

DSQL - an SQL for structured documents

Arijit Sengupta¹ and Mehmet Dalkilic²

¹ Department of Accounting and Information Systems

Kelley School of Business

Indiana University

Bloomington, IN 47405, USA

asengupt@indiana.edu

<http://www.kelley.iu.edu/asengupt/>

² School of Informatics

Indiana University

Bloomington, IN 47405, USA

dalkilic@indiana.edu

Abstract. SQL has been the result of years of query language research, and has many desirable properties. We introduce DSQL - a language that is based on a theoretical foundation of a declarative language (document calculus) and an equivalent procedural language (document algebra). The outcome of this design is a language that looks and feels like SQL, yet is powerful enough to perform a vast range of queries on structured documents (currently focused on XML). The design of the language is independent of document formats, so the capability of the language will not change if the underlying markup language were changed. In spite of its familiarity and simplicity, we show that this language has many desirable properties, and is a good candidate for a viable query language for XML. This paper presents a canonical form of DSQL, showing only the properties of the language that affect the complexity of the language. Remarkably, SQL = core DSQL for flat input and outputs.

1 Introduction

Structured Query Language (SQL) [1] has been an ANSI standard since 1986. The original proposal of SQL was put forward in 1976 as SEQUEL 2 [2] and subsequently renamed [3] as SQL. It then underwent critical human-factors analyses—standardization following an entire decade later. And, some twenty-five years later, SQL has become the de facto standard database query language whose mere mention is virtually synonymous with relational database. Certainly one aspect of SQL’s success has been the success of the relational model, introduced five years earlier by Codd [4]. Until that time, the state of affairs of database was in disarray: too many platforms, models (the most common being network and hierarchical), and most problematic, query languages that depended *directly* upon the architecture of the model. The relational model freed the user from having to “navigate” their models—*data independence* was achieved. The significance of this cannot be over emphasized. For the first time, a user was freed

from having to “program” a query. What came about was a *declarative* query language, SQL, that allowed the user to specify *what* was wanted instead of *how*. Another critical factor in SQL’s success has been its standardization. Standardization provides a single voice that can be listened to and understood by academicians and vendors alike. But SQL’s story is not typical of standardized languages—there are many more languages that are discarded and forgotten soon after standardization. A particularly relevant example in the context of XML is DSSSL (ISO 10179)[5] which included a query language component called Standard Document Query Language (SDQL). In these turbulent times of burgeoning XML standards, reflecting on why and how SQL has become so ubiquitous is instructive and, perhaps, leads to better success in query languages for XML.

We make four critical observations. First, SQL gives data independence, freeing users from having to know the underlying architecture of the model. For example, in a network model if the model changes in some aspect, a query must change as well, since the query is really a procedural account of what needs to occur to retrieve data. An SQL query, on the other hand, does not have to change, since it is independent of how the database is implemented.

Second, SQL (like HTML) is simple and can be used by non-programmers and programmers alike. It is this very accessibility that has made SQL (and HTML) so ubiquitous. SQL has been compared with other languages in the literature [6–9]. Although many of these works have shown some potential flaws in SQL, most of the findings indicate that, compared to other textual languages, SQL is better for simpler tasks. Such research has provided insight into enhancing SQL. The original SQL (and its canonical form) has allowed the standard to be updated with many features, yet the simple canonical form of SQL is the most understood and widely accepted.

Third, SQL reflects well-known and established query language design principles. that have guided much of the database query language research, some of which (adapted from Ramakrishnan, [10]) are as follows: (1) Query languages should *not* be programming languages¹ A query language is meant to be efficient and effective at retrieving information—the accepted tradeoff being the inability to retrieve some kinds of information. A programming language can, after all, result in a non-terminating computation. A query language cannot. Often the solution to more expressiveness is to embed the query language in a host programming language, like Java, C, or C++. One of the most challenging problems in query language research, in fact, is observing how expressivity is enhanced by adding, say, different kinds of looping; (2) Query languages, *wrt* the previous statement, then, are not intended to be used for complex computations. (3) Query languages support easy, efficient access to large data sets. The prime reason for their existence, after all, is to retrieve information.

Fourth, SQL has a formal foundation and is not a collection of ad hoc query constructs. A formal foundation certainly leads to fewer semantic problems, and additionally, has allowed SQL to be optimized.

