Data-Driven Publication of Relational Databases *

Sonia Guéhis[1], Philippe Rigaux[1] and Emmanuel Waller[2]

Abstract

The paper presents a framework for publishing relational databases in textual documents such as mails, HTML pages, LaTeX or BibTex files, plain texts, etc. The publication process relies on a mapping of the relational database to a virtual data graph which supports navigation operators. Applications can express the data they need by navigating in the graph. The result is then obtained by producing, during the navigation, textual fragments whose concatenation constitutes the final document.

These operations are provided by a declarative query language over virtual graphs, named DocQL. The actual evaluation of a DocQL query is done by executing interrelated SQL queries over the relational database. By considering these queries as a whole, a global optimization process takes place during the translation.

DocQL aims at reaching a good tradeoff: allowing a simple, direct and very concise specification of database publishing applications whose semantics depends only on the database instance, while keeping an efficient evaluation. We illustrate its features with the conference management system MyReview.

1 Introduction

One of the major explanation of the relational model success comes from the availability of simple and well-defined query languages. However, whereas SQL and its foundations have been studied at length by the academic community, in practice SQL is almost always *embedded* in traditional programming languages such as C or Java. As a programming language, SQL is indeed very weak and cannot even express some simplistic operations. It also lacks the flexibility (e.g., access to external libraries) commonly required in real-life applications. In practice, the vast majority of applications have to move outside flat relations, both for manipulation and presentation purposes.

Combining a programming language with SQL raises several issues. In the present paper we focus on the following ones:

- first embedding SQL in a programming language is definitely out of the scope of non-expert users;
- second it becomes possible to program everything, including access plans which should be left to the database optimizer.

Many programmers tend to use a relational database as flat files, SQL as a simple scan operator, and implement with the primitives of the programming language some data manipulations that could be expressed more simply and efficiently with SQL. This often prevents the database server from applying the optimizations which are only available on a single-query-at-a-time basis, and not on the set of interrelated queries embedded in a program. It would therefore be desirable to extend database optimization techniques to embedded queries, i.e., to sequences of SQL queries combined with programming constructs. Unfortunately, it is generally considered that their desirable properties (termination, potential for optimization) are undecidable or raise too many technical difficulties.

Although this can be true in general, we believe that in many cases the full power of unrestricted embedded SQL is not necessary, and that restricted forms of SQL programming are sufficient for some common and well characterized classes of applications. In the present paper we develop this intuition in

[1] Lamsade, Univ. Paris-Dauphine, Paris, France, <firstname.lastname>@dauphine.fr
[2] LRI, Univ. Paris XI, France, waller@lri.fr
*Research supported by the CRIT GVD project.

the context of *publishing applications*, for which we adopt the following simple definition: any program that produces a string of characters containing data extracted from a relational database. This definition covers a large range of very common and useful database settings, including the dynamic production of HTML pages in web sites. One of our main goals is actually the definition of high-level specifications for such database-enabled web sites, but the framework proposed in the paper goes beyond this by meeting as well many other publication areas: XML publication of relational data for exchanges purposes (e.g., RSS); mail messages that include database information; LaTeX (or any other text processor) sophisticated layout of database content; Excel spreadsheets, etc.

It turns out that none of these outputs can be produced by SQL, as soon as the structure of the document goes beyond that of a simple table. Our work is motivated by the conviction that the publication requirements can be fully satisfied by restricted programming constructs, with two consequences:

1. high-level, *declarative*, specifications of such programs can be envisaged, bringing, among other advantages, simplicity to non-experts users;

2. a global and declarative specification of the program lets the optimizer manipulate it as a whole, instead of viewing it as a set of SQL queries processed independently by the DBMS.

We aim at defining a very simple, direct and concise language to meet common publishing requirements. To this end we propose a relational language, named DocQL, which combines the following mechanisms: navigation primitives in the relational database and instantiation of *fragments* which contribute to the final result. A design choice of DocQL is that it does not directly rely on the relational database instance, but rather on a representation of this instance as a (virtual) data graph. This is motivated both by simplicity and expressiveness. First, since we aim at producing a graph structure as output, we believe that viewing the input itself as a graph provides an intuitive mechanism for users. Second the graph representation offers a much more convenient support for navigation than SQL cursors which are limited to linear scan of query results. We describe the syntax and semantics of the language, and show that a DocQL query can be easily represented in a formalism which gives rise to rewriting and optimization techniques. Our last contribution is an implementation of DocQL in MyReview, a widely used conference management system which strongly resorts to the production of various documents.

DocQL constitutes a direct approach to data publishing from a relational database which tries to avoid the burden of repetitive programming tasks. An alternative would be to generate an XML document from the relational instance (using, say, the XSQL utility of Oracle [14]), and then to transform this document with an XSLT stylesheet. We believe that the complexity of this process (first export to XML, then parse the document, then submit the document to the XSLT processor) is strongly unadapted to the simplicity of the requirement. It suffices to imagine the heavy infrastructure and development effort that would be necessary just to produce a trivial message containing some values extracted from the database. Moreover this two-steps transformation raises some technical problems with respect to database accesses, and it is in particular difficult to avoid with XSLT the useless materialization of the whole database as an XML document, although a small part of it is usually required [19, 18].

Some recent research works attempt at enabling a composition of XML export and XML publishing languages [16, 11]. SilkRoute for instance [16] combines an RXL query that exports in XML some parts of a relational database with correlated SQL queries, and an XML-QL query over the exported document. The composition algorithm avoids a full materialization. Although our approach is, to some extent, similar, our motivation differs by at least two important points. First we do not consider a middleware-based architecture, where relational data needs a preliminary transformation (in XML) in order to be accessible by other applications. Consequently we avoid the issues raised by the combination of two languages (e.g., XSQL/XSLT, or RXL/XML-QL) and by the necessary infrastructure. Second we advocate a high-level specification of the navigation in the database, from which we derive an appropriate embedded SQL program. In summary, we propose with DocQL a direct, lightweight, approach which straightly navigates in the relational instance and retrieves the data of interest to the document production.

The rest of the paper presents first (Section 2) an informal description of our approach. The model is given in Section 3 and the query evaluation (as well as our implementation in MyReview) is proposed in Section 4. Section 5 discusses related work and Section 6 concludes the paper and outlines future work.

2 Database and Query Model

We illustrate the main intuitions behind our work with a web application, MYREVIEW, which supports the submission phase of scientific conferences. Its main features are undoubtly already familiar to the reader! In short, *authors* can submit *papers* along with a description (abstract, title, topics), whereas *reviewers* are assigned to papers in order to provide an evaluation. An evaluation consists of several pieces of free text (comments, summary, etc.) and of *review marks*, one for each of the evaluation criteria defined by the PC chair.

