An algebra for structured text

Srinivasulu Erva
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Part of the Higher Education Commons

Repository Citation
https://digitalscholarship.unlv.edu/rtds/410

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

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

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

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

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

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

by

Srinivasulu Erva

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
December 1994
The thesis of Srinivasulu Erva for the degree of Master of Science in Computer Science is approved.

Chairperson, Kazem Taghva, Ph.D

Examinin Committee Member, Tom Nartker, Ph.D

Examinin Committee Member, Ajoy Kumar Datta, Ph.D

Graduate Faculty Representative, Shahram Latifi, Ph.D

Graduate Dean, Cheryl Bowels, ED.D

University of Nevada, Las Vegas
December, 1994
The Standard Generalized Markup Language (SGML) is generally used to mark the logical structure of a document. In general, the structure information obtained from SGML documents can be used by an IR system to perform structure-level retrieval. In this thesis, we present a formal model and a modified version of Abiteboul and Beeri's complex object algebra to manipulate the content and structure of SGML documents. Furthermore, we provide an extensive list of queries and their formulations to show the algebra's expressive power for manipulation of textual objects.
List of Figures

2.1 Example of Relational Schema ................................................. 16
2.2 Relational Instance Construction ............................................ 16
2.3 Example of Complex Object Schema ....................................... 17
3.1 SGML Document ........................................................................ 23
3.2 Document Instance ...................................................................... 24
4.1 Document Schema ........................................................................ 26
4.2 Text-union Operator .................................................................... 30
4.3 Text-project and Unnest Operators ........................................... 32
List of Tables

2.1 Inverted Index Example: 1 .................................................. 4
2.2 Inverted Index Example: 2 .................................................. 6
ACKNOWLEDGMENTS

At the outset, I would like to thank my advisor Dr. Kazem Taghva for his help and guidance throughout the course of this thesis and my degree program. But for his constant encouragement and guidance, completion of this thesis would have been beyond my capability. Special thanks to Dr. Tom Nartker, Dr. Ajoy Kumar Datta, and Dr. Shahram Latifi for serving on my committee. I would also like to thank Julie Borsack, Allen Condit, Jeff Gilbreth, Scott Mccord, Raveen Abhisethty and Madhavapeddi Krishna Priya for their support and help in my thesis. I dedicate this thesis to my family.
Chapter 1

INTRODUCTION

The Standard Generalized Markup Language (SGML) is generally used to identify various logical elements of a document such as title, author, sections, paragraphs and sentences. This explicit structure information in an SGML document can be used by an Information Retrieval (IR) system to retrieve logical elements of the document. The data model is a tool that describes certain applications from a logical point of view. The data model for retrieval plays an important role in building an IR system by hiding implementation details.

Information systems based on the Boolean, vector space and probabilistic models consider each document to be a collection of words and thus fail to incorporate structural information. On the other hand Database Management Systems (DBMS) are known for dealing with storage and retrieval of formatted data. Efforts have been made to integrate information retrieval systems and database management systems and extend the functionality of DBMS systems with text handling operations [10]. Such unified systems can have facilities like concurrency control and error recovery implemented once rather than duplicating it in two separate systems. Database management systems are generally based on well-known models such as the network, the hierarchical, or the relational model. These models are called record-based models because the database is structured in fixed format records of various types. The relational model is the most well-known, since it has a strong mathematical background and provides a data manipulation language which is independent of system implementation. The relational model is not suitable to represent both document structure and
content. Complex object models [1] can be used to describe the hierarchical structure of documents.

In this thesis, the model defined in [1] is adopted to manipulate the content and structure of SGML documents.
Chapter 2

INFORMATION SYSTEMS

In IR systems, input information is likely to include the natural language text of the documents or of document excerpts. The output of an information system in response to a search request consists of a set of references. These references are intended to provide the user with information about items of interest. A typical example of an IR system is the system that is used in a library to search through the catalogue. IR is useful in any discipline that requires information from printed documents. The theoretical framework for building an IR system is formalized its model. A model is best described as the logical structure of the system and its behavior. There are three well-known traditional models that are currently used in information retrieval. In the following section, each one of these models is discussed.

2.0.1 Boolean Model

The Boolean model is named primarily for its use of Boolean algebra in formulating user queries. In this model documents are represented by sets of keywords, usually stored in inverted index. The inverted index contains index terms associated with the document reference numbers in which they appear. Each document reference number uniquely identifies a document. Users express information requests in boolean queries. Boolean queries are keywords connected by the Boolean logical operators and, or and not. These operations are implemented by using set union, set intersection and set difference procedures respectively. The following example demonstrates Boolean
queries and their evaluation.

Example 1: Table 2.1 represents an inverted index for the documents whose reference numbers are 1, 2, 3, 4, 6, 7, and the key terms occurring in them are $a, b, c, and d$. The three queries $a \land b$, $a \lor c$ and $a \not\land b$ are considered in explaining Boolean retrieval.

Table 2.1: Inverted Index Example: 1

<table>
<thead>
<tr>
<th>Terms</th>
<th>Document Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>1 3 5 7</td>
</tr>
<tr>
<td>$b$</td>
<td>2 3 4 5 6</td>
</tr>
<tr>
<td>$c$</td>
<td>4 6</td>
</tr>
<tr>
<td>$d$</td>
<td>3 6 7</td>
</tr>
</tbody>
</table>

- **$a \land b$**

  A query which may be used to identify documents containing terms $a$ and $b$ can be stated as $a \land b$. The following procedure can be used to find the corresponding documents:

  1. Use the inverted index to retrieve the set of documents containing $a$; in this case the retrieval set is $\{1,3,5,7\}$.
  2. Use the inverted index to retrieve the set of documents containing $b$; in this case the retrieval set is $\{2,3,4,5,6\}$.
  3. Compute $\{1,3,5,7\} \cap \{2,3,4,5,6\}$ which is $\{3,5\}$.
  4. Retrieve the documents $\{3,5\}$.

- **$a \lor c$**

  The query $a \lor c$ refers to documents which are identified either by the term $a$ or by the term $c$. The following expression explains how this query is evaluated.
1. Use the inverted index to retrieve the set of documents containing \( a \); in this case = \{1,3,5,7\}.

2. Use the inverted index to retrieve the set of documents containing \( c \); in this case = \{4,6\}.

3. Compute \( \{1,3,5,7\} \cup \{4,6\} \) which is = \{1,3,4,5,6,7\}.

4. Retrieve the documents = \{1,3,4,5,6,7\}.

• \( a \) not \( b \)

The query \( a \) not \( b \), should retrieve documents that contain word \( a \) and do not contain word \( b \). Evaluation of this query is explained by the following:

1. Use the inverted index to retrieve the set of documents containing \( a \); in this case the retrieval set is =\{1,3,5,7\}.

2. Use the inverted index to retrieve the set of documents containing \( b \); in this case = \{2,3,4,5,6\}.

3. Compute \( \{1,3,5,7\} \setminus \{2,3,4,5,6\} \), in this case = \{1,7\}.

4. Retrieve documents \{1,7\}.

Although, the syntactic structure of a Boolean query language is quite simple, there are a few drawbacks with the Boolean model.

