Standardizing the Querying Process with SGML
The SQL DTD

Arijit Sengupta [1]

Keywords: Structured Query Language,Query processing,SGML Databases.

Abstract

One of the most exciting applications of SGML which has emerged in the recent years is its use in document databases. The structural information embedded in SGML documents makes it possible to query SGML documents and extract information in an automatic manner; however, this querying process has not been standardized. As a result, different SGML database implementations use their own query language syntax, thus making the migration from one system to another a difficult process. In the relational database domains, however, the query language SQL (Structured Query Language) has been a standard for over ten years and is universally used in most relational database systems. Although originally designed for relational databases, SQL is quite powerful for specifying complex queries in a relatively easy-to-understand syntax. With a small set of extensions to take advantage of the hierarchical structure of SGML, SQL can be easily adapted for use with SGML document databases [TAG-496].

The powerful "generalized" nature of SGML makes it easy to implement SQL as an SGML DTD (Document Type Definition) , so that queries can be expressed as document instances of the SQL DTD. Current SGML authors and users can write queries expressed in this DTD without learning a different language or using a separate editor. Moreover, because of the portable nature of SGML, these queries can be used in any SGML database system and can be converted to regular SQL for use in a relational or Object-Relational/Object-Oriented database system, if necessary. Databases that support the SQL DTD can also store the queries without any extra effort, and subsequently query them for inferring optimization parameters.

This paper presents a representative DTD for the SQL query language, with extensions for use with hierarchically structured documents. It also compares this language with languages proposed and implemented, including SDQL (Standard Document Query Language) — the query language in the DSSSL standard [DSSSL95]. This paper explains the advantages of using this language as a query language in document database systems and the necessity for standardizing the querying process in document databases. Finally, it discusses some implementation issues and complexity measures.

About the Authors

Arijit Sengupta
Arijit Sengupta [2] is a Ph.D. candidate in Computer Science at the Indiana University, Bloomington. He is also the Associate Instructor for the Information Systems sequence course in Computer Science and the system administrator for various SGML systems as well as other commercial and public-domain database systems in the Database Laboratory. His research is directed towards finding new methods for accessing the data from structured document databases, using query languages and interfaces.

(1)

1. Preface

The SGML standard is very suitable as a platform for document databases. SGML describes only the structure of the document, leaving the process of applying semantics to the document structure to the application that processes the document. Following this approach, database applications can make their own access methods for retrieving information from the document. As the popularity of document databases is expanding, an increasing number of database applications are being developed - all of which use their own method for information retrieval. There are many different products that require use of multiple interfaces, different types of style sheets, different types of query languages, different types of translation pseudo-languages, etc. This makes it difficult for users of such systems to migrate from one system to the other. A standard method of information retrieval is thus necessary. The SDQL language in the DSSSL (Document Style Semantics and Specification Language) [DSSSL95] standard provides an answer, but because of its low-level nature, it is not very suitable as a simple query language for document databases. For this purpose, we need something that is easier to use, that is less dependent on the document structure, and that involves less amount of learning from the users' point of view. This makes SQL an obvious choice as a query language for SGML document databases. SQL [SQL86] is very close to English, yet it can pose a complete set of solvable queries with various degrees of complexity.

It is not very difficult to apply SQL in the SGML domain. The basic SQL can be augmented with a relatively small set of

extensions as we proposed in <TAG> [TAG-496]. The generalized nature of SGML, however, makes it very easy to avoid learning another language just to search SGML documents. This is achieved through the use of an SGML-ization of the SQL grammar, with an SQL DTD. This way, users can write their queries directly in SGML, using whatever validating editor/parser they use, and send it to the database engine. For the database engine, presumably they already have a validating parser to start with, as they are dealing with SGML documents, so the parsing of the query should already be built-in. This paper introduces a query language for SGML database, using SGML itself! The DTD described here primarily covers the query language, and does not involve the data definition and manipulation powers of SQL. The goal of this paper is to show that a common method for information retrieval from document databases can be designed without leaving the domain of SGML. The DTD presented here is not complete, but deriving a complete SGML DTD from the SQL language specification is not a very difficult task, and the DTD discussed here shows exactly how it can be done. With the DDL (Data Definition Language) and DML (Data Manipulation Language) properties added, this method has the possibility of leading to a standard way of accessing SGML document databases.

(2)
2. Introduction

SQL and SGML have a lot in common. SQL [SQL86] and SGML [SGML86] were both standardized in 1986, and since then both have undergone changes and extensions. The SGML standard have been quite stable, but other standards like Hytime [HYT94], DSSSL [DSSSL95] have been built on top of SGML in order to give more functionality to SGML. SQL, however, has been revised twice as SQL-89 and SQL-92, and another revision leading towards an object-oriented query language SQL3 is in the works. SQL is primarily a query language, whose roots lie the formal relational algebra and relational calculus traditionally used with the relational database model. SQL, however, also has constructs for data definition and manipulation (creating data structures and adding or updating values in the data structures). SQL-92, also known as SQL2, has facilities for embedding SQL in a host programming language, for using triggers and event handling with active databases, and for making use of many other database-specific facilities. The forthcoming SQL3 will be a complete object-oriented programming language containing facilities for decisions, loops and other programming constructs. The core query language has not been modified much since the original SQL-86, and this core language is our primary area of interest in for accommodating with SGML.

