procedure is called a maximum of $2n - 1$ times. Hence, this algorithm terminates, knowing that the *traverseup* and *traversedown* procedures terminate.

Correctness. The correctness of the evaluation method for simple select queries follows from the semantics of the calculus language DC, on which DSQL is based. Recall that a DSQL query of the form `select A from R where condition` is equivalent to the DC query $z^{\{x|R(x)\wedge condition\}} \circ A$. This indicates that all the evaluation is based on the single accumulator x which is initialized to all of R, and after the conditions are evaluated, the final selection is performed by a path traversal. The correctness of the evaluation of the condition can be determined by noticing that since all the conditions are combined with respect to the same region (that of the accumulator), intersection of the accumulator elements does correspond to conjunction and union of the accumulator elements corresponds to disjunction.

Complexity. To estimate the complexity of the above algorithm, notice that for each of the conditions, in the worst case, there would be one matching operation, one `traverseup` and one `traversedown` operation. In addition, if there are k conditions, we will also have $k-1$ unions or intersections. The complexity of all these operations are linear, since the Pat indexing process and the retrieval using the Pat indices always yield results sorted by the offsets. Hence, the combined complexity of the operations using Pat indices is $O(n \times (k-1))$ where n is the number of nodes in the document tree and k is the number of conditions, as above. Hence for a fixed query, the complexity of processing the query is only linear to the size of the document.

6.4.2.2 Queries Involving Path Expressions

In the above discussion of simple select queries, we have assumed that all path expressions are specified by simply its target GI. Here we present a complete processing of path expressions. Path expressions can appear in different places: (i) in the `SELECT` clause, to specify the output from the query; (ii) in the `FROM` clause, to give aliases to specific paths relative to defined document types; and (iii) in the `WHERE` clause, to specify the region on which a comparison or a complex operation (such as `EXISTS`, `IN`) is to be performed.

Path expressions in the `SELECT` clause primarily signify projections. Unless some optimization strategy causes the projections to be evaluated earlier, when a projection is applied, the evaluation of the query without the projection can be assumed to be complete. Hence, the path expression evaluation can be performed by traversing down the document structure while traversing the path expression. Path expressions in the `FROM` clause also can be evaluated top-down, with the condition that the `FROM` clause does not have any forward referencing in its alias variables.

The path expressions in the `WHERE` clause are somewhat more tricky to evaluate. Although a top-down traversal can be applied to reach the target region for the purpose of the comparison, since all the comparisons are relative to some accumulator, the result needs to be traversed back to the accumulator using an upward traversal. The evaluation of simple select queries described in Section 6.4.2.1 has all three of these cases in a simpler way, since there we assumed that path expressions only consisted of a single GI. The basic strategy, however, does not change if the path expression contains multiple GIs, since the evaluation would still involve repeated application of the *traversedown* algorithm described in Figure 16, followed by the extraction of the elements that match the condition, and a final *traverseup* to reach the accumulator level (see Figure 18 for this process applied to simple select queries). The only difference in path expression evaluation lies in the fact that the *traversedown* procedure is called once for each element in the path expression. The algorithm, and a basic understanding of its correctness of the downward evaluation of path expressions, is given below.

Path expression evaluation Path expressions are evaluated top-down (*i.e.,* starting from the topmost GI in the path and traversing the structure down the tree, following the rest of the GIs of the path). An algorithm performing this traversal will only need to start from a the current accumulator, and for every GI in the path expression, traverse down from the current set to the GI. If the path operator is "**.**", then the traversal involves only a scan through the immediate children of the current node. If the operator is "**..**", the traversal involves a depth-first search through the structure resulting in all the GIs of the required type that are descendants of the elements in the original accumulator. To determine the initial accumulator, the first symbol of the path expression needs to be used. If the first symbol refers to an alias declared in the `FROM` clause, then the accumulator is obtained from a symbol table that stores the aliases. Otherwise the default accumulator from the database in the `FROM` clause is used. The algorithm is described below:

```
evalpedown(string pathexp, GI accumregn,list accumulator:input)
begin
  f = first symbol of pathexp
  if f is an alias
      verify that the accumulator being used corresponds to the same alias,
      return if not
  else if (f != accumregn)
     (accumulator, accumregn) = traversedown(accumulator, accumregn, f);
  endif
  for each of the rest of the symbols g in pathexp do
    templist = empty
    if (connector is .)  /* can not evaluate using Pat */
      for each element e in accumulator do
        if e has a child with label g add the child to templist
      endfor
      accumulator = templist; accumregn = g
    else /* connector is .. */
      (accumulator, accumregn) = traversedown(accumulator, accumregn, g)
    endif
  endfor
end evalpedown
```

Figure 20: Evaluation of path expressions in the from and where clauses

Analysis of path expression evaluation *Termination.* Termination of the `evalpedown` algorithm is trivial to determine, observing that the main loop is on the symbols in the path expression and that a path expression can only have a finite number of symbols. (Note that here we consider only the GIs in the path expression to be symbols. In our setting, path expressions cannot have a variable in the middle, but only in the beginning, which refers to another pre-evaluated path expression in the symbol table.) The inner loop for the evaluation of pat expressions with the "`.`" operation also terminates based on the assumption that there are only a finite number of elements in each accumulator.

Correctness. The `evalpedown` procedure is a simple case of determining the starting position of the traversal and traversing down the document structure through each of the symbols in the path expression. Note that, since the Pat query language operations uses a flat view of the document and does not use the document structure as a tree, the immediate child (`.` operator) cannot be computed using Pat. The only downward traversal operation in the Path query language, *within*, returns descendants of the given element. This is a drawback which cannot be remedied without having more low-level access to the Pat indices. Given this restriction, all path expressions are actually evaluated by treating the `.` operation as a `..` operation. To demonstrate the correctness of `evalpedown`, notice that the traversedown operation retrieves all descendants of the every accumulator element matching the target region. If any candidate match for the path expression is not retrieved by the algorithm, there must be one step where a symbol in the path expression is not reachable as a descendant of the previous symbol, which contradicts the assumption that the candidate matches the path expression.

Complexity. The evaluation method of a path expression with k symbols involves an initial selection, followed by traversal of the structure downwards $k-1$ times. Each of these operations can use a descendant traversal operation (*within*), for which the complexity is linear on the number of corresponding nodes in the document. The absolute worst-case complexity of the algorithm is thus $O(k \times m)$ where k is the number of symbols in the path expression, and m is the total number nodes in the document structure (or the of SGML elements in the document).

6.4.2.3 Queries Involving Products and Joins

Here, we present a complete algorithm for evaluating a query having all the implemented core DSQL features — in particular, products and joins involving more than one hierarchical component. These components can come from multiple DTDs, different branches of the same DTD, or even multiple instances of the same DTD. In the above discussions, we primarily used only a single accumulator. However, for a general query processing algorithm, we need to use multiple accumulators, equal to the number of different hierarchical components on which the query is evaluated. A brief description of the *prodjoin* algorithm is shown in Figure 21.

Analysis of queries with products or joins The queries involving product and join operations use the definition of product and join introduced in Chapter 5. The basic idea behind the product and join operations is the creation of a new GI with the roots of the component documents as immediate children. The current implementation of DocBase uses a binary product and join operation (*i.e.*, only two components can be involved in one particular product or join operation). The creation of new elements is implicit in the implementation. Because of the lack of good DTD processing tools, new DTDs are not created. The newly created document components are stored as "virtual documents" in the storage manager, and the newly created region is added to the catalog. Subsequent projection operations can be used after the joins and products to extract the relevant components of the results. The limitation of not creating output DTDs is often felt — this is planned in the future enhancements of the system.

The algorithm in Figure 21 has 7 component steps. We discuss each of these steps in turn.

1. While discussing simple SELECT queries, we used only one accumulator, because the engine only uses one document tree to process simple select queries, allowing the results to be computed iteratively by keeping one intermediate result and combining each `WHERE` clause condition to the intermediate result.

```
prodjoin(SQL query: input; list accumulator, GI accumregn:output)
begin
  1. Using the FROM clause, determine number of different query components n
     Allocate n accumulators and n accumregn's

  2. Evaluate expressions in the FROM clause, update symbol table with aliases
     initialize each accumulator with the evaluated path expressions.

  3. In the WHERE clause, evaluate simple (non-join) conditions according
      to the order of evaluation determined by precedence. Results of each
      condition is combined with any other condition based on the same
      accumulator. Disjunctions within accumulators can be immediately
      evaluated, however disjunctions between different accumulators are
      delayed until the end (see Step 6).

  4. For each join condition in the WHERE clause do
        4a. Evaluate both sides and store into persistent lists
        4b. Perform sort-merge join on the persistent lists into a new
            structure containing (offset-left, offset-right, data).
            Associate this structure with new catalog entry
        4c. Perform traverseup on each element of each pair of this
            structure to the appropriate accumulator level.
        4d. Combine each of the branches with the computed values of
            the corresponding accumulator from Step 3.
      endfor

  5. Update all accumulators with results from the combined conditions and
     perform inter-accumulator disjunction operations if any..

  6. Resolve dependency between accumulators.

  7. Finally, evaluate the SELECT clause using the different accumulators
                 and the path traversal algorithms.
end prodjoin
```

Figure 21: Evaluation of SQL queries involving products and joins

However, in queries involving joins or products, there are potentially many document trees involved. The different `WHERE` clause conditions may refer to the different trees, which cannot be combined right away. This step of the algorithm identifies the number of required accumulators and allocates them in a symbol table. This stage terminates trivially, since the number of accumulators is the same as the number of objects in the `FROM` clause.

2. This step of the algorithm is to evaluate the path expressions in the `FROM` clause and stores the results in accumulators in step 1. In this implementation, we enforced the rule that the aliases in the `FROM` clause cannot be forward referenced (*i.e.*, `FROM Book B, B.Title C` is valid, but `FROM B.Title C, Book B` is not). This ensures that the `FROM` clause can be processed using a single pass. This stage terminates because there can only be a finite number of objects in the `FROM` clause and no forward references are allowed. Each of the objects can be evaluated using the path expression evaluation algorithms described earlier.

3. The `WHERE` clause conditions are evaluated next, using the accumulators for the corresponding document components. As in the case of simple select queries, the conditions are formed into a tree according to the order of evaluation, and evaluated in their logical order. If there are no disjunctions between different accumulators, all the conditions that do not involve a join can be evaluated at this stage. Because of the way accumulators are used in this step as well as in steps 4 and 5, the evaluation of disjunctions between accumulators is delayed until step 6. For each of these comparisons, the path expressions involved are evaluated top-down as described above and filtered according to the comparison operation. The selected elements are traversed up (as in the simple select queries) to go back to the level of the accumulator they originated from. The termination of this phase is based on the observations that there can be only a finite number of `WHERE` clauses and that each of them can be evaluated using a terminating algorithm described earlier.

4. The join conditions are evaluated next. Using this method, the two components for the join are evaluated first and then stored in the storage manager.

The actual join operation on these two components is then performed in the storage manager, using a sort-merge algorithm on the data counterpart of the (offset-data) structure of the stored accumulators. The sort-merge algorithm is used because of the built-in sort feature of Exodus using the B-tree structure. However, any join algorithm can be used here. The join operation creates a structure which is slightly different from the usual (offset-data pair) structures that are commonly used otherwise - the difference being the additional offset values arising from the join operation. After the join is performed, each of the components of the new structure are traversed up to the level of the accumulator they originated from. If the join follows a conjunction, each of the left and right components of the structure is combined with the appropriate set of matches (evaluated in Step 3). The join conditions are also evaluated by path expression evaluation procedures which were previously shown to terminate. The sort-merge join is a well-known method for computing joins and only requires each of the components to be finite in order to terminate – a requirement satisfied from the assumption that the database is a finite set of documents.

5. Since individual conditions use the original accumulators as starting positions for traversing the path expressions, the original accumulators are not modified during the processing of the `WHERE` clause. An optimization measure that can easily be incorporated in this algorithm is to update accumulators after the evaluation of each condition if the query is completely conjunctive. However, in a query with disjunctions, evaluations of inter-accumulator disjunctions can only be performed after all the conditions have been evaluated. Intra-accumulator disjunctions are performed as they appear in Step 3. Since there can only be a finite number of accumulators and a finite number of disjunctions between them, this step terminates.

6. In DSQL, it is possible to have dependent accumulators, since path expressions are allowed in the `FROM` clause. Hence, after the accumulators are updated, any change in the accumulators is propagated back to the dependent accumulators. Note that in Step 2, these accumulators are intialized using values from

other accumulators that they depend on. However, after all conditions are processed and accumulators updated, the changes need to be propagated again to the dependent accumulators. The updates may be performed using union or intersection operations as necessary. We stated earlier that DSQL only allows backward referencing of accumulator dependencies. Hence, this step terminates.

7. Finally, the `SELECT` clause is processed to generate the results, based on the computed accumulators. Once again, this is computed using the path expression evaluation algorithms described earlier, previously shown to terminate.

Termination. The termination of the algorithm is based on the proper termination of the individual stages, described above.

Correctness. The correctness of the algorithm given here is based on the correctness of the individual step. The basic logic of the algorithm is to first perform the selection conditions (step 3) and then perform the join/product operations (step 4) and the projection operations (step 7) - the usual process followed in evaluating SQL queries in relational databases.

Complexity. A single join operation in the above algorithm requires an initial dump of the respective accumulators into the persistent storage, followed by a standard join operation and the extraction of the components generated by the join operation. The most significant operation among these is the intermediate join on the persistent lists. Since we used the standard sort-merge join operation, this operation carries a worst-case $O(n^2)$ complexity. The initial dump and the final extraction operations have linear complexity. The efficiency of the join operation can be improved by using advanced join techniques such as "hash joins." In the implementation, sort operations were built in to both Pat and Exodus. Thus, the actual evaluation of the join operation only involves a merge. The sort-merge algorithm was chosen for this reason, to simplify the implementation. The built-in sort operation of Pat, however, did not show optimal performance and was discontinued in favor of the B-trees in the implementation.

6.4.3 Query Optimization

Some optimization techniques have been implicitly discussed in the last few sections. These include (i) the use of the catalog to block sections of the document tree from being traversed, (ii) the evaluation of simple selections prior to the computation of joins, and (iii) the incremental updates to the accumulator after the evaluation of each condition in a conjunctive query. Another optimization technique implicit in the implementation is the use of set-oriented evaluations instead of element-oriented evaluation because of the nature of the Pat query language. Apart from these optimization techniques that have been proposed and implemented, many other techniques common in the relational query processing can also be applied in this setting. We have left the implementation of such optimization as future work (see Chapter 8).

Chapter 7

User Interface Design

The success of any system depends not only on the features of the system but also on its "usability." Even if a system is feature-rich, such features are useless if they cannot be easily accessed by the users. "User interface" is a generic term given to the way a system interacts with its users. To design usable systems, the design process needs to incorporate usability considerations into the early stages of the design process. In Chapter 2, we described the essential components of the process for designing for usability. In this chapter, we describe the visual query language that we term "Query By Templates (QBT)." We also discuss the usability analysis process and explain the results obtained from this analysis.

7.1 QBT: A Visual Query Language

This research generalizes the Query By Example (QBE) method described earlier (Chapter 2) for application in databases containing complex structured data. QBE is suitable for relational databases since it uses tabular skeletons (analogous to tables in the relational model) as a means for constructing queries. Thus, the template for presenting queries in QBE is similar to the conceptual structure of the instances in the database. We use this idea to generalize QBE for databases where each data instance, albeit complex, has a simple visual model. We base this assumption on the fact that human beings form a mental model for the tasks that they intend to perform [Boo89]. For example, users performing a search in a dictionary may not know the internal structure and representation of each definition, but they usually have an idea about a visual structure of a dictionary entry, assuming they have used dictionaries in print. In our method, that we term "Query By Templates" (QBT), the basis of

the interface is a visual template representing an instance of the database. Simple examples of templates include (i) a small poem for poetry databases, (ii) a table for relational databases, (iii) a representative word definition for a dictionary database, and (iv) a sample entry in a bibliography database.

QBT is primarily designed to be a simple point-and-click interface for posing queries in document databases without the necessity of knowing and understanding the internal structure of the database and without learning complex query language syntax. In spite of its apparent simplicity, QBT is a powerful language and can express the same class of queries as the core DSQL language introduced in Chapter 5. As in the core DSQL, the current design of QBT does not address nesting of queries.

In this section, we describe the rationale behind the QBT interface. Next, we introduce the concept of templates and describe the various types of templates considered in this design. We then describe the process of formulating queries using templates. Subsequent sections will describe implementation and analysis of the QBT interface.

7.1.1 Rationale

The main rationale for the idea of querying using templates comes from the fact that users tend to form a distinctive mental model for tasks they perform [Boo89]. Simply described, a mental model is a mental image of the expected task (both the process of performing the task as well as the result on completion of the task) that the users conceive of before they actually begin any task. For example, users planning to write a letter may have a mental image of what the letter would look like once it is completed. During the process of carrying out the task, users try to use a tool that can help them achieve their conceptual goal. Analogously, in order to search for information in a repository, users form similar visual images of what they are looking for. This visual image is what we try to capture using the concept of templates.

