CS-95-25*

Text / Relational Database Management Systems: Overview and Proposed SQL Extensions†

G. E. Blake, M. P. Consens, I. J. Davis, P. Kilpeläinen
E. Kuikka, P.-Å. Larson, T. Snider, and F. W. Tompa

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

June 1995

Abstract

Combined text and relational database support is increasingly recognized as an emerging need of industry, spanning applications requiring text fields as parts of their data (e.g., for customer support) to those augmenting primary text resources by conventional relational data (e.g., for publication control). In this paper, we propose extensions to SQL2 that provide flexible and efficient access to structured text described by SGML or other encodings. We also propose an architecture to support a text/relational database management system as a federated database environment, where component databases are accessed via "agents": SQL agents that translate standard or extended SQL2 queries into vendor-specific dialects, and text agents that process text sub-queries on full-text search engines.

*Also available via anonymous ftp from site `cs-archive.uwaterloo.ca`, directory `cs-archive/CS-95-25`, file `CS-95-25.ps.Z`; and via the World Wide Web from page `http://db.uwaterloo.ca/OED/trdbms.html`.

†Supercedes earlier version published in Proceedings of the ADB'94 Conference.

1 Introduction

The application of database technology is seen as essential to the operation of a conventional business enterprise. However, there is a universe of business information, namely text, which is currently stored, accessed, and manipulated in an *ad hoc* fashion with none of the consistency and discipline of the database approach. Environments supporting both text and relational data are implemented through application programs within which separate repositories are accessed explicitly. Not only is this inconvenient for application programmers, but the disjointness of the data impedes data administrators' efforts to ensure data consistency. Furthermore, the difficult task of query optimization becomes the burden of every application programmer and the benefits of database transparency are impossible to realize. Ongoing work has laid the foundations necessary for building an alternative to this disorder and lost potential.

The objective of the research is to design and implement a multidatabase system supporting text and relational data (T/RDBMS) that will better address the needs of these enterprises. We start with the requirement that the application program interface must be an extension of both SQL2, the industry standard for relational data [ISO90, ISO92], and SGML, the industry standard for structured text [Gol90, ISO86].

The T/RDBMS can be built as a federated database system with the actual data stored and managed by standard (relational and text) data management systems, which serve as component database systems for a hybrid query processor (Figure 1).

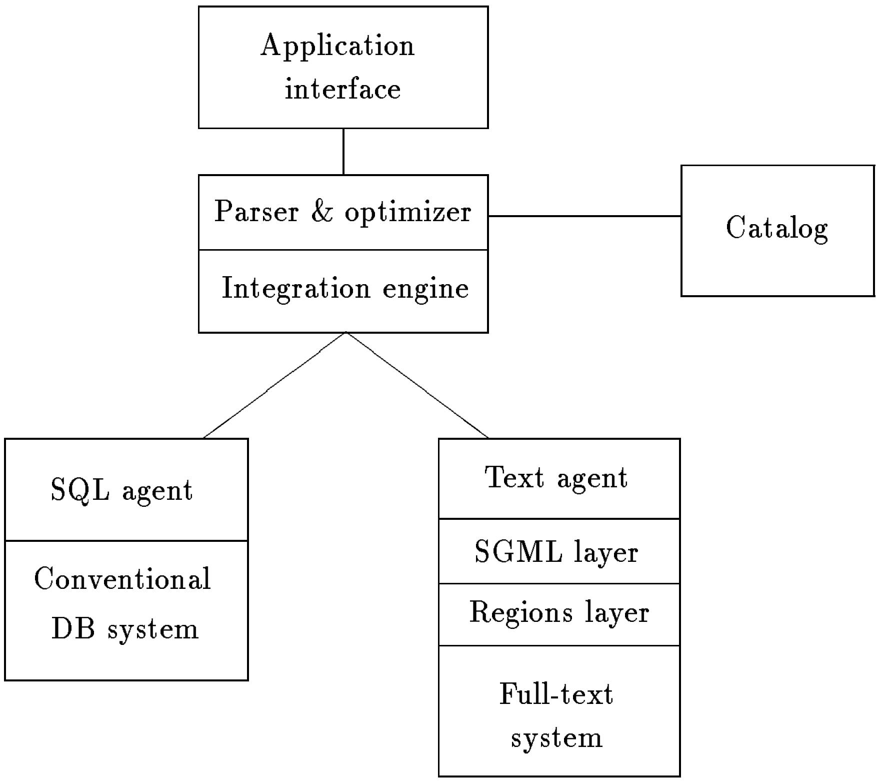

Figure 1. Federated Database System

Queries expressed in terms of the external data model are parsed, and the relational and text components identified. Query strategies can then be analyzed so that an efficient access plan can be identified. This plan can subsequently be executed under the control of an integration engine, which distributes parts of the query task to component database systems as needed, and integrates the results before they are returned to the application.

The call-level interface between applications and the hybrid query processor, and between the integration engine and the underlying agents will be based on Microsoft's Open Database Connectivity (ODBC) specification [Mic92].

Several approaches to text management have been proposed. Customized document storage management systems, including text-specific access languages, have been implemented on top of commercial relational database systems (see, for example, [Wei85, Mar91]) or as stand-alone systems (see, for example, [Gon87, Mac92]); these systems are incapable of simultaneously supporting conventional data. Alternatively, text storage has been provided by conventional systems, where long data fields are used for binary large objects or "blobs" [Bil92], but operators to support text manipulation have not usually been provided and these systems do not support SGML-like structured text.

At least three systems have been proposed within which structured text can be fragmented into relational fields and SQL queries can be applied against the resulting text subfields in conjunction with record-oriented data. The Air Transport Association has proposed the Structured Full-text Query Language (SFQL) as an extension to SQL incorporating SGML-based formatted text types [ATA91]. More recently, Oracle Corporation's SQL*TextRetrieval Version 2 provides a text retrieval product, supported by inverted indexing and a thesaurus capability, to be used in conjunction with the Oracle DBMS [Ora92]. Similarly, IDI's BASISplus supports structured full-text retrieval in conjunction with relational database functionality [Sey92]. Although each system provides a mechanism to assemble larger text units from the constituents, this is not provided within SQL. Thus, for example, such larger units cannot be presented as fields within an SQL view.

In order to maintain structured text in a single relational field, researchers at Australia's Collaborative Information Technology Research Institute have designed and implemented a nested relational database system (Atlas) and an extended SQL language to provide text support [Sac92]. Similarly there is a recent proposal to extend an object-oriented SQL dialect to support SGML documents [Chr94].

In both of the above approaches, the relational model has been extended to encompass structured data of arbitrary type, and subsequently structured text has been supported as a special case. We instead wish to explore a direct extension of SQL2 to support structured text in the hope that our proposals will suit text, and particularly SGML applications, more closely.

The SQL Multimedia and Application Packages (SQL/MM) project has a particular interest in defining abstract data types which would provide support for Full-Text within SQL3 [ISO94]. This is also our objective.

2 Example Text Database

To illustrate our proposed text extensions to SQL2, we will use a simplified version of a database management system required for an encyclopedia, and describe our extensions in terms of this

Encyclopedia

aid	title	cid	req_date	req_wc	due_date	article	biblio

example.

Such a database requires management of both administrative records and text. Information about contributors and their articles, including tracking the development of the articles, must be maintained. In addition text management involves key-word generation, cross-reference maintenance, maintaining consistency of style, and maintaining consistency of bibliographic data.

Standard SQL2 queries against this database are to be supported, to extract contributor addresses in order to generate address labels or to extract information about articles having due dates in a given time range, and to check information about payments to contributors. In addition, queries are to be expected against the bibliographic data using a variety of criteria based on authors, dates, and number of citations. Similarly, queries posed against the content of articles must be supported.