This paper describes a DTD for writing SQL queries using SGML. The DTD is based on the SQL-86 standard [SQL86] and is constructed from the SQL syntax grammar described in [DATE89]. The DTD described here is not complete and is by no means an exact representative of the complete SQL query language, and only includes the relevant section of the query language for selecting documents or components of documents from a document database based on certain search conditions. Queries using this DTD are compared to the same queries using other languages like the standard SQL, the Pat Query language [OT94], the Metamorphosis query language [MM96], the SDQL (Standard Document Query Language) language in the DSSSL standard [DSSSL95] and the Sgrep query language [SGREP96].

(2.1)
2.1. What is SQL?

SQL [SQL86] is nothing new for people acquainted with relational databases. It is the standard query language for all relational database systems that have a language interface for manipulating the internal data. SQL is not a complete programming language, so certain queries are not possible to formulate using SQL. The queries that can be solved in SQL,however, constitute a core set that can be performed by a relational database engine in PTIME (polynomial time complexity) and LOGSPACE (logarithmic space complexity) [ABIT95]. This means that all the queries that one can formulate in SQL will never need an execution time greater than a polynomial of the input size and will never occupy temporary storage more than a logarithm of the input size. This is very important in relational databases, because (i) most queries of interest in relational databases will always be bound to this seemingly small set of queries, and (ii) a relational database implementing SQL will always be able to solve them in polynomial time.

Query languages that are complete programming languages (or, in formal terms, Turing-complete languages) can be used to formulate queries which require beyond polynomial time (for example, exponential time) for computation. Although this makes these languages more expressive than languages like SQL, there is never any guarantee that all queries written in such a language will be solvable in a reasonable time. That is the reason why we like to avoid Turing-complete languages. The SDQL language in the DSSSL standard [DSSSL95] falls in a complete language category. The core query language subset of SDQL is close to the type of query language that we are looking for, but (i) its level is too low and too

restricted requiring explicit navigation through the SGML parse tree, and (ii) it is not very intuitive for end-users, and is also not very easy to implement using a simple user-interface. SQL, however, can be quite easily implemented using a form-based interface, like those in most commercial database packages.

(2.2)
2.2. SQL in the SGML context

SQL was originally designed for relational databases, which is based on table-structured data. Because of the flat structure of tables, the relational model is unsuitable for modeling document databases. The primary reason for this is that modeling hierarchical document structures into flat relational tables requires too much fragmentation of the original data and loses most of the integrity of the original document. Since SQL is primarily a language designed for relational databases, it is also not suitable for querying document databases. Fortunately, a relatively small set of extensions to the core SQL language results in a language that is powerful enough for use with hierarchical document databases. Some recent research-oriented and commercial systems ([ABIT93]; [HOLST95]) have implemented SGML databases using a host Object-Oriented and Object-Relational database, and they use some of the features from the host SQL language for the purpose of query processing. Fortunately, for an SGML query processing system, it is not necessary to implement a complete object-oriented query language.

We introduced a small set of extensions to the core SQL in [TAG-496], [JASIS96]. In these papers, we show how the core SQL with some relatively simple extensions can be used to perform rather complex queries for document databases. In this paper, we take this work one step further - we incorporate this extended SQL language in an SGML DTD, that can be used directly by a common SGML processing system for implementing all the queries that are possible in direct SQL. In this way the query language stays within the domain of SGML - users do not have to learn a new language for writing their queries, and the implementers do not have to write a new parser for a new language.

(3)
3. The "SQL DTD"

The entire SQL-92 BNF (Backus-Naur Form) takes up more than 45 pages [MELT92] - of course most of it consists of interfaces for SQL with various programming languages, embedded SQL, triggers and other advanced features that are primarily targeted towards a relational database language. We are only interested in the querying capabilities of SQL, using a few extensions described in [TAG-496] and [JASIS96]. The three main extensions are:

1. Addition of complex column expressions to specify complex paths from the SGML root to the node of interest,
2. Addition of incomplete path expressions specifying only a source and a target node, leaving the path resolution to the search engine,
3. Selection of complex document structures using a pre-specified document type definition. This input DTD will likely consist of a special rearranging of the output structure from the query and restructure it as specified in the DTD.

(3.1)
3.1. Closure

We give a considerable importance to closure of the language. The main idea of closure is that once an operation has ended successfully, the result is in thes same domain as the input of the operation. We are dealing with SGML documents, so in our case it makes sense to designate SGML as this domain of closure. This implies that the input to the queries consists of one or more SGML documents, and the result of the queries must also consist of SGML documents, conforming to either the input DTDs or DTDs generated as a result of the query. In most cases, the resulting DTD can be constructed from portions of the input DTDs. In cases where more complex reorganization of the output is necessary, it is achieved using an output DTD template which is used to construct a DTD for the structured output.