Let us explain this further with an example. Jane Doe was looking for a poem in a database of poems. She knew that the poem was written by Blake, and she knew that it mentioned the word "tiger" in the first line. However, using the conventional search techniques, she either could not retrieve the poem, or had too many matching

poems. On subsequent brainstorming, she also remembered the occurrence of the word "burning" in the first line, and with some effort, she could retrieve Blake's poem "The Tyger." Of course, the word "tiger" in this particular instance was spelled as "tyger."

One might correctly argue that Jane's problem could be solved using a search method that can perform approximate searches. However, the goal of this research is not to design approximate search techniques. What is more important in the above instance is the fact that Jane acquired a mental image of a poem that she wanted to retrieve, and the only portions of the poem that she could remember were the poet's name and a portion of the first line. Although her initial guess was unsuccessful, a refinement of the guess eventually resulted in a match. In this case, she had a mental image of a poem (similar to Figure 22 a) which resulted in a retrieved instance (in Figure 22 b).

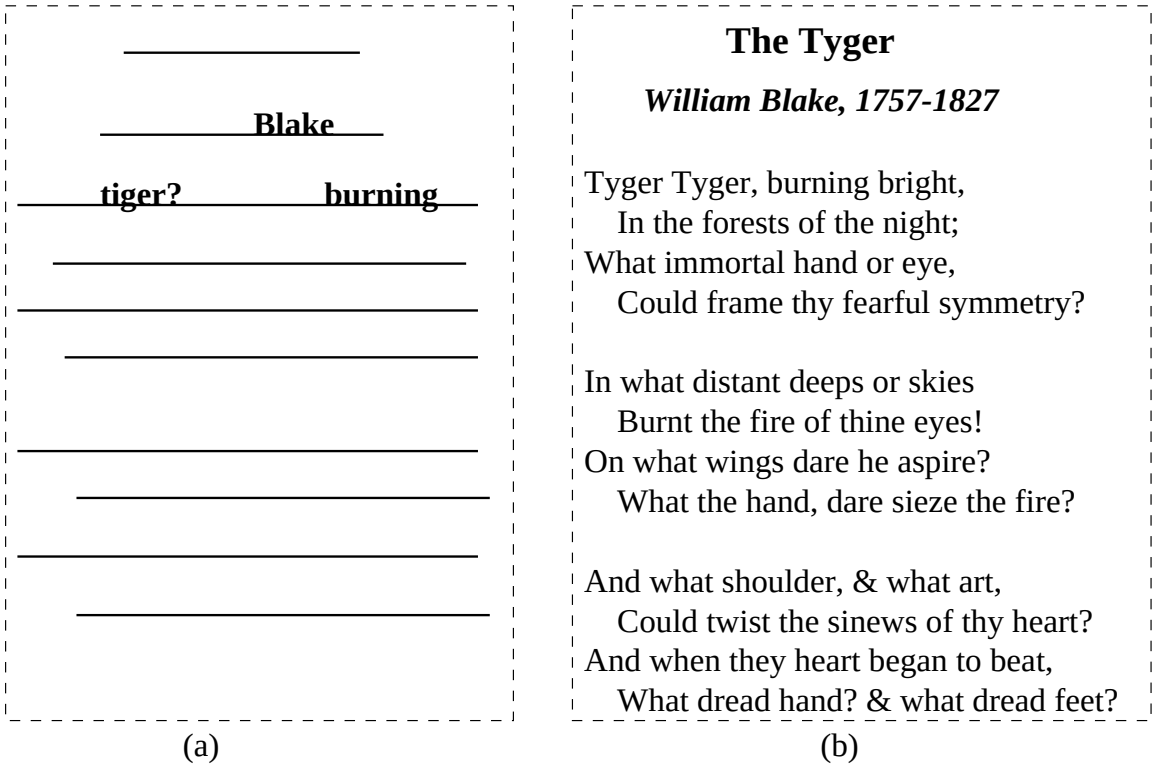


Figure 22: An example of a conceptual image of a search and the retrieved result

As mentioned above, the goal in this research is to capture the mental image that users develop prior to starting a search task. QBT accomplishes this by presenting the search interface using a simple representative of the database instances. Any database that has a simple visual representation of its content can be used with QBT. For databases that do not have a general visual content, we can always revert to tables (or even forms) for use as representative templates. One of the main goals for the design of QBT was to retain all the prominent properties of QBE. The intended properties of QBT that are analogous to those of QBE (as discussed in Section 2.2.2.3) are (i) simplicity, (ii) equivalence, (iii) closure and (iv) completeness. First, QBT is designed to be simple, and it does not require users' knowledge of the complex document structure. Second, it uses templates that are conceptually equivalent to the instances of the databases. Third, QBT is "closed" in its template domain by displaying the results using the same template as the query. Fourth, one can formulate most commonly occurring queries using QBT. In the rest of this section, we describe the various types of templates with illustrations, to elicit the foundation for the design of QBT.

7.1.2 Design Details

A QBT interface, in its simplest manifestation, displays a template for a representative entry of the database. The user sees a sample of the type of data she would expect to find in the database (*e.g.*, a poem in a poetry database). She specifies a query by entering examples of what she is searching for in the appropriate areas of the template, and the system retrieves all the database entries that match the example she provided. To illustrate the interface, we will use a simple template for a poetry database, as in Figure 23. In this figure, we indicate a prominent logical region of the poem by circling it and labeling it with the corresponding region name. Physically, the QBT interface consists of a small template image divided into areas corresponding to the different logical regions in the database, as in Figure 23. Depending on the layout of the regions, the templates can be of several types as discussed in the subsequent sections.

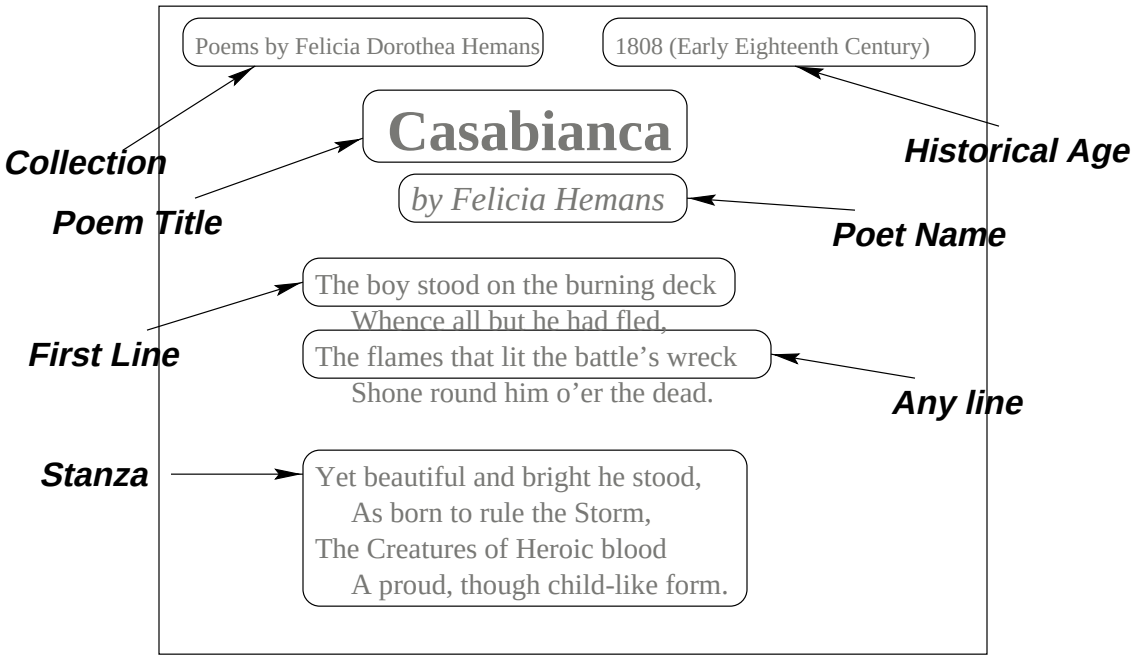


Figure 23: A simple template for poems, with its logical regions

7.1.2.1 Flat Templates

As described in the previous section, QBT relies on the presence of a simple visual template for the instances in the database. In most cases, this template could be planar or flat. This means that all logical regions of the template can be displayed simultaneously in a two-dimensional image without overlapping (see Figure 23). We call these templates "flat templates." Flat templates are usually easier to display and use, as the structural regions can be simultaneously displayed in a plane, possibly by showing multiple instances of some regions. For example, in Figure 23, the *First Line* and *Any Line* regions are sub-regions in *Stanza*. To display these sub-regions, the template needs to include a second stanza that is broken into its components.

7.1.2.2 Nested Templates

Although flat templates are easy to display and navigate, they cannot model structures with deep levels of nesting. In this case, we use templates that can be nested. In nested templates, regions are allowed to overlap. In particular, certain regions can

be completely inside other regions to represent sub-regions. To display embedded logical regions, we use one of the following methods:

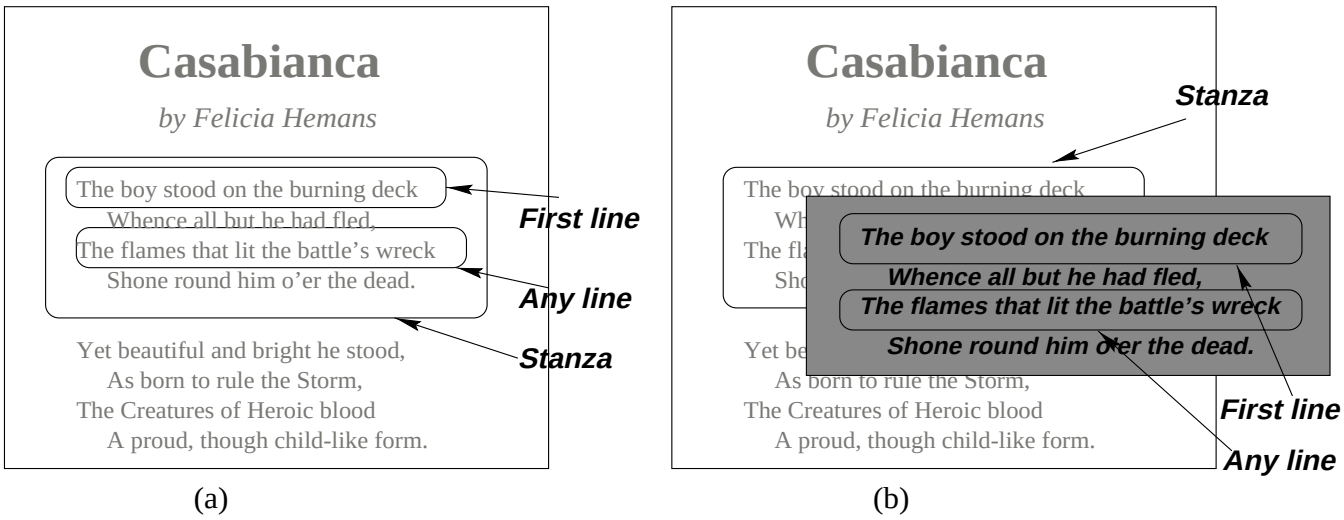


Figure 24: Templates with (a) Embedded Regions and (b)Recursive regions

Embedded Regions In this method, sub-regions are displayed inside the parent region. As in flat templates, all regions are displayed simultaneously in the same plane of the image. Component regions no longer need to be mutually exclusive. This method is a simple extension of flat templates, but it makes templates much more powerful while retaining the simplicity. However, this method is again limited to structures in which the nesting level is not very deep and the top-level region is physically large enough to include all the nested regions without completely obscuring itself. An example of this type of nesting in shown in Figure 24(a).

Recursive Regions This is the most general method of nesting regions. In this method, a region with sub-regions can be subsequently expanded. During traversal, the user may "zoom in" a parent region to display its sub-regions. The magnified portion of the template can be an independent template which can be subsequently magnified to achieve multiple levels of nesting. Although this method can capture any general structure, the templates have to be cleverly designed so that users are not

disoriented by the nested templates. Figure 24(b) shows this method of displaying internal structures for the same sample poem.

7.1.2.3 Structure Templates

Structures, particularly large ones, may get too complex to use nested templates. In these cases, it is often necessary to display the internal structure simultaneously with a template that displays the relative position of the current region. As mentioned earlier, most documents can be conceived as having a hierarchical structure that is conveniently visualized as a tree. The simultaneous display of a template with a hierarchy of logical regions based on context greatly simplifies the visualization of the nested structure. An example of the structure template is shown in Figure 25(b), which is a screen image from the prototype implementation of QBT, described in Section 7.2.

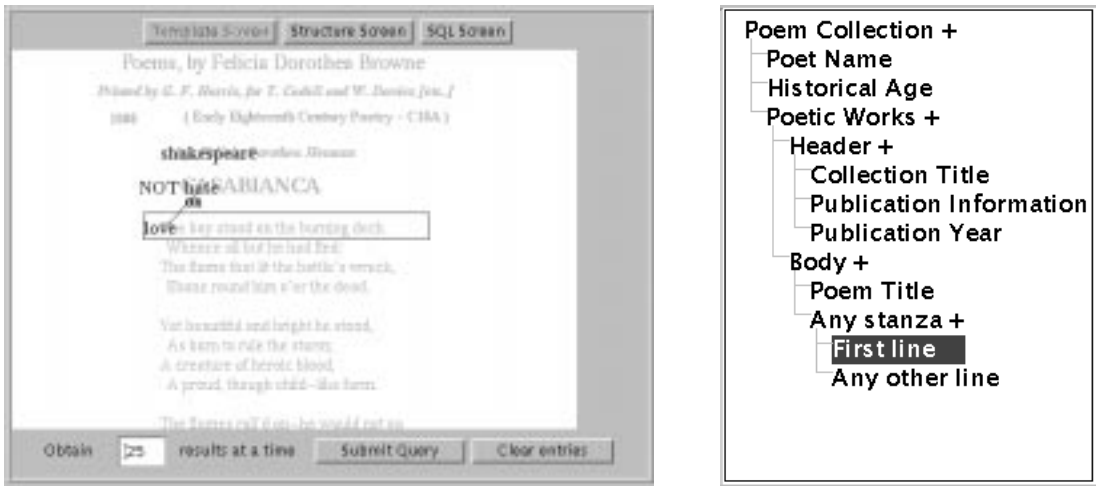


Figure 25: Screen shot of the prototype implementation showing (a) a flat template and (b) the structure template depicting the expanded structure.

7.1.2.4 Multiple Templates

Many queries require the use of more than one template. In relational databases, queries that derive the results from the contents of multiple tables require the constituent tables to be *joined* using a common attribute. QBE implements this by

displaying skeletons for all the constituent tables (see the examples in Chapter 2). QBT incorporates a very similar strategy. Even though "joins" in text databases are less common since the data is implicitly linked in the structure of the documents, they are still necessary and give rise to many interesting queries when the results involve multiple databases or related fragments of the same database. To express these queries, two or more templates, connected with a joining region, are displayed. We give examples of such queries in Section 7.1.3.3.

7.1.2.5 Non-visual Templates

Although the main idea in the QBT formalism is to use visual means for specifying queries, templates can easily be used without any visual structure. In the SGML domain, one might consider an incomplete SGML document to be a template for specifying a query that retrieves the document fragments that satisfies the template. In this case, the template is specified as a pattern which is matched by the query processing engine. However, we are not considering such non-visual templates in the current implementation.

7.1.3 Query Formulation

Normal keyword searches within structural regions are simple and most natural with the QBT interface. As illustrated earlier, users express their queries by indicating the search keywords in the appropriate regions of the template. In this section, we show all the different types of possible searches that can be performed with QBT.

One can treat QBE [Zlo77] as a special case of QBT where the templates used are table skeletons that instantiate tables in the database. In QBE, queries are specified by entering values in proper positions of the tables. These values may be either constants (*i.e.*, strings or numbers), variables (or *examples*, usually differentiated from the constants by underlining), or expressions involving constants and variables combined with arithmetic and comparison operators. The output of the query is specified by marking the regions that need to be presented in the output. QBT uses the same basic principle, with the extension that the templates are not restricted to

table skeletons but can be any visual representation of the database instances.

The primary difference between the method of expressing queries in QBT and in QBE lies in the fact that the templates in QBE are essentially one-dimensional. Although QBE uses two-dimensional tables for querying, the meta-data (attributes of the relations) only appear along the horizontal axis as column headings of the tables. QBE uses the rows along the vertical axis to specify multiple search conditions and logical operations between the search conditions (see examples in [Zlo77]). In QBT, the regions (meta-data) are distributed along both dimensions of the template, utilizing the whole template plane for visualizing the structure. Logical operations between regions can be expressed by physically connecting two or more regions via a logical operator. Logical operations within regions can be formed using logical expressions within the scope of that region. In the rest of this section, we discuss how different types of queries are performed using QBT.

7.1.3.1 Simple Selection Queries

Simple selections include searching for constant strings or numbers within logical regions of the document (the whole document itself being one region). In QBT, such searches are performed by simply entering the search string in the corresponding region of the template. As a result of such a search, database instances that are rooted at a default region and that match all the specified conditions are returned. In other words, the given search criteria are combined using a logical conjunction operation. The result of the query is by default rooted at a pre-selected region defined by the template. However, users can mark the regions that they want returned by placing a print-marker on them.