¹ Turing-complete languages; these include some kind of count operation, looping, and branching.

There is a precedence for using SQL, *e.g.*, [11, 12], demonstrating various levels of success in their respective fields by relying on the familiarity of SQL. One of the most well-known examples is OQL (Object Query Language) [13, 14], a query language for Object-Oriented database systems that is extension of SQL capable of handling the nuances of the Object-Oriented database model.

We demonstrate in this paper that the language SQL itself along with its theoretical basis is already an excellent fit for the domain of structured document querying. With this intent, we present Document SQL (DSQL), an SQL as it applies to the structured document domain. DSQL is just as well-suited for XML. Recalling our critical observations, DSQL remains true to all these. DSQL does not explicitly require the user to “navigate” through the architecture of the data. DSQL is simple and not a programming language. Furthermore, given the user knows SQL, learning DSQL is relatively an easier task than learning a completely new query language. Finally, DSQL has a strong formal foundation.

Like SQL, DSQL has both declarative and procedural counterparts, the document calculus (DC) and document algebra (DA), respectively. These possess essentially the same constructs of the relational calculus and algebra with some refinements in the semantics to apply to the document world *e.g.* paths. DC and DA satisfy the same important properties such as equivalence of DC and DA, PTIME, closure and safety. Remarkably, if the underlying document structures are flat *i.e.*, XML representations of standard relational tables, then SQL queries are the same as DSQL queries.

The following are the main goals for this work:

- Evaluation of SQL as a query language for XML
- Design of a query language that is based on strong and well-known and well-accepted formal semantics
- Design of a query language that is independent of an underlying markup language in its canonical form
- Design of a query language without relying on underlying data format, thus exhibiting data independence.
- Design of a query language for complex structures that is no different from SQL when the inputs and outputs are flat.
- Evaluation of the applicability of well-known query language design principles (low complexity, closure, non Turing completeness)
- Potential human factors analysis of query language design in the domain of document query processing.

The remainder of this work is as follows. The next section provides background and discusses related work. We then discuss general DSQL syntax and give intuitive explanations about its semantics. We then present examples of DSQL (and its XQuery equivalent) for some sample queries as specified in the W3C XQuery use case that are independent of XML-specific issues such as namespaces. We then highlight the DC and DA, providing a flavor of each. We then present important properties about the DC and DA. We conclude with a summary and future work.

2 Background and Related Languages

XML has been widely recognized as the next generation of markup language for the Internet. Government institutions, academic institutions, and private industry are all using XML to represent their organizational data in XML for the purpose of presentation as well as interchange of information. As the concept of platform-independent representation and the independence of information and layout are becoming more relevant, the importance of using the information in the XML data for search and retrieval purposes is also being felt. The World Wide Web Consortium² is currently involved in an effort towards a query language XQuery extract information from XML documents. A brief summary on XQuery and its related languages will provide a sufficient background for this paper. XQuery is presented as a series of related W3C Recommendations:

- XQuery 1.0 (<http://www.w3.org/TR/xquery/>) W3C Working Draft 7/7/01
- XQuery 1.0 formal semantics (formerly XML Query Algebra) (<http://www.w3.org/TR/query-semantics>)
- XQuery 1.0 and XPath 2.0 Data Model (<http://www.w3.org/TR/query-datamodel>)
- XPath 1.0 (XML Path Language) (<http://www.w3.org/TR/xpath>)

W3C has also released related documents specifying the requirements for a query language for XML (XML Query Requirements - <http://www.w3.org/TR/xmlquery-req>), and a set of use cases consisting of samples of queries that demonstrate the properties of such a language (XML Query Use Cases - <http://www.w3.org/TR/xmlquery-use-cases>).

W3C specifies XQuery as a language developed on the basis of a number of previously proposed languages - Quilt [15] XPath [16], XQL [17], and XML-QL [18]. Ideas in XQuery also came from Lorel [19] and YATL [20]. Xquery is designed as a functional language in which a query is represented as an expression. The underlying data model for the representation of inputs and outputs to XQuery queries is based on a representation of documents or document fragments as a nested sequence of nodes.

The proposed language DSQL has a similar flavor to Lorel and YATL mentioned above. DSQL aims, however for expressiveness in a familiar syntax and an equivalence with SQL for flat structures. This results in a language that can be easily incorporated into today's database systems. This then provides interoperability between documents and relational databases. The underlying data model in DSQL is not based on a single document, but on complex relations that may span across documents. Data independence is one of the major design principles in SQL, and such is the case for DSQL as well.