1. The complexity of query formulation and interpretation.

2. The standard Boolean model has no provision for ranking output documents.

3. The Boolean model gives counter-intuitive results for certain types of queries. For example consider a query of the form \( A \ and \ B \ and \ C \ and \ D \ and \ E \). A document indexed by all but one of the above terms will not be retrieved in response to this query. Intuitively, it appears that the user would like to see such a document and that it should be retrieved.
Table 2.2: Inverted Index Example: 2

<table>
<thead>
<tr>
<th>Terms</th>
<th>Document Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1 3 5 7</td>
</tr>
<tr>
<td>b</td>
<td>2 3 4 5 6</td>
</tr>
<tr>
<td>c</td>
<td>4 6 8</td>
</tr>
<tr>
<td>d</td>
<td>3 7 9 11</td>
</tr>
</tbody>
</table>

Query: a and b or c

4. The order in which operations are executed may change the query’s result. This is shown in the following example.

If the evaluation starts at the left and moves right, the set of documents retrieved will be \{3,4,5,6,8\}. If the evaluation is done from right to left the result set will be \{3,5\}.

2.0.2 Vector Space Model

In this model, documents and queries are represented as vectors. If the total textual data set has \(n\) unique key terms, then a document is represented by a vector \((t_1, t_2, t_3, \ldots, t_n)\), where \(t_i\) has a value of 1 if term \(i\) is present, and 0 if term \(i\) is absent in the document. A query can be represented in the same manner. An alternative approach would be to weigh the value of the term in its vector based on its importance to the document or query. There are a number of similarity measures that can be used in the vector space model, but the most common function is the cosine of the angle between two vectors. Document-query similarity can be computed as

\[
D_r \cdot Q_s = \sum_{i,j=1}^{t} a_{ri} a_{sj} T_i \cdot T_j
\]  

(2.1)

Where \(D_r = (a_{r1}, a_{r2}, \ldots, a_{rt})\) and \(Q_s = (a_{s1}, a_{s2}, \ldots, a_{st})\) are document and query vectors respectively. \(T_i\) and \(T_j\) are term vectors.
For this formula to be meaningful, term correlations between $T_i$ and $T_j$, must be known. Since these values are not easily generated in practice, document terms are assumed to be uncorrelated and the formula is reduced to the simple sum of products form:

$$\text{sim}(D_r, Q_s) = \sum_{i,j=1}^{t} a_{ri} q_{sj}$$

(2.2)

The most commonly employed weighting algorithm employs the tf.idf weight which uses the frequency of a term in a single document (tf or term frequency) balanced by the number of documents containing the term in the entire collection (idf or inverse document frequency):

$$w_{ij} = tf_{ij} \cdot \log \frac{N}{df_j}$$

(2.3)

where $w_{ij}$, $tf_{ij}$ are the weight and term frequency of $j^{th}$ term of document $i$ respectively. $df_j$ is the number of documents containing the term $j$ and $N$ is the total number of documents in the collection.

Other term weighting algorithms include the signal-noise ratio and the term discrimination value [7].

The vector space model resolves some of the problems of the Boolean model,

1. The query is easier for the user to formulate since it consists of a set of relevant terms. Logical operators needs to be considered.

2. Since ranking can easily be introduced into the retrieval system through term weighting, the user can be offered a set of ranked documents.

Some of its disadvantages are its assumed term independence and its arbitrary selection of weighting technique and similarity function to determine relevance.

2.0.3 Probabilistic Model

The probabilistic model rests on the premise that document to query relevance is a matter of degree. If the probability of relevance is above some threshold, then the
document is considered sufficiently similar and should be retrieved in response to the given query.

The probabilistic approach is based on two main parameters, \( \Pr(\text{rel}) \), the probability of relevance, and \( \Pr(\text{nonrel}) \), the probability of non-relevance of a record. If relevance is assumed to be a binary property, \( \Pr(\text{nonrel}) = 1 - \Pr(\text{rel}) \). In addition two cost parameters are used, designated \( a_1 \) and \( a_2 \), representing the loss associated with the retrieval of a non-relevant record and the non-retrieval of a relevant record respectively. Because the retrieval of a non-relevant record carries a loss of \( a_1[1 - \Pr(\text{rel})] \), and the rejection of a non-relevant item has an associated loss factor of \( a_2 \Pr(\text{rel}) \), the total loss caused by a given retrieval process will be minimized if an item is retrieved whenever

\[
\Pr(\text{rel}) \cdot a_2 \geq [1 - \Pr(\text{rel})] \cdot a_1
\]

(2.4)

However, before this equation can be satisfied, the probability of relevance, \( \Pr(\text{rel}) \), for a document must be found. Document relevance relies on term relevance. To determine term relevance values, not only must the occurrence characteristics for each term be calculated, the correlation between terms must be considered. Individual term occurrences can be characterized by applying a probability distribution, such as Poisson. Another method generalizes distributions found in similar document collections to characterize term frequency.

Term correlation probabilities cannot feasibly be calculated for all term subsets in a document collection of any size. Therefore, the probabilistic model simplifies these calculations by considering only some of the more important pairwise term relationships. Reduced term dependency may result in the possible exclusion of important term correlations. Further, if each term is considered independently, the probabilistic model becomes a form of the vector space model[8].

The most notable shortcoming of the probabilistic model is its difficulty in calculating representative values for term occurrences.
2.1 Database Concepts

A database is a collection of structured, interrelated data. It provides some level of data abstraction by hiding details of data storage that are not needed by most database users. The data model is the tool which provides this abstraction. A data model is a set of concepts that can be used to describe the structure of a database and to facilitate data retrieval [3]. The user can perceive a conceptual view of the data through the data model. Such systems built on these models are called database management systems. These systems support data sharing, help to perform recovery and maintain integrity.

There are three traditional data models: the network, hierarchical and relational. These models are called record-based models as the database is structured in fixed-format records of several types. Systems built on the relational model are more popular for the following reasons

- Theoretical concepts of relational model has strong mathematical background
- the relational model provides users with simple data schemas and high-level set oriented data definition and manipulation languages.
- Improves logical and physical independence of the data.

2.1.1 Relational Model

The data model proposed by E.F. Codd in 1970 is called $n$-ary relational data model. The basic idea is to consider a database as collection of tables called relations. The mathematical theory of $n$-ary relations is the foundation for this model.

Given $n$ sets $D_1, D_2, \ldots, D_n$, not necessarily distinct, a relation is a subset of the Cartesian product $D_1 \times D_2 \times \ldots \times D_n$. The $D_i$ are called the domains of the relation and $R$ is said to be degree of $n$.

A common representation of a given relation is a table where the columns represent domains and rows represent the elements of the Cartesian product. Such an element is called tuple or record.
Each column is given an attribute name to identify different columns of the table. For instance, given the domains NAME and SUBJECT, we consider the Cartesian product NAME × NAME × SUBJECT. If we want to store information advisor name, student name and the field they are working then, the relation can be defined as

\[ \text{THESIS} (\text{ADVISOR\_NAME: NAME, STUDENT\_NAME: NAME, FIELD: SUBJECT}) \]

PERSON\_NAME, STUDENT\_NAME and FIELD are called the attributes of the relation. Each attribute is associated with a given domain. Relation name, domains and attributes define the schema of the relation. Because relation is a set, all the elements are distinct. A key of a relation is defined to be the set of attributes which uniquely identify the relation tuples.

Figure 2.1 shows an example of relational database on STUDENT, INSTRUCTOR, COURSE, TRANSCRIPT relations.

The set of all values in the relations represent the contents of the database. Most query languages for systems based on the relational model are based on relational algebra.