In the illustrations (see Figure 26), the small tick-mark ($\surd$) is used as a print indicator. In the examples, Figure 26(a) denotes the simple query: "Find the poem titles and poets of all the poems that have the word 'hate' in the title and the word 'love' in the first line." Note that unlike QBE, searches in QBT are substring matches instead of exact matches. So, entering the word 'love' in the region "first line" matches all poems with the first line containing the word 'love' anywhere in the first line. In QBE, this is done by indicating examples before and after the search string.

Figure 26: Query formulation with QBT: (a) Simple selections and (b)Logically combined selections

7.1.3.2 Selections with Multiple Conditions

We have just seen that if multiple conditions are specified in different regions, they are combined using logical conjunctions, so the results returned from the query satisfy all the specified search conditions. If this is not desired, search conditions can be combined using logical operators *AND, OR, NOT*. An individual condition can be negated by placing the keyword "NOT" in front of the string. Implementations of the interface may use some visual mechanisms (such as a negation symbol or a negation button) instead of this negation keyword. Users may combine multiple conditions using binary logical operators "AND" and "OR" by connecting the strings involved using a pointing device and selecting the proper logical operator. Figure 26(b) demonstrates how this is accomplished using the query: "Find the poem titles and poets of all poems that either do not have the word 'hate' in the title or have the word 'love' in the first line." Notice the introduction of the negation and the "OR" connection.

Providing a two-dimensional visualization for a strictly ordered chain of query components connected with logical operations can be somewhat tricky. In our approach, we tried to keep the interface as simple as possible by implying conjunctive connectors when there are no arrows, and explicitly specifying disjunctive or conjunctive connectors when necessary. The algorithm to derive the logical expression from its graphical representation is very similar to a minimal spanning tree algorithm

[CLR89, Chapter 24]. The algorithm is initiated with one of the nodes which does not have any incoming arrows, and a minimal spanning tree is built with all the nodes reachable from the starting node that have not been included in the expression. This process is continued until all nodes have been included. This process ensures that each node is only entered once in the expression. However, it is only a heuristic method, and may or may not correspond exactly to the query the user had in mind. In order to ensure that the proper query is processed, the condition box needs to be used. We discuss condition boxes shortly.

7.1.3.3 Joins and Variables

In this section, we look at a special class of query called "join." A join is an operation which combines multiple fragments of a database (in form of document trees in this case) based on the value of at least one node in each of the components. When a join operation is performed based on the equality of the combining node, it is also referred to as an "equi-join". Joins are indispensable in relational databases, since the relational design involves "normalization" of a schema by breaking it into flat tabular fragments. This fragmentation requires using a join operation to combine the individual fragments together at the time of query processing. However, in document databases, the structure is not normalized into planar fragments but allowed to grow hierarchically, so joins are not required to combine fragments. However, joins are still useful for solving queries that require comparison of different parts of a database or different instances of the same database.

For example, one may try to "find the pairs of poets who have at least one poem with a common title" (as in Figure 27). In this case, we need to generate two instances of the poetry database and run the query comparing the titles of the two poems. This is achieved in QBT by using multiple templates. In the case of the above query, the same template is instantiated twice, and the join attributes are connected together. The connection can be augmented with comparison operators to specify joins other than "equi-joins". As before, in the case of asymmetric comparison operations, the precedence of the operators is determined by the direction of the arrow. To keep the conceptual similarity with QBE, examples are underlined to differentiate them from

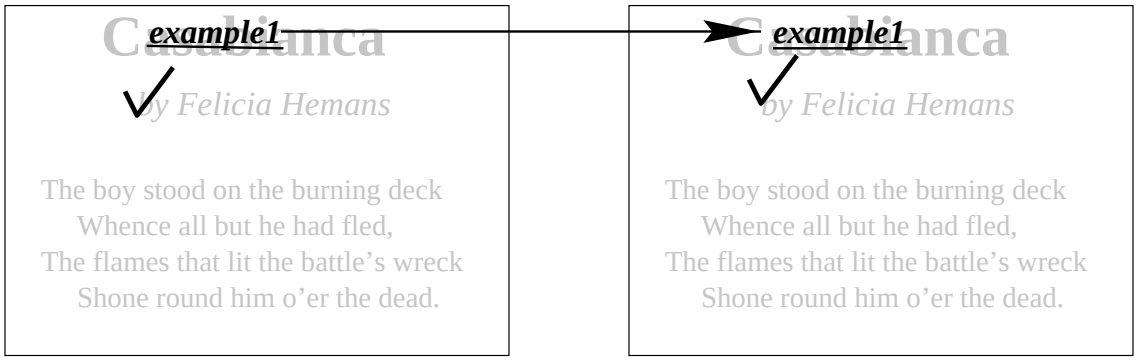


Figure 27: Query formulation with QBT: Joins

constants.

Notice that visualizing the results of join queries may not be possible using the same template as the query itself, but an implementation of QBT can work around this problem by specifying layout characteristics (using stylesheets, for example) to display the results. The closure of the interface is maintained by the fact that the query outputs consist of SGML documents only, so they can be displayed using the same methods used for displaying the template.

7.1.3.4 Complex Queries

Visualization of queries that combine conditions on more than two regions using logical operators is difficult in QBT – a problem arising from its flatness. Connecting the regions together is not always sufficient because the intended order of these operations is important. In QBE, complex situations like this are expressed in a separate area from the skeletons - commonly called the *condition box*. The condition box is simply a small text window in which the complex conditions can be expressed using logical expressions and the order or evaluation denoted using parentheses. The condition box can also be used to override the precedence of operators.

QBT uses a similar mechanism to express complex logical combinations. As search strings and examples are specified, the condition box is automatically updated. The user can then insert parentheses as necessary to change the default precedence. For example, in Figure 28, if the default precedence (left to right) is used, the query

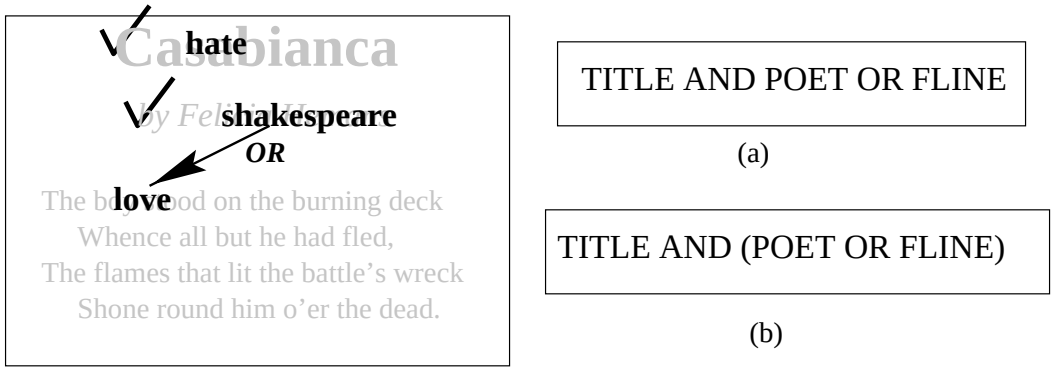


Figure 28: Changing precedence of operations with Condition boxes

evaluates to: "Find the poem titles and poets of the poems in which either the word 'hate' is in the title and the poet is Shakespeare, or the word 'love' is in the first line." The default condition box is shown in Figure 28(a). However, this default can be changed to: "Find the poem titles and poets of the poems in which the word 'hate' is in the title, and either the poet is Shakespeare or the word 'love' is in the first line" (see Figure 28(b)). The condition box can also be used for specifying more complex conditions involving more than two variables in an expression. In this case, QBT's condition box has the same functionality as that of QBE. The main use of the condition box is to provide the power necessary to generalize the querying method to accommodate all types of queries supported by the formal query languages and, hence, add to the expressive power of the language.

7.2 Prototype Implementation of QBT

We built a prototype[1] of the QBT interface using the Java^{TM} programming language [Jav95]. We chose Java over other similar user-interface development languages because of its object-oriented nature and its widespread availability and use on the Web. One of our objectives in building the prototype was to be able to conduct

[1]The current version of the prototype implementation is available on-line at http://blesmol.cs.indiana.edu:7890/projects/SGMLQuery. Note that only the interface is accessible from outside Indiana University. The results of the queries cannot be viewed from a remote location because of copyright restrictions.

usability experiments in the users' familiar environment. Hence the ability to run the system through the widely available WWW using Java-enabled browsers was a bonus. The prototype implements most of the querying constructs described here including the embedded template (without recursive magnification) and the structure template. We have not yet incorporated the condition box in this prototype, but it will be added in a future release. We also included an experimental version of an SQL language translator from the QBT query. Figure 25(a) and Figure 25(b) show two parts of the screen – the template screen showing the nested template and the structure screen showing the structure template. There is a third component of the interface that displays the SQL query equivalent to the template query. This SQL query gets automatically updated as the user modifies her query using the template.

As an experiment, we used the Chadwyck-Healey English Poetry database using the poetry templates similar to those described above. In the prototype system, queries generated using the interface are sent to a query engine through HTTP (HyperText Transfer Protocol), which is run from a web server as a CGI (Common Gateway Interface) executable. The engine generates its output in HTML which is displayed by the clients. We wrote this engine in C/C++, using the API (Application Programming Interface) provided by the Pat [Ope94] software. More details on the implementation of the query engine are presented in Chapter 6.

7.2.1 GUI Implementation with Java^{TM}

This section presents an overview of the implementation of a prototype of the QBT interface. This prototype is built using the Java^{TM} programming language. In this section, we describe the basic components of this prototype, the design considerations, and implementation details of this prototype. The current prototype has three distinct query interfaces, of which only one can be viewed at a time. The QBT interface that we discussed earlier is included in the "template screen", the structure of the database schema is displayed in the "structure screen", and the equivalent DSQL query is shown in the "SQL screen". The subsection sections describe each of these three screens in detail.

7.2.1.1 Interface components

As described above, there are three separate "screens" in the prototype that are closely linked and are designed to work together. Any change made in one of the screens is also reflected in the other two screens. However, in the current implementation, the three component screens of the interface are not displayed simultaneously because of the overhead required. However, users may switch back and forth between the screens using a "tabbed folder" selection method. The top of the interface consists of three buttons that function like three tabs that can be selected to activate the corresponding screen. When a particular screen is selected, the tab corresponding to that screen gets dimmed, highlighting the current selection and also indicating that the users may switch to only the other two screens. The bottom of the screen has two buttons for submitting the query for evaluation and clearing the current query, much like the buttons found in most HTML forms. In addition, the bottom of the screen also includes options for selecting the number of matches that the system should retrieve at a time and for selecting the region that should be displayed as the default result.

The center of the displayed region contains the main query screen. This is the part that the user may change back and forth using the buttons at the top of the screen. By default, the system displays the template screen at start-up.

The Template Screen The template screen consists of a template image in the background. As the user moves the mouse across the template, the position of the mouse activates the underlying region. This highlights the region on the template as well as displays the name of the region on the status bar. A mouse click on the activated region brings up an expression builder for that region. The expression builder consists of at least one entry area for inputting one or more search terms. It also includes a check-box for indicating negation on that region. When checked, the semantics of the search expression in the target region is negated. Currently, the expression needs to be explicitly included in the entry area, but a future version of the interface will have a graphical expression builder that can build boolean combinations of keywords. A screen capture for the template screen is shown in Figure 29.

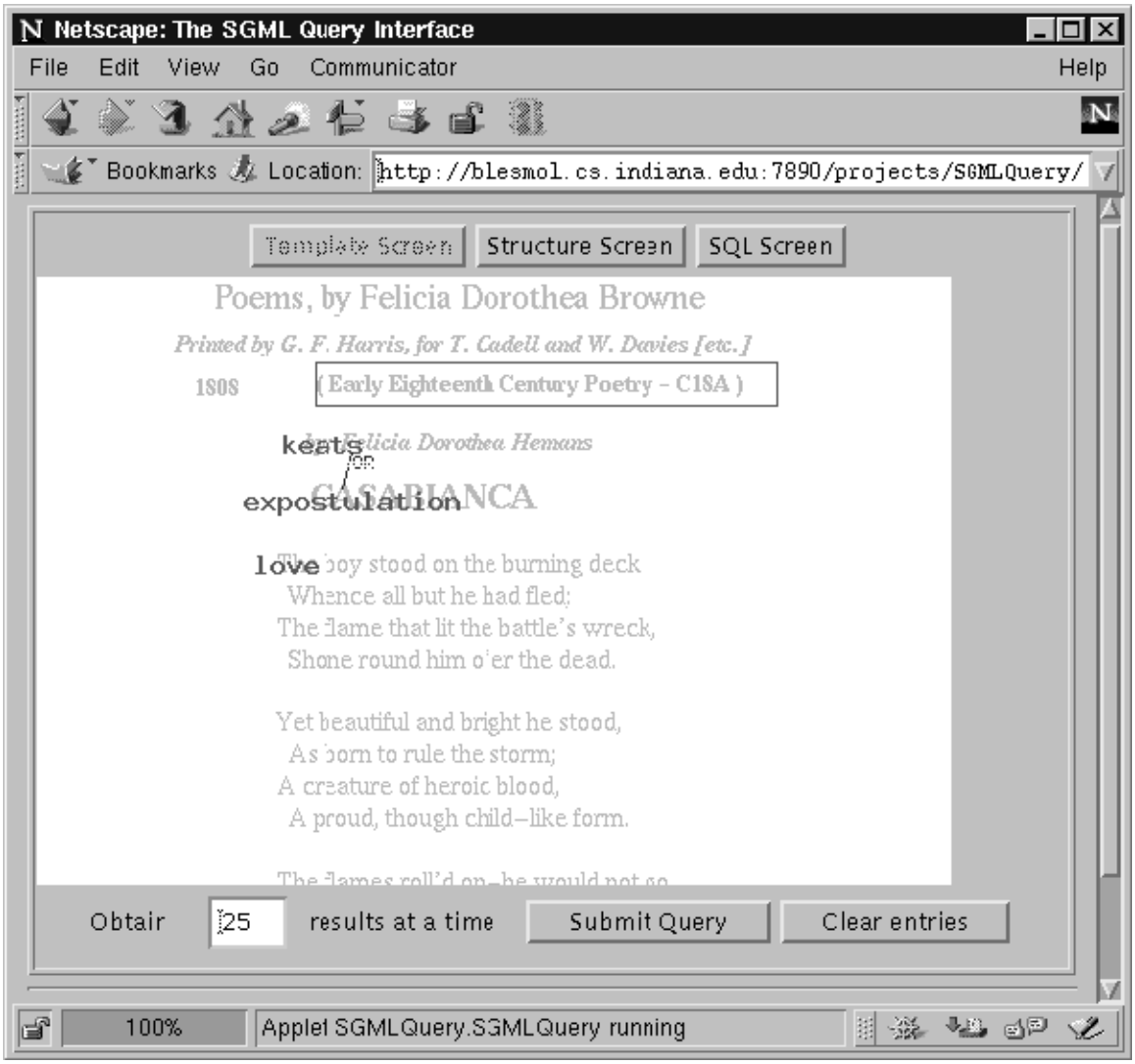


Figure 29: A screen image from the prototype showing the template screen

The Structure Screen The structure screen displays the hierarchical structure of the database. This screen displays the same query as in the templates, by associating a search condition with the corresponding region in the hierarchical display. The structure can be expanded and collapsed by the user as a means for traversing the document structure. Ideally, the structure should be displayed together with the template, with the current region highlighted in both the template and the structure to give the user an idea of the context. In the current implementation, navigation of the structure needs to be manually performed by the user.

The structure screen has two parts: the left half of the screen displays the structure of the database and the right half of the screen displays the query corresponding to the current region highlighted in the structure. The user can change the query by modifying the query text in this section. The condition box is a part of the screen (although it is not implemented in the current prototype); if the user is formulating a query solely using the structure screen, the condition box is the only way to specify boolean combinations of the individual query fragments corresponding to each region.

A screen capture for the structure screen is shown in Figure 30.

The SQL Screen The SQL screen is simply an area where the user can specify the query using the extended SQL described in Chapter 5. This screen is also tightly integrated with the rest of the interface, and a query formulated in any of the other two screens will automatically get reflected in this screen. However, since the current implementation of the template and structure screen does not support joins or nested queries, such SQL queries cannot be automatically translated into the template formulation. However, joins are supported by the internal query engine, so a join formulated in this screen can be submitted for evaluation. A screen capture of the SQL screen is shown in Figure 31.

