A Structured Text ADT for Object-Relational Databases

L.J. Brown, M.P. Consens, I.J. Davis, C.R. Palmer, and F.W. Tompa

Centre for the New OED and Text Research,
Department of Computer Science,
University of Waterloo,
Waterloo, Ontario,
Canada N2L 3G1

ABSTRACT

There is a growing need to develop tools that are able to retrieve relevant textual information rapidly, to present textual information in a meaningful way, and to integrate textual information with related data retrieved from other sources. These tools are critical to support applications within corporate intranets and across the rapidly evolving World Wide Web.

This paper introduces a framework for modelling structured text and presents a small set of operations that may be applied against such models. Using these operations structured text may be selected, marked, fragmented, and transformed into relations for use in relational and object-oriented database systems.

The extended functionality has been accepted for inclusion within the SQL/MM standard, and a prototype database engine that supports SQL with extensions to incorporate the proposed text operations has been implemented. This prototype serves as a proof of concept intended to address industrial concerns, and it demonstrates the power of the proposed abstract data type for structured text.

1. Introduction

The application of database technology is essential to the operation of conventional business enterprise, and it is becoming increasingly important in the development of distributed information systems. However, most database technology, and in particular relational database technology, provides few facilities for effectively managing the vast body of electronic information embedded within text.

Recognizing that text retrieval is central to effective information systems, Fulcrum Technologies Inc., Grafnetix Systems Inc., InContext Corporation (now EveryWare Development Inc.), Megalith Technologies Inc., Open Text Corporation, Public Sector Systems, and SoftQuad Inc., and the University of Waterloo formed the Canadian Strategic Software Consortium (CSSC) in 1993. CSSC's goal was to pursue pre-competitive research relating to the integration of relational databases and text-intensive databases [CSSC94]. Commercial realities dictated that we explore how relational database systems could be extended, so that they might most effectively provide access to structured text in a manner compatible with SQL [Bla94, Bla95].

As well as pursuing research in language design and support, we were also interested in how federated database systems might be constructed on top of existing database and text searching systems, such as Oracle, IMS, DB2/6000, and PAT [Cob92, Zhu92]. We therefore elected to build a prototype hybrid query processor capable of integrating relational data (managed by relational database systems) and text (managed by text engines) [Bri97].

Early versions of our SQL2 extensions have been presented with examples [Bla94, Bla95]. The semantics of these extensions have since been refined and formally defined [Dav96], and the extensions have subsequently been adopted for inclusion within the SQL/MM standard [ISO96s].

In this paper we describe the final text modelling framework that we developed to meet the varied uses of structured text in a database environment. Section 2 highlights the diverse nature of text and briefly reflects on the state-of-the-art. The proposed structured text abstract data type is presented in Section 3. Section 4 describes a set of related applications that illustrate the utility of such structured text objects, and conclusions follow in Section 5.

2. The challenge

A structured text is by definition any text that has an identifiable internal structure. This structure may be explicitly established by the inclusion of appropriate electronic markup [Coo87, Tom89], possibly complemented by an external document type definition (DTD), or it may be implied by the language contained within this text. HTML is an example of a text that contains explicit structural markup used in association with a DTD [Rag97], while Java source code is an example of text whose structure is determined by appropriately parsing the language contained within the text [Fla96]. Elsewhere such data has been termed “semi-structured” to distinguish it from rigidly structured business data as found in relational databases [Suc97].

Many customers required that large texts be searched both vertically with respect to their internal structure and horizontally with respect to their textual content [Wei85]. Texts often need to be fragmented at appropriate structural boundaries. Sometimes selected text needs to be extracted as separate units, but often the appropriate context surrounding selected text must be recovered, and thus the selected text needs to be marked in some manner, so that it can subsequently be located easily within a potentially much larger context [ATA91].