The concept of closure is maintained not only in the query language but also in the interface in which the queries are executed. We proposed an interface that we termed QBT (Query By Templates) based on this closure idea in [JASIS96], in which the interface for specifying queries uses the same template that is used for displaying the outputs. Another use of closure is to enable complex queries by feeding the outputs of certain queries back as inputs of other queries, thus enabling the use of queries as subqueries. This is easy to achieve if the output of queries is in the same format as the inputs. This also enables saving outputs of certain queries as temporary views that can be used as inputs to other queries.

The following two sections show the SQL DTD and a table that describes all the generic identifiers of this DTD.

(3.2)

3.2. The DTD for the SQL Query Language

```
<!-- ***************** SQL Document Type Definition ***************  -->
<!-- Suggested public id: -ANSI X3H2//DTD SQL//EN -->

<!-- Defined Parameter Entities -->
<!ENTITY % Negation "(ASIS | NOT) ASIS">
<!ENTITY % Comparison "(EQUAL | NEQ | LESS | GREATER | LEQ | GREATEQ) EQUAL">
<!ENTITY % Aggregate  "AVG | MAX | MIN | SUM | COUNT">

<!ELEMENT SQL           O O (select, (union, all? ,select)*)>
<!ELEMENT (union|all)   - O EMPTY>
<!ELEMENT select        - O (output, qry-body)>
<!ATTLIST select        selcrit(ALL |DISTINCT)  ALL>
<!ELEMENT output        O O (scalar+ | all | dtd-exp)>
<!ELEMENT dtd-exp       - O (#PCDATA) -- will possibly need change -->
<!ELEMENT qry-body      O O (from, where?, group-by?, having?)>
<!ELEMENT from          - O (db+)>
<!ELEMENT db            - O (#PCDATA)>
<!ATTLIST db            alias  ID  #IMPLIED>
<!ELEMENT where         - O (cond)>
<!ELEMENT group-by      - O (col+)>
<!ELEMENT having        - O (cond)>
<!ELEMENT cond          O O (predicat | (left, logic, right))>
<!ATTLIST cond          neg %Negation; -- (ASIS |NOT) ASIS-->
<!ELEMENT (left|right)  O O (predicat | cond)>
<!ELEMENT logic         - O EMPTY>
<!ATTLIST logic         oper   (AND | OR | FOLLBY | NEAR) AND>
<!ELEMENT predicat      - O (compare | between | like | testnull |
                             in | univqnt | exists)>
<!ELEMENT compare       - O (scalar, (scalar | select))>
<!ATTLIST compare       oper   %Comparison;>
<!ELEMENT between       - O (scalar, scalar, scalar)>
<!ATTLIST between       neg %Negation;>
<!ELEMENT like          - O (col, colatom, escatom?)>
<!ATTLIST like          neg %Negation;>
<!ELEMENT (colatom | escatom) - O (#PCDATA)>
<!ELEMENT testnull   - O (col)>
<!ATTLIST testnull      neg %Negation;>
<!ELEMENT in            - O (scalar, (col | atom+))>
<!ATTLIST in            neg %Negation;>
<!ELEMENT univqnt       - O (scalar, select)>
<!ATTLIST univqnt       neg %Negation;
                        oper   %Comparison;
                        type   (ALL | ANY | SOME) ALL>
<!ELEMENT exists        - O (select)>
<!ATTLIST exists        neg %Negation;>

<!-- The content of scalar is incomplete compared to SQL
    Arithmetic operations are intentionally left out to
    keep the DTD simple, but could be added if necessary -->

<!ELEMENT scalar        O O (atom | col | function )>
<!ELEMENT atom          - O (#PCDATA)>
<!ELEMENT function      - O (countall | distfunc | allfunc | attfunc)>
<!ELEMENT countall      - O EMPTY>
<!ELEMENT distfunc      - O (col)>
<!ATTLIST distfunc      oper   (%Aggregate;) COUNT>
<!ELEMENT allfunc       - O (all?, scalar)>
<!ATTLIST allfunc       oper   (%Aggregate;) COUNT>
<!ELEMENT attfunc       - O (col,attrib)>
<!ELEMENT attrib        - O (#PCDATA)>
<!ATTLIST attfunc       oper   (%Aggregate; | NONE) NONE>
```

```
<!ELEMENT col           O O (source?, thru*, target)>
<!ELEMENT (source|thru|target)  - O (#PCDATA)>
<!ATTLIST source        refdb   IDREF  #CONREF>
```

(3.3)

3.3. Description of the DTD

A preliminary version of the DTD for the query section only is given in the last section. This DTD implements all the extensions described in [JASIS96]. The complex selection using an external DTD could be implemented more effectively, but currently the DTD specifies only a system file name, which the processing application needs to load and process. The complex path expressions are implemented using the complex column element `<COL>`, (having an optional source position which is linked with a database) and a number of intermediate elements that can be used to denote a particular path from the top level to the desired element. In case of multiple paths from a source to a target, they can be explicitly stated or can be left for the processing application to select. Ideally, the processing application will try to find all paths from the source to a target. The examples in the following sections will show how this is used. The following table shows all the generic identifiers (GIs) used in the DTD, with descriptions for each of them.