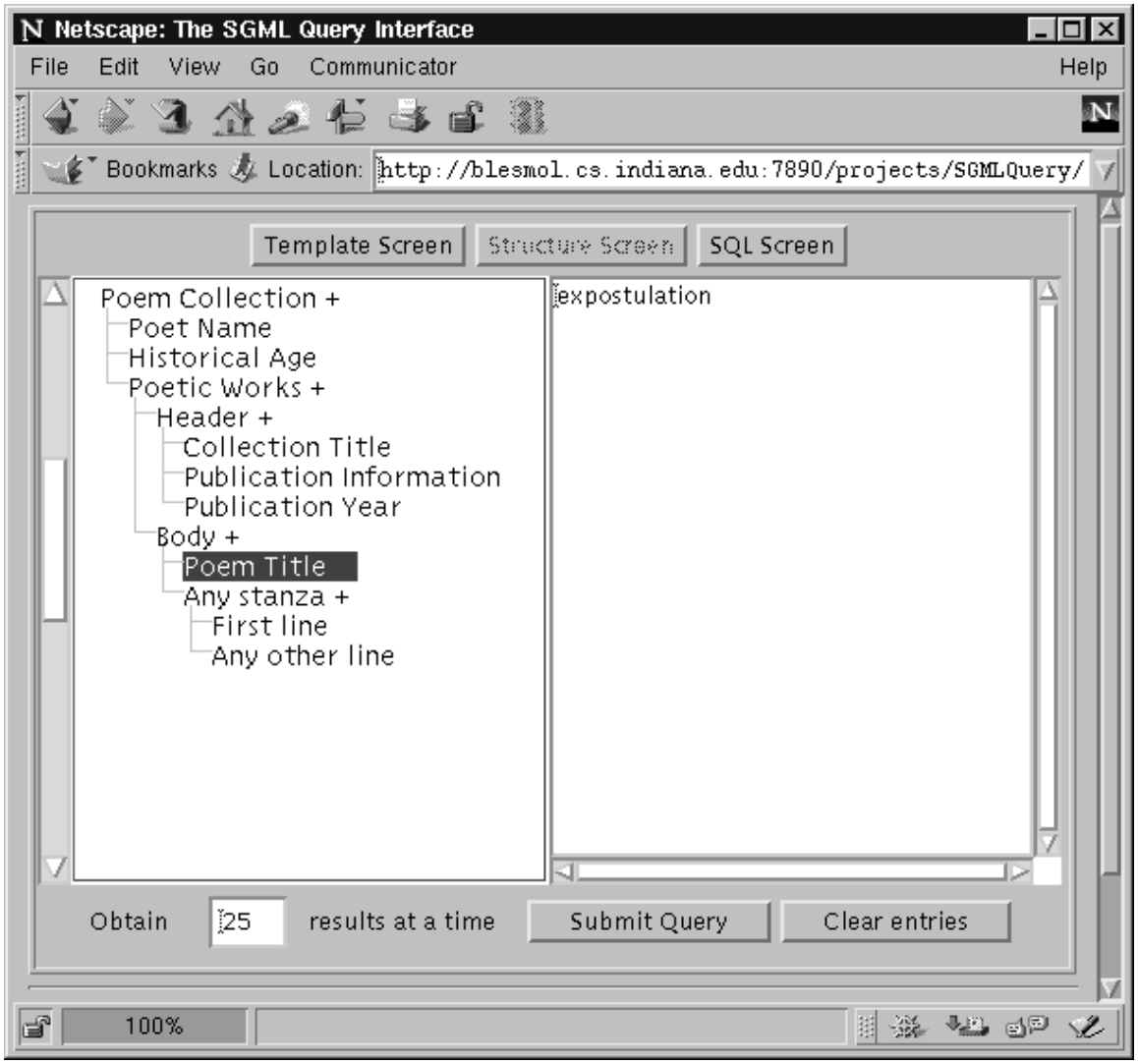


Figure 30: A screen image from the prototype showing the structure screen

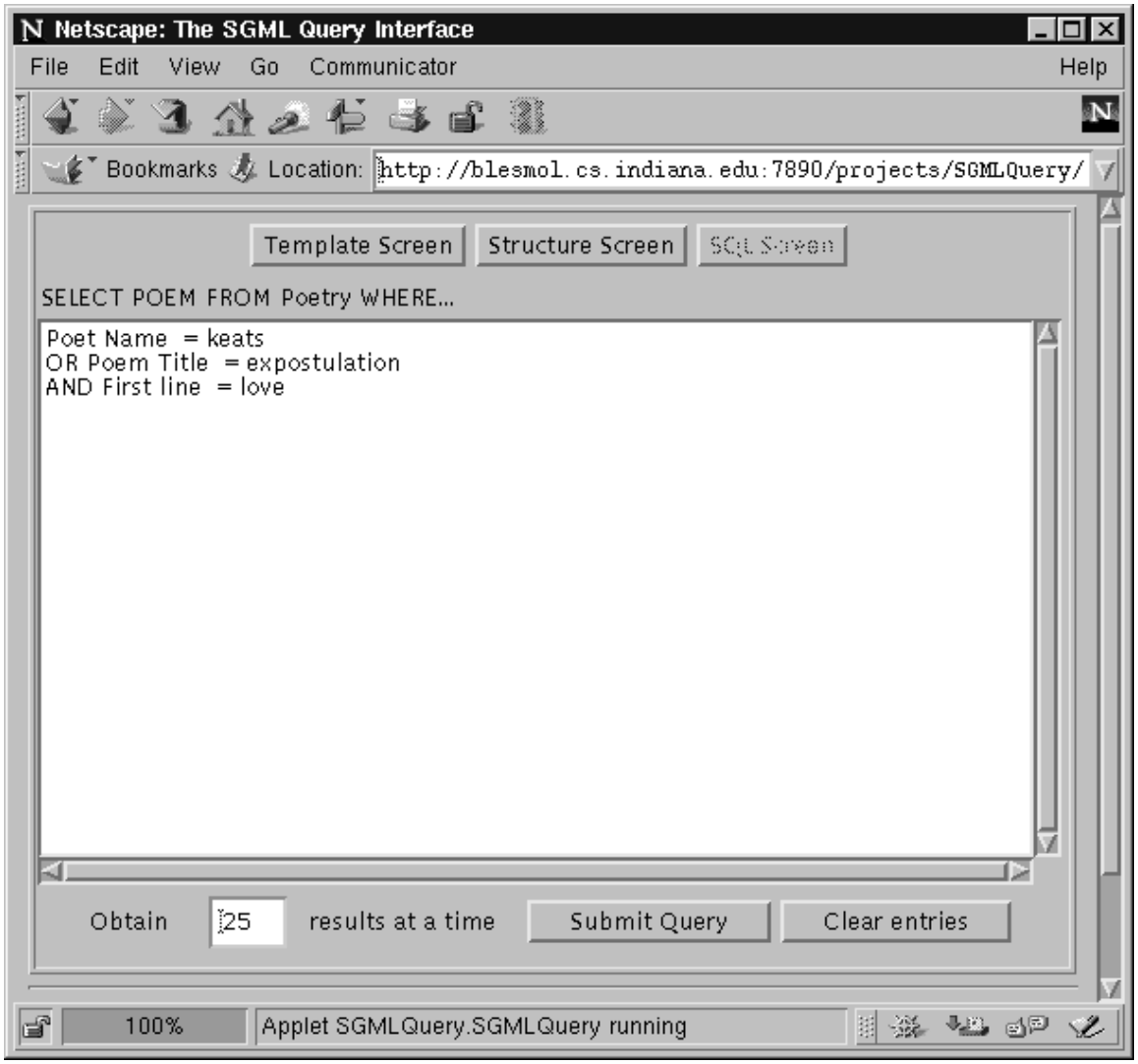


Figure 31: A screen image from the prototype showing the SQL screen

7.2.1.2 Implementation Issues

In this section, we briefly describe some of the issues considered for the implementation of the Java query interface[2]. There are three main modules (Java packages) in this project (see Figure 32 for the class hierarchy):

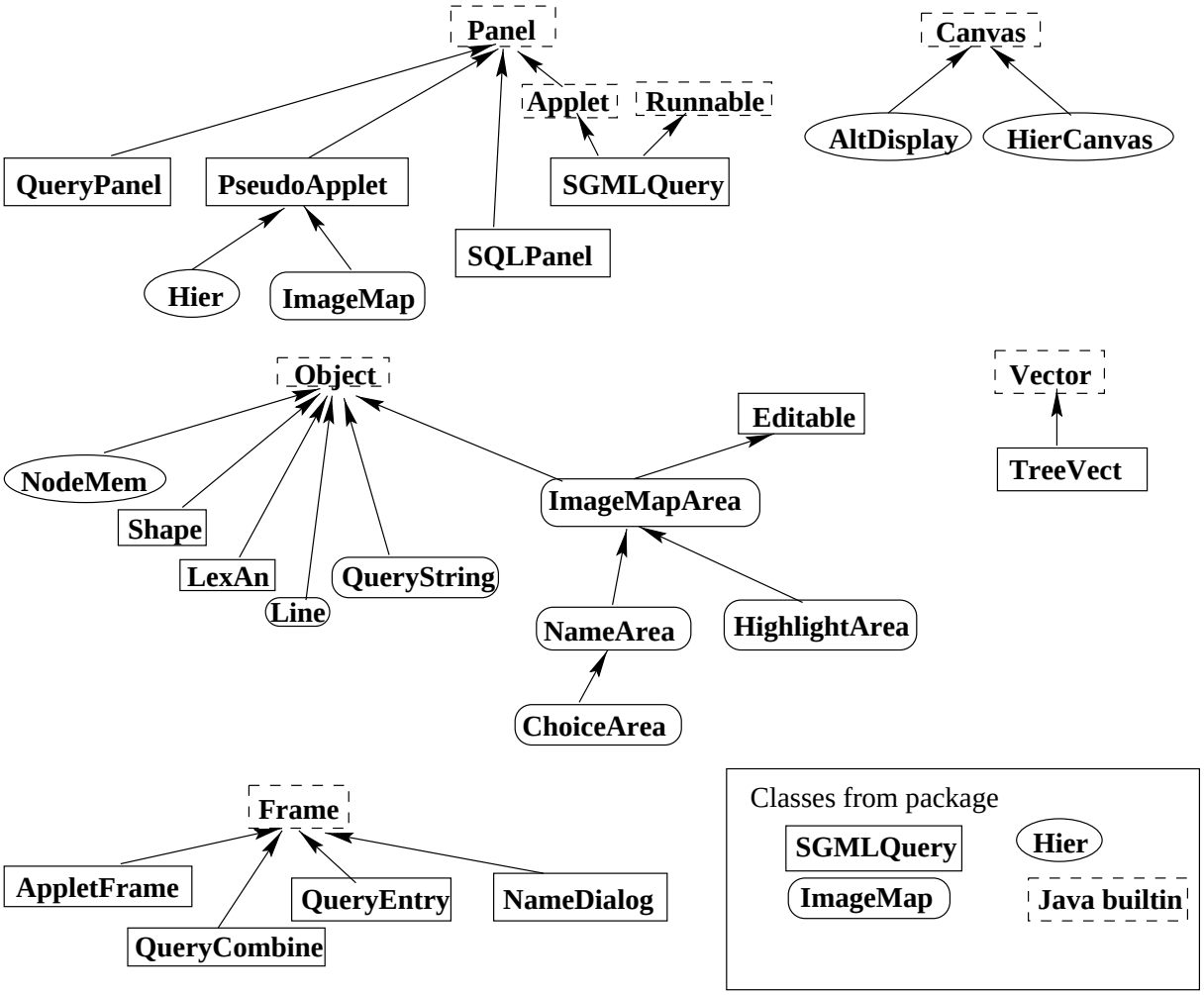


Figure 32: Class Hierarchy of the SGML Query Interface Implementation

[2]The reference manual for the project can be obtained from http://blesmol.cs.indiana.edu:7890/projects/SGMLQuery/doc/packages.html, or by following the SGML Query Interface link from http://www.cs.indiana.edu/hyplan/asengupt.html.

1. *SGMLQuery.* This package serves as the main package and the driver of the basic user interface, query generator and catalog manager.

2. *Hier.* This is the hierarchical structure browser, originally developed by Brogden [Bro95] and adapted for the prototype project.

3. *ImageMap.* This is the primary template screen module. This was originally developed by Sun Microsystems as a demonstration module for Java. This source was used as a starting point with added functionality for the template module.

The SGMLQuery Package The SGMLQuery package includes the main driving class *SGMLQuery* that runs as an applet in a web browser. This class initializes the whole system, including setting up the panel and creating the user interface components. The individual components of the interface generate events which are processed by the method *action* in *SGMLQuery.* Based on the type of the event, this method performs actions such as clearing the query, sending the query to the server, and processing log messages.

Brief description of some of the other classes in this package are given below:

- *AppletFrame.* This class is an experimental class that allows an applet to run as an application without a web browser. Currently, since the query interface requires a web browser to display its results, this class can only display the user interface and is used for quick debugging of the user-interface components.

- *Editable.* This is a Java Interface (classes that cannot be instantiated, but only be inherited from) created to allow multiple query components to share similar editing properties.

- *PseudoApplet.* The Java *Applet* class includes many useful methods such as methods for finding the document URL by accessing the status bar. The *PseudoApplet* class allows non-applet subclasses to inherit these properties.

- *QueryEntry/QueryCombine.* These are subclasses of the builtin Java class *Frame.* *QueryEntry* bring up the entry panel for entering text queries and

QueryCombine brings up the entry panel for specifying the operator combining two query clauses.

- *QueryPanel.* This is primarily a container class that consists of the panel in which the query is displayed. This class uses a *CardLayout* that allows it to switch between the three different views for querying.
- *NameDialog.* This is the class that displays the login dialog box at the start of the application.
- *SQLPanel.* This is the panel that displays the SQL query. It has the capability of automatically generating the SQL equivalent of the query specified by the template and structure screens.
- *LexAn.* A simple lexical analyzer to parse the configuration files.
- *TreeVect.* This is the internal representation of the tree that describes the DTD. Originally, this class was in the *Hier* package, but was moved to the main package so that all the different query components can directly use the same structure instead of having to call methods in the *Hier* package interface.

The Hier Package The *Hier* package is the primary package for displaying and using the hierarchical structure browser in the query interface. The main class in this display is *Hier*, which uses the configuration from the *TreeVect* class to initialize the display and allows a user to interactively expand and collapse the structure on the screen as well as navigate to a particular structure component to specify a query. The other classes in this package are:

- *HierCanvas.* This is the class that displays the structure of the database. It draws the text and the skeleton of the structure and processes user events to expand and collapse the structure.
- *NodeMem.* This is an individual element of the tree structure implemented by the *TreeVect* class which contains the structure information.

The ImageMap Package The *ImageMap* package includes classes that display the template interface. This interface is an extended form of the *ImageMap* demonstration package from Sun Microsystems. An imagemap in a HTML interface consists of an image with specific physical regions associated with different URLs. As described in the design of templates, this resembles the nested template method, and so an imagemap class provided a good starting point for this package. The main class in this package is *ImageMap*, which displays the background image as a template, initializes the different physical areas of the image by associating the classes corresponding to these areas, and processes events generated from user interactions. The primary classes in this package are the following:

- *ImageMapArea.* This class represents individual areas of the imagemap. In the current implementation, each instance of *ImageMapArea* represents a logical region of the document structure, and provides a correspondence between the physical area of the template screen and the logical region of the database by appropriately highlighting the region and displaying the region name in the status area.

- *HighlightArea.* This is a simple class which highlights or un-highlights a particular area when focus is received.

- *NameArea.* This class represents an individual template region, which can handle a query entered as a string.

- *ChoiceArea.* A subclass of *NameArea*, this class allows the query to be selected from a list of choices (set of pre-defined values that can appear in the corresponding region).

- *QueryString.* This is a class that can hold an individual query string and is capable of parsing the embedded logical operators for connecting query components in the same region.

- *Line.* This is a simple class which represents an inter-region connection for the purpose of specifying logical operations between queries specified in different regions.

7.3 Usability Testing

We performed an extensive usability analysis of the prototype interface. The main goal of this analysis was to detect differences between this interface and a standard forms-based interface with similar search capabilities. In particular, we were interested in differences with respect to (1) accuracy, (2) efficiency and (3) satisfaction. This section describes the method used during the experimental evaluation of the prototype Java-based (QBT) interface described above (Section 7.2).

To compare the QBT interface with a normal form-based interface, we created a prototype form interface for searching the database with similar querying capabilities. A screen image of the form interface is shown in Figure 33. This form uses the basic form building blocks provided by HTML, in a style commonly used in web search engines. The output formats for both of these interfaces are the same and are generated by the same query engine.

7.3.1 Experimental Design

The experiment consisted of two primary parts. In the first part, we gave the subjects ten questions – among which we prepared nine and left the tenth question open to the subject's imagination. All subjects were given the same set of questions (see Appendix C). The questions varied in complexity and were designed so that all except one returned some matches. The subjects were divided into two categories based on their familiarity and expertise with the subject. Each subject used one of the two interface types and answered the questions using the assigned search interface by writing down the number of matches returned by the search. The subjects were asked to ascertain that the question was interpreted properly by the searching program. At the conclusion of the experiment, the performance of each subject was evaluated based on their efficiency, accuracy and satisfaction.

The independent and dependent variables for the experiment are outlined below:

Independent variables The independent variables (determining factors) for this analysis were the following:

A. *Interface Type.* (1) the QBT-based interface and (2) the form-based interface

B. *Subject Type.* (1) expert and (2) novice.

Dependent variables The dependent variables (evaluation factors) for the analysis were the following:

1. *Efficiency.* The amount of time in seconds the subjects take to answer each question.
2. *Accuracy.* The degree of accuracy of the queries (*i.e.*, to what extent the queries matched the textual query given to the users). See Appendix C for the actual queries.
3. *Satisfaction.* How satisfied the users were after using the interface (measured by self-reports in written debriefing).