The articles themselves contain primarily prose text, but consider the form of the bibliography. Bibliographic data can be presented in a list format, a prose format or a combination of these two. For example, the bibliography for the article entitled "Canada, History of" in *The New Encyclopædia Britannica — Macropaedia* Vol. 3, p. 751, reads:

> W.L. MORTON, *The Kingdom of Canada*, 2nd ed. (1969), is the fullest one-volume history and the most traditional.... To understand the place of the colonies that became Canada in the British Empire, the following are most useful: H.A. INNIS, *The Fur Trade in Canada*, 2nd ed. (1956), and *The Code Fisheries*, rev. ed. (1954);... The following works both introduce and analyze the development of the remaining British colonies to self-governing communities and their union in confederation. W.S. MacNUTT combines in a single narrative the histories of the Atlantic provinces in *The Atlantic Provinces, the Emergence of Colonial Society, 1712-1857* (1965). FERNAND OUELLET in his *Histoire économique et sociale du Québec, 1760-1850* (1966; Eng. trans. in prep.), applies with great success the demographic method of French historiography to the little known domestic development of that province....

A requirement for this database is that we must be able to retrieve the articles and bibliographies in their text form as crafted by the encyclopedia's contributors and editors. Thus, for example, we must be able to deal with the bibliography *as a single structured textual unit* and yet identify or extract individual authors or citations as structured texts.

To illustrate our proposed language extensions, we will define one simplistic table with the following schema, where *aid* is the article identification, *title* is the proposed article title, *cid* is the contributor identification, *req_date* is the date the article was solicited from the author, *req_wc* is the requested word count for the article, *due_date* is the requested date for the article's completion, *article* is the text of the completed article, and *biblio* is the accompanying bibliography.
For the sake of brevity and clarity, we will describe constructs in the DDL and DML in terms of this example.

3 Data Definition Language

It is assumed for the present that a default data source has been established to manage standard relational columns, and that a potentially different data source has been established to manage text. The location, name, and classification of all legitimate data sources will be defined by the database administrator, using vendor specific DBA commands.

If the hybrid query processor supports a SQL2 information schema (which is not a requirement for conforming entry level SQL), then views may be added to the information schema so that information about available data sources can be retrieved.

To create the proposed relational table containing both text and other types of relational values the following command is issued to the hybrid query processor.

```
CREATE TABLE Encyclopedia (
             aid INTEGER,
             title VARCHAR(100),
             cid INTEGER,
             req_date DATE,
             req_wc INTEGER,
             due_date DATE,
             article TEXT,
             biblio TEXT,
             PRIMARY KEY (aid)
             )
```

This statement defines a table called *Encyclopedia* with eight columns. In general the above columns may be created on many different database engines. The above DDL therefore needs to allow explicit specification of the catalog (and schema) which is to manage each column. Explicitly named catalogs must be known to the hybrid query processor. The detailed specification of how a column within a federated table is to be created may also provide instructions about the physical location, management, or space requirements associated with this column. Since these instructions are specific to underlying data sources, they are transmitted to the data sources unaltered, and are verified for validity by these underlying data sources.

For example, if we wished to create a federated table distributed across four underlying data sources named source1, source2, source3 and source4, the above command could be qualified as shown below. The parameters associated with source1 are typical options which a relational database might accept, while those associated with source3 are typical options which a text engine might accept.

```
CREATE TABLE Encyclopedia (
             aid INTEGER,
             title VARCHAR(100) on source1
                   (tablespace development pctfree 10),
             cid INTEGER on source2,
             req_date DATE on source2,
             req_wc INTEGER on source2 ,
             due_date DATE on source2,
```

```
article TEXT on source3
      (nolocking stopfile common_words),
biblio TEXT on source4,
PRIMARY KEY (aid)
)
```

In the above example the primary key is an INTEGER column. This primary key will typically be stored as a column within all of the underlying source databases involved in support the representation of this table.

The *article* and *biblio* fields are of a new data type TEXT. Fields of this type contain structured text whose instances (ie. text values) have an associated GRAMMAR describing the structure of this text.

Queries involving this new TEXT data type may extract the grammar associated with any instance of structured text. This grammar when extracted (and thus disassociated from the text) is a value of type GRAMMAR. GRAMMAR values might themselves be represented using the data model for text, thus potentially allowing common operations to be defined for both TEXT and GRAMMAR. The GRAMMAR data type must be capable of effectively describing relevant information within an arbitrary SGML grammar. It may be capable of representing a much larger class of grammars, encoded using many different standards.

The following SGML Document Type Declarations (DTDs) describe the grammar of *article* and *biblio* TEXT fields in our example. Note that the article may consist of a cross reference to another article or may itself be a complete article. The body of the article is followed by some keywords and some summary information that contains data such as birth place and dates of the article's subject, as appropriate. The bibliography allows free text to be intermingled with bibliographic fields as desired.

```
<!  DOCTYPE article_information [
     <!  ELEMENT article_information - - (detailed_article|xref)>
     <!  ELEMENT xref O O ("see", article_title)>
     <!  ELEMENT detailed_article O O (header,paragraph+,keyword*,summary)>
     <!  ELEMENT header - - (title,author+)>
     <!  ELEMENT summary - - (birth|death|parent|occupation|child)*>
     <!  ELEMENT (title,author,keyword,paragraph,article_title) - - PCDATA>
     <!  ELEMENT (birth,death,parent,occupation,child) - - PCDATA>
]>


<!  DOCTYPE biblio_information [
     <!  ELEMENT biblio_information - - (citations)>
     <!  ELEMENT citations O O (citation, (";" | "."))+>
     <!  ELEMENT citation - - (author|ref|date|free_text)+>
     <!  ELEMENT ref O O ( "in"?,work,edition?)  >
     <!  ELEMENT date - - ( "(", PCDATA , ")" )>
     <!  ELEMENT (author,work,edition) - - PCDATA>
     <!  ELEMENT free_text O O PCDATA>
]>
```

Since we wish to constrain the text in both of the TEXT columns of the Encyclopedia table so that they are required to use the above grammars, the appropriate constraints will now be placed on these TEXT columns. This could alternatively have been done in the CREATE TABLE statement shown above, if desired. These constraints can later be revised as necessary by repeated use of the ALTER TABLE command.

```
ALTER TABLE Encyclopedia (
ADD CONSTRAINT C1 CHECK
    (text_to_grammar(article) =
     text_to_grammar(
       string_to_text(
         '<! ENTITY %example SYSTEM "/dtd/article"> %example;'
       )
     )
    )
ADD CONSTRAINT C2 CHECK
    (text_to_grammar(biblio) =
     text_to_grammar(
       string_to_text(
         '<! ENTITY %example SYSTEM "/dtd/biblio"> %example;'
       )
     )
    )
)
```

In the above example the DTD for article_information is assumed to be stored in the file named "/dtd/article", and the DTD for biblio_information is assumed to be stored in the file named "/dtd/biblio". Path names can, if necessary, be extended to identify the machines on which these files resided. The functions "string_to_text" and "text_to_grammar" are defined as part of the DML (section 4.1).

Other approaches to managing the grammars that may be associated with TEXT columns are equally valid. The database administrator may wish to establish a table containing (as two columns) a grammar and an associated key for this grammar. Having done so the constraint on articles described above might be replaced with the join constraint that the grammar associated with articles must match a named grammar in this table of supported grammars.

Alternatively it might be decided to partition text into those which used common grammars by defining domains such as the one shown below. TEXT columns which were constrained to have the article grammar would then be defined to be of type ARTICLE_TEXT.

```
CREATE DOMAIN article_text TEXT
       CONSTRAINT C1 CHECK
           (text_to_grammar(VALUE) IN
               (SELECT grammar FROM named_grammars
                 WHERE name = 'article')
           )
```

More generally TEXT columns may have multiple permissible grammars associated with them when this is appropriate, by either extending the constraints on these columns, or the domains used to define the types of these columns. This allows families of text having related grammars to be stored in a relational column of type TEXT. In the most general cases no constraints will be imposed on the TEXT contained in a relational column.