Table 1. Description of the GIs in SQL DTD

GI in the DTD	Description
SQL	Top level GI in the DTD. Optional. Contains one SQL query statement.
union	Signifies the union operation involving the results of two or more select statements.
all	Used as a modifier of the union operation and as a replacement for the "select *" construct of SQL
select	The root of a single select statement. A select statement can be used as a sub-query in various places: as a component in the union of multiple queries, as a scalar output in a comparison, and for quantification, either universal or existential (for all/there exists). The attribute selcrit (selection criterion) can be distinct or all depending on whether duplicate removal is to be performed or not.
output	The output from the query - specified as a list of complex columns, all, or a restructuring DTD
dtd-exp	The DTD expression - currently just the name of the DTD. Some constraints and mappings may be added in future revisions
qry-body	The actual body of the query. Optional
from	The "from" specifier - specifies the input to the query. consists of one or more databases.
db	A database to provide the input to the query. The attribute alias is a short name used for future reference by the complex columns.
where	The condition clause.
group-by	The group-by clause - specifies which columns to group the results by. Must be a subset of the columns in the output clause.
having	Specifies further restrictions in group-by columns.
cond	The conditions for querying. Can be either a predicate or a logical binary expression combined using a logical operator. Can be either true or false. The attribute Neg specifies if the condition is to be negated.
left	The left side of a logical operator. can be either a predicate or another conditional expression.
right	The right side of a logical operator.
logic	The logical operation. Allowed operations are: AND, OR, FOLLBY (followed by) and NEAR. The last two operations are added on top of normal SQL to support proximity queries.
predicat	A relational predicate - can be one of seven operations as given in the DTD. The result is always true or false. All operations can be negated.
compare	The comparison operation. Compares a scalar value with another scalar value or the result of a subquery that returns a scalar value.
between	The between operation performs range queries - decides if the value of a scalar expression is between two different scalar expression
like	The like operation is basically a regular expression match. A complex column is compared with a regular expression formed with a column atom and an escape atom.
colatom	The regular expression. In SQL, the regular expressions use the characters % and _ for zero or more characters and exactly one character respectively.
escatom	Specifies an escape character, in case one of the characters % and _ needs to be used as a data character.
testnull	A null test - to check if a complex column contains a null value
in	An IN expression: tests if the value of a scalar expression is in a set of atoms or a set returned by a select subquery
univqnt	A universal quantifier - can be one of ALL, ANY or SOME, and the comparison can be one of the several comparison operations.
exists	An existential quantifier - determines if the result of a subquery exists.
scalar	A scalar expression - can be a single value or a set of values. Can be a constant (atomic) value, a complex column, or a function of them.
atom	A constant value - normally a character or numeric value.
function	Various aggregate functions allowed in SQL.
countall	The aggregate function count(*) counts the number of tuples (or instances of complex columns) returned by the query, without duplicate elimination
distfunc	Distinct functions - computes aggregate functions on specific complex columns with duplicate elimination
allfunc	All functions - computes aggregate functions on scalar expressions without duplicate elimination
attfunc	Attribute functions - computes functions on attribute values of columns
col	A complex column - targets one or a set of GIs of the underlying database.
source	Source for the complex column - has to be one of the databases in the repository
thru	Path from the source to the target - needs to be a GI of the database
target	The end target of the column for the operation.

(4)

4. The SQL DTD by Example

This section describes a sample scenario demonstrating how queries are formulated using the SQL DTD. This sample DTD is taken directly from [TAG-496], with the addition of one "pages" attribute for the books, which is probably useless in normal circumstances but is included to demonstrate attribute access features of the SQL DTD. Please refer to the above article for the queries using the extended SQL in its native language form. In this paper, although we will examine some examples in the regular SQL, most will use the SQL DTD that we are proposing. The following is the "books" DTD:

```
<!ELEMENT books                O O (book*)>
<!ELEMENT book                 - - (header, body)>
<!ATTLIST book  pages   NUMBER #implied -- possibly redundant -->
<!ELEMENT header               - - (bibl,other)>
<!ELEMENT bibl                 - O (name,(author)+,(editor)*,
   year,publisher)>
<!ELEMENT other                - O (ISBN,price)>
<!ELEMENT (ISBN|price)         - - (#PCDATA)>
<!ELEMENT (name|year)          - - (#PCDATA)>
<!ELEMENT (editor|publisher)   - - (#PCDATA)>
<!ELEMENT author               - O (name,affil)>
<!ELEMENT affil                - - (#PCDATA)>
<!ELEMENT body                 - O (chapter+)>
<!ELEMENT chapter              - O (chtitle,(section)*) +(P)>
<!ELEMENT section              - O (sectitle,P+)>
<!ELEMENT (P|chtitle|sectitle) - O (#PCDATA)>
```