3 DSQL Specification

We now describe the language characteristics of DSQL. DSQL can, in some circumstances, be considered an extension to SQL, since DSQL does have an

² <http://www.w3.org>

added functionality beyond SQL in order to handle the nuances of the domain of structured documents. In the following discussion, we restrict ourselves in explaining the added or changed functionality of DSQL over standard SQL³. Most omitted features are exactly the same as standard SQL.

Every DSQL query has the following basic syntax:

```
SELECT output_structure
FROM   input_specification
WHERE  conditions
       grouping_specs
       ordering_specs
```

As in SQL, only the `SELECT` and the `FROM` clauses are required. The other clauses are optional. Also, multiple `SELECT` queries can be combined using the standard set operations (Union, Intersect, Minus). The following sections describe in detail the above five constructs of DSQL.

3.1 The `SELECT` Clause

A simplified BNF for `output_structure` is as follows:

```
output_structure ::= ['ALL' | 'DISTINCT'] output
output           ::= scalar_exp_list | '*'
scalar_exp_list  ::= scalar_exp [',' scalar_exp]*
scalar_exp       ::= name<scalar_exp_list> | atom | col
col              ::= path_exp
```

The above BNF allows select clauses such as:

```
SELECT *
SELECT output<*>
SELECT result<B.title, B.author>
SELECT booklist<B.author, books<B.title, B.year>>
```

The only omitted part of the actual BNF for the `SELECT` clause involves the use of aggregate functions, which would be discussed in Section 3.4. As is evident from the BNF and the examples above, this extension of the `SELECT` clause allows the creation of structures with arbitrary levels of nesting. Notice that any grouping is not inherent in this specification and is, instead, the task of the `grouping_specs` which will be discussed in Section 3.4.

Semantics of `SELECT *` If the `FROM` clause has only one expression, `SELECT *` returns the root of that expression *e.g.*, `select * from Book.chapter C` would select all chapters (root tag: `chapter`). However, if the `FROM` clause has multiple unrelated expressions, an implicit “cross product” operation is performed, and the result is the root of the cross product. Dependent path expressions (such as the second path expression in `Book B, B.author A`) are not considered for the product. For example, the query `SELECT * FROM Book.chapter C, Authors A` would create structures like the following:

³ A full description and implementation of DSQL is in preparation.

```

<result>
  <chapter>...</chapter>
  <author>...</author>
</result>

```

Semantics of SELECT name<*> Attaching a name to the output expression has the same semantics as attaching no name. Not giving a name chooses a default name (*e.g.*, result). The semantics is to create the following production:

```
outputname ::= head(result)
```

So, `SELECT mybooks<b.title,b.pub>` would generate:

```

<mybooks>
  <name>SQL</name>
  <pub>AWL</pub>
</mybooks>
<mybooks>
  <name>SGML</name>
  <pub>ORA</pub>
</mybooks>

```

Note A restricted form of fixed point is exhibited by the path expression construct, where a recursive path would lead to a fixed point computation (*e.g.*, `p..child` will give all descendants of `p`).

3.2 THE FROM CLAUSE

The BNF for the FROM clause is as follows:

```

input_specification ::= db_list
db_list ::= db [,db]*
db ::= path_exp [alias]
path_exp ::= XPath

```

XPath above represents path expressions following the basic XPath specifications[16]. To maintain the similarity with SQL, we will use the symbol “.” instead of the standard traversal symbol “/” used in XPath.

The above specification of FROM clause allows the following types of expressions in the FROM clause:

```

FROM books.xml B
FROM books.xml B, authors.xml A
FROM http://www.mycompany.com/docs/invoices.xml V, V..items I

```

3.3 THE WHERE CLAUSE

WHERE conditions in DSQL are similar to those in SQL. The main difference in semantics is due to paths, *i.e.*, all expressions in DSQL are path expressions, so operators are often set operators. For example, consider the following query:

	Before grouping	After grouping by A
Query	SELECT result<A,B,C>	SELECT result<A,B,C> ... GROUP BY A
Structure	result ::= A, B, C	result ::= A, (B,C)*
Data	<pre><result> <A>a1b1<C>c1</C> </result> <result> <A>a1b2<C>c2</C> </result> <result> <A>a1b3<C>c3</C> </result> <result> <A>a2b4<C>c4</C> </result></pre>	<pre><result> <A>a1 b1<C>c1</C> b2<C>c2</C> b3<C>c3</C> </result> <result> <A>a2 b4<C>c4</C> </result></pre>

Table 1. Semantics of GROUP BY

```
SELECT result<B.title>
FROM bibdb..book B
WHERE B.title = 'Extending SQL'
```

The WHERE expression evaluates to true if the path expression yields a singleton set containing an atom identical to 'Extending SQL'. Set membership operations such as `in` and `contains` are also available in DSQL.