**Relational algebra**

Relational algebra is a set of operators which can be used to define new relations from existing ones. Using these operators one may manipulate and query the database in homogeneous manner. There are five operators from which many others can be defined. If R and S are two relations then,

1. UNION: Gives the union of two relations. \( R \cup S \)
2. DIFFERENCE: \( R - S \), Gives the set difference of the two sets.
3. PROJECTION \( \pi_X(R) \): Project R on the attribute list X.
4. SELECTION \( \sigma_F(R) \): Select the tuples from R which satisfy the formula F. This formula is defined by atomic formula which may be combined by using logical
### STUDENT

<table>
<thead>
<tr>
<th>STU_NO</th>
<th>NAME</th>
<th>MAJOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>JOHN</td>
<td>CS</td>
</tr>
<tr>
<td>13</td>
<td>MIKE</td>
<td>EE</td>
</tr>
<tr>
<td>14</td>
<td>DEY</td>
<td>CE</td>
</tr>
</tbody>
</table>

### INSTRUCTOR

<table>
<thead>
<tr>
<th>INS_NO</th>
<th>NAME</th>
<th>FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>NETWORKS</td>
</tr>
<tr>
<td>4</td>
<td>CAROLL</td>
<td>DATABASES</td>
</tr>
<tr>
<td>5</td>
<td>OMAR</td>
<td>SYSTEMS</td>
</tr>
</tbody>
</table>

### COURSE

<table>
<thead>
<tr>
<th>CNO</th>
<th>TITLE</th>
<th>INS_NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>115</td>
<td>NETWORKS</td>
<td>3</td>
</tr>
<tr>
<td>351</td>
<td>LANGUAGES</td>
<td>5</td>
</tr>
<tr>
<td>431</td>
<td>DATABASES</td>
<td>4</td>
</tr>
<tr>
<td>555</td>
<td>INFORMATION SYSTEMS</td>
<td>4</td>
</tr>
</tbody>
</table>

### TRANSCRIPT

<table>
<thead>
<tr>
<th>STU_NO</th>
<th>CNO</th>
<th>SESSION</th>
<th>RES</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>115</td>
<td>90/F</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>351</td>
<td>91/S</td>
<td>A</td>
</tr>
<tr>
<td>12</td>
<td>431</td>
<td>91/S</td>
<td>A</td>
</tr>
<tr>
<td>12</td>
<td>555</td>
<td>91/F</td>
<td>B</td>
</tr>
<tr>
<td>13</td>
<td>431</td>
<td>91/S</td>
<td>A</td>
</tr>
<tr>
<td>13</td>
<td>115</td>
<td>91/F</td>
<td>A</td>
</tr>
<tr>
<td>14</td>
<td>555</td>
<td>90/F</td>
<td>B</td>
</tr>
<tr>
<td>14</td>
<td>351</td>
<td>91/F</td>
<td>B</td>
</tr>
</tbody>
</table>

Figure 2.1: Example of Relational Schema
5. JOIN $R \bowtie S$: Join is used to match records from two relations based on the values of the common attributes. Thus a new relation is constructed by taking tuples from $R$ and corresponding tuples from $S$. The correspondence is expressed by comparison between common attributes. If $R(X_1, X_2, \ldots, X_n)$ and $S(Y_1, Y_2, \ldots, Y_p)$ and if $X_i$ and $Y_j$ are attributes constructed on the same domain then

$$R \bowtie S = (R \times S) : X_i = Y_j$$

which means the join is equivalent to a Cartesian product followed by a selection, and projection to eliminate duplicate rows.

Relational algebra can be used to express queries and updates. By considering the schema for the above fig,

\[ \begin{align*}
\text{STUDENT} & : \text{(STU\_NO, NAME, MAJOR)} \\
\text{INSTRUCTOR} & : \text{(INS\_NO, NAME, FIELD)} \\
\text{COURSES} & : \text{(CNO, TITLE, INS\_NO)} \\
\text{TRANSCRIPTS} & : \text{(STU\_NO, CNO, SESESSION, RES)}
\end{align*} \]

Query 1: Numbers and name of the students whose major is CS:

$$\pi_{\text{STU\_NO, NAME}}(\sigma_{\text{MAJOR=CS}}(\text{STUDENT}))$$

Query 2: Course numbers and grades of the students:

$$\pi_{\text{CNO, RES}}(\text{STUDENT} \bowtie \text{TRANSCRIPTS})$$

Adding a row to the STUDENT relation:

$$\text{STUDENT} \cup < 15, \text{Bob}, \text{ME}>$$
Deleting a course unit from COURSES:

COURSES – < 351, Languages, 5 >

Stonebraker[10], explained how sentences and lines can be represented in the relational model and how variable length strings can be expressed in fixed length format. Document retrieval using the relational model can be seen in Macleod [4]. STAIRS is an example of an Information Retrieval Management System, which has the functionality of both IR and DBMS systems.

The relational model has become successful in many application areas such as airline seat reservations, library administration, insurance companies, banks etc.,. But, for information retrieval, the relational model is inadequate to support the structure of a document. CAD/CAM, geographical information systems, office information systems, and many other applications cannot be represented effectively in the relational model because of their deep nested structure of data items. The data types defined in the relational model are not powerful enough to capture the complex structure of advanced applications. Scheck [9] proposed a non-first normal form relational model to overcome the limitations of the relational model.

2.2 Non First Normal Form Relational Model (NF²)

In the relational model, the values of an attribute are atomic. This property of the relational model is formalized as the first normal form. By relaxing the first normal form, a relation can have sets and sets of sets as attribute values and can potentially capture the complex structures. Still the relational model is a special case of the non-first normal form of relational model. All definitions and theoretical conclusions of the relational model are still valid in this model. The relational algebra has been extended with two new operations nest and unnest, which transform first normal form relations to NF² relations [9].

Extensions to the non-first normal form model to effectively support text, vectors and matrices were proposed by Pistor[6]. Similarly, a data model is proposed for
structured text, using non-first normal form and text data [2].

A more general form of the non-first normal form model is the complex object model proposed by Abiteboul[1] which can express a non first normal relation model.

2.3 Complex object Model

This model proposes hierarchical organization of data. It differs from the classical hierarchical model by providing a data manipulation languages in the form of an algebra or calculus. In the relational model, the method by which an instance is constructed is to first apply set constructor, then tuple constructor. Figure 2.2 represents the method diagrammatically.

Figure 2.2: Relational Instance Construction

In the complex object model, set and tuple constructors are used repeatedly to create nested structure.

2.3.1 Definition of complex objects

Complex objects are built from a set of infinite atomic values $D_1, D_2, \ldots D_m$. These are called domains. Also we use an infinite set of names, called attributes. The following is an inductive definition of complex objects:
1. If \( D \) is a domain name, then \( D \) is a type. For each \( a \) in the associated domain, \( a \) is an object of this type.

2. If \( T \) is a type, and \( A \) is a name not used in it, then \( A:T \) is a named type, with name \( A \). Any instance of \( T \) is an instance of \( A:T \).

3. If \( T \) is a type, and \( A \) is an attribute not used in it, then \( \{A:T\} \) is a set type. If \( O \) is a set of objects of type \( T \), then \( O \) is an object of that type.

4. If \( T_1, ... T_n \) are types, and \( A_1, ... A_n \) are attributes not used in them, then \( [A_1:T_1, ..., A_n:T_n] \) is a tuple type. If \( O_1, ... O_n \) are objects of types \( T_1, ... T_n \) resp., then \( [A_1:O_1, ..., A_n:O_n] \) is an object of the type.

The name of a tuple component serves to denote that component, and distinguishes it from other components of the type, or the object. For that reason, the name must be different from that of any other component. The name given to the elements of a set is used in selection predicates applied to the members of the set.

These are illustrated using the following schema, and are shown in Figure 2.3.