Queries using SQL have five main sections, shown in the following two lines taken from the SQL DTD. The first line shows the syntax of the select statement in SQL, which has an output specification and a query body. The output specification can be a list of columns, an "all" specifier, that specifies all the items in the database, or a DTD specifier that gives the name of the DTD which should restructure the output.

```
<!ELEMENT select  - O (output, qry-body)>
<!ELEMENT qry-body   O O (from, where?, group-by?, having?)>
```

The query body contains four sections: (i) a "from" section in which the source for the data is specified, (ii) a "where" section, in which the conditions of the selection are specified, (iii) a "group-by" section, in which the ordering information is specified, and (iv) a "having" section that further constrains the ordering of the output. A sample SQL query, using this SQL DTD, will be similar to the one shown in the following table. This query is equivalent to the search: "Find all the books that are written by Charles Goldfarb".

The query in the following table shows how a simple query in SQL will be coded in SGML using the SQL DTD. Of course there is some of overhead of the SGML tags, some of which can be minimized using the SHORTTAG and DATATAG features of SGML. The above query uses OMITTAG and contains three important sections: *select, from, and where*. The `<select>`" section contains a scalar value which, in this case, is just a complex column with the target book. The `<from>` section contains the source for the query data which is simply the database "Books" (which is given the alias B1 for use with subsequent column references). The `<where>` section contains a comparison predicate, which compares a scalar value containing the column "name" with a scalar value containing the atom (or constant value) "Charles Goldfarb".

Table 2. A sample query in SQL and the equivalent using SQL DTD

Regular SQL	**SQL DTD**
SELECT book **FROM Books B1** **WHERE B1..author.name =** **"Charles Goldfarb"**	<sql><select> <scalar><col><target>book <from><db alias="B1">Books <where> <cond><predicat><compare> <scalar><col><source refdb="B1"><thru>author <target>name <scalar><atom>"Charles Goldfarb" </select></sql>

The complex de-referencing of the column "name" is implemented in the above query. In the SQL syntax, it can simply

be stated as "B1..author.name" (*i.e.*, start from the database B1, take any path to "author", and go to "name" in the content of "author"). Using the SQL DTD, this leads to a source node referring to the database aliased by "B1", a path that goes through the element "author", and a target "name".

(4.1)

4.1. Comparison of SQL DTD with other query languages

In this section, we will take the same six example queries stated in [TAG-496] and show the queries for these six examples using the query languages from the following systems. Note that some of the queries have not been verified; they have been either contributed by one of the responders to our appeal for contributions on the comp.text.sgml newsgroup, or we have constructed them from documentation for the specific system, without actually testing them in the respective system, since the system was not available for our use. The systems considered in this paper are the following:

1. Open Text 5.0 and the Pat query language from Open Text corporation, Waterloo, Canada [OT94]
2. Sgrep 0.99 from Computer Science, Univ. of Helsinki [SGREP96]
3. Metamorphosis 2.13 from Ovidius corporation, Berlin, Germany[MM96]
4. SDQL in the DSSSL Standard [DSSSL95]
5. The SQL DTD

The actual SQL queries are not reproduced here. Interested readers should see [TAG-496] for the queries in SQL or the SQL DTD query for an alternative.

(4.1.1)

4.1.1. Find the books that have more than 200 pages and cost more than $20.

Table 3. Table showing the queries corresponding to Query 4.1.1.

System	Query

Pat	`(region book including`
	`  (region page including (range "0200".."9999")) ^`
	`(region book including`
	`  (region price including (range "0020".."9999"))`

Sgrep	Not possible as such, since requires giving an interpretation to some portions of the text. However, it is possible to find the books having exactly 200 pages and costing exactly $20:
	`NAMED_ELEMS(book) containing`
	`  (NAMED_ATTRVAL(pages) equal "200" in NAMED_STAG(book))`
	`  containing (NAMED_CONTENTS(price) equal "$20")`

Metamorphosis	`source.child[child[?header]`
	`  .child[?other]`
	`  .child[?price].data > 20 &`
	`  child[?header]`
	`  .child[?other]`
	`  .child[?pages].data > 200`
	`  ]`

SDQL	`(setq all-books .....) ;; Create a node-list containing all the books`
	`(define (equal200pages snl)`
	`  (if (= (attribute-string 'pages snl) 200) #t #f))`
	`(define (greater20 snl)`
	`  (if (= (data (child 'price snl)) 20) #t #f))`
	`(nodelist-intersect`
	`  (get-ancestors 'book  (node-list-filter equal200pages all-books))`
	`  (get-ancestors 'book  (node-list-filter greater20 all-books)))`

SQL DTD	`<sql><select><all>`
	`<from><db alias="B1">Books`
	`<where><cond><left><predicat><compare GREATER>`
	`<scalar><function><attfunc><col><target>book<attrib>pages`
	`  <scalar><atom>200  <logic AND>`
	`  <right><predicat><compare GREATER>`
	`  <scalar><col><target>price`
	`  <scalar><atom>20`
	`</select></sql>`

(4.1.2)

4.1.2. Find the books written by an author who has edited another book.

Table 4. Table showing the queries corresponding to Query 4.1.2.