3.4 POST RETRIEVAL OPERATIONS

SQL has several ways of specifying post query formatting and layout generation. In our earlier discussions, we have referred to such specifications as `grouping_specs` and `ordering_specs`. The following are the grouping and ordering specifications in DSQL:

- **ORDER BY** (sorting): DSQL has the same semantics for ORDER BY as SQL. The expressions in the SELECT clause can be ordered by expressions in the order by clause, regardless of whether or not the order by expressions appear in the SELECT clause, as long as the expressions are logically related, i.e., a possible ordering is possible using the ordering expression.
- **Aggregate functions**: DSQL supports the same five basic aggregate functions as SQL (`sum`, `count`, `min`, `max`, `avg`).
- **GROUP BY**: In DSQL, GROUP BY is a restructuring operation but unlike SQL, the aggregate function is optional. The semantics of the restructuring operation is explained using the production that define the created structure, as shown in Table 1

There are a few consequences of this type of grouping:

- *Grouping without aggregate functions* Grouping tasks can be performed with or without aggregate functions. In the case no aggregate functions are specified, grouping is essentially a restructuring operation. With aggregate functions, the aggregate functions are performed by the formed groups.
- *Group by null* An interesting effect of the grouping semantics is the possibility of grouping without any grouping clause which would cause the following restructuring operation:
`result ::= A, B, C` transformed to `result ::= (A, B, C)*`
- *Multiple group-by clauses* A complex structure could be grouped many times. In such cases, the structure to be grouped needs to be specified. For example:

```

SELECT Books<B.Author, Years<B.year, B.title>>
FROM   Bib B
GROUP  Books by B.Author
GROUP  Years by B.Year

```

4 Comparative Examples

The language proposed in this paper is capable of performing most of the queries in the W3C XQuery use case that are independent of XML-specific issues such as namespaces. Some of the queries in the use-case are beyond the expressive power of DSQL by design. Here we take a few samples from that set of queries and show how the same queries can be performed with DSQL.

Query 1. List books published by Addison-Wesley after 1991, including their year and title.

Solution in XQuery	Solution in DSQL
<pre> ----- <bib> { FOR \$b IN document("http://www.bn.com")/bib/book WHERE \$b/publisher = "Addison-Wesley" AND \$b/@year > 1991 RETURN <book year={ \$b/@year }> { \$b/title } </book> } </bib> </pre>	<pre> ----- SELECT bib<B.title> FROM BN.bib.book B WHERE B.publisher = "Addison-Wesley" AND B.attval(year) > 1991 </pre>

Query 2 . Create a flat list of all the title-author pairs, with each pair enclosed in a “result” element.

Solution in XQuery	Solution in DSQL
-----	-----
<pre> <results> { FOR \$b IN document("http://www.bn.com")/bib/book, \$t IN \$b/title, \$a IN \$b/author RETURN <result> { \$t } { \$a } </result> } </results> </pre>	<pre> SELECT result<B.title, B.author> FROM BN.bib.book </pre>

Query 3 . For each book in the bibliography, list the title and authors, grouped inside a “result” element.

Solution in XQuery	Solution in DSQL
-----	-----
<pre> <results> { FOR \$b IN document("http://www.bn.com")/bib/book RETURN <result> { \$b/title } { FOR \$a IN \$b/author RETURN \$a } </result> } </results> </pre>	<pre> SELECT result<B.title, B.author> FROM BN.bib.book GROUP BY B.title </pre>

5 Data Model and Formal Languages

The underlying model for DSQL is the view of a document database as a set of document relations, each relation representing a set of documents of a particular type. The language can dynamically generate other relations as temporary results of the evaluation process of the queries. To define the notion of structured document databases, we first define a few sets: **gi** is a countably infinite set of generic identifiers (GIs) that form the schema of the documents. **doc** \subseteq **gi** is the set of document types that represent the top-level relations in the database. We also define **dom**, which is a countably infinite set of constant character strings, and **var**, a countably infinite set of variables.