In the current SQL/MM standard [ISO96s], there is no ability to use structured text concepts or marking within a Full Text search specification. At present the Full Text specification uses the concepts of character, word, sentence and paragraph within its own search language, without defining or explaining how such concepts relate to the actual material contained within an arbitrary instance of Full Text. This problem could be resolved by viewing these concepts as specific instances of well-defined structure associated with the text being searched and augmenting the SQL/MM Full Text specification so that it allows marking of identified substrings matching Full Text patterns and searching that incorporates structured text concepts. Ideally, full interplay should be allowed between “horizontal” and “hierarchical” text searching and marking,

thus making the resulting language much more expressive. This would also allow the concept of proximity, which is well defined within the Full Text proposal, to be equally effective within our structured text proposals.

Developing a text modelling framework capable of representing a vast variety of properties that any possible application could demand of structured text would probably not be particularly constructive, even if it were possible. What is initially required is a simple text framework that encapsulates most of the significant properties of structured text, coupled with well-defined operations on texts that facilitate effective query, retrieval and update of selected fragments of structured text [Ray96a].

3. Structured text objects

In this section, we describe a framework for tree models for structured text, and we describe operations on such models that provide the functionality needed to select and extract subtexts in a relational database environment. The operators we chose to define the structured text ADT were designed to be compatible with SQL2 and SQL/MM, and they have been influenced by SQL's syntax and semantics. Nevertheless, because the resulting ADT was designed to meet the demands of our industrial partners familiar with commercial applications involving structured text, it forms a useful type definition for any object-oriented or object-relational database environment.

3.1 A framework for tree models of structured text

A structured text subsumes a region of text that may itself contain well-formed subordinate structured texts, such as a chapter containing paragraphs, footnotes, figures, and subchapters. It is assumed that a structured text is finite and that an arbitrary ordering of text may be associated with unordered fragments of text. For example, SGML attributes are considered to be logically unordered [ISO86], and subtexts drawn from a collection of works contained within a single text may have no logical ordering; nevertheless the text is arbitrarily ordered in its presentation. Using these assumptions a structured text can be conceptually represented as an ordered tree having nodes that correspond to the various structures in the text [Mac92, Sal96]. Each node in this tree is labelled with a string that identifies the structure that the node represents conceptually, and each node contains as a second attribute the subtext subsumed by this structure.

To interoperate with texts, we introduce the first two functions of the text ADT:

Function	*Arguments*	*Returns*	*Description*
string_to_text	String , Method	Text	Parses a string to form a text
text_to_string	Text , Method	String	Forms a string from a text

Figure 1. Text creation and interpretation

The function **string_to_text** takes as input two arguments. The first is a string containing the sequence of characters to be parsed and the second is a keyword (passed as a string in SQL) identifying how this text is to be parsed (*i.e.*, which parser and which grammar to apply). If the input string is successfully parsed, the function returns the corresponding instance of structured text, conforming to the model used by the parser.

Two texts that are using identical arguments (*i.e.*, equal input strings and identical methods) are said to share the same *provenance*.

Complementing this, the function **text_to_string** produces a string from a text. A choice of conversion methods is provided, since text can be linearized and presented in many ways. For example, one converter may produce a tagged string, a second might omit all tags, and a third might suppress particular subtexts.