System	Query
Pat	Join operation is not supported.
Sgrep	Not possible, requires joins.

Metamorphosis	`distinct(source.descendant[?author & *n:=child[?name].data]`
	`.ancestor[?book]`
	`.(left|right)[child[?header]  // left| right`
	`.child[?bibl]`
	`.child[?editor].data==*n`
	`]`
	`.(*n)`
	`).(*p:=this)`
	`&`
	`source.child[descendant[?author]`
	`.child[?name].data==*p)];`

SDQL	Join operation is not possible without writing a complete join algorithm using lisp.

SQL DTD	`<sql><select><scalar><col><source refdb="B1"><target>book`
	`<from><db alias="B1">Books<db alias="B2">Books`
	`<where><cond><left><predicat><compare NEQ>`
	`<scalar><col><source refdb="B1"><thru>bibl<target>name`
	`<scalar><col><source refdb="B2"><thru>bibl<target>name`
	`<logic AND>`
	`<right><predicat><compare>`
	`<scalar><col><source refdb="B1"><thru>author<target>name`
	`<scalar><col><source refdb="B2"><target>editor`
	`</select></sql>`

(4.1.3)

4.1.3. Find the total number of chapters in "The SGML Handbook"

Table 5. Table showing the queries corresponding to Query 4.1.3.

System	Query

Pat	`region chapter within (region book including (`
	`region bibl including (`
	`region name including "The SGML Handbook")))`

Sgrep	`sgrep -c 'NAMED_ELEMS(chapter) in \`
	`(NAMED_ELEMS(book) containing \`
	`(NAMED_CONTENTS(name) equal "The SGML Handbook"))'`

Metamorphosis	`count(source.child[child[?header]`
	`.child[?bib]`
	`.child[?name].data=="The SGML Handbook"`
	`].child[?body].child[?chapter])`

SDQL	`(define (giname? name snl) (if (streq? (gi snl) name) #t #f))`
	`(define (child-withgi gi snl)`
	`(node-list-filter (lambda name (giname? name)) snl))`
	`(define (dataval? ... ))  ;; similar as above for data`
	`(define (handbook? snl) (if (streq? (data snl) "The SGML Handbook")`
	`#t #f))`
	`(node-list-count (child-withgi 'chapter`
	`(get-ancestors 'book`
	`(node-list-filter handbook? (child-withgi 'title all-books)))))`

SQL DTD	`<sql><select><scalar>`
	`<function><allfunc COUNT><col><target>chapter`
	`<from><db>Books`
	`<where><predicat><compare>`
	`<scalar><col><thru>bibl<target>name`
	`<scalar><atom>"The SGML Handbook"`
	`</select></sql>`

(4.1.4)

4.1.4. Find the authors all of whose books cost more than $20.

Table 6. Table showing the queries corresponding to Query 4.1.4.

System	Query
Pat	Quantification operation not supported.
Sgrep	Not Possible, requires and numeric interpretation of text content.

Metamorphosis	`source.child`
	`  .child[?header & child[?other]`
	`  .child[?price].data>20]`
	`  .child[?bibl].child[?author]`

SDQL	`(setq all-authors ...) ;; Find the list of all authors`
	`(define (allgt20? snl)`
	`  (node-list every?`
	`  ((lambda (sn) (if (> (child-withgi 'price sn)`
	`  20) #t #f)) (get-ancestor 'book snl))))`
	`(node-list-filter allgt20? all-authors)`

SQL DTD	`<sql><select><scalar><col><source refdb=B1><target>author`
	`<from><db alias="B1">Books`
	`<where><predicat><exists NOT>`
	`<select><all>`
	`<from><db alias="B2">Books`
	`<where><cond><left><predicat><compare>`
	`<scalar><col><source refdb=B1><thru>author<target>name`
	`<scalar><col><source refdb=B2><thru>author<target>name`
	`<logic AND>`
	`<right><predicat><compare LESS>`
	`<scalar><col><source refdb="B2"><target>price`
	`<scalar><atom>20`
	`</select></sql>`

(4.1.5)

4.1.5. Find all books written by the author of "Practical SGML"

Table 7. Table showing the queries corresponding to Query 4.1.5.

System	Query
Pat	Join Operation not supported.
Sgrep	Not possible, requires joins.

Metamorphosis	`source.child[?header]`
	`  .child[?bibl]`
	`  .child[?name & data=="Practical SGML"]`
	`  .child[?author].(*p:=(child[?name].data))`
	`  &`
	`  source.child[?book &`
	`  .child[?header]`
	`  .child[?bibl]`
	`  .child[?author]`
	`  .child[?name & data==*p]`
	`  ];`

SDQL	Join Operation not possible without writing a complete join algorithm using lisp.