Courses:{Course:[Course.No, Course.Name, Instructor.Name, students{Student:[Student.No, Student.Name]}]}


Courses

Course

Course_Name

Course_No

Instructor_Name

Students

Student

Student_No

Student_Name

Set Type

STR type

Tuple Type

INT type

Figure 2.3: Example of Complex Object Schema.
Chapter 3

MARKUP

Markup is the information added to indicate a document's structure. For example, tags added to explicitly indicate author's name or any other logical component of the document such as paragraphs, section, and chapters. There are two kinds of electronic markup instructions in common use.

- Specific Markup instructions.
- Generalized Markup instructions.

3.0.2 Specific Markup

The Specific markup instructions describe the format of a document by use of instructions that are specific to the program used to generate or output the text. These kind of instructions have immediate effect on the appearance of the text. They can either affect the appearance of the text, by changing the type face or affecting spatial relationships between the text.

Many word processing applications employ their own specific markup to format the documents that are printed or viewed. Given the fact that there are many word processing applications, there are different specific markup instructions associated with each word processing application. As this markup is machine specific and application specific, it is difficult for authors to change systems or transfer the documents from one system to other. Changing printing or viewing devices causes some of the codes in the markup to be redefined.
3.0.3 Generalized Markup (SGML)

The concept of a generalized markup is postulated by Goldfarb to overcome the problems posed by the specific markup systems. Generalized markup describes a document's structure rather than its physical characteristics. The ideas of Goldfarb formed the basis of IBM's Document Composition Facility Generalized Markup Language (DCF GML) [5]. With some changes in GML, the Standard Generalized Markup Language (SGML) was developed by Goldfarb. It has been accepted as the standard markup language by the International Standards Organization (ISO).

SGML recognizes logical structure and layout structure that are associated with a document. For example, a report logical structure includes: the title, author name, date of the publication and other information associated with the front matter such as section headings, sections, subsections, paragraphs and sentences in the main body, and references on the end matter of the report. The layout structure includes font information, text justification and paragraph layout like whether it is a two-columned paragraph or three-columned report etc.,. A document's logical structure can possibly have many layout structures depending on the medium it is printed or viewed on. SGML has the capability to represent document logical structure along with the intended layout structures. SGML can have external references to other content. These can be special symbols or files that are held outside document. SGML has some optional features by which markup can be minimized.

3.1 SGML document

SGML is based on rules. Nothing may be included in the document that is not declared as permitted content in a certain context. All these rules are stated in the SGML declaration and Document type Definition (DTD). There are three parts in SGML[5].

- the SGML declaration,
- the DTD,
3.1.1 Declarations in SGML

Declarations in SGML enable parser to understand the conventions used in the document. It can be omitted if the document is processed exclusively by one system[book].

The declaration of SGML document includes the following:

- Character sets: Character sets declaration specifies what character sets are used in the document, for example ISO 646, which has 256 characters and includes the ASCII characters.

- Capacity sets: Capacity sets declaration can be used to determine how much storage is required for different forms of SGML markup.

- Concrete reference syntax: This syntax specifies which characters are used for delimiters, maximum lengths of the names, starting characters of the names. Shunned characters that are prohibited as they are used as operating system control codes.

- Application specific information: Information that is should be conveyed to the application to understand the document.

3.1.2 DTD

A document type Definition (DTD) is the most important part of an SGML document. It represents the documents structure. The DTD is a set of rules that apply to a type of document. The rules can be made upon the following declarations.

- Element: The declaration starting with the key word ELEMENT specifies a tag to markup a logical component of the document and it's content model i.e, how it's content is made up of nested elements. A regular expression type notation is used to denote the relation between elements.
? means optional
+ means one or more
* means zero or more
| means or
& means and ( in any order )
, means and ( required order )
( ) means subexpression grouping

• Attribute: If an element has any attributes, they are declared using the keyword ATTLIST. This lists all the attributes along with their data types. It also includes information as to whether these attributes are required or optional.

• entity: This declaration specifies reference strings to externally held files or short forms to strings used inside the document. This is declared using the keyword ENTITY.

Figure 3.1 shows an example of a DTD. In this figure the element document has a content model that includes one title element, one or more author elements, zero or more section elements, and finally zero or more reference elements in that order. Two hyphens in the element declaration indicate that markup should not be omitted. The key word CDATA means that data value of the element or attribute is character data. The REQUIRED key word indicates that value for the attribute is required.

3.1.3 Document Instance

Document instance is the actual text entered with the markup according to the rules specified in document declaration and DTD.

An instance conforming to the above DTD is shown in the Figure 3.2

The structure shown in the above document is used in the following chapter to model a document database.
Figure 3.1: SGML Document
SGML is a markup language.

Refer to the book "Standard Generalized Markup Language" in JASIS.

Figure 3.2: Document Instance
Chapter 4

TEXT MODELING

This chapter describes a retrieval model for the structured document presented in the previous chapter a complex object algebra and new operations built from this algebra. It defines a document database schema using the complex object definitions of the previous chapter.

4.1 Schema of the database

In the following definitions, the Word_No, Sentence_No, Paragraph_No and Section_No, are the ordinal values of the Word, Sentence, Paragraph and Section.

- Atomic domains are integer and string. The types of these domains are INT and STR respectively.

- Word is a tuple type of [Word_No:INT, Lit:STR] and word is a name type.

- Sent is a tuple type of [Sentence_No:INT, Words:{Word}]

- Para is a tuple type of [Paragraph_No:INT, Sents:{Sent}]

- Sect is a tuple type of [Section_No, Section_Type:STR, Section_Heading {Word:[Word_No:INT, Lit:STR]}, Paras:{Para}]

- Author_Type is a tuple type of [Author_No:INT, Author_Name:[First:STR, Last:STR, Mid:STR]]
- **Ref** is a tuple type of \([\text{Cite}:\text{STR}, \text{Ref\_Author}:\text{Author\_Type}, \text{Ref\_Title} : \{\text{Word}\}, \text{Journal\_Name} : \{\text{Word}\}, \text{Date\_Pub} : \{\text{Word}\}]\)

- **Document** is a tuple type of \([\text{Doc\_No}:\text{INT}, \text{Title} : \{\text{Word}\}, \text{Author} : \{\text{Author\_Type}\}, \text{Sects} : \{\text{Sect}\}, \text{Refs} : \{\text{Reference}\}\]

- **Database** is a set type \(\text{Docs} : \{\text{Document}\}\)

---

**Figure 4.1: Document Schema**

When expanded this database type is going to be,

- \(\text{Docs} : \{\text{Document} : [\text{Doc\_No}:\text{INT}, \text{Title} : \{\text{Word} : [\text{Word\_No}:\text{INT}, \text{Lit} : \text{STR}]\}], \text{Authors} : \{\text{Author} : [\text{Author\_No}:\text{INT}, \text{Author\_Name} : [\text{First} : \text{STR}, \text{Last} : \text{STR}, \text{Mid} : \text{STR}]]\}], \text{Sects} : \{\text{Sect} : [\text{Section\_No}:\text{INT}, \text{Section\_Type} : \text{STR}, \text{Section\_Heading} : \{\text{Word} : [\text{Word\_No}:\text{INT}, \text{Lit} : \text{STR}]\}], \text{Paras} : \{\text{Para} : [\text{paragraph\_No} : \text{INT}, \text{Sents} : [\text{Sen} : [\text{Sentence\_No} : \text{INT}, \text{Words} : [\text{Word} : [\text{Word\_No} : \text{INT}, \text{Lit} : \text{STR}]\]]]\}], \text{Refs} : \{\text{Ref} : [\text{Cite} : \text{STR}, \text{Ref\_Authors} : \{\text{Ref\_Author} : [\text{Author\_No} : \text{INT}, \text{Author\_Name} : [\text{First} : \text{STR}, \text{Last} : \text{STR}, \text{Mid} : \text{STR}]\}], \text{Ref\_Title} : \{\text{Word} : \ldots\}\]}}\)
Figure 4.1 shows the pictorial representation of the docs type.