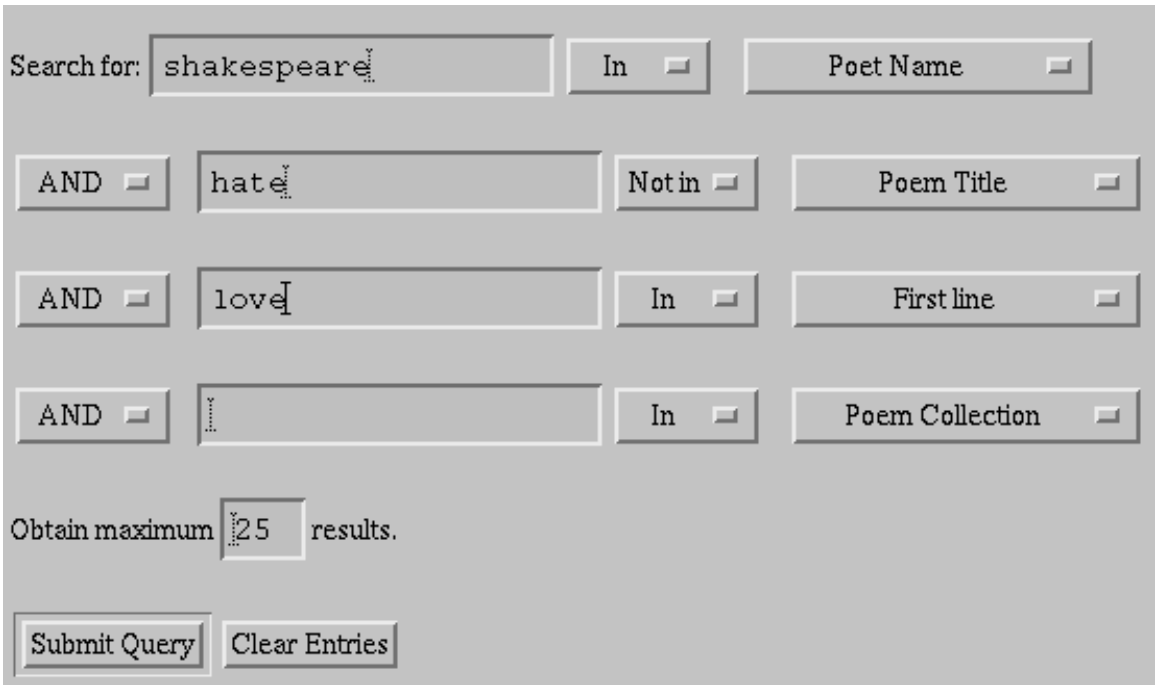


Figure 33: The form implementation of the query interface used in the usability analysis

7.3.2 Subjects

A usability analysis procedure with a pilot test was conducted as part of this research. The first experiment was designed as a pilot test for the actual usability process. In this experiment, four subjects, one in each category (novice-form, novice-QBT, expert-form, expert-QBT) participated in this study. The main purpose of this study was to determine the appropriateness of the analysis technique and ways the experimental design could have been improved. The rest of this section refers to the final usability analysis experiment.

Twenty subjects participated in the final usability analysis. We structured the study using a "between-users" strategy [Rub94], where two distinct groups of users use the two platforms. In our experiment, ten subjects were given the Java-based interface (see Figure 25), while the other ten users were given the form-based interface (see Figure 33). Each subject was placed in one of two distinct groups of five experts and five novices.

We chose the subjects from students who volunteered to participate in the research. The only restriction imposed on the subjects was that they all be Indiana University affiliates because of the copyright restrictions on the database which we used in the experiment. We divided the subjects into two groups based on their experience with computers and databases. The subjects classified as "novices" had minimal computer expertise – generally limited to only e-mail and occasional World Wide Web access. The subjects classified as "experts" were people accustomed to using databases and the web as well as designing and programming graphical user interfaces. We made no distinctions between male and female subjects or young and old subjects, since sex and age were not considered as independent variables in this analysis. Eleven female and nine male subjects, all within the age group of 20–35, participated in this study.

7.3.3 Equipment – Software and Hardware

We performed all the experiments using Netscape 2.0 for the Java-based interface and either Netscape 2.0 or 1.1 for the form-based version. For the Java-based interface,

we restricted experiments to machines having 16MB or more system memory, since Netscape's Java performance is sub-standard with less memory. No memory restriction was enforced for the second interface as the HTML forms do not have additional memory requirements.

As described earlier, all the sessions were held in the users' familiar environments. Only one of the subjects (novice) did not have access to a specific computer environment. In this case, we performed the session at the usability laboratory at the School of Library and Information Science at Indiana University. The rest of the sessions were held at the subjects' homes or offices or the laboratories that they were primarily accustomed to. Although this meant that the client machines varied in many ways, this did not make much difference in terms of their efficiency since most of the search processing was done on the server side (which was the same for all cases).

7.3.4 Data Collection

We collected two types of written data: (1) the subjects' responses to the survey questions and (2) the subjects' responses to the number of matches for each search problem (See Appendix C for the actual questions). The subjects were timed automatically by the server and the query engine that was actually executing the queries. The server also kept a detailed log on the actions performed by the users during the experiments, including the actual query that was executed.

7.3.4.1 Basic Procedure

The subjects were introduced to the experiment and the target interface. After an initial introduction, the subjects were given the experimental search problems and asked to obtain the search results by composing queries sequentially using their target interface. Once the system responded with a result, they recorded the number of matches returned. They were also asked to verify their results in order to check for possible typographical errors by checking the response from the database and viewing sample results from their search. After they finished the searches, they were given a set of survey questions. They were also asked to orally describe their feelings and

general reactions about the functionality and appropriateness of the systems.

7.3.4.2 Experimental Search Queries

A set of nine queries (see Appendix C) was given to each subject. For the tenth query, they were asked to search for something of their own interest. The purpose of this tenth query was primarily to decide the types of questions that are usually asked by users, and use the response for determining the scope of the languages and future usability studies. The first and the easiest query was primarily meant to acquaint the subjects with the system, and the last query was mainly to see what types of questions users were interested in. The other queries ranged from very simple searches involving a single clause in a field to complex searches involving up to four clauses combined together. Note that the QBT interface had no restrictions on the number of clauses that could be specified, but the form interface was limited to only four clauses, which is why we did not involve any query with more than four clauses.

7.3.4.3 Timing Techniques

The subjects were timed by electronic means. Whenever a user submitted a query using either interface, the server logged the access time. The query engine that we designed also logged timing and other detailed information about the queries sent by the users. The Java interface sent logging messages to the server in response to actions performed by the user. This allowed the server to keep track of all the actions (such as button press and query selection) that the user took over the course of submitting the queries.

Examples of the log messages kept at the server side are shown in Figure 34. In the session denoted by this log, the user authenticates himself as "Alan" and specifies two queries. These logs keep track of when particular query string is specified for any specific region, when the query is submitted to the query engine and when the user switches between the different screens of the interface. The date and times are used to calculate the actual time taken by the user to formulate the queries. For example, in this log, the user takes 42 seconds to formulate the first query and 33 seconds to

formulate the second query (in the first case, the time is calculated by the difference between the authentication and submission, and in the second case, from the restart and submission).

```
<none>~4/9/96 14:45:11~init~~
<none>~4/9/96 14:45:57~start~~
alan~4/9/96 14:47:50~auth~~
alan~4/9/96 14:48:20~query~Poem Title~casabiana
alan~4/9/96 14:48:30~query~Poem Title~casabianca
alan~4/9/96 14:48:32~submit~head=casabianca~
alan~4/9/96 14:48:37~stop~~
alan~4/9/96 14:50:23~start~~
alan~4/9/96 14:50:28~query~Poem Title~NOT casabianca
alan~4/9/96 14:50:29~submit~head=NOT+casabianca~
alan~4/9/96 14:50:37~stop~~
alan~4/9/96 14:50:44~start~~
alan~4/9/96 14:50:52~query~Poem Title~
alan~4/9/96 14:51:15~query~Poet Name~rilke
alan~4/9/96 14:51:17~submit~poet=rilke~
alan~4/9/96 14:51:19~stop~~
alan~4/9/96 14:51:23~start~~
alan~4/9/96 14:51:29~query~Poet Name~
alan~4/9/96 14:51:30~switch~0~1
alan~4/9/96 14:51:52~switch~1~0
alan~4/9/96 14:51:53~switch~0~1
```

Figure 34: Sample log messages stored at the server

7.3.4.4 Survey Questions

In addition to the queries, we gave the users a small set of questions to assess their experience, preferences, and degree of satisfaction with the interface. They were also asked to point out features that they liked or disliked in the interface they used. The survey questions are also listed in Appendix C. The primary purpose of the survey was to determine the degree of satisfaction reached by the users and the comparability of the interface they used with other search interfaces that they have experienced on the web prior to this experiment.

7.3.4.5 General Feedback

After the experiment was over, the subjects were asked to comment on their general feelings about the project; their comments and suggestions were noted. This data was primarily used for the purpose of designing improved features for the current interface.

7.4 Usability Evaluation

This section describes in detail the results that we obtained from the usability analysis. We divide the results into three different sections, one each for the dependent variables – *accuracy*, *efficiency*, and *satisfaction*. We used a statistical measure to determine whether or not the data gathered had enough information to sufficiently support any claim for significance. A common statistical method used in determining significance results is ANOVA or Analysis of Variance (for an introduction to ANOVA, see [WW90, Chapter 10]). The ANOVA technique analyzes variance within samples and provides a method for determining whether two or more samples showing significant difference based on one factor (simple ANOVA) or multiple factors (multivariate ANOVA). The result of an ANOVA computation with a sample of observations of two different events provides a confidence level for determining whether the two events were different. For an ANOVA analysis, based on the degrees of freedom (number of factors affecting the event), a significance level is decided (usually a small value such as .05), and a sample is only considered to be exhibiting significant difference if the value is lower than this threshold.

For each of the measures, we performed a multivariate ANOVA test with a .05 significance level. Here we show the mean and standard deviation values for the effects of interface and expertise for each of the dependent variables, and comment on the result of the analysis.

In the following analysis, for the independent variable "Interface type," the Java interface (Figure 25) is given a value of 1 and the form interface (Figure 33) is given a value of 2. For the independent variable "Subject type," the values of 1 and 2

are assigned to experienced and novice users, respectively. The tasks are denoted as "Task 1" through "Task 10."

7.4.1 Accuracy

We measured accuracy by evaluating the answers to each question on a $0 - 5$ scale. Perfect answers were given 5 points and completely wrong answers (of course, there were none in the experiment) were given 0 points. Partially incorrect answers were given a value in the range of 1 to 4, inclusive, based on the type of mistake. Table 9(a) shows the cumulative means and standard deviations for all tasks on the accuracy value. Appendix C shows the actual accuracy measures for all the tasks.

	Interface 1	**Interface 2**	**Overall**
Expert	4.64 (0.98)	4.76 (0.87)	4.7 (0.92)
Novice	4.78 (0.61)	4.62 (1.02)	4.7 (0.84)
Overall	4.71 (0.82)	4.69 (0.95)	

Source of Variation	**SS**	**DF**	**MS**	**F**	**Sig. of F (p)**
Within + residual	15.00	16	0.94		
Interface	0.02	1	0.02	0.02	0.886
Expertise	0.00	1	0.00	0.00	1.000
Interaction: interface and expertise	0.98	1	0.98	1.05	0.322

Table 9: Effect of Interface and expertise on accuracy: (a) Summary of mean(standard deviation) over all tasks, (b) Results of the F tests and significance values

The tasks 1, 2, 4 and 10 had 0 standard deviation since all users had correct answers for these tasks. For the rest of the tasks, the cumulative effect of expertise or interface was non-significant at the $P < 0.5$ level as shown in Table 9(b). $F(1, 16) = 0.02, p = 0.886$ for interface effects, $F(1, 16) = 0.00, p = 1.000$ for expertise effects, and $F(1, 16) = 1.05, p = 0.322$ for their interaction: none of which show significance at $P < 0.5$ level.

7.4.2 Efficiency

For the efficiency measure, we used the time (in seconds) between two successive submissions of queries. The absolute times at which (1) the system was first accessed and (2) the queries were submitted, were logged by the query processing system. We calculated the difference between these times to get the time taken for each task by the subjects. For the first task, we used the time difference between the first access of the search page and the submission of the first task. This turned out to be a problem (as indicated by the results), since the Java interface page did not have any other text besides the search interface itself. On the other hand, the form interface contained some instructions; most of the subjects spent time reading these instructions before composing the first query. Table 10(a) shows the cumulative means and standard deviations for all tasks with respect to efficiency. Appendix C shows the actual efficiency measures for all the tasks

	Interface 1	**Interface 2**	**Overall**
Expert	69.64 (39.56)	85.71 (102.1)	77.59 (77.16)
Novice	123.96 (73.4)	170.58 (152.41)	147.27 (121.30)
Overall	96.8 (64.70)	128.57 (136.16)	

Source of Variation	**SS**	**DF**	**MS**	**F**	**Sig. of F (p)**
Within + residual	143935.39	15	9595.69		
Interface	40268.65	1	40268.65	4.20	0.058
Expertise	241839.89	1	241839.89	25.20	0.000
Interaction: interface and expertise	14194.36	1	14194.36	1.48	0.243

Table 10: Effect of Interface and expertise on efficiency: (a) Summary of mean(standard deviation) over all tasks, (b) Results of the F tests and significance values

Table 10 (b) displays the results we obtained from the multivariate tests of significance. Here, the effect of the interaction of expertise and interface was non-significant at the $P < 0.5$ level ($F(1, 15) = 1.48, p = 0.243$). However, expertise had significant effect on efficiency ($F(1, 15) = 25.20, p = 0.000$). The means clearly suggest that

the experts were significantly more efficient than the novices using both the interfaces, so the different interfaces did not effect experts' performances. The effect of interface on efficiency, however, was marginally non-significant at the $P < 0.5$ level ($F(1, 15) = 4.20, p = 0.058$). Although this indicates that the interface does not necessarily made users significantly more efficient, this also suggests that the easier interface does not make them any slower either.

Univariate tests of significance on individual tasks, however, show significant effects of interface only for Task 1 ($F(1, 15) = 85.626, p = 0.00$) and Task 7 ($F(1, 15) = 16.385, p = 0.001$), while the rest of the tasks did not show any significance. For Task 1, the subjects using the Java interface performed significantly better than the subjects using the form interface, because of the time they spent looking at the help information that was absent for the Java interface. For Task 7, the users of the form interface performed significantly better than the users of the Java interface. On a subsequent analysis of the subjects' actions, we discovered that this task required the users to switch to a different screen for the Java interface. Unfortunately, most of the users could not understand the necessity for this action. This situation will be rectified when all three screens are displayed simultaneously; then, users will not have to switch to a different screen in order to perform this query.

7.4.3 Satisfaction

For the satisfaction measure, the users were asked to grade the interface that they used in a scale of five qualitative values: *Much better, Little better, About the same, Worse, Absolutely worse.* These five classes were assigned the ranks 5,4,3,2 and 1 respectively. This data was taken after all the actual tasks were performed and was not calculated on a task-by-task basis. Table 11(a) shows the mean and standard deviation of the satisfaction measure taken for this observation, and Table 11(b) shows the results of the tests of significance using unique sums of squares ANOVA method. From this table, we observe significant effect on satisfaction of the interface at $P < 0.05$ level ($F(1, 16) = 7.53, p = 0.014$). However, the expertise and the interaction of expertise and interface do not show any significant results.

	Interface 1	**Interface 2**	**Overall**
Expert	4.6 (0.54)	3.8 (0.83)	4.2 (0.78)
Novice	4.8 (0.44)	4.0 (0.70)	4.4 (0.69)
Overall	4.7 (0.48)	3.9 (0.73)	

Source of Variation	**SS**	**DF**	**MS**	**F**	**Sig. of F (p)**
Within + residual	6.80	16	0.42		
Interface	3.20	1	3.20	7.53	**0.014**
Expertise	0.20	1	0.20	0.47	0.503
Interaction: interface and expertise	0.00	1	0.00	0.00	1.000

Table 11: Effect of Interface and expertise on satisfaction: (a) Summary of mean(standard deviation) over all tasks, (b) Results of the F tests and significance values

7.5 Summary

QBE and forms are both quite popular means for querying in the relational domain. The main advantage of the form interface is that it is very simple to implement and easy to use for small databases. However, forms do not adapt very well for databases with a complex structure, and most text-based databases do tend to have a complicated structure (*e.g.*, the Chadwyck-Healey database used in the prototype contains over fifty logical regions.) A form interface that can search on only a few of these areas is easy to construct, but if the number of searchable regions is increased, the interface tends to get too crowded too quickly. With QBT, the query interface stays simple regardless of the complexity of the underlying structure, and the depth of structure navigation can be controlled by the users using the nested template or the structure template. For complex hierarchies, the focus can also be concentrated in the regions of interest using advanced methods like differential magnification [KR96].