SQL DTD	`<sql><select><all>`
	`<from><db>Books`
	`<where><predicat><compare EQUAL>`
	`<scalar><col><thru>author<target>name`
	`<select><scalar><col><thru>author<target>name`
	`<from><db>Books`
	`<where><predicat><compare>`
	`<scalar><col><thru>bibl<target>name`
	`<scalar><atom>"Practical SGML"`
	`</select></select></sql>`

(4.1.6)

4.1.6. Find the authors who have not edited any book.

Table 8. Table showing the queries corresponding to Query 4.1.6.

System	Query
Pat	Join operation not supported.
Sgrep	Not possible, requires Joins.

Metamorphosis	`(*p:=distinct(source.child[?header].child[?bibl].child[?editor].data))`
	`&`
	`source.child[?book]`
	`.child[?header]`
	`.child[?bibl]`
	`.child[?author & *p~==child[?name].data]`
	`;`

SDQL	Join Operation not possible without writing a complete join algorithm using lisp.

SQL DTD	`<sql><select><scalar><col><thru>author<target>name`
	`<from><db alias="B1">Books`
	`<where><predicat><exists NOT>`
	`<select><all>`
	`<from><db alias="B2">Books`
	`<where><predicat><compare>`
	`<scalar><col><source refdb=B2><target>editor`
	`<scalar><col><source refdb=B1><thru>author<target>name`
	`</select></select></sql>`

(5)

5. Motivations for a standard query language

Although SGML has been a standard for a decade now, there has been little attempt at developing standards for using SGML as a language for describing database schema and for querying the data in the database. The SDQL language in DSSSL [DSSSL95] does provide an answer. Although SDQL is well suited for manipulating formatting styles of the document, its level is too low to be of use as a query language. SQL has been the query language of choice in relational databases, and there are many justifications for its use as the query language in SGML databases. The selection becomes obvious when we use the DTD version of SQL. The DTD presented here is just a quick implementation to show what can be done and is by no means an end in itself. Designing a DTD with appropriate SGML declaration enabling DATATAG and SHORTTAG minimization will make the language almost as simple as its non-SGML counterpart, and at the same time, keep it completely portable. Some of the motivations behind having this SQL DTD as a standard query language in SGML are as follows:

- It is powerful enough to specify a reasonably complete set of queries that can be evaluated within practical time limits.
- It is easy to understand and visualize, especially when using it in tandem with an SGML-aware authoring system.
- It conforms to the idea of closure. The query language is itself in SGML while using SGML documents as input and produces SGML documents as output.
- Since the queries are themselves in SGML format, they can be stored in the database. These SGML queries can be subsequently queried for determining optimization and other tuning parameters for the database system. Also, in database systems, optimization is normally performed by constructing the query tree and applying optimization rules to the tree to reduce the depth and span of the query tree. This is achieved by moving the expensive operations higher up in the tree, so that they are computed with input that is much smaller in size compared to the original input. Using the SGML version, the parse tree of the query can be easily used as a starting point for applying transformation rules to construct more optimized queries.
- Probably the biggest advantage of using such queries is portability. Since the queries are in a generalized format, they can be easily converted to regular SQL for use in relational databases or into QBE for use in a visual query formulation. The queries are also general enough to be converted into other proprietary query languages and into the SDQL language for the purposes of style specification and transformation. In fact, a system implementing SQL needs to first convert the query into a procedural format, and the procedures described in SDQL can be used for such purposes.

(6)

6. Implementation Issues

A problem with very general languages is implementation difficulty. In contrast, we can formally prove that all queries

written in SQL can be solved in PTIME (polynomial complexity for time) and LOGSPACE (logarithmic complexity for temporary space) [ABIT95]. Our current work [SENG96]
[3] is aimed at showing that PTIME implementations of SQL queries for structured documents are possible using index-based approaches.

(6.1)
6.1. Language Implementation

Implementations of document databases can be roughly grouped in two categories - (i) mapping approaches which map the SGML schema into an Object-Oriented or Object-Relational schema and convert the document accordingly (such as in [ABIT93], [HOLST95]) and (ii) index-based approaches in which indices are built on top of the original documents to expedite searches (such as in [OT94]). Index-based approaches have the advantage of keeping the authoring separate from the database counterpart, since the original documents are kept intact and the indices can be rebuilt when the documents are updated. Although these updates are expensive, in most cases, the indices can be incrementally updated. On the other hand, the mapping approach of converting documents into totally different database systems often require that the authoring is also built into such systems. Users do not have access to the original raw SGML document, but can only export the internal database content in SGML form if necessary.

SQL-like queries are already built into systems that map documents into an Object-Relational or Object-Oriented database, since the host database systems usually support SQL queries. Our approach is to show that such queries can also be supported in index-based systems, using some specialized indices for certain functions and operations. The "join" operation, which requires the use of a variable (which is usually missing in the index-based approaches) is the first operation that we implement, and in [SENG96] we show that joins can be efficiently implemented using indices on the GI(s) used in the "join" operation. Implementations strategies for other first-order operations in SQL will be investigated subsequently.