Types We only consider two types:

1. *Basic type* β . The base type comprises of character strings, with **dom** being the range of values.
2. *Complex type* τ . The complex types form the set **gi**. Within this set, the subset **doc** defines document types which are special complex types in the database. In the presence of a DTD (or schema) Any GI in a DTD defines a complex type, and various operations in the language create new productions to generate a new DTD or schema.

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

- *Document*. Conceptually, a document is a strictly hierarchical structure with a root GI that defines its type. If a DTD or schema is available, the document type can be modeled as a grammar represented by a quadruple $d = (\tau, \mathcal{G}, \mathcal{C}, \mathcal{P})$ where $\tau \in \mathbf{doc}$ is a document type, $\mathcal{G} \subset \mathbf{gi}$ is a set of generic identifiers, and $\mathcal{C} \subset \mathbf{dom}$ is a set of constants. \mathcal{P} is a set of production rules describing the structure of conforming document instances.
- *Document relations*. A document relation can now be viewed as a set of documents of a given document type.

A database, in this setting, is a set of Document relations.

Note A document conceptually does not coincide with a physical document file. A single file may include multiple documents, or a single document. A set of documents can be in one file, or spread over multiple files.

Path expressions Path expressions provide the second means of difference between the relational and document models. Relational models are predominantly flat, so the only concept of paths is based on tables and attributes (a path is thus a traversal from a table **T** to an attribute **A**, written in SQL as **T.A**). We use a notion of simple path expressions (SPEs), that includes the standard “.” operator to traverses to a child node, and a new “..” operator that traverses to a descendant. So **Book.title** returns titles directly underneath book, whereas **Book..title** returns titles reachable by traversing down from book, which may lead to chapter titles, section titles, figure titles, etc.

To highlight the properties of this language without getting bogged down into the details, we are going to make a few simplifying assumptions. The demonstrated properties of this language do not change even if the assumptions are violated.

1. *String data types only* For the purpose of the formal language discussion, we assume that string is the only atomic data type in the language. Numeric interpretations of strings are allowed. Since atomic data types do not affect the complexity of languages, we do not lose any generality from this assumption.

2. *Simple path expressions* The formal languages use the notion of simple path expressions (SPEs) described above. DSQL has support for full XPath, although the support may be limited to a PTIME subset of XPath if XPath turns out to be more expressive.
3. *Element-Normal form* We will not consider attributes for this discussion, and assume that all documents are in Element-Normal Form (ENF) [21], where no attributes are present. Any document with attributes is equivalent to a corresponding document in ENF, so this assumption does not change our results as well.

5.1 The Document Calculus (DC)

We now describe Document Calculus (DC) as a calculus for documents. DC is an extension of the relational calculus, and uses SPEs as terms. We will use the symbol τ for types, and the symbol \circ to represent one of the path expression operators $.$ and $..$ in the following discussion.

Terms Terms in DC include constants ($c \in \mathbf{dom}$); variables ($x_\tau \in \mathbf{var}, \tau \in \mathbf{gi}$); and path terms ($x \circ \mathcal{P}$) where $\circ \in \{., ..\}$.

Operators Basic comparison operators and logical operations are supported in this language. Comparison operators include atom to set comparison operators (\ni and $\not\ni$) and set to set comparison ($=, \neq, \subseteq, \subsetneq, \cap, \not\cap$). The last two operators are intersection operators, *i.e.*, $R \cap S$ is true if R and S intersect (*i.e.*, have a common element). In addition, DC supports the standard logical operators \wedge (and), \vee (or), \neg (not).

Predicates The predicates supported are document predicates such as $D \in \mathbf{doc}$, path and path term predicates (of the form $D \circ \mathcal{P}$ or $x \circ \mathcal{P}$).