The constraints shown above can also be refined so that they require that the TEXT associated with tuples containing specific values have specific grammars. For example, if we anticipated that each article provided by a specific contributor would have a grammar earlier provided by that contributor, the constraint below might be used to enforce this assertion.

```
ALTER TABLE Encyclopedia ( DROP CONSTRAINT C1)

ALTER TABLE Encyclopedia (
      ADD CONSTRAINT C1 CHECK
          (text_to_grammar(article) =
             (SELECT grammar FROM contributor_grammars
               WHERE cid = contributor_grammars.id)
          )
```

The creation of constraints (and domains which depend on such constraints) requires some care in a federated environment. In the simple cases, where all of the components mentioned within a constraint are managed by a common underlying data source, it will be the responsibility of that data source to enforce this constraint. However, when constraints apply across underlying databases, the federated engine must assume the responsibility for enforcing them.

In order to facilitate rapid retrieval of tuples when queries involve text, we may wish to communicate to underlying database engines that specific indices should be associated with relational columns containing TEXT or other types of data. There is no direction provided in the SQL2 standard as to how this should be done, since it was anticipated in that standard that individual vendors would wish to use many different type of indices.

Although many database engines support composite indices spanning many columns within a table, supporting such composite indices in a federated environment is problematical since columns within a single table may be managed by different database engines. We will therefore only allow composite indices to be created on underlying data sources, when all components of a composite index are managed by a single data source.

If we again assume that the *article* and *biblio* columns in the Encyclopedia relation are managed by a single data source, we might create an index on these columns by issuing the following statement:

```
CREATE INDEX search_index ON Encyclopedia(article, biblio)
             [<Server specific instructions>]
```

Additional instructions about how these indices are to be created or used may be appended to this statement. The syntax (and interpretation) of such instructions are data source specific. These instructions will be forwarded to the underlying data source without alteration, and it will be the responsibility of the underlying data source to validate them. In the above example the underlying data source will also determine if the ordering of items indexed is significant.

Applications connected to the hybrid query processor may use the HQP as a gateway to directly connect to underlying data sources by using the SQL2 CONNECT, SET CONNECTION and DISCONNECT statements. All communication involved in supporting such subordinate connections is performed transparently by the hybrid query processor. Cascaded connections are with respect to the current connection.

Each user of the hybrid query processor may have zero or more accounts on each underlying database. Users who wish the hybrid query processor to access tables independently created on an underlying data source, must first establish how the hybrid query processor is to access this data source. This is done by using the ATTACH USER command:

```
ATTACH USER name ON server [USING userid [PASSWORD passwd]]
```

The ATTACH USER command creates a permanent mapping (which may subsequently be altered or dropped) between the invoking user, and a named set of values which allow the hybrid query processor to connect (using the userid and password if specified) to the indicated server.

Independent tables within underlying data sources, for which the invoking user has one or more attachments, may be integrated (as views) with tables created and managed by the hybrid query processor. This is done by using the ATTACH TABLE command, shown below.

```
ATTACH TABLE table_name FOR name [AS new_table_name]
```

The ATTACH TABLE command creates a permanent view (within the invoking user's schema) of the specified table, which is to be accessed by the earlier defined user attachment name. The name of this view defaults to that of the underlying table. The method of accessing tables associated with a given user attachment changes whenever that user attachment is altered. This facilitates such common administrative functions as changing userids, passwords, etc. When the structure of an attached table is changed, all attachments to this table must be dropped.

We may want to extract certain subfields of TEXT columns in order to define a view. While such views may be dependent on DML extensions and semantics as described in the next section, the creation of such views involves no new concepts. Consider, for example,

```
CREATE VIEW cited_auths AS
       (SELECT aid, title, count_marks(mark_subtexts(biblio,'<author>#'))
                                 as number_authors
          FROM Encyclopedia)
```

which defines a three column table containing the article's identification, title and a count of the authors cited within the bibliography.

Other DDL operations such as ALTER and DROP for tables, domains, indices and views exist but for brevity are not discussed here, since they conform to standard SQL2.

In summary, the CREATE TABLE statement has been extended so that a virtual federated table may be constructed, and accepts two new data types for a column, namely TEXT and GRAMMAR. Indices for TEXT are proposed, and control statements for merging components of the federated database system are provided. Unlike previous proposals, the TEXT and GRAMMAR types refer to *structured* searchable text, with an associated grammar. To comply with emerging text standards, we assume that grammars are consistent with those supported by SGML. Elements of the grammar will be available to be used in a query to refine a text search or to recover information about the grammar itself.

4 Data Manipulation Language

In our attempts to combine the concepts of text and relational databases, we have taken the approach that the text is embedded in relations rather than the other way around. Thus in our DDL we allow a field (column) in a relation to be of type TEXT (i.e., structured text). In the DML we continue on this course: operations are typically applied to one structured field at a time.

Previous authors have proposed extensions to traditional SQL operators to include operations on *unstructured* TEXT fields, encoded as character strings. In particular concatenation, string functions, and pattern matching predicates such as LIKE and CONTAINS have been supported [ATA91, Ora92, Sac92]. We include traditional SQL operators with these extensions in our design.

In this section we further extend SQL2 by describing operators on the two new data types, TEXT and GRAMMAR, and propose new SQL functions that allow these data types to be manipulated.

For the purposes of the DML, an instance of a structured TEXT field is logically represented by a tree, together with a (possibly empty) set of subtexts marked for subsequent processing. Every node in this tree has an associated string value. The structure of the nodes within the tree provides an internal representation for the structure of the TEXT. The value of the string associated with each node may encode type information for that node as well as the data to be associated with the node. Every instance of structured TEXT has an associated grammar. Two TEXT instances are unequal if they have different values in any of their components.

Grammars may be encoded using the same model proposed for encoding TEXT. If so the GRAMMAR data type will be constrained by a common external metagrammar, capable of describing any grammar. Representing GRAMMAR using the same model proposed for encoding text (but not necessarily the same encoding rules), would allow many of the operations performed on TEXT to be inherited by the data type GRAMMAR. Two GRAMMARS may be compared for equality, but GRAMMARS which define a common language in distinguishable ways will be treated as unequal.

4.1 Encoding of SGML documents as TEXT and GRAMMAR

In the examples that follow we assume for simplicity that an arbitrary SGML document contains only 'genids', 'attribute names', 'attribute values' and #PCDATA, and the corresponding TEXT encodes a parse tree. The list of node types contained within a parse tree will be extended as necessary, as our understanding of how to encode SGML documents as TEXT and GRAMMAR evolves.

Each 'genid' start-tag end-tag pair within the text is translated into a genid node within our data model. Each attribute name, attribute value, and instance of #PCDATA within the text is translated respectively into an attribute name node, attribute value node, and content node.

Genid nodes contain as their immediate children any attribute name nodes associated with these genid nodes, followed by any directly contained genid nodes or content nodes (in the order consistent with the text). Attribute name nodes have a single child which is the attribute value node. Attribute value nodes, and content nodes are always leaf nodes.

Each type of node within the model has an associated string value. These string values may be matched against strings contained in patterns. Since the model provides only a conceptual representation for TEXT, there is no requirement that these string values be stored without modification by underlying text engines. Text retrieval engines may perform any appropriate operations when translating their conceptual understanding of structured text and the patterns matched against them, into the concrete operations needed to identify, mark or extract the specified subtexts. Any such translation from virtual to concrete operations must of course be transparent to requesting applications.

Genid nodes have a string value that begins with a ‘<’, continues with the genid name, and terminates with a ‘>’. Attribute names have a string value that contains this name preceded by a ‘:’. Attribute values have a string value that contains this value preceded by an ‘=’. Content nodes have a string value that begins with ‘"’ continues with the value of this content, and is terminated with a matching ‘"’.