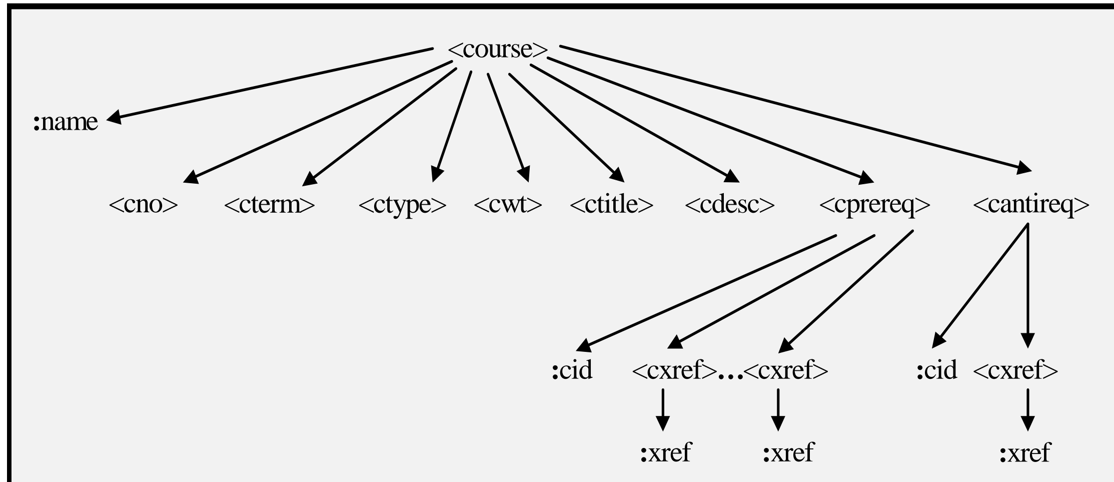
As an example, consider the fragment of the University of Waterloo calendar [UW96] shown in Figure 2. This fragment could be encoded as a tagged string as in Figure 3(a) and interpreted by a parser as the labelled tree shown in Figure 3(b), where the texts subsumed by each node within the tree are not shown. The text corresponding to Figure 3(b) may have been produced by applying the **string_to_text** function to the string in Figure 3(a) using the keyword "calendar" to identify SGML parsing with a DTD appropriate for the calendar.

CS 370 F,W 3C 0.5
Numerical Computation
Principles and practices of basic numerical computation as a key aspect of scientific computation. Visualization of results. Approximation by splines, fast Fourier transforms, solution of linear and nonlinear equations, differential equations, floating point number systems, error, stability. Presented in the context of specific applications to image processing, analysis of data, scientific modeling.
Prereq: MATH 235, 237 and one of CS 230, 246
Antireq: CS 337

Figure 2. Part of Chapter 16 of the University of Waterloo calendar

```
<COURSE NAME="CS370"><CNO>CS 370</CNO><CTERM>F,W</CTERM><CTYPE>3C
</CTYPE> <CWT>0.5</CWT><br><CTITLE>Numerical Computation</CTITLE><br><CDESC>
Principles and practices of basic numerical computation as a key aspect of scientific computation.
Visualization of results. Approximation by splines, fast Fourier transforms, solution of linear and
nonlinear equations, differential equations, floating point number systems, error, stability. Presented in
the context of specific applications to image processing, analysis of data, scientific modeling.</CDESC>
<br><CPREREQ CID=478><cxref xref="MATH235">MATH 235</cxref>, <cxref
xref="MATH237"> 237</cxref> and one of <cxref xref= "CS230">CS 230</cxref>, <cxref
xref="CS246">246</cxref> </CPREREQ><br><CANTIREQ CID=479> <cxref xref="CS337">CS
337</cxref></CANTIREQ> <br></COURSE><p>
```

(a) Tagged encoding as a character string



(b) Schematic representation in the model

Figure 3. An encoding for a fragment of the calendar

In this sample model, labels in the tree are "typed" as being SGML generic identifiers [ISO86] by the convention of using enclosing angle brackets, whereas attribute names are preceded by a colon. A node representing a generic identifier subsumes the subtext within the corresponding tags, and a node representing an attribute name subsumes the attribute's value. Note that not all text need be subsumed by leaf nodes: in the example, the text "and one of" is subsumed by `<course>` and `<cprereq>`, but not by any `<cxref>` (nor any other leaf). Furthermore, in this example some SGML markup has been ignored by the modeller, as has the case used in the string for generic identifiers and attribute names. Other SGML types (such as entity references) may be similarly encoded.

Continuing with the example, the four `<cxref>` course cross references that are cited as prerequisites for course CS370, may be considered to form either a list or a set. Applications that wish to treat such cross references as an unordered collection will avoid attaching unwarranted significance to the ordering of these subtexts within the above encoding. However, applications must be careful to preserve the intended semantics of the text. When considering results derived from the above encoding, for example, even though CS370 lists four prerequisites, the text states that students need not satisfy all four prerequisites prior to enrolling in CS370.

No assumptions are made in the framework about what constitutes structural information within an arbitrary text; this is imposed by the process that parses a character string to interpret it as a structured text. Similarly, no assumptions are made about how the physical structure within the text is encoded within a model, since such assumptions would limit the usefulness of the model [Mac92].