Formulas DC formulas are functions from a set of variables to the boolean values **true** and **false**. The formulas in DC include the following:

1. *Atomic formulas.* $\mathcal{R}(x)$ is an atomic DC formula, where \mathcal{R} is a predicate.
2. $x \circ \mathcal{P} \theta c$ is a DC formula, where $\theta \in \{\ni, \not\ni\}$, $x \circ \mathcal{P}$ is a path term and $c \in \mathbf{dom}$ is a constant.
3. $t_1 \theta t_2$ is a DC formula, where $\theta \in \{=, \neq, \subseteq, \subsetneq, \cap, \not\cap\}$ and $t_1 = x_1 \circ \mathcal{P}_1$ and $t_2 = x_2 \circ \mathcal{P}_2$ are two path terms.
4. If φ and ψ are formulas, so are $\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$.
5. If $\varphi(x, x_1, x_2, \dots, x_n)$ is a DC formula with $n+1$ free variables x, x_1, x_2, \dots, x_n ($n \geq 1$) then the following are DC formulas:
 - $\exists x \varphi(x, x_1, x_2, \dots, x_n)$ (existential quantifier).
 - $\forall x \varphi(x, x_1, x_2, \dots, x_n)$ (universal quantifier).
6. If φ is a formula, so is (φ) .

In the above setting of formulas, it is not possible to guarantee that only a finite combination of the free variables satisfies the formula. For example, the query “all documents not in a relation” can be represented by the formula $\neg D(x)$ and can be satisfied by an infinite number of values of x . Such formulas are unsafe, since queries that include such formulas can never be computed in a finite time. To avoid this problem, we also define the notion of *safe formulas*⁴.

Tuple Construction The path expressions provide a means for extracting components of a composite object. DC supports dynamic creation of composite types by creating new generic identifiers with already existing generic identifiers as children. This is achieved by providing a tuple construction expression of the form: $z = R\langle x_1, x_2, \dots, x_n \rangle \quad n \geq 1$ The tuple construction operation can be treated as a formula that always returns a truth value, and thus can be combined with other formulas using a conjunction.

Queries A query is an expression denoting a set of documents described by a safe DC formula φ . Queries give the closure property to the language by ensuring that the output of the query will be valid documents. All queries in DC are of the form $\{x \mid \varphi(x)\}$. Consider the following DTD:

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

- Ex. 1. The query ‘Find all poems that contain the word “love” in the poem title’ would be written in DC as $\{x \mid x^\psi..title \ni \text{“love”}\}$. Here, ψ is given by the query $\{z \mid poem(z)\}$.
- Ex. 2. Consider the query ‘Find the pairs of names of poets who have at least one common poem title’. Suppose $\psi = \{z \mid poem(z)\}$. The query is then expressed as:
- $$\{v \mid (v = R\langle x..poet, y..poet \rangle) \wedge (\exists x, y (x^\psi..title \cap y^\psi..title \wedge x^\psi..poet \neq y^\psi..poet))\}$$

5.2 The Document Algebra (DA)

DA is predictably an extension of the relational algebra. As any algebraic language, DA is defined in terms of special set operators that map one or more sets of documents to a new set of documents. One important property of DA is that it always produces documents that adhere to our original notion of documents, *i.e.*, always conforming to specific document types. To achieve this, operations in DA may add one or more productions to the existing DTD. Further, all operations in DA are closed under the SGML domain - they all generate document instances as their output.

⁴ For the complete definition, please see [22].

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

Table 2. Types of Document Algebra operations and new created types.

Every DA expression E^τ represents a set of documents of a particular type τ . We define DA expressions inductively by first defining the basic document expression and then defining the operations *path selection* (\circ), *cross product* (\times), *selection* (σ), *union* (\cup), *intersection* (\cap) and *set difference* ($-$). Other useful operations such as *join* (\bowtie), projection (π), generalized product (\amalg) and root addition (ρ), can be constructed from these primitive operations. All the primitive operators generate documents of specific types, as shown in Table 2.

Intuitive explanation of each of the above operators is as follows:

Document The expression D represents the set of all documents in the document relation D .

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

Union (\cup) Union is the usual set union operation, without the restriction that both operands of the union be of the same type. Given two DA expressions $E_1^{\tau_1}$ and $E_2^{\tau_2}$, the result of the union $E_1^{\tau_1} \cup_R E_2^{\tau_2}$ is a set of documents of a new type R which is created by adding a production for R with τ_1 and τ_2 in an option group.

Intersection (\cap) The intersection operation is the usual set intersection operation $E_1^\tau \cap E_2^\tau$, containing the set of documents that in both E_1^τ and in E_2^τ .