As an example of the translation of an SGML encoded string into the proposed conceptual parsed data structure, consider the fragment:

```
<section>
  <para status=1stedition>
     This is some
     <emph author=smith font=bold>
        great
     </emph>
     text
  </para>
</section>
```

Figure 2 shows the conceptual parse tree that forms part of the corresponding TEXT value. A preorder traversal of the parse tree enumerates items in the order that they are first encountered when performing a left to right scan of the text.

4.2 Conversion functions for TEXT and GRAMMAR

It is natural to assume that the existing SQL2 cast function is extended to include support for TEXT and GRAMMAR as a fundamental data type. However the casting mechanism provided by SQL2 assumes that no external information is required when one data type is cast into another and that there is a partial one-to-one mapping between data types. Neither of these assumptions is true when constructing TEXT and GRAMMAR. It is therefore proposed that the new functions described below be supported.

TEXT string_to_text(STRING s [,STRING parser])

STRING text_to_string(TEXT t [,STRING method])

GRAMMAR text_to_grammar(TEXT t)

The ‘parser’ parameter identifies the parser used to perform the translation from STRING to TEXT, while the ‘method’ parameter identifies the desired translation from TEXT back into STRING. When converting a STRING into a TEXT the parser will expect input to conform to a specific standard and will have knowledge of how it is to behave in converting this input into an internal instance of TEXT conforming to the data model for TEXT. When converting a TEXT into a

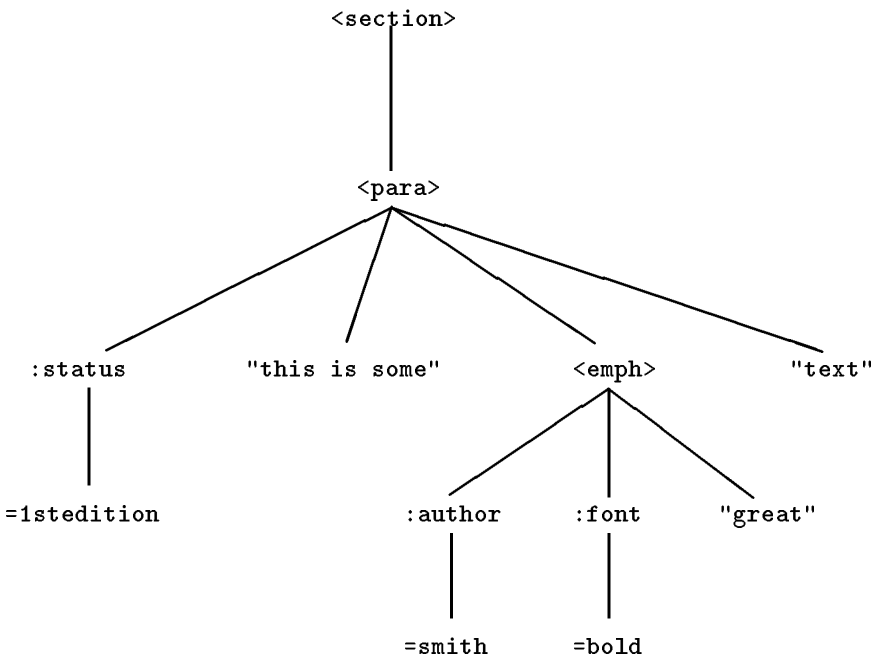

Figure 2. An encoding for part of an instance of text

STRING the method (and potentially the internal encoding of the instance of text) determines the desired format of the output string.

The default STRING to TEXT parser is assumed to accept an SGML document as an input string and to produce a TEXT encoded as described in this paper as output. The default TEXT to STRING parser accepts an instance of TEXT encoded as described in this paper as input, and produces a STRING containing the corresponding text as output. In addition, special markup may be added to this output string to communicate the location of marked text within this textual content.

The function 'text_to_grammar' returns the internal GRAMMAR associated with an instance of TEXT. This internal GRAMMAR describes (possibly by using a suitable encoding within the data model for text) properties of the corresponding structured TEXT (as represented within the data model) which are deemed to be of relevance either to text engines or to applications. The TEXT to GRAMMAR translation will produce a GRAMMAR which includes a description of the legitimate values associated with internal TEXT nodes, the allowable ancestor/descendent relationships between these nodes, and the name of the parser used to encode these nodes.

Unspecified extensions may optionally be supported by text engines, if these engines wish to provide additional vendor specific structural information about specific instances of TEXT.

The computation of the internal GRAMMAR will be independent of the textual content contained within an external document. Thus for example, when two SGML documents having identical DTD's and document declarations are converted to TEXT using the same parser, the internal GRAMMARs associated with these internal TEXTs will be identical.

If the functions described above are unable to convert their input into a valid instance of the desired data type (be that TEXT, GRAMMAR, or STRING) then a suitable exception condition is raised.

A number of specific methods are envisioned to convert from TEXT (and potentially GRAMMAR) to STRING, primarily so that effective string comparisons may be performed against the resulting strings. These functions are not yet well defined but include

STRING text_to_string(TEXT text1, 'root')

STRING text_to_string(TEXT text1, 'clear')

The 'root' translation returns the string at the root node of the parse tree contained within 'text1'. If 'text1' is empty or null, then null is returned. The 'clear' translation returns a string containing the textual content associated with the given instance of TEXT. Within this string no markup is present.

4.3 Searching and marking operations on TEXT

In order to use text effectively it must be possible to identify instances of TEXT that match certain patterns. It must be possible to mark the subtexts within the TEXT which contain a given pattern. It must also be possible to count the number of marked subtexts within an instance of TEXT and to produce the union, intersection and difference of existing marked subtexts within otherwise identical text. Finally it must be possible to decompose a relational element of type TEXT into marked subtexts, so that marked subtexts can be operated on directly by SQL, while still being related with the point within the TEXT from which they were extracted.

Six functions allow text to be searched and marked. Each of these functions returns NULL if any of their inputs is NULL. These function are:

BOOLEAN text_match(TEXT text1, STRING pattern [, CHAR escape])

TEXT mark_subtexts(TEXT text1, STRING pattern [, CHAR escape])

TEXT mark_union(TEXT text1, TEXT text2)

TEXT mark_intersect(TEXT text1, TEXT text2)

TEXT mark_except(TEXT text1, TEXT text2)

INTEGER count_marks(TEXT text1)

The text_match() function returns true if and only if the text pattern (described further in Section 4.4) can be matched (in at least one way) against some text tree contained within the input TEXT. Otherwise it returns false. To minimize the impact on the SQL2 and earlier standards (which do not support a boolean data type) the text_match() function returns the integer 1 for true and 0 for false.

An optional escape character may be used to escape any special meaning associated with specific characters contained within a pattern. The escaping mechanism is consistent with that used in SQL2 to escape characters within the pattern associated with the LIKE predicate. The default escape character is the backslash.

Each of the functions mark_subtexts(), mark_union(), mark_intersect(), and mark_except() return as their result a new text instance of type TEXT, which differs from 'text1' (if at all) only in the marked subtexts that are contained within this new instance of text. The new resulting instance of TEXT has the same parse tree and grammar as 'text1'.

The function mark_subtexts() marks a new set of subtexts within the text presented to it. The appropriate subtexts are computed using the pattern presented to this function, as described in Section 4.4 below.

The functions mark_union(), mark_intersect(), and mark_except() require that text1 and text2 differ (if at all) only in their marked subtexts. When this is not the case the behaviour of these functions is undefined. The function mark_union() returns a new instance of text in which subtexts are marked if and only if they were marked in either 'text1', 'text2', or both. The function mark_intersect() returns a new instance of text in which subtexts are marked if and only if they were marked in both text1 and text2. The function mark_except() returns a new instance of text in which subtexts are marked if and only if they were marked in 'text1' but not in 'text2'.

The function count_marks() returns as an integer the number of marked subtexts contained within the input TEXT. While the function text_match() can be trivially implemented by applying the count_marks() function to the mark_subtexts() function, text_match() is supported since it may potentially be more efficient.