Another advantage of the template method is its direct relationship to the internal structure of the database. Forms always look the same, whether the underlying database is a poem, a dictionary, a quotation collection, or even a relational database. However, templates can be custom-designed for different types of databases. This way, templates can provide a direct reflection of the users' mental models [Boo89, Chap. 6],

a significant factor in the design of good user-interfaces. Moreover, templates use the principle of familiarity [Nor90], which is demonstrated to work well for novice users. The only disadvantage of templates is that good templates require expensive graphics terminals, while forms work quite well with terminals without graphics capabilities. However, with the advance in technology, non-graphics terminals are less common, so the assumption of a graphics-capable terminal is not very demanding.

The implementation of QBT in this work is in an early developmental stage and has substantial potential for improvement. The experiment we performed clearly indicated some of the ways it could be improved. However, in spite of being a prototype interface, this QBT implementation demonstrates that QBT is suitable for querying textual databases using a simple graphical interface. Moreover, QBT is at least as accurate and efficient as the general form-based approach and is significantly more satisfying to the users. We believe that the idea behind QBT will give us a starting point for query interfaces in future text database systems. A significant portion of the current research is aimed towards the theoretical stability and soundness of the QBT concept, and once established, this method has the potential of becoming the standard querying mechanism for text databases.

Chapter 8

Conclusion and Future Work

Current relational models as well as other advanced database models such as the complex-object and object-oriented models lack the ability to properly model documents with complex hierarchical structure. It is usually possible to map a document structure into a database schema, but such mappings are not always one-to-one, and often result in loss of information contained in the original documents as a result of the mapping operation. The SGML standard [ISO86] provides a uniform system-independent and paltform-independent method of encoding documents with a complex hierarchical structure. The process of modeling documents in SGML resembles that of modeling databases, using a DTD as a schema and conforming documents as instances. The research in this dissertation used this property of SGML to provide SGML repositories with database-like properties, using the SGML data model and a set of minimal yet powerful query languages. A proof-of-concept implementation of the model and a considerable subset of each query language was implemented on top of standard storage managers and indexing utilities.

This chapter describes the contributions made by this dissertation and presents directions for future research in database systems for structured documents.

8.1 Contributions

The most significant contributions in this thesis are the design ideas for building a database system for structured documents. In achieving this result, this research also makes the following contributions:

1. *Proposal of a formal model for structured documents.* In Chapter 4, we described an elegant model for structured documents using SGML.

2. *Design of low complexity query languages.* In Chapter 5, we proposed a simple query language with some minor extensions over the relational languages as well as some new semantics. We also showed that this language has the desired properties of a low-complexity, closed query language and forms a core language on which more powerful languages can be built.

3. *Proposal for a standard query language for SGML databases.* In Chapter 5, we proposed a practical language for SGML users, using SGML itself, and demonstrated its special form of closure and other desirable properties.

4. *Design and implementation of a query processing infrastructure for document databases.* In Chapter 6, we described an architecture of a query processing system for document databases that does not require any transformation process for converting documents into a different database format.

5. *Design and implementation of a prototype system with most of the desired features.* In Chapter 6, we described the implementation of **DocBase**, a prototype system for posing queries in a document database. DocBase accepts queries using either SQL or a simple visual interface to formulate queries.

6. *Design of a generalized method for current SGML systems to support SQL-like queries.* The prototype system described in Chapter 6 uses Pat, a commercial system popularly used for searches on SGML data, and builds an SQL query processing infrastructure on top of this system. The same technique can be used with most current SGML applications.

7. *Design and implementation of a generalized visual query language.* In Chapter 7, we described a query formulation interface based on a simple template metaphor, which proved to be an effective alternative to forms-based query interfaces.

8.2 Future Work

- *Full SQL implementation.* As described in Chapter 6, the implemented language is a subset of the complete SQL language described in Chapter 5. We described earlier how some of the features that are yet to be implemented, can be incorporated in this system. The similarity between the method of processing of queries in the implementation here and in relational databases indicates that many of the methods already used in relational databases would also apply in the domain of document databases. In particular, processing of nested queries can be performed using a tuple-substitution method [SAC+79]. Further research is necessary to evaluate application of advanced techniques such as in [Day87] for nested query processing, grouping, ordering and aggregation operations.

- *Query optimization.* As described in Chapter 6, query optimization issues were considered during the processing and evaluation of queries, and some optimization techniques were implicit in the evaluation algorithms presented earlier. However, it was also stated that in order to efficiently use the Pat indices, most of the algorithms needed to resort to set operations even for purposes such as selection on the same document component. More control over the Pat structures in addition to the operations provided by the Pat query language would allow more efficient means for performing queries. However, further research needs to be performed in order to determine more efficient query evaluation and optimization techniques.

- *Immediate parent and child traversal using Pat indices.* During the analysis in Chapter 6, we noticed that, because of the way sets are constructed for the traversal operations in Pat, it is not possible to traverse to the immediate child or immediate parent of a node (*i.e.,* use of the "**.**" operator in the path expressions). One of the reasons this problem was not addressed in detail was the lack of availability of the internal details for the Pat region indices, as they are proprietary structures of Open Text. Collaboration with Open Text

towards a solution of this problem would definitely be an important step towards building complete SQL support with a system such as Pat.

- *Selectors in path expressions.* In Chapter 5, we needed to introduce the concept of *distinguished queries* in which all the free variables of formulas had to be explicitly rooted to a unique name. This was necessary to ensure that individual components of queries could be extracted from the query. General path expressions (such as in [dBV93]), however, allow positional notations on labels in the path expressions (*e.g.,* book.chapter[1].section[2].title, denoting the titles of the second sections of the first chapters of books). Typically, these positional expressions can be variables, thus increasing the expressiveness of the language. Constant selectors can be trivially introduced in the current language without introducing many changes in its properties. Further research is, however, necessary to determine whether general path expressions can be evaluated in PTIME, and if not, whether reasonable restrictions can guarantee the desired low complexity.

- *Parallelization of DocBase.* The implementation of the query language indicated that information is always local to specific sections of the data. This indicates that it should be possible to distribute the data across processors or systems and to evaluate the final result by combining the resulting fragments.

- *Full QBT implementation.* The visual interface in the current prototype implements a major subset of the QBT technique, however it is still missing some key components which might significantly change the properties of this language. Usability evaluation needs to be performed again after the implementation is completed in order to assess and compare the degree of effectiveness of this design.

8.3 Applicability

One significant aspect of this research is its potential application in a number of areas. With the increase in popularity of HTML and the Internet, we are experiencing an

explosion in the amount of information on the web. Most search engines on the web suffer from their lack of the ability to perform complex searches. The main types of searches are restricted to keyword searches which tend to result in *too many* matches. The ability of users to write SQL-like queries on the web would enable them to restrict searches to certain portions of the documents, and thus reduce the number of unnecessary matches. Based on this and related considerations, this research can be applied for various purposes as stated below:

- *Complex Web Searches.* The current structure of the web is based on HTML. Although HTML uses a mixed markup model, most of the HTML tags are generic and semantics are only associated to them by the browsers. Moreover, HTML has already been incorporated as an SGML DTD [BLC95], so the current research can be easily applied for building complex SQL-capable search engines for the Internet.

- *XML Search Engines.* With the advent of XML [W3C97], custom user-defined tags are becoming standard. This work has the advantage of using structured documents in their native format and process queries based on tags in the documents. Moreover, XML has been proposed as a subset of SGML and backwards compatible to HTML. So this research can be easily applied for searching XML documents in their native format.

- *Modular Design.* The modular design adopted in the implementation allows the replacement of either or both of the underlying external systems (Exodus and Open Text in the current implementation) by other systems. This provides a method for enabling SQL query support in many current SGML processing systems.

- *Advanced SGML Features.* In addition to the SGML features used here, advanced SGML features such as CONCUR, SUBDOC can be used for more complex document operations. CONCUR can be used to provide a concurrent physical layout description for a document, which allows users to search on physical characteristics of documents (such as position of particular objects in

a page). In addition, the SUBDOC feature of SGML can be used to embed queries in SGML documents for dynamic content generation.

8.4 Finale

SGML and SQL were two languages designed for entirely different purposes and standardized in the same year (1986). Although SQL has gained tremendous popularity in the database context, SGML has only recently started to gain popularity as a publishing standard. Because of the highly general nature of SGML, it has the potential for becoming a standard modeling tool for not only documents but any structured data in general. Query languages and processing techniques such as those presented in this dissertation would immensely influence the applicability of SGML as a universal data representation format. The Internet and the World Wide Web is very definitely a sure step towards this future.

Bibliography

[AB95] Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4(4):727–794, October 1995.

[AC75] M. M. Astrahan and D. Chamberlin. Implementation of a structured english query language. *Communications of the ACM*, 18(10), October 1975. Also published in/as: 19 ACM SIGMOD Conf. on the Management of Data, King(ed), May.1975.

[ACM93] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and updating the file. *Proceedings, 19th Intl. Conference on Very Large Data Bases*, pages 73–84, 1993.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Reading, Mass. : Addison-Wesley, 1995.

[AV97] Serge Abiteboul and Victor Viannu. Regular path queries with constraints. In *Proceedings: ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 122–133, Tucson, Arizona, May 1997.

[BBB+88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, and F. Velez. The design and implementation of O2, an object-oriented database system. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Sys.*, volume 334 of *Lecture Notes in CS*, page 1. Springer-Verlag, September 1988.

[BCM96] Tim Bienz, Richard Cohn, and James R. Meehan. *Portable Document Format Reference Manual.* Adobe Systems Incorporated, version 1.2 edition, November 27 1996.

[BGBG95] Ronald M. Baecker, Jonathan Grudin, William A. S. Buxton, and Saul Greenberg. *Readings in Human-Computer Interaction: Toward the Year 2000*, chapter 2. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1995.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Reading, Mass.: Addison-Wesley Publishing Co., 1987.

[BLC95] T. Berners-Lee and D. Connolly. *Hypertext Markup Language - 2.0.* MIT/W3C: HTML Working Group, RFC: 1866 edition, November 22 1995. Available on-line from http://www.w3.org/pub/WWW/MarkUp/html-spec.

[Boo89] Paul Booth. *An Introduction to Human-computer Interaction.* Laurence ErlBaum Associates Publishers, 1989.

[Bro95] Bill Brogden. Hierarchical browser in java. available on the WWW at http://www.bga.com/ wbrogden/javatest.html, 1995.

[Bur92] Forbes J. Burkowski. An algebra for hierarchically organized text-dominated databases. *Information Processing & Management*, 28(3):333–348, 1992.

[BYG89] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Efficient text searching of regular expressions. *Proceedings, 16th International Colloquium on Automata, Languages, and Programming*, pages 46–62, 1989.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. *SIGMOD RECORD*, 23(2):313–324, June 1994.

[CCB95] Charles L.A. Clarke, G.V. Cormack, and F.J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

[CCM96] Vassilis Christophides, Sophie Cluet, and Guido Moerkotte. Evaluating queries with generalized path expressions. In H.V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings, ACM SIGMOD 1996*, volume 25, pages 418–422. Association of Computing Machinery, June 1996.

[CDF+86] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, M. Muralikrishna, Joel E. Richardson, and Eugene J. Shikita. The architecture of the EXODUS extensible DBMS. In Klaus R. Dittrich and Umeshwar Dayal, editors, *Proceedings, 1996 International Workshop on Object-Oriented Database Ssytems*, pages 52–65, Pacific Grove, California, USA, September 23-26 1986. IEEE-CS.

[Cha94] Chadwyck-Healey. *The English Poetry Full-Text Database*, 1994. The works of more than 1,250 poets from 600 to 1900.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship model – toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, March 1976.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1989.

[Cod70] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 6(13):377–387, June 1970.

[D'A95] Al D'Andrea. Improved database technology for document management. In Yuri Rubinsky, editor, *Proceedings, SGML '95*, pages 113–122. Graphic Communications Association, December 1995.

[Dat89] C.J. Date. *A guide to the SQL Standard: A user's guide to the standard relational language SQL*. Addison-Wesley Publishing Co., 1989.

[Day87] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In Peter M Stocker and William Kent, editors, *Proceedings: International Conference on Very Large Data Bases (VLDB)*, pages 197–208, Brighton, England, September 1-4 1987. Morgan Kaufmann.

[dBV93] Jan Van den Bussche and Gottfried Vossen. An extension of path expressions to simplify navigation in object-oriented queries. In Stefano Ceri, Katsumi Tanaka, and Shalom Tsur, editors, *Proceedings of the third international conference on Deductive and Object-Oriented Databases (DOOD)*, number 760 in Lecture Notes in Computer Science, pages 267–282, Phoenix, Arizona, December 1993. Springer-Verlag.

[DGS86] B.C. Desai, P. Goyal, and F. Sadri. A data model for use with formatted and textual data. *JASIS*, 1986.

[DR93] Joseph S. Dumas and Janice C. Redish. *A practical guide to usability testing.* Ablex publishing corporation, 1993.

[Ebe94] Ray E. Eberts. *User Interface Design.* Prentice Hall, 1994.

[Emb89] D.W. Embley. NFQL: The natural forms query language. *ACM Transaction on Database Systems*, 14(2):168–211, June 1989.

[GBY91] Gaston H. Gonnet and R. Baeza-Yates. Lexicographical indices for text: Inverted files vs pat trees. Technical Report TR-OED-91-01, University of Waterloo, 1991.

[GNU92] GNU Project. *Unix Commands Refernece Manual*, Sep 1992.

[Gol90] Charles F. Goldfarb. *The SGML Handbook.* Clarendon Press, Oxford, 1990.

[Gou95] John D. Gould. How to design usable systems. In Ronald M. Baecker, Jonathan Grudin, William A. S. Buxton, and Saul Greenberg, editors,

Readings in Human-Computer Interaction: Toward the Year 2000, chapter 2, pages 93–121. Morgan Kaufmann Publishers, San Francisco, California, 1995.

[GPG89] M. Gyssens, J. Paredaens, and D. Van Gucht. A grammar based approach toward unifying hierarchical data models. *SIGMOD*, pages 263–272, 1989.

[GT87] Gaston H. Gonnet and Frank W. Tompa. Mind your grammar: a new approach to modeling text. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *Proceedings: 13th International Conference on Very Large Data Bases*, pages 339–346, Brighton, England, September 1-4 1987. Morgan Kaufmann.

[GZC89] Guting, Zicari, and Choy. An algebra for structured office documents. *ACM TOIS*, 1989.

[Hel88] Martin Helander. *Handbook of Human-Computer Interaction*. North Holland, 1988.

[Hol95] Sebastian Holst. Database evolution: the view from over here (a document-centric perspective). In Yuri Rubinsky, editor, *Proceedings, SGML '95*, pages 217–223. Graphic Communications Association, December 4-7 1995.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Inf95] Inc. Inforium. Livepage^{TM}: A system for open information exchange, 1995. Software Information Brochure.

[ISO86] International Organization for Standardization, Geneva, Switzerland. *ISO 8879: Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, 1986.

[ISO94] International Organization for Standardization and International Electrotechnical Commission, Geneva, Switzerland. *ISO/IEC DIS 10179: Document Style Semantics and Specification Language: DSSSL*, 1994.

[Jav95] Sun Microsystems. *The Java^{TM} Language Specification: Version 1.0 Beta*, 1995.

[JK96] Jani Jaakkola and Pekka Kilpeläinen. The sgrep online manual. Available online at http://www.cs.helsinki.fi/ jaakkol/sgrepman.html, 1996.

[JMG95] Manoj Jain, Anurag Mendhekar, and Dirk Van Gucht. A uniform data model for relational data and meta-data query processing. In *Proceedings of the Seventh International Conference on Management of Data (COMAD)*, pages 146–165. Tata McGraw-Hill Press, December 1995.

[JW83] Barry E. Jacobs and Cynthia A. Wasczak. A generalized query-by-example data manipulation language based on database logic. *IEEE Transactions on Software Engineering*, SE-9(1):40–56, January 1983.

[KKS92] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying object-oriented databases. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 393–402, June 1992.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu86] Donald E. Knuth. *The TEXbook*. Addison Wesley, 1986.

[KR96] T. Alan Keahey and Edward L. Robertson. Techniques for non-linear magnification transformations. In *Proceedings, Visualisation '96 Information Visualization Symposium*. IEEE, October 1996.

[Lam94] Leslie Lamport. *LATEX A Document Preparation System*. Addison Wesley, 2nd edition, November 1994.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex & yacc*. O'Reilly & Associates, 2nd ed. edition, 1992.

[McG77] W. McGee. The information management system IMS/VS, part I: General structure and operation. *IBM Systems Journal*, 16(2), June 1977.

[MK76] O. L. Madsen and B. B. Kristensen. LR-parsing of extended context free grammars. *Acta Informatica*, 7(1):61–73, 1976.

[MW93] Udi Manber and San Wu. Glimpse: A tool to search through entire file systems. Technical Report TR 93-34, University of Arizona, October 1993.

[MW95] A.O. Mendelzon and P.T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, December 1995.

[Nor90] Donald Norman. *The Design of Everyday things*. Doubleday Currency, 1990.