The complex object algebra is a functional language, based on a set of primitive operators, and on a set of combinators that produce new operations from the given operations. This algebra allows restructuring of the complex objects.

The language contains the following components:

1. Domain names, \( D_1, D_2, \ldots \)

2. Attributes.

3. Constants. (Each constant is assumed to belong to one of the domains, i.e., it is of an atomic type.)

4. Input parameters, \( R_1, R_2, \ldots \).

5. Two particular predicates: \( \in_{ST} \) (membership predicate), and \( =_{SS} \) (equality predicate) for all types \( S, T = \{S\} \).

6. Two constructors: A tuple constructor, denoted by matched brackets "[", "]," and a set constructor, denoted by matched set brackets "{", "}".

7. Filtering brackets "<", ">".

8. A set of operations, as listed below.

   (a) Set operators \( \cap \), \( \cup \), and \( \setminus \) —

   (b) \textit{Rename}

   (c) \textit{Set-collapse}

   (d) \textit{Replace}

   (e) \textit{Powerset}

   (f) \textit{Text-union}
This list includes four newly defined operations which deal with the manipulation of textual objects. They are text-union, text-project, unnest and collapse. Tuple-collapse, text-union, text-project, unnest, and selection are defined using replace operation.

4.1.1 Operations

- **Set operators:** intersection \( \cap \), union \( \cup \), and set difference \(-\): Arguments must be of the same set type, and produce a result of the same type. The resultant set and its element can be named.

- **Rename:** If \( R \) is the type of \( T \), \( A \) appears in \( T \), and \( B \) does not appear in \( T \), then \( \text{rename}_{A\rightarrow B} \) is an expression of type \( T' \) where \( T' \) is obtained from \( T \) by replacing \( A \) by \( B \). The operation does not change the value of its argument, only the name used in it.

- **Powerset:** If \( R \) is of the type \( T \), and \( B \) does not appear in \( T \), then \( \text{powerset}_B(R) \) is an expression of type \( \{B:\{A:T\}\} \). Its value, when applied to a set \( R \), is the collection of the subset of \( R \)

- **Set-collapse:** This operation is simply a union operation, where the argument is a set of sets, and the result is the union of the member sets. If \( R \) is an expression of type \( \{B:\{A:t\}\} \), then \( \text{set-collapse}(R) \) is an expression of type \( A:T \)

- **Tuple-collapse:** This operator collapses each tuple of tuples in a set into a (flat) tuple. If \( R \) is an expression of type \( A:[a, B:[b,c]] \), then \( \text{tuple-collapse}(R) \) is an expression of type \( A:[a,b,c] \)
- **Text-union** The purpose of this operation is to merge the STR domain types in to a set type. If \( R \) is the expression of type \( a[:b:INT, c:STR, d:{e:STR}] \) then

\[
text - \text{union}_{i,h}(R) = a : [b : INT, h : \{i : STR\}]
\]

\( i \) is used to name the \( STR \) type object and \( h \) is used to name the set type of the \( STR \) domain object. If \( R \) is an expression of type \( a[:b:STR, c:{d:str}] \), then

\[
text - \text{union}_{h,i}(R) = h : \{i : STR\}
\]

Pictorially the text-union operation is shown in the Figure 4.2.
The operation $\text{text}_\text{union} H, I(A)$ converts the argument $A$ shown in (a) to the type shown in (b). $H$ is the new name for the resultant set type and $I$ is the new name for the STR type.

The operation $\text{text}_\text{union} H, I(A)$ converts the argument $A$ shown in (c) to the type shown in (d). $H$ is the new name for the resultant set type and $I$ is the new name for the STR type.

Figure 4.2: Text-union Operator
• **Text-project:** This operation projects the specified structure from a given type. If \( R \) of the type \( a: \{ b: [c: \text{INT}, d: \{ e: [f: \text{INT}, g: \text{STR}] \}] \} \) then,

\[
\text{text-project} < \{ b: [d: \{ e: \{ g: \text{STR} \}] \}> (R) = a: \{ d: \{ g: \text{STR} \}] \}
\]

While projecting, a structure like \( p: \{ q: \{ s: \text{STR} \} \} \) is reduced to \( p: \{ s: \text{STR} \} \). Pictorial representation of text-projection is shown in Figure 4.3.

• **Collapse:** This is a higher level operation than set-collapse and can be defined in terms of set-collapse. It takes a set object whose nesting level is more than two and collapses it into a single set. If \( R \) is of type \( a: \{ b: \{ c: \{ d \} \} \} \) then \( \text{collapse}(R) \) is \( c \) i.e, the union of all the underlying sets \( c \).

• **Unnest:** If \( R \) is an expression of type \( a: \{ b: [c: \text{INT}, d: \{ f: \text{str} \}] \} \), then \( \text{unnest}(R) \) is of the type \( a: \{ b: [c: \text{INT}, f: \text{str}] \} \). Figure 4.3 shows a diagrammatic representation of this operation.

• **Replace:** This operation takes replace specifications for a given type and applies them to the set of that type. Restructuring and filtering of the types can be done by using the replace operation. The formal definitions of the replace specification are as follows.

\[ P(-, R_1, \ldots, R_n) \] is a replace specification which converts the input \( R \) of type \( \{ A: T \} \) to \( \{ A: T' \} \). The - denotes \( R \). \( R \) is considered to be implicit input of the replace specification. \( R_1 \) of type \( T_1 \) \( \ldots \) \( R_n \) of type \( T_n \) are considered as explicit input.

\[
\text{replace}<P>(R)
\]

Above is the syntax of the replace operation. The effect of this operation is to replace each element \( r \) of \( R \) for which \( P \) is defined by \( P(r, R_1, \ldots, R_n) \), which is of type \( T' \). That is

\[
\text{replace}<P>(R) = \{ P(r, R_1, \ldots, R_n) \mid r \in R \land P(r, R_1, \ldots, R_n) \text{ is defined} \}
\]
Project:\ A: C: ( D: F, G: H )) \rightarrow (A), \text{ where } A \text{ is of type shown in (a), resultant type going to be of figure shown in (b). Structure of } C: ( D: F, G: H ), \text{ when } F \text{ projected out } C \text{ will be reduced to } C: ( G: H ))

Unnest (A), where A is of type shown in (c) gives the result of type shown in (d.)

Figure 4.3: Text-project Operator
4.1.2 Replace Specification

The construction of the replace specification for a type is obtained by the following rules.

1. Basis: \( q, R_i, \) and \( S \) are replace specifications for \( A:T \), where \( q \) is a constant, \( R_i \) is an input parameter, and \( S \) is a constant name of \( A:T \). If \( z \) of type \( T \), and \( z_i \) of type \( T_i \) then

\[
q(z) = q; R_i(z, z_i) = z_i; S(z) = z.S
\]

2. Tuple construction: if \( P_1, \ldots, P_n \) are replace specifications from \( A:T \) to \( T_1, \ldots, T_n \) respectively., then \([A_1:P_1, \ldots, A_n:P_n]\) is a replace specification from \( A:T \) to \([A_1:T_1, \ldots, A_n:T_n]\). The resultant tuple can be named.