2.1 The relational database

- Paper (**idPaper**, title, year)
- Person (**email**, firstName, lastName)
- Author (***idPaper***, ***email***)
- Review (***idPaper***, ***email***, comment)
- ReviewMark (***idPaper***, ***email***, **criterion**, mark)

Figure 1: The schema of the MYREVIEW database (excerpt)

Figure 1 shows an excerpt of the relational schema. Primary keys are in boldface and foreign keys in italics. Figure 2 shows a sample of the relational database. Each paper corresponds to a tuple in *Paper* and is is associated to one or several authors (table *Author*). Persons are identified by their email in every table, and table *Person* gives the first and last name of the person corresponding to the email. The assignment of a paper to its reviewers is represented as tuples in the table *Review*. Finally reviewers give their comments (in *Review*) and a list of marks (table *ReviewMark*), each for some criteria.

idPaper	title	year
128	Do computer think?	1951

Paper

email	firstName	lastName
turing@nw.com	Alan	Turing
r1@sw.com	John	Doe
r2@ew.com	Bill	Smith

Person

idPaper	***email***
128	turing@nw.com

Author

email	***idPaper***	comment
r1@sw.com	128	Ridiculous
r2@ew.com	128	Outstanding

Review

email	***idPaper***	**criterion**	mark
r1@sw.com	128	quality	3
r1@sw.com	128	relevance	4
r2@ew.com	128	quality	5

ReviewMark

Figure 2: A database instance

The representation is quite standard. Each tuple consists of a set of attributes values, and is identified by its key. The key is used to refer to a tuple t: for instance the referee of a paper (table *Review*) is referred to by means of its email, which is the key of *Person* tuples. In the following we assume that the database is in third normal form.

2.2 Documents

The MYREVIEW system, as many others of its kind, produces documents built from information extracted from the database. Most user requests to the system result in the production of an HTML page, and

several other kinds of documents are also produced on demand. For instance the report on the reviewers' evaluations regarding a given paper exists in the following formats:

- an HTML page;
- a message, sent by email to the contact author at notification time;
- a LaTeX document whose output can be printed by the PC chair.

In each case the goal is to publish a document that complies to a specific grammar (or no grammar at all in the case of emails) and contains database information. Here is for instance a (simplified) HTML version of the report, produced from the instance of Figure 2.

```
...
Paper: Do computer think?<br/>
Author: Alan Turing<br/>
Abstract: ...<br/>

<h2>Reviewer: John Doe</h2>
 <ol>
    <li>Comments: Ridiculous</li>
    <li>Quality: 3</li>
    <li>Relevance: 4</li>
  </ol>
<h2>Reviewer: Bill Smith</h2>
  <ol>
     <li>Comment: Outstanding</li>
     <li>Quality: 5</li>
  </ol>
...
```

The LaTeX version is an alternative presentation of the same data:

```
...
\section{Do computer think?}

Paper written by Alan Turing

\subsection*{Review of John Doe}

\begin{enumerate}
   \item \textbf{Comments}: Ridiculous
   \item \textbf{Quality}: 3
   \item \textbf{Relevance}: 4
\end{enumerate}
...
```

There is no way to produce such documents with pure SQL. So, in practice, a program with embedded SQL queries has to be written. Such programs always follow the same internal organization since, even if the formats of these documents differ greatly, they all rely on the same data skeleton: a paper is associated to an enumeration of all its reviews, and each review in turn comes with its marks. Specifying this organization is, conceptually, extremely simple: we just have to follow the paths from a paper to its reviews, then from a review to its marks. This navigation is driven by the database content. It is important to underline that this "database-driven" mechanism not only includes the mere iteration on instances found in the database, but also some decisions made occasionally to follow or not some links, and thus to instantiate or not some specific parts of the documents.

Here is a last example that illustrates this decision process. The notification mail output the reviews on a paper, following the general layout of the above documents, and contains also some fragments associated respectively to the 'accepted' or 'refused' status.

```
Dear Alan Turing,

We are pleased to inform you that your paper entitled
"Do computer think?" has been accepted [...]

Please take into account the suggestions of our reviewers.
We are looking forward ...

Reviewer 1:
  Quality: 3
  Relevance: 4
  Reviewer's comments: Ridiculous

Reviewer 2:
  ...
```

Some parts of the notification message change whether the paper is accepted or refused. The choice depends on some attributes values, found in the database, which determine the behavior of the publication process. Actually most of the practical functionalities required to publish textual documents from a relational database comply to the general mechanism illustrated by the previous examples. In particular, the programs commonly used to produce such documents rely intensively on iterations and tests whose conditional statements only consider values extracted from the database.

2.3 The Data Graph

In order to support the conceptual navigation mechanism mentioned above, we model the database as a directed graph and each tuple in the database as a vertex in the graph. Each link (foreign-key, primary-key) is modeled as a directed edge between the corresponding tuples. The graph is virtual, and must be partially materialized during the query evaluation process.

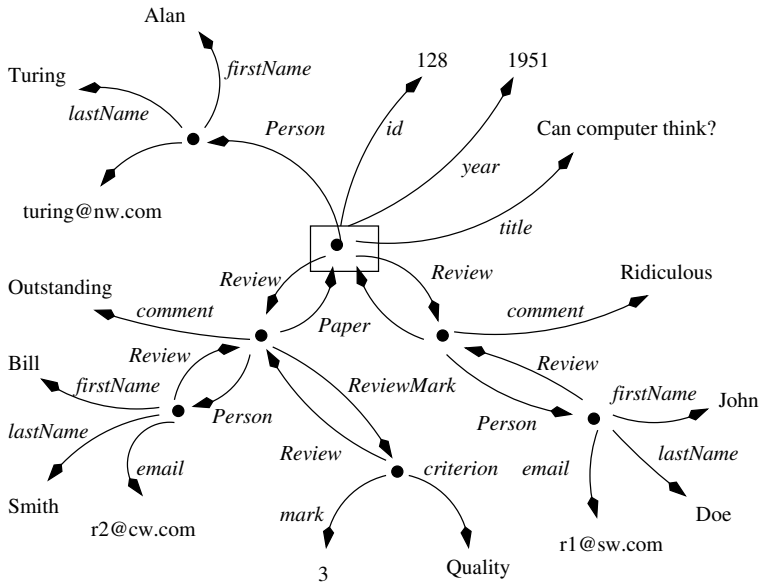


Figure 3: A part of the virtual graph

A part of the data graph is shown in Figure 3 for our sample database (the content of the *ReviewMark* table for instance is partially represented). The main feature of this representation is the simplicity and concision of the concepts in use: each vertex corresponds either to a tuple or a value; each edge is labeled and represents an association between vertices. The edges cover all the various links which are usually distinguished in a relational database, namely tuple-to-attribute and tuple-to-tuple.