Set difference ($-$) The set difference is the usual set difference operation $E_1^\tau - E_2^\tau$, containing the set of documents in E_1^τ not in E_2^τ .

Cross product (\times) Given two DA expressions $E_1^{\tau_1}$ and $E_2^{\tau_2}$, the expression $E_1^{\tau_1} \times_R E_2^{\tau_2}$ is a DA expression, and it represents a set of documents with a new type R , the members of which contain two subcomponents: one from the set $E_1^{\tau_1}$ and the other from $E_2^{\tau_2}$. Every member in the resulting set is hence of type R , and each member of the set $[[E_1^{\tau_1}]^M]$ is paired with each member of the set $[[E_2^{\tau_2}]^M]$ under an instance of R .

Selection (σ) The selection operation $\sigma_\gamma E^\tau$ extracts a subset of documents from the input set E^τ that satisfy a selection condition γ .

We can again consider the example queries we used before:

Ex. 1. The first example is intuitively: $\sigma_{poem..title \ni \text{“love”}} Poem$.

Ex. 2. The second example clearly requires a cross product between poem and itself. To simplify the expression, we use the derived projection, join and root addition operators.

$$\pi_{P_1..poet, P_2..poet} (\sigma_{P_1..poet \neq P_2..poet} (\rho_{P_1} Poem \bowtie_{P_1..title \cap P_2..title}^R \rho_{P_2} Poem))$$

Given the above specification of DC and DA, we have proved the following significant results (the proofs are beyond the scope of this paper, please see [22] for the actual proofs):

Theorem 1 (DC = DA). Any query expressed in DC can be expressed in DA and vice versa. This enables us to talk about the properties of both of these languages equivalently.

Theorem 2 (DC and DA are safe). Queries expressed in DC and DA will never result in an infinite set of results, or infinite size of result tuples.

Theorem 3 (DA and DC are closed). Queries expressed in DC and DA use structured documents (result proved using SGML) as input, and generate documents in the same format as output.

Theorem 4 (DC and DA are in PTIME). Any query written in DC or DA can be processed by an algorithm that has a worst case complexity of a polynomial to the size of the input.

Theorem 5 (DC and core DSQL are equivalent). The basic querying portion of DSQL that does not include the grouping, ordering and aggregate functions is semantically equivalent to DC.

6 Results and Contributions

In this paper we presented a language entirely based on SQL, but with properties that are highly desirable of query language. Here we present a summary of these properties.

1. **DSQL properties** Because of the equivalence with the calculus and algebra, DSQL is implicitly safe, closed, and in PTIME.
2. **DSQL Macros** DSQL can be augmented with macros that allow structuring operations that do not change any complexity, such as decision and link traversal. IF-ELSE queries can be implemented using DSQL UNION (as shown above), and finite link traversal (such as in [23] can be performed using EXISTS.
3. **No ordering ambiguity** The sequential nature of documents is built into DSQL. A single document is always sequential, but a document is fragmented using a path expression creates unordered elements (since the result of a path expression is a set).
4. **Set-Atom ambiguity** All non-atomic expressions (queries and path expressions) in DSQL are set expressions, so there is no ambiguity in comparison of different expressions.

5. **Similarity with Relational domain** Because of the analogous nature of DSQL with SQL, most of the language processing methods work unchanged. For example, all the optimization equalities hold in DSQL, which indicates that the same optimization principles will be applicable in processing DSQL queries.
6. **DSQL=SQL for flat structures** For flat structures as input, and intended flat structures as output, DSQL queries are exactly the same as the corresponding SQL queries. This is based on similar results for nested relations [24].
7. **Merging relational and document databases** DSQL can easily support queries which include local documents, remote documents via a URL, and relational tables all in the same FROM clause. This is an easy yet elegant technique for incorporating distributed heterogeneous databases in this query language.
8. **Implementation architecture** We built a prototype implementation of DSQL in DocBase[22] that also demonstrates the applicability of the concepts described here.

7 Future Research Issues

A number of issues need to be handled in order to apply DSQL in XML. Specific XML features (such as namespaces, attributes, etc.) are not included in DSQL because of its markup-independent design. XML-specific features can easily be added on to the language in order to satisfy all the W3C XML Query requirements. DSQL does not support recursion, and we have no intention of supporting direct unbounded recursion in the language. Similar to SQL, DSQL can be embedded in a host language or a 4GL-type language for such language features. We are also in the process of performing an empirical study comparing DSQL with XQuery based on a human-factors analysis involving users.