3. Set construction: If \( P_1, \ldots, P_n \) are replace specifications from \( A:T \) to \( T' \), then \( G_1, \ldots, G_n \) is a replace specification from \( A:T \) to \{A:T'\}. The resultant set and its element can be named.

4. Conditional: If \( P, H, K \) are replace specifications for \( A:T \), where the output type of \( K \) is \( T_k \), then if \( P \theta h \) then \( K \) is a replace specification from \( A:T \) to \( T_k \). The predicate \( \theta \) is one of the two predicates =, \( \in \) and the types of the arguments should be compatible.

5. Application of an operation (except for replace): If \( P_1, \ldots, P_n \) are replace specifications from \( A:T \) to types \( T_{P_1}, \ldots, T_{P_n} \), respectively, and \( op(R_1, \ldots, R_n) \) is an algebraic operation of type \( T_{P_1}, \ldots, T_{P_n} \) to it \( T' \), then \( P_1, \ldots, P_n \) is a replace specification from \( A:T \) to \( T' \).

6. Application of replace: If \( P_1 \) is a replace specification from type \( A':T' \) to type \( A':T'' \), and \( P_2 \) is a replace specification from type \( A:T \) to type \( A:\{A':T'\} \), then \( replace < P_1 >(P_2) \) is a replace specification from A type \( A:T \) to type \( A:\{A':T''\} \).

7. The empty tuple [], is a replace specification for any type.
\[ [](x) = \{[]\} \]

8. The result of replace< [] >(R), when R is assigned a set R is \{[]\}, when R is nonempty, and \{\} when R is empty. Thus it is a predicate for testing set emptiness.

### 4.2 Query Construction

A query is a function whose input and output are of set type. The construction of queries, in this model, are governed by following rules.

1. **Basis** For every constant \( q \), \( \{q\} \) is a query.
   
   If \( R \) is a set type then \( R \) is a query.

2. **Application of an operation** (except for replace): If \( Q_1, \ldots, Q_n \) are queries, with set output types \( T_{Q_1}, \ldots, T_{Q_n} \) respectively, and \( op \) is an algebraic operation of type \( T_{Q_1} \rightarrow \cdots \rightarrow T_{Q_n} \) to set type \( T' \), then \( op(Q_1, \ldots, Q_n) \) is a query, with output type \( T' \).

3. **Application of replace**: If \( Q \) is a query, when \( R \) is of type \( \{A:T\} \) and if \( P \) is a replace specification for \( A:T \) such that all input parameters appearing in \( P \) are of set types, then \( \text{replace } < P > (Q) \) is a query.

### 4.2.1 Properties of queries and replace specification

These properties can be used in constructing queries.

1. For every query \( Q \), and for every type \( A:T \), \( Q \) is a replace specification for \( A:T \).

2. The set of replace specifications, for any given type, is closed under composition.
   
   That is, if \( P(R, R_1, \ldots, R_n) \) and \( P_1(R,S_1,\ldots, S_n) \) are replace specifications for the type \( A:T \), and the output type of \( P_1 \) is the same as the type of \( R_1 \), then \( P(R, P_1(R, S_1, \ldots, S_n), R_2, \ldots, R_n) \) is also replace specification for \( A:T \).
3. The set of queries is closed under composition. That is, if \( Q(R_1, \ldots, R_n) \) and \( Q_1(S_1, \ldots, S_m) \) are queries, and the output of the type \( Q_1 \) is same as the type \( R_1 \), then \( Q(Q_1(S_1, \ldots, S_m), R_2, \ldots, R_m) \) is also a query.

4. If \( \text{replace} \ <P(R_1, R_2, \ldots, R_n)> \) is a query, and \( Q, Q_1, \ldots, Q_n \) are queries, with output types corresponding to the types of \( R, R_1, \ldots, R_n \), then \( \text{replace} <P(R, Q_1, \ldots, Q_n)> (Q) \) is also a query.

5. If \( \text{replace} <P(R_1, R_2, \ldots, R_n)> (R) \) is a replace specification, where the type of \( R \) is \( \{A:T\} \), and if \( P_1, \ldots, P_n \) are replace specifications for \( A:T \), and \( P' \) is a replace specification from \( B:T' \) to \( \{A:T\} \), then \( \text{replace} <P(R, P_1, \ldots, P_n)> (G') \) is also a replace specification for \( B:T' \).

6. For any query \( R \), \( \{R\} \) is also a query.

7. Let \( P \) be a replace specification for \( A:T \), and let \( A':T' \) be a (tuple) type such that \( A \) is a constant name of \( T' \), and \( T'.A \) is of type \( T \). Then \( P \) is also replace specification for \( A':T' \).

Proofs for the above properties are given in [1]. The following section lists sample queries possibly asked in a document.

### 4.3 Queries

1. Find all the documents that contain words \( a, b, c \).

2. Find all the sentences that have words \( a, b \) in them.

3. Find out all the paragraphs written by author whose first name is \( a \) and last name is \( b \) and contain words \( c \) or \( d \) and do not contain word \( e \).

### 4.4 Explaining the queries in terms of the above specified operations

Find all documents that contain words \( a, b, c \).

Applying text_project to type Docs.
The resultant type is going to be

Docs:{Doc: [Doc_No:INT, Title:{Lit:STR}, Authors:{Author:[Author_Name:[First:STR, Last:STR, Mid:STR]]}, Sects:{Sect:[Section_Heading:{Lit:STR}, Paras:{Para:{Sents:{Sen:{Words:[]}}}}]}}, Refs:{Ref:[Cite:STR, Ref_Author:[Author_Name:[First:STR, Last:STR, Mid:STR]], Ref_Title:{Lit:STR}, Journal_Name:{Lit:STR}, Date_Pub:{Lit:STR}]}]

Applying tuple-collapse on Author, Ref_Author and collapse on Title, Section_Heading, Paras, the result is going to be,

Docs:{Doc: [Doc_No:INT, Title:{Lit:STR}, Authors:{Temp_Author:{Lit:STR}}, Sects:{Temp_Sect:{Lit:STR}}, Refs:{Ref:[Cite:STR, Temp_Ref_Author:{Lit:STR}, Ref_Title:{Lit:STR}, Journal_Name:{Lit:STR}, Date_Pub:{Lit:STR}]}]}

Applying text-union on Author giving resultant set name Temp_Author and str type name Lit, Sect giving set name Temp_Sect and str type name Lit and Ref_Author giving resultant set name Temp_Ref_Author and str type name Lit,

Docs:{Doc: [Doc_No:INT, Title:{Lit:STR}, Authors:{Temp_Author:{Lit:STR}}, Sects:{Temp_Sect:{Lit:STR}}, Refs:{Ref:[Cite:STR, Temp_Ref_Author:{Lit:STR}, Ref_Title:{Lit:STR}, Journal_Name:{Lit:STR}, Date_Pub:{Lit:STR}]}]}
Applying collapse on Authors and Sects

Docs: {Doc: [Doc_no: INT, Title: {Lit: STR}, Authors: {Lit: STR},
Sects: {Lit: STR}, Refs: [Ref: {Cite: STR, Ref.Author: {Lit: STR},
  Ref.Title: {Lit: STR}, Journal.Name: {Lit: STR}, Date_Pub: {
    Lit: STR}}]}]

Applying text-union on Ref giving a name Lit for naming str type and Temp_Ref for output set type name.

Docs: {Doc: [Doc_No: INT, Title: {Lit: STR}, Authors: {Lit: STR}, Sects: {
  Lit: STR}, Refs: {Temp_Ref: {Lit: STR}}]}

Applying set-collapse on Refs

Docs: {Doc: [Doc_No: INT, Title: {Lit: STR}, Authors: {Lit: STR}],
Sects: {Lit: STR}, Refs: {Lit: STR}}

Applying text-union on Doc and giving the name Content as output set type name

Docs: {Doc: [Doc_No: INT, Content: {Lit: STR}]}

Selecting those documents which have a given set of words in them.