2.4 The DocQL language

DocQL allows to integrate data extracted from the relational database to textual fragments. These fragments are concatenated so as to create documents. Basically a DocQL query consists of the following components:

1. a tree of *path expressions* (sometimes called the *graph query*) in the following, specifies the part of the data graph (called the *subgraph*) which contains the data of interest to the output document;
2. each path expression p in the graph query denotes a set of vertices, called the *terminal vertices* of p; p is associated to a *decoration template* which describes the textual fragment to produce for each of the terminal vertex.

The following example shows a DocQL query over our sample database (we assume an additional attribute *accepted* in table *Paper*). The query produces a document for PC chairs which summarizes the status of a paper, along with its reviews. Path expressions and templates are organized as *rules* of the form $@path\{body\}$. Syntactic details can be ignored for he moment, and will be detailed later on.

```
@paper[idPaper=128]{
  This paper, entitled @title, written by
  @author.person.firstName @author.person.lastName, has been
  evaluated as follows:

   @accepted['Y']
      {The paper is going to be accepted ...}
   @accepted['N']
      {The paper is going to be rejected ...}

   @review{
     - Reviewer name: @reviewer{@firstName @lastName}
     - Comments:   @comment
     - Marks:
        @reviewMark{name: @mark}
   }
}
```

The interpretation of this query over the instance of Fig. 3 can be described as follows. First one accesses the paper whose id is 128 (framed by the box in the figure). From this vertex the paths `title`, `author.person.firstName` and `author.person.lastName` lead to terminal vertices whose value are inserted in the document. The rule `accepted` shows a decision to instantiate a fragment based on the value of the `accepted` attribute. Then a new rule is triggered for each path `review` starting from the current vertex. The interpretation of this rule is similar.

2.5 Query evaluation

Intuitively, the evaluation of a DocQL query q can be decomposed in two steps. The data graph $\mathcal{G}_I$ which supports the querying process is virtual, and defined by a mapping $\mathcal{M}$ of the relational database I. The navigation operations which are necessary to produce the result of a query must visit a subgraph of the data graph: the first step is the materialization of this subgraph. This is done by running a dynamically generated SQL program P_q which retrieves all the necessary tuples from all the involved tables, and maps these tuples to an in-memory representation of the subgraph. During the second step, the final document is produced thanks to a navigation through this subgraph.

The commuting diagram of Figure 4 summarizes the DocQL evaluation mechanism. The query q can be decomposed in a *graph query* which defines a subgraph $q(\mathcal{G}_I)$, and in *decoration templates* which produce a document from this subgraph. The query evaluation runs an embedded SQL program P_q over I, such that $\mathcal{M}(P_q(I)) = q(\mathcal{G}_I)$. The decoration templates can then be applied to $q(\mathcal{G}_I)$.

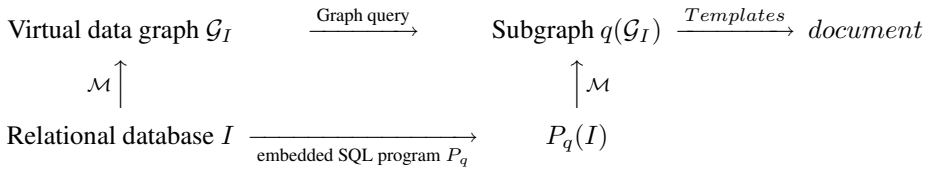


Figure 4: Evaluation of a DOCQL query q

The description is conceptual and gives rise to many variations in practice. In particular the evaluation is not necessarily strictly decomposed in two successive processes as explained above, but can enable some degree of pipelining between the subgraph materialization and the decoration. Note also that the embedding of the program instructions (mostly loops and tests) and SQL queries is now fully under the control of the query evaluator which chooses the appropriate strategy. We shall discuss these aspects further in the section devoted to query evaluation.

In summary, our claim is first that most publishing programs are limited to a few data-driven mechanisms, and second that the data graph view is the most convenient conceptual approach to supporting these mechanisms. In the next section we describe more formally the language that expresses this navigation. Moreover we show that this corresponds, in terms of embedded SQL, to a simple class of programs which gives rise to many optimization perspectives.

3 The model

3.1 The data graph

Let $\mathcal{T}, \mathcal{R}, \mathcal{A}$ be sets of symbols pairwise disjoint, $\mathcal{T}$ finite, and $\mathcal{R}, \mathcal{A}$ countably infinite. The elements of $\mathcal{T}$ are called *atomic types*, those in $\mathcal{R}$ *relation names*, and those in $\mathcal{A}$ *attribute names*.

Definition 1 (Schema) *A* (graph database) schema *is a directed labeled graph* (V, E, λ, μ) *with the following structure.*

1. $V \subseteq \mathcal{T} \cup \mathcal{R}$ *is a set of vertex, and* $E \subseteq (V \cap \mathcal{R}) \times V$ *is a set of edges;*
2. λ *is a labeling function from* E *to* $\mathcal{R} \cup \mathcal{A}$ *such that, if* e *and* e' *are two edges with same initial vertex* r*, then* $\lambda(e) \neq \lambda(e')$ *;*
3. μ *is multiplicity function from* E *to* $\{1, *\}$*; if* $\mu(e) = 1$*, this indicates that there can be at most one instance of* e *in the database for a given initial vertex; if* $\mu(e) = *$*, multiple instances are allowed;*
4. *if* $e \in E$ *is of the form* $r \overset{\lambda(e)}{\rightarrow} s$*, with* $r, s \in \mathcal{R}$*, there exists an edge* $e' \in E$ *of the form* $s \overset{\lambda(e')}{\rightarrow} r$*, called the* reverse edge *of* e *;*

In the following we adopt the standard graph terminology. An edge e is an ordered pair (a, b), where a is the initial vertex, denoted $initial(e)$, and b is the terminal vertex, denoted $terminal(e)$. Figure 5 shows the graph schema of our sample database. The simple choice adopted here for the labeling function is to associate to each edge $r \rightarrow v$ either the corresponding attribute name if $v \in \mathcal{T}$, or the name of the referred table if $v \in \mathcal{R}$. In general the schema may be a multi-graph, i.e., a pair of vertex may be connected by more than one edge. In such a case the simple labeling mechanism used for the schema of Figure 5 must be refined. The extension is trivial.

Now let $\mathcal{I}$ be a countably infinite set of *tuple identifiers*, and for each atomic type $\tau \in \mathcal{T}$ let be given the set of values of this type, denoted $[\tau]$.