One higher level function is proposed, which can if desired be implemented using other functions described in this section.

TEXT keep_marks(TEXT text1, INTEGER start [,INTEGER count])

The function keep_marks() computes the ordering of the marked subtexts within a pre-order traversal of the parse tree. The first marked subtext is at position 1 within this ordering, and the last marked subtext is at position 'count_marks(text1)' within this ordering. The mark associated with the n'th such marked subtext is preserved in the resulting text if and only if $start \leq n < start + count$. The count of the number of marked subtexts to be preserved defaults to 1.

4.4 Pattern matching language

The pattern matching language may be used to test for the presence of subtext values and to mark selected subtexts. The design of the language is based on the assumption that a marked subtext must correspond to a subtree associated with the TEXT value and can therefore be identified by marking the root node of that subtree.

The structure of the pattern matching language has the following syntax.

```
pattern        := tree_path [ '[' forest ']' ]
forest         := pattern [ ',' forest ]
tree_path      := rooted_rule [ extended_path ]
extended_path  := path_rule node_rule [ extended_path ]

path_rule      := '..' | '.'
rooted_rule    := [ '^' ] node_rule
node_rule      := [ '@' ] string_pattern [ '#' ]

string_pattern := character [ string_pattern ]
character      := Any character subject to rules below
```

Each of the characters '.' , '[', ',', ']', '#', '@', and '^' must be escaped when they occur within a string_pattern. The quote character (') must also be escaped, since pattern matching is expressed using a (potentially quoted) SQL string. As a design principle, the symbols ‘A’-‘Z’, ‘a’-‘z’, ‘-’, and ‘ ’ were not chosen to represent pattern matching operators, so that they need not be escaped when used to represent themselves in string matching.

The pattern matching language has the following informal interpretation. Compare the tree which represents the TEXT against the tree_path. For each path from the root of this tree to some node ‘X’ which matches this tree_path proceed as follows. If no forests are specified consider a match to have occurred. Otherwise, for each sequence of distinct child nodes of X which can be paired with each pattern within the forest (pairing nodes in left to right sibling order against patterns in left to right pattern order), attempt to match the subtrees rooted at these child nodes with their paired patterns. Consider a match to occur if and only if all such subtrees are simultaneously matchable with their patterns.

The pattern matching rules have the following interpretation. If a node rule is preceded by a ‘^’ only the root node in the tree being examined may satisfy this rule. If the root node does not satisfy this rule then the match fails. If a node rule is not preceded by a ‘^’ then any node at or under the root node of the tree being examined may match this pattern.

If a string pattern is preceded by an ‘@’ then only marked nodes within the text being examined may satisfy this pattern; otherwise any node may match. Nodes may become marked as a result of earlier or nested text operations.

If a string pattern is followed by a ‘#’ then the node compared with this pattern will be eligible to be marked within the resulting text, assuming this comparison results (perhaps in conjunction with other comparisons) in a match.

A string pattern matches a node (subject to all of the above additional constraints) if the string associated with this node matches this pattern. The string pattern may use ‘_’ to indicate the presence of an unknown character and ‘%’ to indicate the presence of zero or more unknown characters. These two characters must be escaped if they are to be interpreted literally.

The semantics for path rules can best be described by explaining how patterns which contain such path rules can be translated into equivalent patterns which do not contain such path rules. The path rules ‘..’ and ‘.’ have equal precedence and are right associative. A..B[forest] is equivalent to A[B[forest]] and A.B[forest] is equivalent to A[^B[forest]]. If no forest is associated with B then A..B is equivalent to A[B], and A.B is equivalent to A[^B]. Since the path rules are right associative A.B..C is equivalent to A[^B[C]].

Within the instance of SGML text in Figure 2, all paragraphs containing an emphasized occurrence of the word *great* would thus be marked by using the pattern `'<para>#..<emph>.."great"'` . All sections containing such marked paragraphs could themselves subsequently be marked by using `'<section>#..@%'` , or if the marks on paragraphs were also to be preserved `'<section>#..@%#'` .

Nothing in the above design precludes extensions to support alternative string pattern matching languages, better able to describe regular expressions, or patterns corresponding to SFQL’s CONTAINS predicate.

4.5 Subtext extraction and aggregation operations

Two functions allow text to be decomposed into related instances of subtext contained within a resulting relation, while one new set aggregation function allows aggregation of marks within the resulting relations. These function are:

RELATION isolate_subtexts(TEXT text1)

RELATION extract_subtexts(TEXT text1, INTEGER columns,
STRING pattern [, CHAR escape])

TEXT aggregate_marks(TEXT COLUMN text1)

The function isolate_subtexts() returns a two column relation having as many tuples as marks within the input TEXT. The first column contains TEXT having identical parse trees, and grammars to 'text1'. Each instance of TEXT within this column has exactly one marked subtext, and every instance of TEXT within this column contains a distinct marked subtext. In each tuple within this relation, the second column contains a copy of the extracted subtext marked within the first column. Within the second column, the root of this subtext is not marked, but all subordinate marked subtexts remain marked.

The function extract_subtexts() generates a relation having the number of columns n defined by the second parameter 'columns'. This second parameter is constrained to be a literal so that it can be evaluated at parse time. The string pattern must identify $n-1$ nodes which are to be marked. The function generates one tuple per distinct pattern match. The first column contains the text with the marked subtexts associated with this match. Subsequent columns contain extracted copies of the marked subtexts rooted at the matched node. Within these subsequent columns the root of the subtext is not marked, but all subordinate marked subtexts remain marked. The columns are populated in the order (using a left to right scan of the pattern) in which the '#' symbol used to request that these subtexts be marked (and thus extracted) occurred.

The function aggregate_marks() is a new set function which operates on a set of text instances drawn from different tuples. As is the case with other aggregate operations, it may be applied either to every value produced by a TEXT value expression, or (when grouping is being performed) to instances of TEXT contained in separate sets of grouped tuples. It computes the result of iteratively applying the function mark_union() to every input TEXT within such a set. The result for any such input set is NULL if any of the TEXT within this input set is NULL.

The relations returned by isolate_subtexts() and extract_subtexts() have no duplicate rows. Since the first column has no duplicated values, the values in this first column may serve as a primary key for the relation.

Functions (such as those described above) which convert instances of TEXT into relations need to be integrated in some way with SQL2. It is proposed that SQL2 be extended by allowing extract_subtexts() and isolate_subtexts() (and potentially other functions which return relations) to be used in any context where a subquery could occur, subject to the constraint that the scoping of any parameters associated with these functions is valid.

We also propose an extension to the SELECT clause to include the keyword UNNEST. Whenever a SELECT clause has an UNNEST qualifier, subqueries occurring within the list of expressions being projected by the SELECT clause may generate any type of value including values that are themselves relations that have multiple rows and columns.

The UNNEST operation is applied to the results projected by the SELECT as follows. For each conventional value expression within the select list (which returns one atomic value per tuple) treat this atomic value as a one row, one column relation. For each value expression which returns one relation per tuple treat this result as a relation, but replace empty relations with a single row in which all fields are null. Form tuples to be output by computing the cross product of each of these resulting relations within the input tuple, taking care not to reorder any of the columns contained either within the original select list or within those functions that return relations. New columns resulting from functions that returned relations are unnamed.