This framework allows the full generality of structured text models to be exploited by diverse applications. Out of necessity, we have chosen one particular realization of the framework for this example. In general, individual *enterprise designers* determine the structures that are to be identified in the conceptual model and how type information is to be encoded within node labels. Similarly, *data providers* choose the mechanisms for encoding structured text as character strings and ensure the existence of *parsers* that can

be used to interpret strings' values as structured text, and thus populate the model. Through the use of standards such as SGML [ISO86] and DSSSL [ISO96d], applications and data providers can ensure that the data stored within the model exhibits the appropriate conceptual structures within a text. Thus the data provider assumes responsibility for the management of physical texts, the model provides a mechanism for describing how these physical texts may be accessed, and the data consumer remains responsible for deciding how a text is to be interpreted and manipulated.

An ordered hierarchical model for structured text seems an intuitive one, but may be unduly restrictive. In practice many loosely structured texts (and particularly those on the Web) violate the assumption that the markup within them is correctly nested. For example, font changes may occur at arbitrary points within a text, rather than within well-defined structural boundaries. The framework allows multiple hierarchical structures to be encoded as independent substructures, but does not allow relationships between distinct structural hierarchies to be modelled directly. For example, physical page boundaries impose a secondary structure on many documents, but these physical boundaries cannot readily be related to the logical document structure within our proposed framework. If the ordered hierarchical model is considered too limited, it might be possible to generalize the concept of containment and ordering used within the model, so that these concepts can be applied to overlapping regions of text (see, for example, [ISO89, Ray96b]).

3.2. Marking structured text

Previous proposals [ATA91] have recognized the importance of allowing fragments of subtext to be marked so that, for instance, these fragments may be highlighted when viewed. In environments that support update and storage of marked subtexts, such marks may also be used to store the state necessary to support interactive hierarchical text navigation and browsing, through a stateless SQL interface.

To allow fragments of structured text to be identified, any node within a text tree may be either marked or unmarked. Thus, an instance of structured text not only identifies spans of subtexts, but also includes a set of zero or more marks that identify selected structured subtexts.

The functions in Figure 4 allow marks within a structured text to be manipulated. Where multiple texts are provided as parameters, they must share the same provenance (*cf. compatibility* in SQL); otherwise an appropriate exception is raised. For all six functions, the text returned shares the same provenance as the text parameters.

Function	*Arguments*	*Returns*	*Description*
mark_subtexts	Text, Pattern	Text	Mark matched subtexts
union_marks	Text, Text	Text	Combine marks from two instances
intersect_marks	Text, Text	Text	Intersect marks from two instances
except_marks	Text, Text	Text	Subtract marks from two instances
keep_marks	Text, Int, Int	Text	Preserve marks in a given range
aggregate_marks	TextSet	Text	Union marks over several instances

Figure 4. Functions that manipulate marks in a text

The function **mark_subtexts** takes as input an instance of text and a string containing instructions about how the resulting text is to be marked (*cf*. [Kil93]). The structured text pattern matching language used to encode these instructions within the string was designed to be compact, yet expressive, and to be compatible with full-text searching as defined for SQL [ISO96s]. The pattern language is illustrated in Figure 5, assuming the text model implied by Figure 3. (Following SQL's conventions, '%' matches zero or more consecutive characters within a text label. The hash mark '#' identifies which nodes are to be marked upon a successful match.) The language is formally defined using the BNF for *<pattern>* in Figure 6. An alternative syntax using more descriptive function names rather than the compact notation presented here has also been defined [Dav96].