References

1. American National Standards Institute New York: ANSI X3H2 standards group. (1996)
2. Chamberlin, D.D., Astrahan, M.M., Eswaran, K.P., Griffiths, P.P., Lorie, R.A., Mehl, J.W., Reisner, P., Wade, B.W.: SEQUEL 2: A unified approach to data definition, manipulation and control. IBM Journal of Research and Development (20) 560–575
3. Denny, G.: An introduction to SQL, a structured query language. Technical Report RA 93, IBM Research, San Jose, California (1977)
4. Codd, E.: A relational model for large shared data banks. Communications of the ACM **6** (1970) 377–387
5. International Organization for Standardization and International Electrotechnical Commission Geneva, Switzerland: ISO/IEC DIS 10179: Document Style Semantics and Specification Language: DSSSL. (1994)

6. Reisner, P.: Use of psychological experimentation as an aid to development of a query language. *IEEE Transactions on Software Engineering* **SE-3** (1977) 218–229
7. Reisner, P.: Human factors studies of database query languages: A survey and assessment. *Computing Surveys* **13** (1981) 13–31
8. Welty, C., Stemple, D.W.: Human factors comparison of a procedural and a non-procedural query language. *ACM Transactions on Database Systems* **6** (1981) 626–649
9. Welty, C.: Human factors studies of database query languages: SQL as a metric. *Journal of Database Administration* **1** (1990) 2–10
10. Ramakrishnan, R., Gehrke, J.: *Database Management Systems*. Second edition edn. McGraw-Hill (1999)
11. Mendelzon, A., Mihaila, G., Milo, T.: Querying the world wide web. *International Journal of Digital Libraries* **1** (1997) 68–88
12. Gadia, S.K., Chopra, V.: A relational model and sql-like query language for spatial databases. In Adam, N.R., Bhargava, B.K., eds.: *Advanced Database Systems*. Volume 759 of *Lecture Notes in Computer Science*. Springer (1993) 213–225
13. Cattell, R., Barry, D., Bartels, D., Berler, M., Eastman, J., Gamberman, S., Jordan, D., Springer, A., Strickland, H., Wade, D.: *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann (1997)
14. Cluet, S.: Designing OQL: Allowing objects to be queried. *Information Systems* **23** (1998) 279–305
15. Chamberlin, D., Robie, J., Florescu, D.: Quilt: an xml query language for heterogeneous data sources. In Suciu, D., Vossen, G., eds.: *The World Wide Web and Databases, Third International Workshop WebDB 2000, Dallas, Texas, USA, May 18-19, 2000, Selected Papers*. Volume 1997 of *Lecture Notes in Computer Science*. Springer (2001) 1–25
16. Clark, J., DeRose, S.: XML path language XPath version 1.0. <http://www.w3.org/TR/xpath> (1999) W3C Working Draft.
17. Robie, J., Lapp, J., Schach, D.: XML Query Language (XQL). W3C. (1998) Available on-line from <http://www.w3.org/TandS/QL/QL98/pp/xql.html>.
18. Deutsch, A., Fernandez, M., Florescu, D., Levy, A., Suciu, D.: XML-QL: A query language for XML. W3C. (1998) Available on-line from <http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/>.
19. Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.: The lorel query language for semistructured data. *International Journal on Digital Libraries* **1** (1997) 68–88
20. Cluet, S., Simeon, J.: YATL: A functional and declarative language for xml. available from author's website <http://www-db.research.bell-labs.com/user/simeon/> (1999)
21. Layman, A.: Element-normal form for serializing graphs of data in XML. Based in part on an earlier paper, *Serializing Graphs of Data in XML*, by A. Bosworth, A. Layman, M. Rys, in *XML Europe '99, Granada, April 1999* (1999)
22. Sengupta, A.: *DocBase - A Database Environment for Structured Documents*. PhD thesis, Indiana University (1997)
23. Mendelzon, A., Wood, P.: Finding regular simple paths in graph databases. *SIAM Journal on Computing* **24** (1995) 1235–1258
24. Paredaens, J., Gucht, D.V.: Converting nested algebra expressions into flat algebra expressions. *ACM Transactions on Database Systems* **17** (1992) 65–93