[NP93] J. Nielsen and V. Phillips. Estimating the relative usability of two interfaces: Heuristic, formal, and empirical methods compared. In *Proceedings: INTERCHI'93*, pages 214–221. ACM, 1993.

[Ope94] Open Text Corporation, Waterloo, Ontario, Canada. *Open Text 5.0*, 1994.

[Oss76] J.F. Ossanna. Nroff/troff user's manual. Technical Report Comp. Sci. Tech. Rep. 54, Bell Laboratories, Murray Hill, NJ, October 1976.

[OW93] Gultekin Ozsoyoglu and Huaqing Wang. Example-based graphical database query languages. *Computer*, 26(5):25–38, May 1993.

[Paw82] Z. Pawlak. Rough sets. *International Journal of Computer and Information Sciences*, 11:341–356, 1982.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[PT86] P. Pistor and R. Traunmueller. A database language for sets, lists, and tables. *Information Systems*, 11(4):323–336, 1986.

[Rub94] Jeffrey Rubin. *Handbook of Usability Testing: How to plan, design and conduct effective tests.* John Wiley & Sons, Inc., 1994.

[SAC+79] Patricia G. Selinger, Moton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings: Special Interest Group on Management of Data (SIGMOD)*, pages 23–34, Boston, MA, May 30-June 1 1979. ACM.

[Sal91] Gerard Salton. Developments in automatic text retrieval. *Science*, 253:974–980, 1991.

[SB88] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24:513–523, 1988.

[Sch97] Bruce R. Schatz. Information retrieval in digital libraries: Bringing search to the net. *Science*, 275:327–334, January 1997.

[Sen96] Arijit Sengupta. Demand more from your SGML database! bringing SQL under the SGML limelight. *<TAG>*, 9(4):1–7, April 1996.

[Sha84] B. Shackel. The concept of usability. In J. Bennett, D. Case, J. Sandelin, and M. Smith, editors, *Visual Display Terminals: Usability Issues and Health Concerns*, pages 45–87. Prentice Hall, Englewood Cliffs, N.J., 1984.

[Shn87] Ben Shneiderman. *Designing the user interface : strategies for effective human-computer interaction.* Reading, Mass. : Addison-Wesley, 1987.

[SQL86a] American National Standards Institute, New York. *ANSI X3.135-1986, Database Language SQL*, 1986.

[SQL86b] ANSI X3.135-1986, Database Language SQL, 1986.

[SR90] Tengku M.T. Sembok and C.J. Van Rusbergen. SILOL: A simple logical-linguistic document retrieval system. *Information Processing and Management*, 26(1):111–134, 1990.

[Sri89] P. Srinivasan. Intelligent information retrieval using rough set approximatioins. *Information Processing and Management*, 25(4):347–361, 1989.

[Sri90a] P. Srinivasan. A comparison of two-poisson, inverse document frequency and discrimination value models of document representation. *Information Processing and Management*, 26(2):269–278, 1990.

[Sri90b] P. Srinivasan. On generalizing the two-poisson model. *Journal of the American Society for Information Science*, 41(1):61–66, 1990.

[Suc97] D. Suciu, editor. *Proceedings on the Workshop on Semistructured Data*, Tucson, Arizona, USA, May 1997.

[Syb94] Sybase, Inc., Emeryville, CA. *SYBASE SQL Server^{TM} Reference Manual: Volume 1. Commands, Functions and Topics*, 1994.

[Sys85] Adobe Systems. *Postscript language reference manual.* Reading, Mass. : Addison-Wesley, 1985.

[SYY75] G. Salton, C.S. Yang, and C.T. Yu. A theory of term importance in automatic text analysis. *Journal of the American Society of Information Science*, 26(1):33–44, 1975.

[TF86] S.J. Thomas and P.C. Fischer. Nested relational structures. In P.C. Kanellakis, editor, *Advances in Computing Research III, The Theory of databases*, pages 269–307. JAI Press, 1986.

[Tic85] Walter F. Tichy. Rcs – a system for version control. *Software – Practice & Experience*, 15(7):637–654, July 1985.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume Vol 1. Computer Science Press, Rockvill, MD, 1988.

[Uni93] University of Wisconsin, Madison. *Using the Exodus Storage Manager V3.1*, November 1993.

[W3C97] W3C. *Extensible Markup Language (XML) W3C Working Draft 07-Aug-97*, August 7 1997. Available on-line from http://www.w3.org/TR/WD-xml-lang.

[WW90] Thomas H. Wonnacott and Ronald J. Wonnacott. *Introductory Statistics*. John Wiley & Sons, 1990.

[Zha95] Jian Zhang. Oodb and sgml techniques in text database: An electronic dictionary system. *SIGMOD RECORD*, 24(1):3–8, March 1995.

[Zlo77] M. M. Zloof. Query by example: A database language. *IBM Systems Journal*, 16(4), 1977.

Appendix A

DSQL Language Details

This appendix gives the details of the practical query languages described in Chapter 5. The BNF representation of the complete DSQL language is first presented, and then the DTD for the DSQL language is presented along with descriptions of all the generic identifiers.

A.1 The DSQL Language BNF

This section gives a complete BNF (Backus-Naur Form) representation of the DSQL language that we are proposing. The BNF is a modified version of the one from [Dat89, pages 144-146].

$$
\begin{array}{rcll}
\textit{query-exp} & ::= & \textit{query-term} \mid \textit{query-exp}\ \text{UNION [ALL]}\ \textit{query-term} & \\
\textit{query-term} & ::= & \textit{query-spec} \mid (\textit{query-exp}) & \\
\textit{query-spec} & ::= & \text{SELECT [ALL} \mid \text{DISTINCT]}\ \textit{output}\ \textit{qry-body} & \\
\textit{output} & ::= & \textit{target} \mid \textit{outputname}(\textit{target}) \mid \textit{dtd-exp} & (1) \\
\textit{target} & ::= & \textit{scalar-exp-list} \mid\ * & (2) \\
\textit{scalar-exp-list} & ::= & \textit{scalar-exp}\ [,\textit{scalar-exp}]^* & \\
\textit{dtd-exp} & ::= & \text{DTD}\ \textit{filename} & (3) \\
\textit{qry-body} & ::= & \textit{from-clause}\ [\textit{where-clause}]\ [\textit{group-by-clause}\ [\textit{having-clause}]] & \\
\textit{from-clause} & ::= & \text{FROM}\ \textit{db-list} & \\
\textit{db-list} & ::= & \textit{db}\ [,\textit{db}]^* & \\
\textit{db} & ::= & \textit{rooted-path}\ [\textit{alias}] & (4) \\
\textit{where-clause} & ::= & \text{WHERE}\ \textit{search-cond} & \\
\textit{group-by-clause} & ::= & \text{GROUP BY}\ \textit{col-list} & \\
\textit{col-list} & ::= & \textit{col}\ [,\textit{col}]^* &
\end{array}
$$

col	::=	*complete-path*	(5)
having-clause	::=	HAVING *search-cond*	
search-cond	::=	*bool-term* | *search-cond* OR *bool-term*	
bool-term	::=	*bool-factor* | *bool-term* AND *bool-factor*	
bool-factor	::=	[NOT] *bool-primary*	
bool-primary	::=	*predicate* | (*search-cond*)	
predicate	::=	*comp-pred* | *between-pred* | *like-pred* | *testnull*	
		| *in-pred* | *univqnt* | *existqnt*	
comp-pred	::=	*scalar-exp ops* {*salar-exp* | *subquery*}	
ops	::=	$=$ | $\neq$ | $>$ | $<$ | $\geqslant$ | $\leqslant$	
between-pred	::=	*scalar-exp* [NOT] BETWEEN *scalar-exp* AND *scalar-exp*	
like-pred	::=	*col* [NOT] LIKE [*atom* | *prox-exp*]	(6)
prox-exp	::=	*atom* [NOT] *prox-ops atom*	(7)
prox-ops	::=	*NEAR* | *FBY*	(8)
testnull	::=	*col* IS [NOT] NULL	
in-pred	::=	*scalar-exp* [NOT] IN {*subquery* | *atom* [, *atom*]*}	
univqnt	::=	*scalar-exp ops* [ALL | ANY | SOME] *subquery*	
existqnt	::=	[NOT] EXISTS *subquery*	
subquery	::=	(*query-spec*)	
scalar-exp	::=	*atom* | *col* | *function*	(9)
function	::=	COUNT(*) | *distfunc* | *allfunc* | *attfunc*	(10)
distfunc	::=	{AVG | MAX | MIN | SUM | COUNT} (DISTINCT *col*)	
allfunc	::=	{AVG | MAX | MIN | SUM | COUNT} ([ALL] *scalar-exp*)	
attfunc	::=	ATTVAL(*col*, *attrib*) | {AVG | MAX | MIN | SUM | COUNT}	
		(ATTVAL(*col*, *attrib*))	(11)
path-exp	::=	*path-list* [..*path-list*]*	(12)
path-list	::=	*gi* [.*gi*]*	(13)
rooted-path	::=	*root* {. | ..} *path-exp*	(14)
complete-path	::=	*root* {. | ..} *path-exp* {. | ..} *leaf*	(15)

In the above BNF, the numbered lines are the lines that are modified from or added to the original SQL grammar. The non-terminals (*outputname, filename, alias, gi, root, leaf, attrib*) are all atomic and have not been explicitly shown. The non-terminal *outputname* is the name which will be given to the output DTD as the result of the query. The symbol *filename* is the name of the explicitly described DTD. The symbol *alias* is a variable name associated with a complex column. The symbols *gi*, *root*, *leaf* are all generic identifiers in the input database DTD. The symbol *root*refers to the root of one of the input DTDs, and *leaf* must be a data group. The *attrib* is the SGML attribute name for the GI at the leaf of the complete path with which it is associated. In addition, for comparison of path expressions, the terminating ".leaf" is omitted, as all comparisons are performed at the leaf level.

A.2 The DSQL DTD

This section presents the DTD for the extended DSQL language described above, the primary difference is the implicit handling of operator precedence using SGML tags instead of in the grammar itself.

```
<!-- *************** SQL Document Type Definition **************  -->
<!-- Suggested public id: -//ANSI X3H2//DTD SQL//EN -->

<!-- Defined Parameter Entities -->
<!ENTITY % Negation "(ASIS | NOT) ASIS"
                -- As is or negate - defaulted to as is -->
<!ENTITY % Compare  "(EQUAL | NEQ | LT | GT | LEQ | GEQ) EQUAL"
                -- All the comparison operators -->
<!ENTITY % Aggr     "AVG | MAX | MIN | SUM | COUNT"
                -- Aggregate operators for arithmetic ops on numbers -->

<!-- The top level SQL element - containing one or more
   select statements connected by union operations -->
<!ELEMENT SQL           O O     (select, (union, all? ,select)*)>

<!ELEMENT (union|all)   - O     EMPTY  -- union operation -->

<!-- The components of the select clause - the output to be generated,
and the main body of the query containing the conditions. -->
<!ELEMENT select        - O     (output, qry-body)>
<!ATTLIST select        selcrit (ALL |DISTINCT)  ALL
        -- Selection criteria - can be all or only distinct results -->
```

```
<!-- The output of the query - can be in form of a scalar expression,
     the default (all) - that combines everything that the from
     expression builds, and a dtd expression which is for now, a
     filename containing the DTD. Can also specify an optional name
     for the output(which becomes the name for the output DTD.  -->

<!ELEMENT output        O O     (scalar+ | all | dtd-exp)>
<!ATTLIST output        name    CDATA #IMPLIED>
<!ELEMENT dtd-exp       - O     EMPTY -- will possibly need change -->
<!ATTLIST dtd-exp       dtdfile CDATA #REQUIRED>

<!-- The body of the query - as in SQL, contains a from clause
 (required), and optional where, group-by and having clauses -->

<!ELEMENT qry-body      O O     (from, where?, group-exp?)>

<!-- The from clause - needs to be a rooted path. Details on building
 path expressions can be found at the end. Each rooted path can be
 given an alias for linking later from the conditions -->

<!ELEMENT from          - O     (db+)>
<!ELEMENT db            - O     (pathexp)>
<!ATTLIST db            alias   ID      #IMPLIED>

<!-- The where clause - contains the search condition expression -->
<!ELEMENT where         - O     (cond)>

<!-- The group-by clause - contains the complex columns that form the
 scalar part of the output. Group by cannot be specified if the output
 is a dtd expression. The having clause specifies further conditions
 in the groups -->

<!ELEMENT group-exp     - O     (group-by, having?)>
<!ELEMENT group-by      - O     (col+)>
<!ELEMENT having        - O     (cond)>

<!-- The search condition. The main difference between the SQL BNF and
  this method of specification is that the precedence is incorporated
  in the way the tags are structured, not possible with tagless SQL
  -->

<!ELEMENT cond          O O     (predicat | (cond, logic, cond))>
<!ATTLIST cond          neg     %Negation;>
<!ELEMENT logic         - O     EMPTY>
<!ATTLIST logic         oper    (AND | OR ) AND>
<!ELEMENT predicat      - O     (compare | between | like | testnull |
                                in | univqnt | exists)>
<!ELEMENT compare       - O     (scalar, (scalar | select))>
<!ATTLIST compare       oper    %Compare;>
```

```
<!ELEMENT between       - O     (scalar, scalar, scalar)>
<!ATTLIST between       neg     %Negation;>
<!ELEMENT like          - O     (col, (atom | prox))>
<!ATTLIST like          neg     %Negation;>
<!ELEMENT prox          - O     (atom, atom)>
<!ATTLIST prox          neg     %Negation;
                        proxop  ( NEAR | FBY) FBY>
<!ELEMENT testnull      - O     (col)>
<!ATTLIST testnull      neg     %Negation;>
<!ELEMENT in            - O     (scalar, (select | atom+))>
<!ATTLIST in            neg     %Negation;>
<!ELEMENT univqnt       - O     (scalar, select)>
<!ATTLIST univqnt       oper    %Compare;
                        type    (ALL | ANY | SOME) ALL>
<!ELEMENT exists        - O     (select)>
<!ATTLIST exists        neg     %Negation;>

<!-- The content of scalar is incomplete compared to SQL --
  -- Arithmetic operations are intentionally left out to --
  -- keep the DTD simple, but could be added if necessary -->

<!ELEMENT scalar        O O     (atom | col | function )>
<!ELEMENT atom          - O     (#PCDATA)>
<!ELEMENT function      O O     (countall | distfunc | allfunc | attval)>
<!ELEMENT countall      - O     EMPTY>
<!ELEMENT distfunc      - O     (col)>
<!ATTLIST distfunc      oper    (%Aggr;) COUNT>
<!ELEMENT allfunc       - O     (all?, scalar)>
<!ATTLIST allfunc       oper    (%Aggr;) COUNT>
<!ELEMENT attval       - O      (col,attrib)>
<!ATTLIST attval       oper     (%Aggr; | NONE) NONE>
<!ELEMENT attrib        - O     (#PCDATA)>

<!ELEMENT col           - O     (pathexp)>
<!ELEMENT pathexp       O O     (pathlist)+>
<!ATTLIST pathexp       refdb   IDREF  #CONREF>
<!ELEMENT pathlist      - O     (gi)+>
<!ELEMENT GI            - O     (#PCDATA)>
```

A.2.1 Description of the DTD Elements

Tables 12 and 13 give short descriptions of the elements in the above DTD. The essential query constructs are equivalent to the DSQL constructs. The primary difference lies in the handling of operator precedence as described above.

GI in the DTD	Description
SQL	Top level GI in the DTD. Optional. Contains one SQL query statement.
union	Signifies the union operation involving the results of two or more select statements.
all	Used as a modifier of the union operation and as a replacement for the "select *" construct of SQL
select	The root of a single select statement. A select statement can be used as a sub-query in various places: as a component in the union of multiple queries, as a scalar output in a comparison, and for quantification, either universal or existential (for all/there exists). The attribute selcrit (selection criterion) can be distinct or all depending on whether duplicate removal is to be performed or not.
output	The output from the query - specified as a list of complex columns, all, or a restructuring DTD
dtd-exp	The DTD expression - currently just the name of the DTD. Some constraints and mappings may be added in future revisions
qry-body	The actual body of the query. Optional
from	The "from" specifier - specifies the input to the query. consists of one or more databases.
db	A database to provide the input to the query. The attribute alias is a short name used for future reference by the complex columns.
where	The condition clause.
group-by	The group-by clause - specifies which columns to group the results by. Must be a subset of the columns in the output clause.
having	Specifies further restrictions in group-by columns.
cond	The conditions for querying. Can be either a predicate or a logical binary expression combined using a logical operator. Can be either true or false. The attribute Neg specifies if the condition is to be negated.
left	The left side of a logical operator. can be either a predicate or another conditional expression.
right	The right side of a logical operator.
logic	The logical operation. Allowed operations are: AND, OR, FOLLBY (followed by) and NEAR. The last two operations are added on top of normal SQL to support proximity queries.
predicat	A relational predicate - can be one of seven operations as given in the DTD. The result is always true or false. All operations can be negated.