Pattern	Marks
<cref>#	Every <cref> node in the text
^%#	The root of the text (regardless of label)
<cref>[^%#]	Every child of any <cref> node in the text
@<course>#[<cprereq>]	Every marked course having a <cprereq> child
<course>#[<cprereq>[:xref{CS370}]]	Every course that lists CS370 as a prerequisite
<c%req>#[:xref{CS230}&:xref{CS246}]	Relations to both CS230 and CS246 in either order
@<course>#[<cwt>,<cterm>]	Marked courses whose weight appears before term
<course>[:name{CS370}&<cwt>#]	The course weight of CS370
%[<cno>#,<ctitle>#]	Every pairing of <cno> followed by <ctitle>

Figure 5. Examples of how structured text patterns match a text

```
<pattern>           ::= <node_rule> [ <descendants> ] | <node_rule>
<descendants>       ::= <set> | <list>
<set>               ::= <pattern> & <set> | <pattern>
<list>              ::= <pattern> , <list> | <pattern>

<node_rule>         ::= <rooted_rule>
<rooted_rule>       ::= ^ <marked_rule> | <marked_rule>
<marked_rule>       ::= @ <marking_rule> | <marking_rule>
<marking_rule>      ::= <node_pattern> # | <node_pattern>
<node_pattern>      ::= <node_label> { <text_expression> } | <node_label>

<node_label>        ::= <characters>
<text_expression>   ::= <characters>
<characters>        ::= <characters> <character> | <character>
<character>         ::= !! Any appropriately escaped character !!
```

Figure 6. The structured text pattern matching language

The *<pattern>* and *<descendants>* productions, allow a simple one-dimensional representation of a partially ordered pattern tree to be expressed. Within this expression, each *<pattern>* within a *<list>* (e.g., *B*, *C*, and *D* in the pattern '*A*[*B*,*C*,*D*]') constitutes an ordered descendant of the *<node rule>* immediately preceding this *<list>* within the pattern, and each *<pattern>* within a *<set>* constitutes an unordered descendant of the immediately preceding *<node rule>*.

The structured text pattern matches a subset of the nodes in an instance of structured text when

(a) every <*node rule*> is associated with exactly one distinct node in the structured text,
(b) every ancestor/descendant relationship between <*node rule*>s in the structured text pattern holds between the corresponding matched nodes within the text,
(c) ordered lists of nodes within the pattern appear in the same order as the nodes that they match within the text,
(d) any <*node rule*> containing the symbol '`^`' matches a node whose parent node (if any) is also simultaneously matched by its corresponding <*node rule*>,
(e) each <*node rule*> containing the symbol '`@`' matches a marked node within the input text,
(f) every <*node label*> agrees with the corresponding node label within the text, and
(g) the text subsumed by a matched node satisfies the <*text_expression*> (if present).

The function **mark_subtexts** identifies all possible matches (if any) between nodes in the input text and the structured text pattern, and marks any node within the matched text that corresponds to a <*node rule*> containing the hash symbol '`#`'.

The rules governing how node labels and subsumed text are matched against strings within the pattern tree should be compatible with the environment within which the structured text abstract type is supported. For SQL, a <*node label*> may use the symbols '`%`' and '`_`' as wildcards, such a <*node label*> is compared with structured text labels using the SQL 'like' predicate [ISO92], and this comparison is case insensitive. Furthermore, the <*text_expression*> must be a valid SQL/MM 'contains' clause [ISO96s]; when applied against the subsumed text, it identifies structured text nodes matching this expression. As a possible extension, a <*text expression*> could be an arbitrary SQL predicate (potentially containing more than just a Full Text search specification); this would increase the power of the pattern matching language considerably, and it might simplify the detection of cases where certain complex text operations could be optimized.

Because chain patterns are commonly used in text searching, the pattern matching language is extended with two syntactic shorthands: `A..B` represents an ancestor-descendant relationship (equivalent to `A[B]`), and `A.B` represents a parent-child relationship (equivalent to `A[^B]`). Thus, for example, every course that lists CS370 as a prerequisite could be marked by the pattern `<course>#.<cprereq>..:xref{CS370}`.

The functions **union_marks**, **intersect_marks** and **except_marks** take as input two instances of text with the same provenance and return a new text of that same provenance having marks that are respectively the union/intersection/set difference of the marks in the input texts. For example,

```
intersect_marks(
  mark_subtexts(calendar, '<course>#[<cprereq>[:xref{CS370}]]'),
  mark_subtexts(calendar, '<course>#[<cwt>,<cterm>]')
 )
```

marks courses in the calendar that have CS370 as a prerequisite and list the course weight before the term in which the course is offered.