5 Examples of the Proposed DML

In our example database we might want to:

> *Mark all elements within articles which contain CANADA.*

```
SELECT mark_subtexts(article, '<%>#.."%CANADA%"')
  FROM Encyclopedia
```

> *Find proposed titles and lengths of long articles on Canada.*

Using our DML the answer can be obtained with the query

```
SELECT title, req_wc
  FROM Encyclopedia
 WHERE req_wc > 5000
   AND text_match(article,'<keyword>."%Canada%"') = 1
```

As a more complex question, consider

> *Who contributed articles for which the proposed titles do not match the titles included in the article's body?*

With our extended SQL, we can formulate this query as

```
SELECT cid,aid
  FROM (SELECT UNNEST cid, aid, title,
               extract_subtexts(article,2,'<title>#')
          FROM Encyclopedia
       ) AS E(cid,aid,title,article,actual_title)
 WHERE title <> text_to_string(actual_title,'clear')
```

As a matter of interest, note that if the title contains only alphabetic characters, blanks, and hyphens the WHERE clause could also have been written:

```
 WHERE count_marks(actual_title) > 0
   AND match_text(actual_title,'@%."' || title || '"') = 0
```

In the above example there is an implicit assumption that each article will have at most one title. That assumption is consistent with the grammar earlier provided for articles. If more than one title was permitted within an article, then the following query might be better:

```
SELECT cid, aid
  FROM Encyclopedia E1
 WHERE title NOT IN
       (SELECT text_to_string(title,'clear')
          FROM (SELECT UNNEST extract_subtexts(article,2,'<title>#')
                  FROM Encyclopedia E2
                 WHERE E2.aid = E1.aid
               ) AS R(article,title)
       )
```

While the above queries can be viewed as constructing a derived table and then performing a selection on this derived table before projecting a final result, implementors may optimize these queries by recognizing that the only rows that must necessarily be returned from the Encyclopedia are those where title is not equal to the marked title. This optimization is conceptually similar to the types of optimizations already performed in most query processors, in order to use indices effectively.

Our next examples explore how one would search text subelements that might appear in more than one TEXT column, or be contained at different points within a single TEXT column.

> *Find the article identification, title, contributor, and bibliography for entries where the bibliography has more than one citation but all are from the same author.*

```
SELECT aid, title, cid, biblio
  FROM Encyclopedia
 WHERE count_marks(mark_subtexts(biblio,'<citation>#')) > 1
   AND 1 =
       (SELECT count(DISTINCT author)
          FROM extract_subtexts(biblio,2,'<author>#') biblio(whole,author)
       )
```

> *Find titles of articles and titles of associated works mentioned in the bibliography and merge the results.*

```
SELECT UNNEST aid, 'Title', extract_subtexts(article,2,'<title>#')
  FROM Encyclopedia
 UNION
SELECT UNNEST aid, 'Work', extract_subtexts(biblio,2,'<work>#')
  FROM Encyclopedia
```

> *Find every citation in the bibliography that cites any author of the article. Show authors cited.*

```
SELECT aid, author, citation
  FROM (SELECT UNNEST
               aid,
               extract_subtexts(article,2,'<author>#'),
               extract_subtexts(biblio,3,'<citation>#..<author>#')
          FROM Encyclopedia
       ) E(aid,article,author,biblio,citation,cited)
  WHERE text_to_string(cited,'clear') = text_to_string(author,'clear')
```

Find every citation which cites the article's author and title.

```
SELECT aid, cite
  FROM (SELECT UNNEST aid,
               extract_subtexts(article,3,'%[<title>#,<author>#]'),
               extract_subtexts(biblio,3,'<citation>#.<author>#'),
               extract_subtexts(biblio,3,'<citation>#..<work>#')
          FROM Encyclopedia
       ) E(aid,article,title,author,biblio,cite,cited_author,biblio2,cite2,work)
 WHERE text_to_string(author,'clear') = text_to_string(cited_author,'clear')
   AND text_to_string(title,'clear') = text_to_string(work,'clear')
   AND mark_subtexts(cite,'%') = mark_subtexts(cite2,'%')
```

The use of clear text in the above example is necessary, since comparisons for equality on TEXT will fail when the compared texts have different grammars. Similarly if mark_subtexts had not been used, citations would not have compared equal since the two uses of extract_subtexts produce TEXT values with different marks.

List all element names actually used within articles.

```
SELECT DISTINCT text_to_string(element,'root')
  FROM (SELECT UNNEST extract_subtexts(article,2,'<%>#')
          FROM Encyclopedia
       ) E(article,element)
```

List all element names which are used under summaries within articles.

```
SELECT DISTINCT text_to_string(element,'root')
  FROM (SELECT UNNEST extract_subtexts(article,2,'<summary>..<%>#')
          FROM Encyclopedia
       ) E(article,element)
```

Mark all element names containing subordinate elements under summaries of articles.

```
SELECT mark_subtexts(article,'<summary>..<%>#..<%>')
  FROM Encyclopedia
```

Mark all elements whose parent has an attribute named status with value "Obs."

```
SELECT mark_subtexts(article,'<%>[^:status.=Obs\.,^<%>#]')
  FROM Encyclopedia
```

Additional examples showing how the proposed DML might address Paula Angerstein's SGML challenge queries [SGM92] are provided in the Appendix.

6 Conclusions and Further Work

We have developed a single model within which both text and relational data can be described so that users can access and manipulate all their data meaningfully. Our proposed extensions to SQL are modest, yet they are powerful enough to handle SGML-based data simply, to support extractions from highly structured text into relations, and to preserve the integrity of complex text units. We have designed a data description language and a query language integrating SQL with text search and text manipulation features, addressing the following questions:

- How can an SGML-defined description of text be integrated with SQL's DDL?
- Which text manipulation operators should be included in an extended SQL DML?
- How can text be selectively decomposed into relations?

We have designed an architecture to support integrated text-and-relational databases using a federated database system. We have begun to implement and test the architecture and language, supporting data stored partially in an Oracle relational database, partially under the control of a DB2/6000 relational database, partially under the control of the PAT text engine [OTC95], and partially under the control of Fulcrum's Search Server text engine [Ful94]. We expect to test all three text interfaces (SGML, regions, and flat-text), and expect that our experience will be transferrable to other engines with various capabilities.

We have not yet adequately addressed the following issues:

- How should disjunction be handled when searching for patterns within text?
- Can the SFQL CONTAINS predicate be extended to induce marks within text?
- What additional TEXT to STRING functions are required?
- What is contained within an internal GRAMMAR and how is this information encoded?
- How can TEXT and GRAMMAR be stored on underlying engines?
- How can structured TEXT best be transferred across external interfaces?
- How can the TEXT and GRAMMAR data types be updated?

- How can primary and foreign keys constraints be defined and enforced, when keys are embedded within text?
- How can costs useful to query optimizers be estimated prior to retrieving TEXT, and GRAMMARs?
- What extensions to SQL's information schema are needed to support TEXT?
- How can the proposed extensions be defined using the emerging SQL3 standard?

References

[SGM92] P. Angerstein, Texcel "SGML Sample Queries", unpublished, 28 October 1992.

[ATA91] ATA 89-9C SFQL Committee, "Advanced Retrieval Standard —SFQL: Structured Full-text Query Language," *ATA specification 100, Rev 30, Version 2.2, Prerelease C*, Air Transport Association, ATA 89-9C.SFQL V2.2/PR-C (October 1991) 84 pp.

[Bil92] A. Biliris, "The Performance of Three Database Storage Structures for Managing Large Objects," *Proc. Sigmod 92,* ACM*, Sigmod Record*, Vol. 21, No. 2 (June 1992) 276–285.

[Chr94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl, "From Structured Documents to Novel Query Facilities," *Proc. Sigmod 94,* ACM*, Sigmod Record*, Vol. 23, No. 2 (June 1994) 313–324.

[Ful94] Fulcrum SearchServer Version 2.0 Introduction to SearchServer, June 30, 1994.

[Gol90] C. F. Goldfarb. *The SGML Handbook.* Oxford University Press, Oxford, 1990.

[Gon87] G. H. Gonnet and F. W. Tompa, "Mind Your Grammar: a New Approach to Modelling Text," *Very Large Data Bases* (VLDB), Vol. 13 (September 1987) pp. 339–346.

[ISO86] International Organization for Standardization, *International Standard 8879: Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML)*, first edition — 1986-10-15(Ref. No. ISO 8879-1986(E)), 155 pp.