Table 12: Description of the GIs in the SQL DTD

GI in the DTD	Description
compare	The comparison operation. Compares a scalar value with another scalar value or the result of a subquery that returns a scalar value.
between	The between operation performs range queries - decides if the value of a scalar expression is between two different scalar expression
like	The like operation is basically a regular expression match. A complex column is compared with a regular expression formed with a column atom and an escape atom.
colatom	The regular expression. In SQL, the regular expressions use the characters % and _ for zero or more characters and exactly one character respectively.
escatom	Specifies an escape character, in case one of the characters % and _ needs to be used as a data character.
testnull	A null test - to check if a complex column contains a null value
in	An IN expression: tests if the value of a scalar expression is in a set of atoms or a set returned by a select subquery
univqnt	A universal quantifier - can be one of ALL, ANY or SOME, and the comparison can be one of the several comparison operations.
exists	An existential quantifier - determines if the result of a subquery exists.
scalar	A scalar expression - can be a single value or a set of values. Can be a constant (atomic) value, a complex column, or a function of them.
atom	A constant value - normally a character or numeric value.
function	Various aggregate functions allowed in SQL.
countall	The aggregate function count(*) counts the number of tuples (or instances of complex columns) returned by the query, without duplicate elimination
distfunc	Distinct functions - computes aggregate functions on specific complex columns with duplicate elimination
allfunc	All functions - computes aggregate functions on scalar expressions without duplicate elimination
attfunc	Attribute functions - computes functions on attribute values of columns
col	A complex column - targets one or a set of GIs of the underlying database.
source	Source for the complex column - has to be one of the databases in the repository
thru	Path from the source to the target - needs to be a GI of the database
target	The end target of the column for the operation.

Table 13: Description of the GIs in the SQL DTD (continued)

Appendix B

Guide to the DocBase Source Code

This appendix briefly describes the source structure for the implementation of the query engine (see Chapter 6) as well as the visual query interface (see Chapter 7). In addition, it includes the skeleton for a parser for the SQL language described in Chapter 5.

B.1 Guide to DocBase Source Code

The source code of DocBase has two primary components: (i) a query language processing component and (ii) a visual query interface component. The full DocBase distribution also includes some sample data for testing the application. When the DocBase package is extracted, the following directories are created. Each of these directories contains a `README` file that describes the files and their use. The root level directory also contains a file named `INSTALL` that explains all the configuration options and installation instructions.

- *src.* This directory contains the source of the query engine. The top-level directory includes a top-level makefile and the virtual classes, and the directories under src includes the sub-classes.

- *scripts.* This directory contains a few useful scripts for helping with some of the configuration options. Since there are no graphical configuration setting mechanism, currently only these scripts can be used to set up configuration files (or the files could be manually edited if necessary).

- *data.* This directory contains some sample data used for testing the system, including an SGML-converted version of the "pubs2" database from the Sybase

distribution [Syb94], and the complete normalized source of this thesis.

- *SGMLQuery.* This directory contains the Java source and compiled class files for the visual interface, as well as the reference manual and other documentation.

B.2 Running DocBase

Since DocBase is primarily a research-oriented system, not much attention has been given towards portability issues. A future version will hopefully include easy compilation capabilities using GNU autoconf or imake. Currently, DocBase can be compiled by manually editing the makefiles and a few header files to specify the default parameters and positions of the directories and other similar configuration options. The file named 'INSTALL' in the top-level source directory includes details on the changes that need to be made for specific platforms. Currently, the DocBase query engine source has only been compiled on Solaris 2.5.

Once DocBase is compiled, the following steps need to be performed in order to set up one of the sample databases:

1. *Start the storage manager server.* The source code comes with the capability of using the Exodus Storage Manager or Sybase as the basic storage manager, depending on how the configuration options were selected. The appropriate server needs to be started and needs to be running to use any of the DocBase clients (other than the parsing utilities).

2. *Create the structure configuration file for the data.* If one of the sample databases is used, this step can be omitted, since the some sample configuration files are included in the sample databases. The main configuration file is the catalog configuration file, using which the indexable regions and their descriptions are specified. DocBase comes with a script called `parse_dtd.pl` that can read an SGML DTD and a simple text file containing the names of the searchable regions and their descriptions. The format of this file is as follows:

```
# Lines starting with the number sign are ignored
```

```
# Each line contains regionname/description
book/Book
chap/Chapter
l/Line
P/Paragraph
```

The above example shows a simple configuration file for a book database in which four regions are to be indexed. The parse_dtd.pl script will create a full structure configuration file from the DTD and the above file. If no sample configuration file like the above is provided, parse_dtd.pl will create a default file indexing all the GIs in the DTD, and using the GI names as their descriptions.

3. *Create the template.* The template using which the graphical query processing will be performed needs to be created next. This is usually an image representing the database. At this point, only the image needs to be created, and the coordinates for each of the regions need to be noted.

4. *Create the GUI applet configuration file.* The GUI applet configuration file is an HTML document containing a reference to the GUI applet and the parameters. An interactive applet configuration script, called `sgmconfig.pl` is available in the scripts directory that asks for the template image filename, and all the regions, and generates the HTML file.

5. *Test the configuration.* Once all the above configuration files are created, the configuration can be tested by running `src/parser/psql` and giving it an SQL query.

6. *Test the system.* The final DocBase system can be tested by bring up the HTML file containing the applet configuration information, and trying out some simple queries.

B.3 SQL Parser Implementation

In this section, we present the yacc implementation of the skeleton parser for DSQL. The source code distribution includes all the parsers described in Chapter 6. The source code for the skeleton parser given below is only included here to show the implementation of the basic parsing method.

```
%{

/*  $Id: sql.y,v 1.6 1997/11/27 02:56:07 asengupt Exp $  */

#include <stdio.h>
#include <string.h>

#define YYDEBUG 1

extern char* yytext;
extern FILE* yyin;
void yyerror(char *s);
int yylex(void);
int yyparse();

%}

%union {
  int intval;
  double floatval;
  char *strval;
}

/* symbolic tokens */

%token <strval> NAME VARREF
%token <strval> STRING
%token <intval> INTNUM
%token <floatval> APPROXNUM

/* types associate with non-terminals */
%type <strval> query_exp query_term query_spec output query_body target
%type <strval> explist exp from_clause where_clause group_clause
%type <strval> dblist db pathexp pathlist search_cond collist col predicate
%type <strval> comp_pred between_pred like_pred testnull in_pred
%type <strval> univquant existquant ops atom atomlist subquery prox_exp
%type <strval> function countfunc distfunc allfunc attfunc aggops

/* operators */

%left AND OR NOT
```

```
%left EQ_OP NEQ_OP LT_OP GT_OP LEQ_OP GEQ_OP
%left PLUS MINUS
%left STAR DIV

/* literal keyword tokens */

%token UNION ALL ANY SOME DISTINCT
%token SELECT FROM WHERE GROUPBY HAVING NOTEXISTS EXISTS
%token DTD
%token IS NULL_T
%token AVG MAX MIN SUM COUNT ATTVAL
%token DOT DOTDOT
%token BETWEEN LIKE IN_PRED NOTLIKE NOTIN
%token NEAR FBY NOTNEAR NOTFBY

%%

query_exp: query_term {}
| query_exp UNION query_term {}
| query_exp UNION ALL query_term {}
;

query_term: query_spec {}
| '(' query_exp ')' {}
;

query_spec: SELECT output query_body {}
| SELECT ALL output query_body {}
| SELECT DISTINCT output query_body {}
;

output: target {}
| NAME '(' target ')' {}
| DTD NAME {}
;

target: STAR {}
| explist {}
;
explist: exp {}
| explist ',' exp {}
;

query_body: from_clause {}
| from_clause where_clause {}
| from_clause where_clause group_clause {}
| from_clause group_clause {}
;

from_clause: FROM dblist {}
```

```
;

dblist: db {}
| dblist ',' db {}
;

db: pathexp {}
| pathexp NAME {}
| VARREF {}
| VARREF NAME {}
;

where_clause: WHERE search_cond {}
;

group_clause: GROUPBY collist {}
| GROUPBY collist HAVING search_cond {}
;

collist: col {}
| collist ',' col {}
;

col: pathexp {}
;

search_cond: search_cond AND search_cond {}
| search_cond OR search_cond {}
| NOT search_cond {}
| '(' search_cond ')' {}
| predicate {}
;

predicate: comp_pred {}
| between_pred {}
| like_pred {}
| testnull {}
| in_pred {}
| univquant {}
| existquant {}
;

comp_pred: exp ops exp {}
| exp ops subquery {}
;

ops: EQ_OP {}
| NEQ_OP {}
| LT_OP {}
| GT_OP {}
```

```
| GEQ_OP {}
| LEQ_OP {}
;

between_pred: exp BETWEEN exp AND exp {}
| exp NOT BETWEEN exp AND exp {}
;

like_pred: col LIKE atom {}
| col NOTLIKE atom {}
| col LIKE prox_exp {}
| col NOTLIKE prox_exp {}
;

prox_exp: atom NEAR atom {}
| atom NOTNEAR atom {}
| atom FBY atom {}
| atom NOTFBY atom {}
;

testnull: col IS NULL_T {}
| col IS NOT NULL_T {}
;

in_pred: exp IN_PRED subquery {}
| exp NOTIN subquery {}
| exp IN_PRED atomlist {}
| exp NOTIN atomlist {}
;

atomlist: atom {}
| atomlist ',' atom {}
;

univquant: exp ops ALL subquery {}
| exp ops ANY subquery {}
| exp ops SOME subquery {}
;

existquant: EXISTS subquery {}
| NOTEXISTS subquery {}
;

subquery: '(' query_spec ')' {}
;

exp: atom {}
| col {}
| function {}
;
```

```
function: countfunc {}
| distfunc {}
| allfunc {}
| attfunc {}
;

countfunc: COUNT '(' STAR ')' {}
| COUNT '(' col ')' {}
| COUNT '(' DISTINCT col ')' {}
;

distfunc: aggops '(' DISTINCT col ')' {}
;

allfunc: aggops '(' ALL exp ')' {}
| aggops '(' exp ')' {}
;

attfunc: COUNT '(' ATTVAL '(' col ',' NAME ')' ')' {}
| aggops '(' ATTVAL '(' col ',' NAME ')' ')' {}
| ATTVAL '(' col ',' NAME ')' {}
;

aggops: AVG {}
| MIN {}
| MAX {}
| SUM {}
;

pathexp: pathlist {}
| pathexp DOTDOT pathlist {}
;

pathlist: NAME {}
| pathlist DOT NAME {}
;

atom: STRING {}
| INTNUM {}
;

%%

void yyerror(char *s)
{
        printf("%s at %s\n", s, yytext);
}

main (int argc, char **argv)
```

```
{
  char *filename;
  if (argc > 1) {
    if (strncmp(argv[1], "-v", 2) ==0) {
      yydebug = 1;
      if (argc > 2) filename = argv[2];
    }
    else filename = argv[1];
  }
  yyin = (FILE *)fopen(filename,"r");
  if (yyparse ()) {
    fprintf(stderr,"Sorry, your SQL did not parse properly\n");
  }
  else {
    fprintf(stderr,"Parse successful!\n");
  }
}
```

Appendix C

Usability analysis questions and tables

This appendix lists the questionnaire used in our usability analysis, as well as the detailed results from the analysis which is summarized in Chapter 7.

C.1 Queries Performed by the Subjects

As described in Chapter 7, the subjects were asked to pose a set of ten queries using the target interface. Among these ten queries, the first query was primarily for the purpose of getting accustomed to the particular interface, and the next eight were the experimental queries. The last question was left to the subject to formulate. The following were the set of questions asked:

1. Find the poems written by Shakespeare.
2. How many poems were written in the Middle English Period age (MEP)?
3. Find all the poems written in the Early 19th Century period (C19A) that have the word "burning" in the first line.
4. Find the poems that have the word "hate" in the title and the word "love" in the first line.
5. Find the poems not written by "Hemans" that have the word "wreck" somewhere in a stanza.
6. Find the poems written during the Early 18th Century (C18A) which have the word "love" in the collection title, as well as in the poem title, but not in the first line.

7. Find the poems that have the phrase "expostulation and reply" anywhere in the body of the poem.

8. Find the poems written by Keats that do not have the word "mortal" in any of the stanzas.

9. Find the poems written by Shakespeare that has the phrase "to be or not to be" somewhere in the poem body.

10. Write a query of your own from your interest in poems, and indicate the number of matches you found for that query.

C.2 Detailed Usability Analysis Results

In Chapter 7, we presented a summary of the results obtained from the usability analysis on the QBT prototype. Here we show the actual data on which the ANOVA measures where performed and some visual representations of the data.

In the following, Table 14 gives the details of the times taken by every user for every task. Here we denote the Java interface by "I1" and the form interface by "I2"; and the experts by "Exp" and the novices by "Nov." Table 15 shows the detailed accuracy measures in the scale 1–5 for every user for every task, and Table 16 shows the detailed satisfaction measure in the scale 1–5 for every user.

Int. type	**Subj. Type**	**Time in Seconds for Task no.**									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
I1	Exp	80	47	51	51	45	119	51	37	48	98
I1	Exp	60	232	52	41	50	173	158	49	64	44
I1	Exp	14	43	50	62	64	121	78	96	89	31
I1	Exp	55	50	62	43	67	103	125	32	62	44
I1	Exp	45	35	50	55	66	113	85	72	68	52
I1	Nov	71	162	81	70	179	172	180	65	79	82
I1	Nov	91	170	158	382	197	219	265	178	174	165
I1	Nov	76	91	52	54	114	156	173	70	84	151
I1	Nov	50	62	91	57	82	138	248	97	103	300
I1	Nov	27	43	70	38	107	138	181	85	47	68
I2	Exp	378	77	68	33	51	71	76	50	61	55
I2	Exp	549	51	50	-	47	49	64	35	50	44
I2	Exp	86	42	31	56	30	65	49	28	58	35
I2	Exp	342	52	66	53	64	82	72	55	68	45
I2	Exp	379	98	65	66	55	82	53	42	67	55
I2	Nov	489	102	63	82	142	116	7	105	81	121
I2	Nov	614	114	52	67	138	206	44	124	82	141
I2	Nov	737	157	70	65	81	433	110	161	102	304
I2	Nov	452	152	109	85	132	285	162	145	117	119
I2	Nov	542	166	89	63	114	176	106	134	107	84

Table 14: Detailed values of the efficiency measures

Int. type	**Subj. Type**	**Scores (out of 5) for Task no.**									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
I1	Exp	5	5	5	5	5	5	2	5	4	5
I1	Exp	5	5	5	5	5	5	1	5	3	5
I1	Exp	5	5	5	5	1	5	5	5	5	5
I1	Exp	5	5	5	5	5	5	3	5	3	5
I1	Exp	5	5	5	5	5	5	5	5	5	5
I1	Nov	5	5	5	5	5	3	3	5	5	5
I1	Nov	5	5	5	5	5	5	5	5	4	5
I1	Nov	5	5	5	5	5	5	3	5	3	5
I1	Nov	5	5	5	5	5	5	5	5	5	5
I1	Nov	5	5	5	5	5	5	5	3	5	5
I2	Exp	5	5	5	5	5	5	5	5	5	5
I2	Exp	5	5	5	5	5	5	5	5	5	5
I2	Exp	5	5	5	5	1	5	3	5	5	5
I2	Exp	5	5	5	5	1	5	5	5	3	5
I2	Exp	5	5	5	5	5	5	5	5	5	5
I2	Nov	5	5	5	5	5	5	3	5	4	5
I2	Nov	5	5	5	5	5	5	5	5	5	5
I2	Nov	5	5	2	5	1	5	5	5	5	5
I2	Nov	5	5	5	5	1	5	2	5	3	5
I2	Nov	5	5	5	5	5	5	5	5	5	5

Table 15: Detailed values of the accuracy measures

Interface Type	**Subject Type**	**Satisfaction measure (out of 5)**
I1	Exp	4
I1	Exp	3
I1	Exp	5
I1	Exp	4
I1	Exp	5
I1	Nov	5
I1	Nov	5
I1	Nov	5
I1	Nov	4
I1	Nov	5
I2	Exp	5
I2	Exp	4
I2	Exp	3
I2	Exp	4
I2	Exp	4
I2	Nov	5
I2	Nov	4
I2	Nov	4
I2	Nov	3
I2	Nov	4

Table 16: Details on the Satisfaction measures

Appendix D

About this dissertation

This dissertation itself is created using SGML to demonstrate the applicability and usefulness of this document model. A modified version of the DTD used in the Electronic Thesis and Dissertation (ETD) project at Virginia Tech (http://etd.vt.edu) was used for modeling the thesis. The primary differences from the original ETD document type definition were the following:

1. Use of the standard ISO 8879:1886 special symbols and entity references.
2. Use of the standard CALS table model instead of the simple row/column model used in ETD
3. Slight modification of footnote and other referencing methods
4. Additional parameters in preformatted regions.
5. Additional tags to embed LaTeX commands and mathematical symbols.

The printed version of the dissertation was created using a perl stylesheet based on the SGMLS.pm package written by David Megginson (http://home.sprynet.com/sprynet/dmeggins/). The on-line SGML version used Panorama Pro (http://www.sq.com) stylesheets. The thesis was prepared using the "Adept^{TM}" family of products from ArborText (http://www.arbortext.com). It was also indexed and incorporated into DocBase for posing SQL queries.

Index