Select< Content: {Lit: a, Lit: b, Lit: c} ∈ powerset₄(Content) > (Docs).

The above query will produce all the documents that contain reference words.

Find all the sentences that contain words a, b in them.

Text-project<{Doc: [Doc.No: INT, Sects: {Sect: [Section.No: INT, Paras: {
    Lit: STR}]}]}]}]}] > (docs)

The resultant type going to be
Applying unnest on Sects

Docs:{Doc: [Doc_No:INT, Sects:{Sect:[Section_No:INT, Paras:{Para:[Paragraph_No:INT, Sents:{Sen:[Sentence_No:INT, Words:{Lit:STR}]})}]})}}

Applying unnest on Docs

Docs:{Doc: [Doc_No:INT, Section_No:INT, Paragraph_No:INT, Sentence_No:INT, Words:{Lit:STR}]}

Selecting those sentences which have given words.

Select < Words:{Lit:a, Lit:b} \subseteq \text{powerset}(\text{Words}) > (Docs).

Find all paragraphs written by author whose first name is a and last name is b which contain words c or d and do not contain word e

Applying text-project on type Docs.


The resultant type is going to be

Applying tuple-collapse on Author and set-collapse on Sents then, the resultant type is going to be,

\[
\]

Selecting only documents whose Author's first name is \(a\)

\[
\text{Select } <\text{replace}<\{\text{[]}\}> (\text{replace}<\text{if First = a then a }>(\text{Authors})) = [\text{[]}(\text{Authors}) > (\text{Docs})].
\]

Selecting only documents whose Author's last name is \(b\)

\[
\text{Select } <\text{replace}<\{\text{[]}\}> (\text{replace}<\text{If Last = b, then b }>(\text{Authors})) = [\text{[]}(\text{Authors}) > (\text{Docs})].
\]

Resultant type is going to be a document whose Last Name of the author is \(b\) and first name of the author is \(a\)

\[
\]

Projecting out Authors, using text-project

\[
\]

The resulting structure going to be

\[
Applying unnest on Sects, resulting type going to be

\[
\]

Applying unnest on Docs

\[
\]

Selecting paragraphs which contain either \(c\) or \(d\).

\[
\text{Select} < \text{replace} < \emptyset > (\text{replace} < \{h\{p, q\}, h:\{p\}, h:\{q\}\} \cap (\text{powerset}_p (\text{powerset}_h (\text{Sents}))) = \emptyset (\text{Sents}) > (\text{Docs})
\]

The resulting structure will produce all the paragraphs that contain either word \(c\) or \(d\).

\[
\]

In order to select sentences which do not contain word \(e\),

\[
\text{Select} < e \notin \text{Sents} > (\text{Docs})
\]

The resulting structure give all the paragraphs written by author whose last name is \(b\), first name is \(a\) and contain words either \(c\) or \(d\) and do not contain word \(e\)

4.5 Construction of text-project, text-union, unnest, select and collapse

In this section, we will show the construction of the new operators in terms of the operations given.
4.5.1 **Text-project**

The construction of *text-project* is explained using the following example

\[
\text{Text-project} < \text{Docs}: \{ \text{Doc} : \{ \text{DocNo} : \text{INT}, \text{Sects}: \{ \text{Sect} : \{ \text{SectionType} : \text{STR}, \text{SectionHeading} : \{ \text{Word} : \{ \text{Lit} : \text{STR} \} \}, \text{Paragraphs}: \{ \text{Para} : \{ \text{Sents}: \{ \text{Sent} : \{ \text{Words} : \{ \text{Lit} : \text{STR} \} \} \} \} \} \} \} > (\text{Docs}) = \text{Docs}: \{ \text{Doc} : \{ \text{DocNo} : \text{INT}, \text{Sects}: \{ \text{Sect} : \{ \text{SectionType} : \text{STR}, \text{SectionHeading} : \{ \text{Lit} : \text{STR} \}, \text{Paragraphs}: \{ \text{Sents}: \{ \text{Words} : \{ \text{Lit} : \text{STR} \} \} \} \} \} \}
\]

where docs is the following

\[
\text{Docs}: \{ \text{DocNo} : \text{INT}, \text{Sects}: \{ \text{SectNo} : \text{INT}, \text{SectionType} : \text{STR}, \text{SectionHeading} : \{ \text{Word} : \{ \text{WordNo} : \text{INT}, \text{Lit} : \text{STR} \} \}, \text{Paragraphs}: \{ \text{ParaNo} : \text{INT}, \text{Sents}: \{ \text{SentenceNo} : \text{INT}, \text{Sen}: \{ \text{Word} : \{ \text{WordNo} : \text{INT}, \text{Lit} : \text{STR} \} \} \} \} \} \}
\]

The effect of the above text-project operation is to remove all integer types except for the DocNo from docs. The construction of this *text-project* as follows

These replace specifications are constructed from bottom up.

\[ P_1 = \text{Lit} \]

\[ P_1 \] is a replace specification whose implicit input is \text{Lit:STR}

The resultant type is going to be \text{Lit}

\[ \text{Lit} \] is also replace specification for implicit input \text{Word}.

\[ P_2 = \text{Lit} \text{ where implicit input is Word: [WordNo: INT, Lit: STR] } \]

The resultant type is going to be \text{Lit}

\[ P_3 = \text{replace< Lit > (Sen)} \]

is a replace specification, implicit input is \text{Sen}. The resultant type is going to be \text{Sen: [Lit]}

\[ P_3 \] is a query which takes set input and gives set output so it can be replace specification for \text{Sent}

\[ P_4 = \text{replace< Lit > (Sen)} \] is a replace specification for the type \text{Sent.}
$P_6 = \text{replace} < \text{replace} < \text{Lit} > (\text{Sen}) > (\text{Sents})$ is a replace specification, implicit input is $\text{Sents}$. The resultant type is going to be $\text{Sent} : \{\text{Sen} : \{\text{Lit}\}\}$

$P_6 = \text{replace} < \text{replace} < \text{Lit} > (\text{Sen}) > (\text{Sents})$

$P_6$ is query, so it will be replace specification for $\text{Para}$

$P_6 = \text{replace} < \text{replace} < \text{replace} < \text{Lit} > (\text{Sen}) > (\text{Sents}) > (\text{Paragraphs})$

is a replace specification, implicit input is $\text{Paragraphs}$. The resultant type is going to be $\text{Paragraphs} : \{\text{Sent} : \{\text{Sen} : \{\text{Lit}\}\}\}$

$P_6$ is a query, so it is a replace specification for $\text{Sect}$

$P_7 = \text{replace} < \text{replace} < \text{replace} < \text{Lit} > (\text{Sen}) > (\text{Sents}) > (\text{Paragraphs})$

$P_7$ is a replace specification for $\text{Sect}$

$P_8 = \text{Lit}$ is a replace specification whose implicit input is $\text{Lit} : \text{STR}$

$P_9 = \text{Lit}$ Where implicit input is $\text{Word} : [\text{Word.No} : \text{INT}, \text{Lit} : \text{STR}]$ is also replace specification.

$P_{10} = \text{replace} < \text{Lit} > (\text{Section.Heading})$

is a replace specification the resultant type is going to be $\text{Section.Heading} : \{\text{Lit}\}$

$P_{10}$ is a query So it is also replace specification for $\text{Sect}$

$P_{11} = \text{Section.type}$ is a replace specification for the type $\text{Sect}$.