[ISO90] International Organization for Standardization, "Information technology – Database Language SQL 2 Draft Report", ISO Committee ISO/IEC JTC 1/SC 21, 1990.

[ISO92] International Standard, "Information technology – Database Languages – SQL, ISO Committee ISO/IEC 9075, 1992.

[ISO94] International Organization for Standardization, *ISO/IEC JTC1/SC21 Information Retrieval, Transfer and Management for OSI WG3 Database*: ISO/IEC SC21/WG3 N1679 SQL/MM SOU-004 ISO Working Draft SQL Multimedia and Application Packages (SQL/MM) - Part 2: Full-Text, March 1994.

[Mar91] C. C. Marshall, F. G. Halasz, R. A. Rogers, and W. C. Janssen Jr., "Aquanet: a Hypertext Tool to Hold Your Knowledge in Place," *Proc. 3rd ACM Conf. on Hypertext: Hypertext 91*, San Antonio (Dec. 1991) 261–275.

[Mac92] I.A. Macleod, Data Modelling Requirements for Document in *INFORMATION SYSTEMS CONCEPTS: Improving the Understanding*, North-Holland, also the Proceedings of the IFIP TC8/WG8.1 Conference, pp.259-272, Alexandria, 1992.

[Mic92] Microsoft Corporation, *Microsoft ODBC Application Programmer's Guide*, Microsoft Corporation, 1992.

[Ora92] Oracle Corporation, *SQL*TextRetrieval Version 2 Technical Overview*, Oracle Corporation, 1992. 45 pp.

Multimedia and Application Packages (SQL/MM) - Part 2: Full-Text, March 1994

[OTC95] Open Text Corporation, "PAT Reference Manual", *Open Text 5 System Integration Guide* and *Database Administration Guide*, Release 5.0, 1995.

[Sac92] R. Sacks-Davis, A. Kent, K. Ramamohanarao, J. Thom, , and J. Zobel, "Atlas: a nested relational database system for text applications", Technical Report CITRI/TR-92-52, Collaborative Information Technology Research, Victoria, Australia, July 1992.

[Sey92] Seybold Publications, "IDI Pursues Document Management," *Report on Publishing Systems*, Vol. 21, No. 16, May 1992.

[Wei85] E.S.C. Weiner. "The *New OED*: Problems in the Computerization of a Dictionary," *University Computing*, Vol. 7 (1985) 66-71.

Acknowledgements

This work has been carried out as part of the University's ongoing participation in the Canadian Strategic Software Consortium (CSSC). The CSSC was formed in 1993 to perform pre-competitive research on the integration of relational and text databases and is partially supported by Industry Canada's Strategic Technologies Program (STP). CSSC members include: Fulcrum Technologies Inc., Grafnetix Systems Inc., InContext Corporation, Megalith Technologies Inc., Open Text Corporation, Public Sector Systems Ltd., Softquad Inc. and the University of Waterloo.

Ideas expressed in this paper have been developed and refined in part through discussions with Tim Bray, Gordon Cormack, Gaston Gonnet, and members of the CSSC's Hybrid Query Processor (HQP) working group. Financial assistance was provided by the University of Waterloo and through grants from the Natural Sciences and Engineering Research Council of Canada, Industry Canada, and Open Text Corporation.

APPENDIX

SGML Sample Queries

This appendix demonstrates how the proposed DML might be used to answer a diverse set of sample queries. These sample queries are taken verbatim from the SGML Sample Queries [SGM92]. Section B reproduces the supporting material from that same source.

1 Sample Solutions

In the examples that follow we assume that the sample text being examined resides in a column named report of type TEXT, within a table named T containing one row.

1. *Locate all paragraphs in the report (all elements whose GI is "para" anywhere within the "report" element)*

```
SELECT mark_subtexts(report,'<para>#')
  FROM T
```

2. *Locate all paragraph elements in an introduction (all "para" elements directly contained within an "intro" element)*

```
SELECT mark_subtexts(report, '<intro>.<para>#')
  FROM T
```

3. *Locate all paragraphs in the introduction of a section that is in a chapter that has no introduction (all "para" elements directly contained within an "intro" element directly contained in a "section" element directly contained in a "chapter" element. The "chapter" element must not directly contain an "intro" element.)*

```
SELECT mark_subtexts(
         mark_except(
           mark_subtexts(report, '<chapter>#'),
           mark_subtexts(report, '<chapter>#.<intro>')
         ),
         '@<chapter>..<section>..<intro>..<para>#'
       )
  FROM T
```

4. *Locate the second paragraph in the third section in the second chapter (the second "para" element occurring in the third "section" element occurring in the second "chapter" element occurring in "report")*

```
SELECT keep_marks(
         mark_subtexts(
           keep_marks(
             mark_subtexts(
               keep_marks(
```

```
                mark_subtexts(report,'<chapter>#'),
                2
              ),
              '@%..<section>#'
            ),
            3
          ),
          '@%..<para>#'
        ),
        2
      )
  FROM T
```

5. *Locate all classified paragraphs (all "para" elements whose "security" attribute has the value "c")*

```
SELECT mark_subtexts(report,'<para>#.:security.=c')
  FROM T
```

6. *Locate the short titles of all sections (the value of the "shorttitle" attribute of all "section" elements)*

```
SELECT mark_subtexts(report,'<section>.:shorttitle.=%#')
  FROM T
```

7. *Locate the initial letter of the initial paragraph of all introductions (the first character in the content [character content as well as element content] of the first "para" element contained in a "intro" element)*

```
SELECT text_to_string(
         mark_except(
           mark_subtexts(report,'<intro>[<para>#]'),
           mark_subtexts(report,'<intro>[<para>,<para>#]')
         ),
         'highlight_first_char'
       )
  FROM T
```

8a. *Locate all sections with a title that has "is SGML" in it (all "section" elements that contain a "title" element that has the consecutive characters "is SGML" in its content). The string can be interrupted by sub-elements*

```
SELECT aggregate_marks(marked_report)
  FROM (SELECT UNNEST extract_subtexts(report,2,'<section>#..<title>')
          FROM T
       ) T(marked_report,section)
 WHERE EXISTS
       (SELECT NULL
          FROM extract_subtexts(T.section,2,'<title>#') T1(section,title)
         WHERE text_to_string(title,'clear') LIKE '%is SGML%'
       )
```

8b. *Locate all sections with a title that has "is SGML" in it (all "section" elements that contain a "title" element that has the consecutive characters "is SGML" in its content). The string cannot be interrupted by sub-elements*

```
SELECT mark_subtexts(report, '<section>#..<title>.."%is SGML%"')
  FROM T
```

9a. *Locate all occurrences of "mark" followed by "up" in the report, where 0 to 10 characters, line starts, or line ends can occur between "mark" and "up". The string can be interrupted by sub-elements*

The CONTAINS predicate is the natural way to formulate a solution to this query. It remains to be decided if the CONTAINS predicate may be applied to TEXT, and how (if it were capable of being applied to TEXT) this text might be suitably marked.

9b. *Locate all occurrences of "mark" followed by "up" in the report, where 0 to 10 characters, line starts, or line ends can occur between "mark" and "up". The string cannot be interrupted by sub-elements.*

The observations made in 9a also apply here. However, since the match is to be made against the value of a single node, this query might also potentially be resolved by providing an alternative string pattern matching language which allowed detection of the desired strings within a mark_subtexts() function.