(6.2)
6.2. Visual Implementation

We mentioned earlier that the simplest visual implementation of this query language will be the familiar authoring interface for SGML authors. We need to keep in mind, however, that all users searching for information in SGML documents are not authors, so it is necessary to consider other visual alternatives. A significant portion of our current efforts is towards the development of a simple yet powerful visual approach to interfacing query languages. This visual approach is based on the Query By Example (QBE) language for relational databases. An implementation of this method for structured documents, which we call Query By Templates (QBT) [JASIS96], is available on-line [QBT96].

(7)
7. Conclusion and Discussion

In this paper, we have presented an elegant way for querying SGML databases. We have demonstrated here that not only is this method useful, but also reasonably simple to implement. Most current commercial systems will simply require a front-end to the current language to accommodate SQL. Making a consistent query language for structured documents will also make the process more uniform for many people, including both the users and the developers. For this purpose, a complete programming language with a number of predefined procedures is not necessary. It would make much more sense to use the currently available technology and know-how on query languages and to apply these lessons for querying SGML documents. The common standard does not have to be SQL, but it definitely is a very appropriate solution. This will be a first step toward a universal method for searching digital libraries across platforms - maybe a step towards Nelson's Xanadu!

Acknowledgments
This paper would be incomplete unless I acknowledge the persons who helped me compile the solutions to the sample queries in the various query languages. I would like to thank Bernhard Weichel and Hans Kerkman of the Ovidius Corpn. for their solutions to the queries using Metamorphosis, and Klaus Fenchel for his help and support in communicating the solutions to me. I would also like to thank Pekka Kilpelainen, Computer Science, University of Helsinki for his solutions using Sgrep. Thanks also goes to Ryan Sweet of the Open Text Corporation, for verifying the queries using Open Text 5.0. I would also like to thank Bob Ducharme from ACM, Gordon V. Cormack from the Multitext corporation, and Art

Pollard for their support in getting answers to my questions. Last but not least, I would like to thank my colleague Mehmet Dalkilic for his help with the lisp codes used for the SDQL language.

References

[ABIT93] Serge Abiteboul, Sophie Cluet, Tova Milo. "Querying and Updating the File". Proceedings, 19th Intl. Conference on Very Large Databases, 73-84, 1993

[ABIT95] Serge Abiteboul, Richard Hull, Victor Vianu. "Foundations of Databases". Reading, Mass. Addison-Wesley, c1995

[DATE89] C. J. Date. "A Guide to the SQL Standard" - second edition. Addison Wesley Publishing Company, 1989

[DSSSL95] ISO/IEC DIS 10179.2. "Document Style Semantics and Specification Language (DSSSL)". Working Draft, 1995.

[HOLST95] Sebastian Holst. "Database Evolution: the View From Over Here (A Document-centric perspective)". Proceedings of the SGML '95 Conference, December 1995.

[HYT94] ISO/IEC 10744, in DeRose(1994): Steven J. Derose, David G. Durand. "Making Hypermedia Work: A User's Guide to HyTime". Kluwer Academic. 1994.

[JASIS96] Arijit Sengupta and Andrew Dillon. "Extending SGML with database functions: A methodological overview." To appear in the Journal of the American Society for Information Science (JASIS) special issue on Structured Information/Standards for Document Architectures; August, 1996.

[MELT92] Jim Melton and Alan R. Simon. "Understanding the New SQL: A Complete Guide". Morgan Kaufmann Publishers, 1992.

[MM96] Ovidius Corporation. "The Metamorphosis Manual". Available online at http://www.ovidius.com/mmmanual/toc.html. 1996

[OT94] Open Text Corporation. Open Text 5.0 Software and Reference Manuals. 1994.

[QBT96] Arijit Sengupta, Andrew Dillon and Shawn P. Morgan. An Implementation of QBT as the SGML Query Language. Demonstration available at http://blesmol.cs.indiana.edu:8080/projects/SGMLQuery.

[SENG96] Arijit Sengupta and Dirk Van Gucht. A PTIME Query Language for Structured Document Databases. Work In Progress, August 1996

[SGML86] ISO 8879. "Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML)", 1986.

[SGREP96] Jani Jaakkola and Pekka Kilpeläinen. "The Sgrep online manual". Available at http://www.cs.helsinki.fi/~jjaakkol/sgrepman.html. 1996.

[SQL86] ANSI X3.135-1986. "Information Technology - Database Languages - Structured Query Language (SQL)". American National Standards Institute. New York, 1986.

[TAG-496] Arijit Sengupta. "Demand more from your SGML database! Bringing SQL under the SGML limelight." in <TAG> The SGML Newsletter; 9(4); pages 1-7; April, 1996.

[1]Indiana University, Computer Science Dept.
Lindley Hall 215
Bloomington, IN 47405
USA
Phone: 812 855 4318
Fax: 812 855 4829
E-mail: asengupt@indiana.edu
WWW: http://www.cs.indiana.edu/hyplan/asengupt.html

[2]Partially supported by US Dept. of Education award number P200A502367 and NSF Research and Infrastructure grant, award number NSF

CDA-9303189.

[3]Work in progress - a copy of a preliminary version of this work can be obtained by contacting the author.