The function **keep_marks** takes as input a text and an integer range (expressed as a start position and a length). Marks in the input text are assigned ordinals (starting from 1) consistent with the order that they would be visited by a pre-order traversal of the text

tree; those marks within the input text (if any) having ordinals lying in the specified range are preserved in the resulting text. This function provides a simple mechanism to identify the subtexts based on their order in the text.

The function **aggregate_marks** takes as input a collection of texts having the same provenance, and returns a new instance of text having this provenance and containing the union of all marks in the collection of input texts. This function may be applied as an aggregation function in SQL, for example, after grouping related rows in a table.

Two other related functions are also defined, one to test for matching text without marking and one to count marks in a text:

Function	*Arguments*	*Returns*	*Description*
text_match	Text, Pattern	Boolean	Matches text against a hierarchical tree pattern
count_marks	Text	Integer	Counts the number of marks in a text

Figure 7. Other functions associated with structured text

3.3. Extracting structured subtext

Each of the functions shown in Figure 8 extracts from an input text a collection of subtexts, returning a relation that contains the extracted subtexts.

Function	*Arguments*	*Returns*	*Description*
isolate_subtexts	Text	Relation	Extracts all marked subtexts within a text
extract_subtexts	Text, Int, Pattern	Relation	Extracts subtexts matching a given pattern

Figure 8. Functions that extract subtexts from a text

The function **isolate_subtexts** creates a binary relation. It takes an instance of text as input and, for each mark within this text, produces an output row. The first attribute within this row contains text having the same provenance as the input text, but having only the solitary mark within this text that caused the row to be generated. The second attribute in the row contains, as a new instance of text, the subtext rooted at this mark. The mark on the root is removed from the resulting subtext, but all other marks within the resulting subtext are preserved.

This form of output preserves the context of a match together with the isolated text on its own. The pairing in a single row maintains the relationship between these two texts in a manner that suits SQL's value-based semantics. Subsequent operations in SQL can select and project data from this relation to suit various applications' needs.

The function **extract_subtexts** takes as input an instance of text, an integer, and a structured text pattern as described for **mark_subtexts** above. It produces a multi-column relation with one row for every possible complete match, as described below. The number of columns in the resulting relation depends in principle on the text pattern, but in environments where this value must be known at compile time, the middle parameter indicating the expected number of columns in the resulting relation must be included. In SQL2, for example, this third argument must be an integer constant, and the function **extract_subtexts** will raise an appropriate exception if the resulting relation does not contain exactly the number of columns indicated.

In the pattern, let the number of *<node rule>*s flagged with a hash mark be *n*. For every distinct matching of the *n* flagged *<node rule>*s within the structured text pattern against nodes in the structured text (matching the entire structured text pattern against the text in at least one way) an output row is produced with *n+1* columns. The first column contains a text with the same provenance as the input text, while the remaining *n* columns contain the subtexts that matched the flagged *<node rule>*s, in the left to right order (pre-order) that they occurred within the structured text pattern. Each subtext remains marked in the innermost extracted ancestor within this tuple. No other marks are present in the texts contained with the output tuple.

For example, if the operation:

```
extract_subtexts(calendar, 3, '<course>[:name#,
<cxref>..:xref#]')
```

is applied to the subtext shown in Figure 3 the relational rows shown in Figure 9(a) are returned in no specific order. (Within this figure marked subtexts within a text are shown in bold.) This relation differs substantially from that returned by the corresponding use of **isolate_subtexts** as shown in Figure 9(b):

```
isolate_subtexts(
  mark_subtexts(calendar,'<course>[:name#, <cxref>..:xref#]')
)
```

<course **name="CS370"**... ... <cxref **xref="MATH235"**>MAT ... <p>	name="CS370"	xref="MATH235"
<course **name="CS370"**... ... <cxref **xref="MATH237"**>237< ... <p>	name="CS370"	xref="MATH237"
<course **name="CS370"**... ... <cxref **xref="CS230"**>CS 230< ... <p>	name="CS370"	xref="CS230"
<course **name="CS370"**... ... <cxref **xref="CS246"**>246< ... <p>	name="CS370"	xref="CS246"