Definition 2 (Instance) *Let* $S = (V, E, \lambda, \mu)$ *be a schema. An* instance $\mathcal{G}_I = (V_I, E_I)$ *of* S *is a mapping from* S *to rooted labeled graphs defined as follows:*

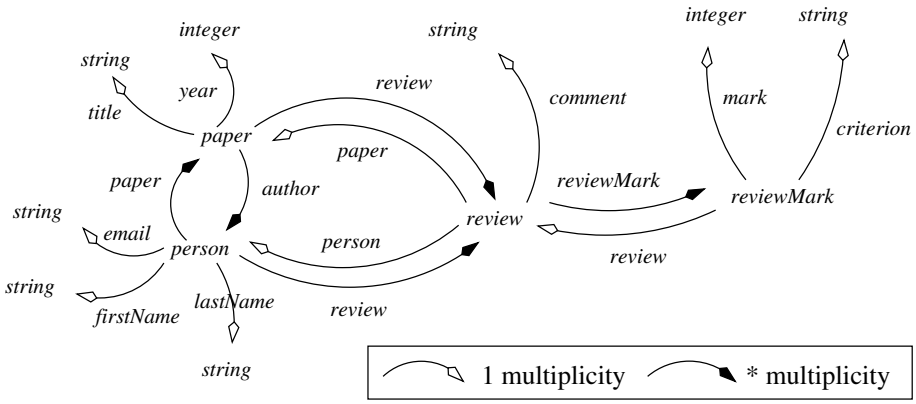


Figure 5: The schema of the data graph

1. *for each* $v \in V$ $\begin{cases} V_I(v) \subset \mathcal{I}\ if\ v \in \mathcal{R}\ (tuple - to - tuple) \\ V_I(v) \subset [v]\ if\ v \in \mathcal{T}\ (tuple - to - value) \end{cases}$

2. *if* $e \in E$*, then each instance of* e *is of the form* $x \xrightarrow{a} y$*, with* $x \in V_I(initial(e))$*,* $y \in V_I(terminal(e))$*, and* $a = \lambda(e)$*; moreover, if* $\mu(e) = 1$*, there does not exist two instances of* e *with the same initial vertex;*

3. *there exists a* root vertex $\mathtt{db}$ *in* V_I *such that, for each* $r \in V \cap \mathcal{R}$ *and for each* $v \in V_I(r)$*,* $\mathtt{db} \xrightarrow{r} v \in E_I$.

If r is a relation name, $V_I(r)$ is the set of vertex in r. Any access to the database must be through the root vertex `db`, whose out-edges refer to all the vertex/tuples of the database instance. Given a relational database, there exists a straightforward mapping between the relational schemas and instances and the graph schemas and instances.

3.2 The language

We now turn to the language definition. It consists of *path expressions* in the data graph, and *rules* which are triggered for each vertex denoted by a path. Syntactically, our paths expressions correspond to a subset of the XPath language [22]. In its simplest form, a path expression is a sequence of labels of edges pairwise connected in a graph schema S. A path expression may contain *predicates* which are Boolean combinations of atomic formulas of the form $q = value$ where q is a path expression. More precisely

- if l is a label or $l = \mathtt{db}$, l is a path expression;
- if q is a path expression and l is a label, $q.l$ is a path expression;
- if q is a path and p is a predicate, $q[p]$ is a path expression.

The general form of a path expression is $l_1[p_1].l_2[p_2].\cdots.l_n[p_n]$. A path expression is is *valued* if its last label is an attribute name. It is *valid* if the path denoted by $l_1.l_2.\cdots.l_n$ is connected in the graph schema, and if each path expression in a predicate is valued. A path expression is *absolute* if it begins with `db`, else it is relative.

Example 1 *Here are some examples of path expressions over our sample schema.*

1. $db.paper[id = 128].review.person.lastName$
2. $db.paper[id = 128].review$
3. $person.lastName$

A (valid) path expression q is interpreted with respect to a vertex v in the graph instance, called the initial vertex of q. Unlike XPath, the interpretation is the subgraph that consists of all the instances of q connected to v.

Definition 3 (Path interpretation) *Let q be a path expression in S, $\mathcal{G}_I = (V_I, E_I)$ an instance of S, and v a vertex in V_I. The interpretation of q over $\mathcal{G}_I$ from v is a graph $q(\mathcal{G}_I, v) = (V_I(q,v), E_I(q,v))$ defined inductively as follows.*

1. *if $q = \texttt{db}$, $E_I(q,v) = \emptyset$ and $V_I(q,v) = \{\texttt{db}\}$.*
2. *if $q = l$, where l is a label, $E_I(q,v) = \{e \in E_I \mid e \; is \;\; of \; the \; form \; v \xrightarrow{l} v'\}$, and $V_I(q,v) = \{v\} \cup \{terminal(e), e \in E_I(q,v)\}$*
3. *if $q = q'.l$, where q' is a path expression and l is a label, then*

$$\begin{cases} E_I(q,v) = E_I(q',v) \cup \{e \in E_I \mid initial(e) \in V_I(q',v) \; and \; \lambda(e) = l\} \\ V_I(q,v) = V_I(q',v) \cup \{terminal(e), e \in E_I(q,v)\} \end{cases}$$

4. *if $q = q'[p]$ where q' is a path expression and p is a predicate of the form $path = value$ then*

$$\begin{cases} V_I(q,v) = \{v' \in V_I(q',v) \mid value \in V_I(path, v')\} \\ E_I(q,v) = E_I(q',v) \cup \{e \in E_I(q',v) \mid terminal(e) \in V_I(q,v)\} \end{cases}$$

Consider the data graph of Figure 3. The interpretation of $db.paper[id = 128].review.person.lastName$ is a connected subgraph which consists of two paths with initial vertex `db` (not shown on the figure) and with terminal vertex, respectively, "Smith" and "John". Now, let Σ be a finite alphabet and '.' the concatenation operator in Σ^*. Rules are defined as follows.

Definition 4 (Rules) *A* rule *is a 3-tuple (q, b, e), where q is a path expression, and b and e are finite sequences of words and rules over Σ, called respectively the* body *and the* exception *of the rule.*

A *query* is simply a rule $r(q, b, e)$ such that q is an absolute path.

Definition 5 (Rules semantics) *Let $\mathcal{G}_I$ be an instance of S, and v a vertex in $\mathcal{G}_I$, called the* initial vertex. *The semantics $[r(\mathcal{G}_I, v)]$ of $r(q, b, e)$ over $\mathcal{G}_I$ from v is defined inductively as follows:*

1. *Let $q(\mathcal{G}_I, v) = \{v_1, \ldots, v_k\}, k \geq 0$, then* $\begin{cases} if \; k > 0, [r(\mathcal{G}_I, v)] = \|b(\mathcal{G}_I, v_1)\|. \cdots .\|b(\mathcal{G}_I, v_k)\| \\ else \; [r(\mathcal{G}_I, v)] = \|e(\mathcal{G}_I, v)\| \end{cases}$
2. *if s is a sequence of words and rules of the form $s_0.r_1.s_1.\cdots.r_n.s_m, m \geq 0$, and w is a vertex from $\mathcal{G}_I$, then $\|s(\mathcal{G}_I, w)\| = s_0.[r_1(\mathcal{G}_I, w)].s_1.\cdots.[r_n(\mathcal{G}_I, w)].s_n$.*

As a special case, the semantics of a query r is $[r(\mathcal{G}_I, \texttt{db})]$. The definition is constructive. The number of "steps" is the depth k of imbrication of rules in the program, and the size of a step depends on the number of vertex returned by the paths of the step.