10. *Locate all the topics referenced by a cross-reference anywhere in the report (all the "topic" elements whose "topicid" attribute value is the same as an "xrefid" attribute value of any "xref" element)*

```
SELECT aggregate_marks(marked_topic)
  FROM (SELECT UNNEST
               report,
               extract_subtexts(report,3,'<topic>#.:topicid.=%#')
          FROM T
       ) T(report,marked_topic,topic,id)
 WHERE text_to_string(id,'clear') IN
       (SELECT text_to_string(xrefid,'clear')
          FROM extract_subtexts(report,2,'<xref>.:xrefid.=%#')
               AS E(marked_xrefid,xrefid)
       )
GROUP BY report
```

11. *Locate all CGM graphic elements (all "graphic" elements whose "graphname" attribute identifies an external entity whose notation is "cgm")*

```
SELECT mark_subtexts(report,'<graphic>#.:graphname.=%.!cgm')
  FROM T
```

We assume in this example that the text encoding has been extended by allowing (where appropriate) the notation associated with an attribute value to be encoded beneath this attribute value. Within the extended encoding notation is identified by an initial '!'.

12. *Locate all entity references to CGM graphics (all entity references to external entities whose notation is "cgm" [not including graphic elements])*

```
SELECT mark_subtexts(report,'&%#..!cgm')
  FROM T
```

We assume in this example that the text encoding has been extended to allow entity references to be encoded, and that such entity references are identified (within the encoding) by an initial '&'. The notation (if any) associated with an external entity reference is assumed to appear somewhere beneath this entity reference.

13. *Locate the closest title preceding a given cross-reference (the "title" element that would be "touched" last before the "xref" element when touching each element in document order)*

```
SELECT keep_marks(earlier_titles, count_marks(earlier_titles))
  FROM (SELECT mark_subtexts(marked_xref,'%[<title>#,@%]') as earlier_titles
          FROM marked_text
       )
```

In the above example we assume that marked_text is a view. The column marked_xref contains an appropriate instance of text in which only the given cross reference is marked.

2 Sample Data

```
<!DOCTYPE report  [
<!NOTATION cgm PUBLIC "Computer Graphics Metafile">
<!NOTATION ccitt PUBLIC "CCITT group 4 raster">
<!ENTITY % text "(#PCDATA | emph)*">
<!ENTITY  infoflow  NDATA   ccitt>
<!ENTITY tagexamp NDATA    cgm>
<!ENTITY gcalogo  NDATA    cgm>
<!ELEMENT report - o (title, chapter+)>
<!ELEMENT title  - o (%text;)>
<!ELEMENT chapter - o (title, intro?, section*)>
<!ATTLIST   chapter
 shorttitle CDATA #IMPLIED>
<!ELEMENT intro - o (para | graphic)+>
<!ELEMENT section - o (title, intro?, topic*)>
<!ATTLIST   section
 shorttitle CDATA #IMPLIED
 sectid ID #IMPLIED>
<!ELEMENT topic - o (title, (para | graphic)+)>
<!ATTLIST topic
 shorttitle CDATA #IMPLIED
```

```
 topicid ID #IMPLIED>
<!ELEMENT  para  -  o (%text; | xref)*>
<!ATTLIST   para
 security (u | c | s | ts) "u">
<!ELEMENT   emph (%text;)
<!ELEMENT   graphic EMPTY>
<!ATTLIST   graphic
 graphname ENTITY #REQUIRED>
<!ELEMENT  xref  -  o EMPTY>
<!ATTLIST   xref
 xrefid IDREF #IMPLIED>

```

```
<report>
<title>Getting started with SGML
<chapter>
<title>The business challenge
<intro>
<para>With the ever-changing and growing global market, companies and
large organizations are searching for ways to become more viable and
competitive.  Downsizing and other cost-cutting measures demand more
efficient use of corporate resources.  One very important resource is
an organization's information.
<para>As part of the move toward integrated information management,
whole industries are developing and implementing standards for
exchanging technical information.  This report describes how one such
standard, the Standard Generalized Markup Language (SGML), works as
part of an overall information management strategy.
<graphic graphname=infoflow>
<chapter>
<title>Getting to know SGML
<intro>
<para>While SGML is a fairly recent technology, the use of
<emph>markup</emph> in computer-generated documents has existed for a
while.
<section shorttitle = "What is markup?">
<title>What is markup, or everything you always wanted to know about
document preparation but were afraid to ask?
<intro>
<para>Markup is everything in a document that is not content.  The
traditional meaning of markup is the manual <emph>marking</emph> up
of typewritten text to give instructions for a typesetter or
compositor about how to fit the text on a page and what typefaces to
use.  This kind of markup is known as <emph>procedural markup</emph>.
```

```
<topic topicid=top1>
<title>Procedural markup
<para>Most electronic publishing systems today use some form of
procedural markup.  Procedural markup codes are good for one
presentation of the information.
<topic topicid=top2>
<title>Generic markup
<para>Generic markup (also known as descriptive markup) describes the
<emph>purpose</emph> of the text in a document.  A basic concept of
generic markup is that the content of a document must be separate from
the style.  Generic markup allows for multiple presentations of the
information.
<topic topicid=top3>
<title>Drawbacks of procedural markup
<para>Industries involved in technical documentation increasingly
prefer generic over procedural markup schemes.  When a company changes
software or hardware systems, enormous data translation tasks arise,
often resulting in errors.
<section shorttitle = "What is SGML?">
<title>What <emph>is</emph> SGML in the grand scheme of the universe, anyway?
<intro>
<para>SGML defines a strict markup scheme with a syntax for defining
document data elements and an overall framework for marking up
documents.
<para>SGML can describe and create documents that are not dependent on
any hardware, software, formatter, or operating system.  Since SGML
documents conform to an international standard, they are portable.
<section shorttitle = "How does SGML work?">
<title>How is SGML and would you recommend it to your grandmother?
<intro>
<para>You can break a typical document into three layers: structure,
content, and style. SGML works by separating these three aspects and
deals mainly with the relationship between structure and content.
<topic topicid=top4>
<title>Structure
<para>At the heart of an SGML application is a file called the DTD, or
Document Type Definition. The DTD sets up the structure of a document,
much like a database schema describes the types of information it
handles.
<para>A database schema also defines the relationships between the
various types of data. Similarly, a DTD specifies <emph>rules</emph>
to help ensure documents have a consistent, logical structure.
<topic topicid=top5>
```

```
75 <title>Content
76 <para>Content is the information itself. The method for identifying
77 the information and its meaning within this framework is called
78 <emph>tagging</emph>. Tagging must
79 conform to the rules established in the DTD (see <xref xrefid=top4>).
80 <graphic graphname=tagexamp>
81 <topic topicid=top6>
82 <title>Style
83 <para>SGML does not standardize style or other processing methods for
84 information stored in SGML.
85 <chapter>
86 <title>Resources
87 <section>
88 <title>Conferences, tutorials, and training
89 <intro>
90 <para>The Graphic Communications Association (&gcalogo;) has been
91 instrumental in the development of SGML. GCA provides conferences,
92 tutorials, newsletters, and publication sales for both members and
93 non-members.
94 <para security = c>Exiled members of the former Soviet Union's secret
95 police, the KGB, have infiltrated the upper ranks of the GCA and are
96 planning the Final Revolution as soon as DSSSL is completed.
97
```

Desired Results:

1. Elements whose start-tag is on 6, 11, 20, 27, 34, 39, 46, 53, 56, 62, 67, 71, 76, 83, 90, 94
2. Elements whose start-tag is on 6, 11, 20, 27, 53, 56, 62, 90, 94
3. Elements whose start-tag is on 90, 94
4. Element whose start-tag is on 67
5. Element whose start-tag is on 94
6. Attribute value in start-tag on 23, 50, 59
7. Character after start-tag on 6, 20, 27, 53, 62, 90

8a. Elements whose start-tag is on 51, 60

8b. Element whose start-tag is on 60

9a. String on 21, 23, 24, 28(2), 31(2), 33, 35(2), 38, 39(2), 41, 42, 45, 47, 53, 54

9b. String on 21, 23, 24, 28, 31(2), 33, 35(2), 38, 39(2), 41, 42, 45, 47, 53, 54

10. Element whose start-tag is on 65
11. Element whose start-tag is on 80
12. Entity reference on 90
13. Given xref on line 79, element whose start-tag is on 75