(a) Result returned by **extract_subtexts**

<course **name="CS370"** ... xref="MATH235">MAT ... <p>	name="CS370"
<course name="CS370" ... **xref="MATH235"**>MAT ... <p>	xref="MATH235"
<course name="CS370" ... **xref="MATH237"**>237< ... <p>	xref="MATH237"
<course name="CS370" ... **xref="CS230"**>CS 230< ... <p>	xref="CS230"
<course name="CS370" ... **xref="CS246"**>246< ... <p>	xref="CS246"

(b) Result returned by **isolate_subtexts** on a text marked using the same pattern

Figure 9. Comparison of text extraction operators

The ability to extract a subtext while preserving the context within which it was extracted is significant. This avoids information loss and allows reassembly of subtexts back into the text from which they were earlier extracted.

3.4. Associating a schema with text

The encoding shown in Figure 3 provides a conceptual model of the structure of the text shown in Figure 1, but it fails to provide key information needed by a user who wishes to access this text. Such a user, and more importantly an application program, may have no *a priori* knowledge about the node labels present in the encoding, their interpretation, and

the valid relationships between the various node labels within the text encoding. It is not possible to deduce from Figure 3 that university courses may also have corequisites associated with them, and nothing indicates if course cross references occur elsewhere within the calendar, or potentially within the course descriptions subsumed by nodes labelled `<cdesc>`. Thus, when a parser converts a string into a text tree, it associates with this text the grammar that it used for parsing.

The schema shown in Figure 10 describes the actual information content present within Chapter 16 of the University of Waterloo calendar, and constitutes part of the schema for the calendar. Chapter 16 is partitioned into source files, each describing one or more departments. Departments have a name and associated courses. Course listings may include many details, such as descriptions, ancillary information, prerequisites, antirequisites and corequisites. Repeating elements within this schema have been marked with a '+' to improve comprehension.