3.3 Rule syntax and examples

The concrete syntax for rules is $@p\{b\}\{e\}$. The exception of the rule can be omitted, in which case it is the empty string. When the path of a rule is valued (i.e., the last label is an attribute name), the body can also be omitted, and is assumed to be the value of the attribute. The syntax also features some syntactic extensions which are convenient in practice. They are illustrated with some examples, still based on the data graph of Figure 3.

The following query outputs a document with the first name and last name of the reviewers.

```
@db.Person{First name=@firstName Last name=@lastName}
```

This query contains three rules with respective paths expressions `db.Person`, `firstName` and `lastName`. The first expression is evaluated with respect to the initial vertex `db`. One obtains a set of paths whose terminal vertices correspond to tuples of the table *Person*. Each of these vertices is used in turn as an initial vertex for the evaluation of the rules `@firstName` and `@lastName`. On obtains the list of the reviewer's first and last names.

The next query produces a document with the title of the paper 128, and the comments of the reviewers:

```
@db.Paper[id=128]{
  @Review{
    Comment for the paper '@initial.title': @comment
 }
}
```

Note that there can be several reviews for a paper, hence the title of the paper is repeated for each review. The result of this query over the instance of Figure 3 is:

```
Comment for the paper 'Can computer think?': Outstanding
Comment for the paper 'Can computer think?': Ridiculous
```

This query illustrates the special path expression `initial` which simply denotes, in the body of a rule, the initial vertex of the rule evaluation. In this case the rule `@Review` is always evaluated with respect to an initial vertex `Paper`, which can be referred to by the relative path `initial` in the body of the rule.

Note that `initial` always denotes a vertex which has already been visited in the data graph. It can sometimes be equivalent to another path expression: in the above example, `initial` is equivalent to `Paper`, the reverse path of `Review`. However this is not always the case, as shown by the following example which outputs the name of the reviewer:

```
@db.Paper[id=128]{
  @Review{
   @Person{
    Comment of @firstName @lastName
       for the paper '@initial.initial.title': @initial.comment
  }
 }
}
```

Then the path expression `initial.comment` in the body of the rule `@Person` is *not* equivalent to `Review.comment`: the former denotes the vertex `Review` for the paper 128 (which can itself be referred to by the path `initial.initial`), whereas the latter denotes the reviews of all the papers assigned to the reviewer.

As a syntactic facility, we allow the definition of variables to denote the initial vertex of a rule. The following query is equivalent to the previous one, with variables `$P` and `$R` that denote respectively the vertices `Paper` and `Review`.

```
@db.Paper[id=128]::P{
  @Review::R{
   @Person{
    Comment of @firstName @lastName
       for the paper '$P.title': $R.comment
  }
 }
}
```

The "body" and "exception" parts of a rule can support the expression of "if-then-else" programming construct, as shown by the following example which partitions the set of submitted papers in two categories, "accepted" and "rejected", depending on the value of the `accepted` attribute.

```
@db.Paper{
  @self[accepted='Y']{
    The paper @title is accepted.
  }
  {
    The paper @title is rejected.
  }
}
```

The keyword `self` is, as expected, the path expression that refers to the initial node itself. Note that the "exception" part of rule is instantiated when the evaluation of a rule's path expression returns an empty set, and that the initial vertex in the exception part of a rule r is the initial vertex of r itself.

```
@db.Paper[id=128]{
  @Review{
     (...)
  }
  {
    The paper @title has not yet been assigned!
  }
}
```

The previous examples also shows how to express negation (i.e., the exception is instantiated for all the papers that do *not* have a reviewer).

4 Query evaluation

Since the graph is a view of the underlying relational database, we must generate and execute, for each DocQL query, an embedded SQL program that materializes an appropriate part of the graph (see Figure 4). This materialization can then be used as a support for our navigation operators (i.e., paths) and the instantiation of templates (bodies and exceptions).

4.1 Materialization strategies

In a traditional embedded SQL program, SQL queries are provided by the programmer, and each query is optimized independently. In contrast, a DocQL query does not mention SQL operations, and provides a global view of all the data required to construct the result. The DocQL interpreter has therefore the freedom to generate the "best" possible program. Here, "best", can be interpreted in several, non exclusive ways:

1. the program materializes a part of the data graph which is minimal with respect to the query needs;
2. the program is the most "efficient" one (here, efficiency can be defined regarding several criteria: I/O cost, client/server exchanges cost, etc.);
3. the program never materializes twice the same vertex or the same edge.

The evaluation strategy described in this section aims at materializing a minimal subgraph, i.e., the subgraph which is necessary and sufficient to produce the result. We also try to avoid redundant materialization (i.e., SQL queries that retrieve the same tuple(s)), although this may happen in some specific cases (.e.g, independent paths that lead to the same vertex).

Efficiency is considered with respect to the cost of client/server exchanges. It is very common, in a traditional setting where SQL queries are embedded in a programming language, to adopt an object-oriented approach where each database tuple is seen as an object, and retrieved from the database thanks to a query encapsulated in OO methods. This provides a quite convenient framework from the software engineering point of view, but generates a lot of small SQL queries, each accessing a few tuples (typically

only one tuple). From a database viewpoint, this constitutes a degenerate use of SQL [20]. The cost of parsing and optimization becomes quite significant with respect to the cost of data access. In addition such applications exploit poorly the network exchanges between the client application and the database server because this prevents the clustered transmission of groups of tuples (see for instance the client cache and fetch options in JDBC [15]). The embedded SQL program dynamically generated tries to minimizes the number of SQL queries, and favors the grouping of queries that access the same tables and perform the same joins.

Note finally that one could envisage additional constraints such as the maximal amount of available memory which can be assigned to the query. The materialized subgraph can indeed be seen as a cache managed by the DOCQL application. The cache avoids repeated SQL query evaluation and client/server exchanges, but the storage cost can be significant. We ignore this issue in the present paper.