$P_7, P_{10}, P_{11}$ are replace specification for $\text{Sect}$, then

$P_{12} = [\text{Section.type} : \text{STR}, \text{replace} < \text{Lit} > (\text{Section.Heading}), \text{replace} < \text{replace} < \text{replace} < \text{Lit} > (\text{Sen}) > (\text{Sents}) > (\text{Paragraphs})]$

Using tuple construction. Implicit input is $\text{Sect}$.

$P_{13} = \text{replace} < ([\text{Section.type} : \text{STR}, \text{replace} < \text{Lit} > (\text{Section.Heading}), \text{replace} < \text{replace} < \text{replace} < \text{Lit} > (\text{Sen}) > (\text{Sents}) > (\text{Paragraphs})] > (\text{Sects})$

is a replace specification for $\text{Sects}$

$P_{13}$ is a query which is a replace specification for $\text{Doc}$ and $P_{14} = \text{Doc.No}$ is a replace specification for $\text{Doc}$.

$P_{15} = [\text{Doc.No} : \text{INT}, \text{replace} < ([\text{Section.type} : \text{STR}, \text{replace} < \text{Lit} > (\text{Section.Heading}), \text{replace} < \text{replace} < \text{replace} < \text{Lit} > (\text{Sen}) > (\text{Sents}) > (\text{Paragraphs})] > (\text{Sects})$
Paragraphs]) > (Sects])

is a replace specification for Doc by tuple construction.

\[ P_{15} = \text{replace}<([\text{Doc.No}:\text{INT}, \text{replace}<([\text{Section.type}:\text{STR}, \text{replace}<\text{Lit}>(\text{Section.Heading}), \text{replace}<\text{replace}<\text{replace}<\text{Lit}>(\text{Paragraphs})](\text{Sents}))>(\text{Sects}))>(\text{Docs})] \]

Using set pull up, the resultant type is

Docs:{ Doc:[Doc.No:INT, Paras:{Sents:{Sent:{Lit}}}]} }

### 4.5.2 Text-Union

Constructing the Text-Union on the following structure


\[ P_1 = \text{Title} \] is a replace specification for the implicit input Doc.

So are

\[ P_2 = \text{Authors} , P_3 = \text{Sects}, P_4 = \text{Refs}, P_5 = \text{Doc.No}, P_6 = \text{Doc.Tag} \]

Using set construction on Doc.Tag and naming the resultant set type to Temp.Tag and renaming Doc.Tag to Lit,

\[ P_7 = \text{Temp.Tag}:{\text{Lit}} \] is also a replace specification for Doc

Applying union operation on Authors, Sects, Refs and Temp.Tag and giving resultant set type name Temp.Words

\[ P_8 = \text{U(Authors, Sects, Refs,Temp.Tag)} \] is a replace specification for Doc

\[ P_9 = [\text{Doc.No}, \text{U( Authors, Sects, Refs,Temp.Tag)}] \] is a replace specification for Doc using tuple construction

This is also replace specification for Docs using replace.

\[ P_{10} = \text{replace}<[\text{Doc.No}, \text{U( Authors, Sects, Refs,Temp.Tag)}]>(\text{Docs}) \]

The resultant type is going to be

Unnesting the following structure


Constructing replace specifications bottom up

$P_1 = \text{Lit}$ is a replace specification for $\text{Lit}$.

$P_2 = R_{\text{Sentence}_\text{No}}$

is a replace specification for $\text{Words}$ since $R_{\text{Sentence}_\text{No}}$ is an explicit input parameter.

$P_3 = [R_{\text{Sentence}_\text{No}}, \text{Lit}]$ is also replace specification for $\text{Lit}$.

$P_4 = \text{replace}<[R_{\text{Sentence}_\text{No}}, \text{Lit}]>(\text{Words})$

the above is also replace specification for $\text{Sen}$, $\text{Sentence}_\text{No}$ is a replace specification for $\text{sen}$ and whose output type same as $R_{\text{Sentence}_\text{No}}$, so,

$P_5 = \text{replace}<[\text{Sentence}_\text{No}, \text{Lit}]>(\text{Words})$

$P_6 = \text{replace}<\text{replace}<[\text{Sentence}_\text{No}, \text{Lit}]>(\text{Words})>(\text{Sents})$

$P_7 = \text{setcollapse}(\text{replace}<\text{replace}<[\text{Sentence}_\text{No}, \text{Lit}]>(\text{Words})>(\text{Sents})$

The resultant type is going to be

Sents: {[[\text{Sentence}_\text{No}, \text{Lit}]]}

When passed $\text{Paragraph}_\text{No}$ as a parameter,

$P_8 = \text{replace}<[R_{\text{Paragraph}_\text{No}}, \text{Sentence}_\text{No}, \text{Lit}]>\text{setcollapse}(\text{replace}<\text{replace}<[\text{Sentence}_\text{No}, \text{Lit}]>(\text{Words})>(\text{Sents})$

is also a replace specification for $\text{Sents}$

This is also replace specification for $\text{Para}$, substituting $\text{Paragraph}_\text{No}$ for $R_{\text{Paragraph}_\text{No}}$

$P_9 = \text{replace}<[\text{Paragraph}_\text{No}, \text{Sentence}_\text{No}, \text{Lit}]>(\text{set-collapse}(\text{replace}<\text{replace}<[\text{Sentence}_\text{No}, \text{Lit}]>(\text{Words})>(\text{Sents}))$

$P_{10} = \text{replace}<\text{replace}<[\text{Paragraph}_\text{No}, \text{Sentence}_\text{No}, \text{Lit}]>(\text{set-collapse}(\text{replace}<\text{replace}<[\text{Sentence}_\text{No}, \text{Lit}]>(\text{Words})>(\text{Sents})>)>(\text{Paragraphs})$
4.5.4 Select

If \( R \) is of \( S: \{ B: \{ C, D: \{ E \} \} \} \), then selecting from the \( S \), the tuples where the first component is a member of the second component. The expression for this is 
\[
\text{replace< if } C \in D \text{ then } B > (S).
\]
The above can be described as using traditional select 
\[
\text{select<} C \in D > (S)
\]

4.5.5 Collapse

If \( R \) is of type \( A: \{ B: \{ C: \{ D: \text{STR} \} \} \} \) then 
\[
\text{collapse}(R) = C: \{ D: \text{STR} \} \text{ is equivalent to set-collapse(set-collapse}(R) = C: \{ D: \text{STR} \}
\]
Chapter 5

CONCLUSIONS

IR systems that are capable of retrieving logical structural elements of a document such as title, author, and sections, are more useful. Traditional IR systems can only retrieve a document in its entirety. This is due to their simple conceptual models which consider a document as only a collection of words. In the past few years, ISO standards for document description have been developed. These standards view documents as hierarchical objects. SGML is one of those standards that is used to mark up the logical structure of a document. Structure information available in a SGML document can be used to build an IR system capable of performing structure-level retrieval. In this thesis topic, I have proposed a conceptual model for structured documents using complex objects. With the ability to represent nested hierarchical structure, complex objects are natural candidates for representing structure described by a SGML document. Complex object algebra with replace specifications approach has the required ability to restructure and filter textual objects. A natural way of manipulating nested objects is achieved by replace operator. In the context of text, the higher level operators text-project, text-union and the other useful operators are useful are constructed using the existing algebra. Typical content and structure based queries are explained in this model. Further research includes adding the functionality of retrieval-based ranking to this model.
Bibliography