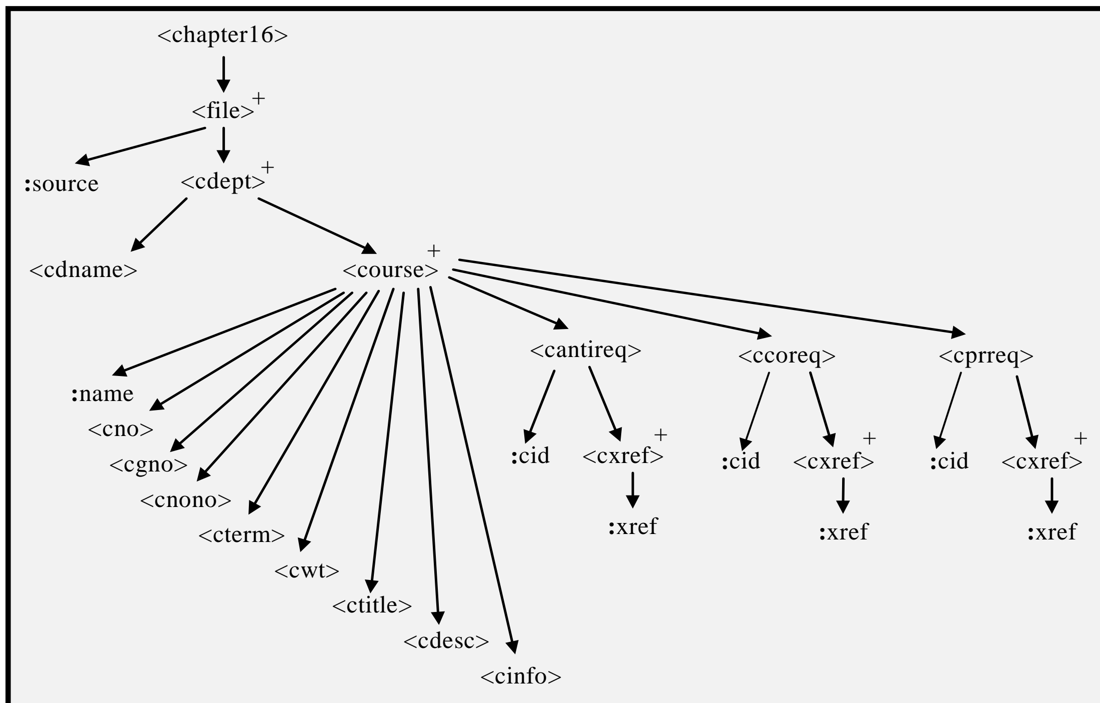


Figure 10. A schema for Chapter 16

The grammatical schema for a text is equivalent to a particular structured text having a fixed model, as witnessed by the similarity between Figure 3(b) and Figure 10. Thus, if the schema is itself converted to structured text, the operators defined earlier in this section can be applied to manipulate it. Furthermore, certain additional operators are also defined to simplify application code intended to manipulate a text's grammar. Specifically, the functions shown in Figure 11 provide rudimentary access to the grammar associated with a text and enable recovery of the schematic information presented above.

Function	*Arguments*	*Returns*	*Description*
text_to_grammar	Text	Grammar	Returns the grammar for a given text
grammar_root	Grammar	String	Extracts the name of the root of a grammar

grammar_elements	Grammar	Relation	Returns element names with their descriptions
grammar_hierarchy	Grammar	Relation	Returns child/descendant relationships
grammar_to_text	Grammar	Text	Full textual description of a text's grammar

Figure 11. Functions on the schema of a text

The function **text_to_grammar** returns the grammar associated with a given text. (Note that although all examples here use SGML, this is not a requirement imposed by the model. However, in our implementation, the grammar is automatically extracted from the DTD when an SGML parser is applied to create the text.)

The function **grammar_root**, when applied to such a grammar, returns the label of the root of the schema for this grammar. The function **grammar_elements**, when applied to a grammar, returns a binary relation describing each distinct node label in the schema associated with this grammar. For example, this function returns a table such as that shown in Figure 12 when applied to the calendar schema; the descriptive information forms part of the information to be provided with the parser (for example, as comments in the DTD).

Element name	Description
<chapter16>	Chapter 16
:name	Course name abbreviation
<cdept>	Department course listings
<course>	A course description
…	**…**

Figure 12. Part of the relation returned by **grammar_elements**

The function **grammar_hierarchy** when applied to a grammar returns a relation describing the transitive closure of all ancestor/descendant relationships within the grammar schema. This function returns the table shown in Figure 13 when applied to the calendar schema.

Ancestor	Descendant	Relationship
<chapter16>	<file>	Child
<chapter16>	:name	Descendant
<chapter16>	<cdept>	Descendant
<cdept>	<cdname>	Child
…	…	…

Figure 13. Part of the relation returned by **grammar_hierarchy**

By including all descendants explicitly, an application can determine which paths are legal in text patterns without having to call first a transitive closure function (which is non-standard in SQL). For applications that desire direct descendants only, a simple selection of tuples having `Relationship='Child'` produces the desired result.

The **grammar_to_text** function is not the inverse of the **text_to_grammar** function, but instead produces a structured text corresponding to a parsed string-form of the grammar. For example, the text of the grammar associated with an SGML document represent the actual document type definition (DTD) used when parsing and validating this SGML document. If no such text exists the function returns null. Providing a textual representation of the grammar allows the full power of the proposed text extensions to be

employed not only against an arbitrary structured text, or a particular abstract grammar, but against an arbitrary textual description of the grammar associated with a text.

It should be stressed that the above approach associates a grammatical schema with every instance of text rather than merely with a collection of texts residing in a single relational column or belonging to a particular set of textual objects. Based on industrial practice, it is unrealistic to demand that only texts having exactly the same grammatical schema can be grouped into collections. However, because it is difficult to perform set-at-a-time operations against collections of text that share little in common, applications may choose to impose constraints on text collections to ensure that the grammars of all related texts share certain features.