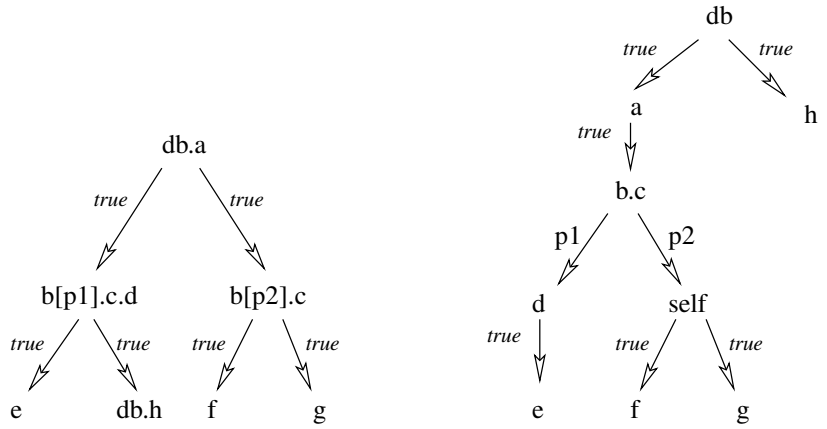
4.2 Graph query representation

We model a DOCQL query q as a *query graph*. It represents an abstraction of an embedded SQL program which materializes the subgraph that supports the evaluation of q. The program alternates cursors on the results of SQL queries, and tests on some values of these results. The query graph representation is as follows:

1. each vertex is a path expression;
2. each edge is labeled with a predicate.

When a DOCQL query is parsed, the interpreter creates an initial representation with a node for each rule, each edge being labeled with $true$. Consider the following example.

```
@db.a{
  @b[p1].c.d{ @e @db.h{...} }

  @b[p2].c{
      @f @g
  }
}
```



a. Initial representation b. After rewriting

Figure 6: Query graphs

The direct translation as a query graph is given in Figure 6, left part. Basically the graph represents the following embedded SQL program (the syntax is meant to be intuitive):

```
for each $v1 in (SELECT * FROM a)
loop
  map($v1) // Map the tuple to the data graph
  for each ($v2 in SELECT * FROM ($v1 JOIN b JOIN c JOIN d)
                   WHERE p1)
  loop
    map($v2)
    for each ($v3 in SELECT * FROM h) loop
     map($v3)}
    end loop
  end loop

  for each ($v5 in SELECT * FROM ($v1 JOIN b JOIN c)
                   WHERE p2)
  loop
   map($v5)
  end loop
end loop
```

That is, each node in the query graph, except the leaves (which correspond to attribute names) defines a cursor on the result of an SQL query. Each tuple is mapped to the data graph representation using the `map` function. The execution of the program produces the subgraph from which the DOCQL query can be evaluated.

The above program is clearly quite inefficient. The loop on the h relation, for instance, is potentially repeated several times. The join between $\$v1$, b and c must also be carried out twice. Our goal is to define a set of rewriting rules to avoid these invariant or redundant SQL evaluations. Still considering the above example, our rewriting strategy yields the query graph of Figure 6, right part. The direct translation as en embedded SQL program is now as follows:

```
for each $v1 in (SELECT * FROM a)
loop
  map($v1)
  for each ($v2 in SELECT * FROM ($v1 JOIN b JOIN c))
  loop
   map($v2)
    if (p1), for each $v3 in (SELECT * FROM $v2 JOIN d)
    loop
     map($v3)
    end loop
  end loop

  if (p2) ...
end loop

for each $v4 in (SELECT * FROM h) map($v4)
```

Note that the query on h is now executed once, and the join between b and c is represented as a single SQL query. This limits the cost of parsing and optimization on the database server side, and allows the packed transmission of tuples. The program above is a "direct" translation because each node from the graph query is mapped to an SQL query. There exists several other possibilities. For instance, from the representation of Figure 6.b, one can obtain the following program which joins `a`, `b` and `c` instead of using a nested loop.

```
for each $v1 in (SELECT * FROM a JOIN b JOIN c)
loop
  map($v1)
  if (p1), for each $v3 in (SELECT * FROM $v2 JOIN d)
  loop
```

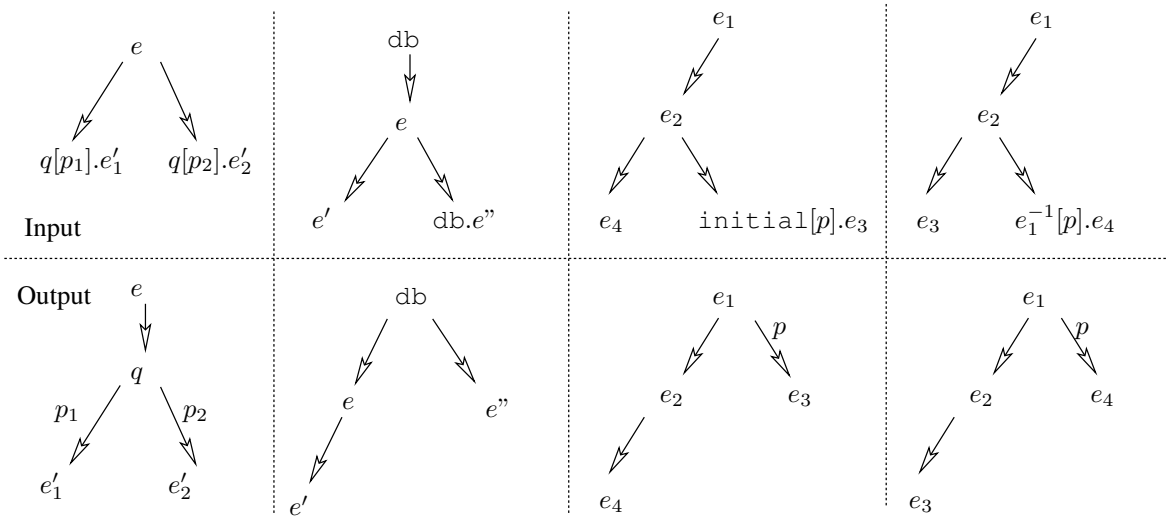
```
    map($v3)
  end loop

  if (p2) ...
end loop

for each $v4 in (SELECT * FROM h) map($v4)
```

The `map($v1)` materializes tuples from `a` and `c`, creating a subgraph that decodes the tabular representation delivered by the cursor. This program executes only once the join `a` $\bowtie$ `b` $\bowtie$ `c`, instead of hard-coding an execution plan that first scan the `a` table, and then sends to the database server a join `b` $\bowtie$ `c` for each tuple retrieved from `a`. Note that most programers would choose this later solution, since it avoids the burden of decoding the tabular SQL result of the full join, which contains many redundant informations (a tuple from `a` must be repeated for each of its associated tuples from `b` and `c`). Hiding the low-level programming primitives (tests and loops) to the database developper lets him design his programs with the convenient concepts (basically, the graph navigation advocated here), and yet provides some easy rewriting mechanisms to obtain an efficient and non-redundant organization of the client-server exchanges. Moreover this enables a "pure" query layer in database programs, accessible to database experts who can tune the data accesses independently from the application logic.

4.3 Rewriting rules

The rewriting rules presented below perform reorganizations from the query graph obtained from the direct translation of a DOCQL query. They mainly consist of rules that apply to *redundant paths* and rules that apply to *invariant paths*.

If e is a path expression in a query graph, and e_1, e_2 are two children of e, e_1 and e_2 are *redundant* if they have a non-empty common prefix. If e_1 (resp. e_2) is of the form $p.e'_1$ (resp. $p.e'_2$), the rewriting rule transforms the tree $(e \to e_1, e \to e_2)$ in the tree $(e \to p, p \to e'_1, p \to e'_2)$ (see Figure 7).



a. Redundant paths b. Invariant paths (1) c. Invariant paths (2) d. Reverse paths

Figure 7: Some rewriting rules

If p is a path expression in a query graph, and p' is a child of p, p' is *invariant* with respect to p if for each terminal vertices v and v' of p in the data graph, $p'(\mathcal{G}_I, v) = p'(\mathcal{G}_I, v')$ (i.e. the interpretation of p' with respect to v and v' are identical). Syntactically, invariant paths are characterized as follows:

1. a rule with an absolute path is always invariant with respect to all its ancestors;

2. if the path of a rule r is of the form `$v.e`, where `$v` is a variable, then r is invariant with respect to all its ancestors up to the rule that defines `$v` (cf. Figure 7.b(1));

3. if the path of a rule r is of the form `initial.`e, then r is invariant with respect to its parent (cf. Figure 7.b(2)). .

A final rewriting rules concerns *reverse paths*. The following example illustrates the issue.

```
@Paper[id=128]{
  @Review.ReviewMark[criterion='Quality']{
     Quality of the paper '@Review.Paper.title': @mark
  }
}
```

This query outputs, for each review of the paper 128, the mark given by the reviewer for the 'Quality' criterion. The title of the paper, denoted by the path expression `Review.Paper.title`, is repeated for each review. In general, it is necessary to execute an SQL query to evaluate a path expression. In this specific case, however, it turns out that `Review.Paper` is the reverse path of `Review.ReviewMark`, the path expression of the context. Moreover, the multiplicity of the path in the graph schema is 1 (see Figure 5). It follows that the query can be rewritten as:

```
@Paper[id=128]{
  @Review.ReviewMark[criterion='Quality']{
     Quality of the paper '@initial.title': @mark
  }
}
```

and the rewriting rule for `initial` holds. The rewriting can be determined from the schema (e.g., $\mu(review.Paper) = 1$), and avoids a useless query (Figure 7.d). Note that the constraint on the multiplicity is necessary. The following query for instance does not give rise to the same rewriting:

```
@Paper[id=128]{
  @Review.Person{
     @firstName @lastName evaluates the paper '@Review.Paper.title'
  }
}
```

`Review.Paper` is indeed the reverse path of `Review.Person` in the schema graph, but the path expression `Review.Paper` actually denotes all the papers assigned to one of the reviewer of the paper 128, and not the paper 128 itself.

4.4 Implementation in MyReview

DocQL has been implemented and is available in the test version of MyReview 2.0. We believe that it will allow non-expert users to produce simple and useful documents in various formats: messages that must be sent to authors or reviewers, list of emails, list of accepted papers, in HTML, LaTeX or plain text. More complex documents, which require a higher level of expertise, can now be produced and maintained much more easily. We plan to provide a library of commonly required documents, available to the chair person and to MyReview administrators.

It would be desirable to provide DocQL querying facilities to reviewers, but this gives rise to some security issues, since each reviewer is only allowed to access the papers that s/he has been assigned to. A useful feature in this context would be the composition of DocQL queries in order to enable the traditional view-based security mechanism. This is part of future work.

The administrator menu of MyReview now contains a DocQL interface (Figure 8). Queries can be entered in a form window, and evaluated over the MySQL database. The resulting document is displayed or saved in a file for further processing. This makes very easy the production of LaTeX or even Excel documents with information extracted from the database, which can then be exploited in another context.

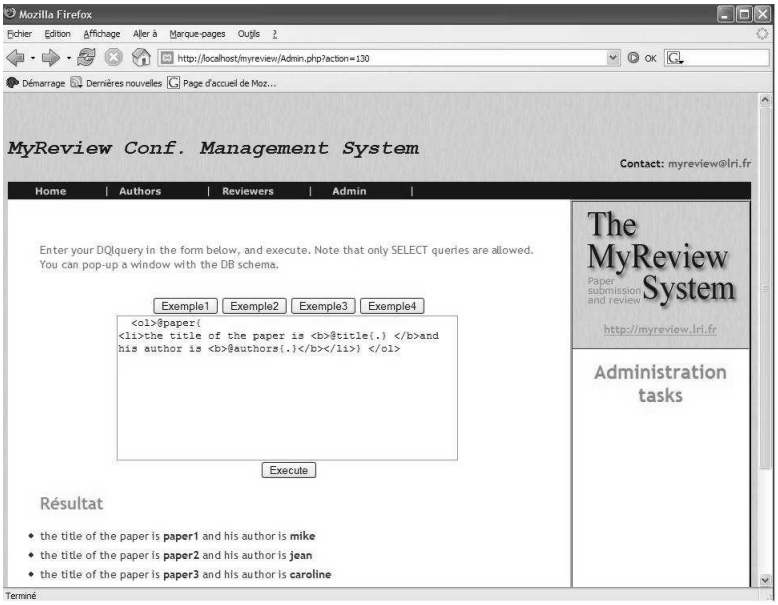


Figure 8: Screenshot of the MYREVIEW interface for DOCQL queries.

The current implementation must be extended in several ways. First the exception part of the rules must be implemented, along with several practical details. In the context of web sites for instance, the HTTP input which often contains parameters can be represented in the data graph as a specific `db.input` subgraph, allowing a consistent management of all the data involved in the document production. Second the translation from a query graph to an embedded SQL program is direct: each node in the query graph yields an SQL query in the evaluation program. The clustering of nodes must be investigated.

Finally our evaluation strategy merely constructs the subgraph, and uses this subgraph as a support for the navigation operations and the application of decoration rules. More subtle techniques must be envisaged. In particular a major issue is the size of the materialized subgraph. Since it is hardly the case that this subgraph must be fully resident in main memory at each step of the query evaluation process, we should devise a progressive evaluation approach which progressively instantiates the subgraph, and pipelines the produced vertices and edges to the decoration rules. Again this is part of near future work.

5 Related work

There exists a few proposals of declarative languages explicitly devoted to relational database publishing. A close work, at least in its motivations, is SuperSQL [21] which extends SQL to insert tuples in textual fragments. The language remains strongly limited by the absence of recursion or nesting. In general, our framework can be seen as an application to relational databases of formalisms developed in the context of semi-structured databases [1]. In particular the processing model is similar to that of the XSLT language [23] or UnQL [8].

There has been recently a lot of publications with the goal of using a relational database management system (RDBMS) to store and/or query XML data. Several papers deal with *XML publishing*, i.e., exporting existing relational data in an XML view. There exists a significant difference between the XML publishing motivation and the approach advocated in the present paper. XML publishing mostly aims at defining an XML representation of the database in order to make data accessible to the external world. The XML views behave then as a support for further querying, using XML query languages such as XPath or XQuery. Some of the main issues in this context are the SQL extension to specifying the XML output [16], the efficient materialization of the XML document [9, 17, 10, 2, 6], and the verification of a view validity with respect to a DTD [3]. In contrast, our language works directly on the relational database, and uses a high-level

specification of the required data, from which SQL queries can be inferred through rewriting operations.

In practice, all the relational database systems provide facilities to manage and generate XML documents [14, 12, 13]. Basically they represent the result of a query as an XML document with three levels (result set, row and attribute). From this export, an XSLT stylesheet can be applied. The combination of XML publishing and XSLT constitutes a candidate solution to our problem. However, although useful, this approach leads to highly complex architectures and necessitates the definition of two distinct programs (one for the publication, another for the transformation). Since they are independent from one another, nothing guarantees that the XML view corresponds to the data required by the XSLT program. The works [19, 18] show that the *composition* of XML publishing with XSLT transformation is a highly difficult problem. As already discussed in the Introduction, a successful approach is SilkRoute [16] which composes two specialized languages (i.e., RXL for the XML view, and XML-QL for the query language). Our approach, which aims at different goals, makes the intermediate XML materialization superfluous.

The specification of the exported data as a tree of co-related SQL queries can be viewed as an abstraction of nested cursors over result sets, each defined with respect to its parent(s) [4, 11]. A noticeable proposal is the ROLEX system [5] which offers a DOM interface on a relational database, and supports the DOM operators which are converted on the fly in SQL queries, thereby supporting navigation over a virtual XML document. An incremental evaluation strategy is also presented in [6].

6 Concluding remarks

We proposed in the present paper a simple and concise publication language, DOCQL. One of our main objectives is to limit the task of the user to the minimal set of instructions necessary to specify the resulting document. This comes from the observation that the current solutions (roughly SQL + general purpose language) is both too strong and inadapted to such simple tasks. Relational database publishing currently requires repetitive programming patterns and the manipulation of several languages.

DOCQL enables a data-driven language which avoids to resort to classical programming languages and relies only on the schema and instance of the database, uniformly represented in a graph. We defined the main characterictics of the language (syntax, operational semantics), showed that is supports optimization techniques, and briefly described its implementation and use in MYREVIEW. Many important aspects remain to be investigated: typing (i.e., compile-time verification of the output document validity with respect to a given grammar), closure and composition of queries, optimal query evaluation and rewriting, etc.

The rewriting rules and evaluation strategy aim at eliminating redundant queries or repetitive execution of invariant queries. They show the potential and flexibility of the language for optimization of large sets of interrelated SQL queries. We plan to use this flexibility in the context of object-oriented interfaces (e.g., hibernate, *http://www.hibernate.org*) which promote a one-tuple-at-a-time access strategy to relational databases. In such a context our query graph can be interpreted as the "profile" of a program, allowing the OO interface to anticipate the fetching of tuples and their grouped transfer from the server. The materialized subgraph acts as a structured client cache, raising several issues regarding the amount of allocated memory, the partitioning of the query graph in SQL queries, and the asynchronous transfer between the client and the server. Such an application represents an extension of the motivation that underlies the present paper: separating the programming aspects from the client-server SQL exchanges.

Finally, although DOCQL eliminates the programming effort and reduces the complexity of the document production, it still requires a good knowledge of the graph schema. In the next step we plan to investigate a "show-and-publish" approach, where a user gives as input a sample document that contains values extracted from the database (e.g., a message to reviewers) and marked as placeholders. The system should then infer the DOCQL query that produces the documents and all its possible instances.

Acknowledgements. We are grateful to C. du Mouza and D. Gross-Amblard for their useful comments on early drafts on this paper.

References

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[2] S. Amer-Yahia, Y. Kotidis, and D. Srivastava. XML Publishing: Look at Siblings too! In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 711–713, 2003.

[3] M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. DTD-Directed Publishing with Attribute Translation Grammars. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 838–849, 2002.

[4] P. Bohannon, H. Korth, and P. Narayan. The Table and the Tree: Online Access to Relational Data through Virtual XML Documents. In *Intl. Proc. on WebDB 2001 Workshop on Databases and the Web*, 2001.

[5] P. Bohannon, H. Korth, P.P.S. Narayan, S. Ganguly, and P. Shenoy. Optimizing View Queries in ROLEX to Support Navigable Tree Results. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2002.

[6] Philip Bohannon, Peter Buneman, Byron Choi, and Wenfei Fan. Incremental Evaluation of Schema-Directed XML Publishing. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 503–514, 2004.

[7] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proc. Intl. Workshop on Database Programming Languages*, pages 9–19, 1991.

[8] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.

[9] M. Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.

[10] Surajit Chaudhuri, Raghav Kaushik, and Jeffrey F. Naughton. On Relational Support for XML Publishing: Beyond Sorting and Tagging. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 611–622, 2003.

[11] B Choi, W Fan, X Jia, and A Kasprzyk. A Uniform System for Publishing and Maintaining XML Data. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 1301–1304, 2004.

[12] IBM corp. *DB2 XML Extender*. http://www-306.ibm.com/software/data/db2/extenders/xmlext/index.html.

[13] Microsoft corp. *A survey of microsoft sql server 2000 xml features.* http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml07162001.asp.

[14] Oracle Corp. *XML, XSLT and Oracle8i.* http://technet.oracle.com/tech/xml/xsql_servlet/.

[15] Sun Corp. *JDBC Technology*. http://java.sun.com/products/jdbc/.

[16] M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. SilkRoute: A framework for publishing relational data in XML. *ACM Trans. on Database Systems*, 27(4):438–493, 2002.

[17] M. F. Fernandez, A. Morishima, and D. Suciu. Efficient Evaluation of XML Middle-ware Queries. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2001.

[18] C. Li, P. Bohannon, H. F. Korth, and P. P. S. Narayan. Composing XSL Transformations with XML Publishing Views. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 515–526, 2003.

[19] G. Moerkotte. Incorporating XSL Processing Into Database Engines. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2002.

[20] Dennis Shasha and Philippe Bonnet. *Database Tuning: principles, experiments, and troubleshooting techniques*. Morgan- Kaufmann, 2002.

[21] M. Toyama and T. Nagafuji. Dynamic and Structured Presentation of Database Contents on the Web. In *Proc. Intl. Conf. on Extending Data Base Technology*, pages 451–465, 1998.

[22] The XPath language recommandation (1.0). World Wide Web Consortium, 1999. http://www.w3.org/TR/xpath.

[23] The Extensible Stylesheet Language Family (XSL). World Wide Web Consortium, 1999. http://www.w3.org/Style/XSL.