

Declarative Embedding of Relational Data in Textual Documents ^{*}

Philippe Rigaux⁽¹⁾ and Emmanuel Waller⁽²⁾

⁽¹⁾ LAMSADE, Univ. Paris-Dauphine, France, rigaux@lamsade.dauphine.fr

⁽²⁾ LRI, Univ. Paris-Sud Orsay, France, waller@lri.fr

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 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. Both operations can be expressed by a declarative query language called DocQL.

Our approach can be seen as an extension of the relational algebra to some restricted programming structures. DocQL allows a very concise specification of database publishing applications whose behaviour depends only on the database instance. 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.

A relational query maps a set of finite relations (instance of the database) to a finite relation (the query result). This is a strong constraint which permits powerful optimization mechanisms, but falls short in practice because the vast majority of applications have to move outside flat relations, both for manipulation and presentation purposes. Adding a programming language to SQL represents a complete shift in expressive power since it becomes possible to program everything, including access plans which should be left to the database optimizer. It would therefore be desirable to extend database optimization techniques to embedded queries, i.e., to sequences of SQL queries combined with

^{*} Research supported by the CRIT GVD project.

programming constructs. Unfortunately, it is generally considered that their desirable properties (termination, low data complexity, 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 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 many 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 the following capabilities:

- XML publication of relational data for exchanges purposes (e.g., RSS);
- mail messages that include database information ;
- L^AT_EX (or any other text processor) sophisticated layout of database content;
- reports of various kinds.

It turns out that none of these outputs can be produced by SQL. Therefore programmers commonly use the SQL language embedded in a general-purpose one, because the latter provides programming constructs (loops, tests, variables, output primitives) missing in SQL. Our work is motivated by the conviction that the publication requirements can be fully satisfied by restricted programming tools, with two consequences:

1. understanding/proving properties of such programs is likely to be much more easier, and makes it possible to study optimisation techniques for SQL queries embedded in such a (restricted) programming language;
2. high-level, *declarative*, specifications of such programs can be envisaged.

Publishing applications offer an opportunity to define an intermediate solution between SQL and embedded SQL. The main assumption of the current work is that the required query language represents only a slight extension of SQL, and stays therefore close to the simplicity of the relational algebra. Yet we need to augment the language with more powerful output constructs in order to enable the production of rich structures beyond flat relations. In the present paper we focus on the production of data graphs, since they are naturally suited for representing documents. Another important issue is the high-level specification of publishing programs. Since our goal reduces to extend SQL with a few programming primitives, we can expect that there exists a declarative counterpart of this language which hides the programming constructs. By “declarative”, we mean, as it is customary in the database realm, a syntax which hides the algorithmics or imperative aspects and let the system determine the best execution strategy. In other words, the language should allow a global specification of

the program in order to let the optimizer manipulate it as a whole, instead of viewing it as a set of unrelated SQL queries.

We propose a relational publishing language named DOCQL which combines the following mechanisms:

1. navigation primitives in the relational database;
2. instantiation of *fragments* which contribute to the final result.

A design choice of DOCQL is that it does not rely on the relational database instance, but rather on a view 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 powerful navigation than SQL cursors which are limited to linear scan of query results.

The rest of the paper presents first (Section 2) a motivating example. The model is given in Section 3 and a simple evaluation strategy is proposed in Section 4. Section 5 discusses related work and Section 6 concludes the paper and outlines future work.

2 Motivation

We shall illustrate the main intuitions behind our work with a web application, MYREVIEW, which supports the submission phase of conferences. Its main functionalities are probably already familiar to the reader: *authors* can submit *papers* along with a description (abstract, title, topics), whereas *reviewers* are assigned to papers in order to provide an evaluation. The system uses a relational database for data storage and retrieval. As most web applications, it relies intensively on documents exchanges between the various actors. Let us consider these aspects in turn.

The relational database

Figure 1 shows a sample of the relational database. Each paper corresponds to a tuple in *Paper* and is associated to one or several authors (table *Author*). Persons are identified by their email in every table, and *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 (*Review*) and a list of marks, for some criteria (table *ReviewMark*).

The representation is standard. Each tuple consists of a set of attributes values, and is identified by its key (in boldface). The key is used to refer to a tuple *t*: this requires the storage of the key value in the referring tuple as an additional attribute (foreign key, in italics; note that they can be part of a primary key as well). For instance the referee of a paper (*Review*) in a paper is referred to by means of its email, which is the key of *Person* tuples.

<i>idPaper</i>	title	year
128	Do computer think?	1951

Paper

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

Person

<i>idPaper</i>	<i>email</i>
128	turing@nw.com

Author

<i>email</i>	<i>idPaper</i>	comment
r1@sw.com	128	Ridiculous
r2@ew.com	128	Outstanding

Review

<i>email</i>	<i>idPaper</i>	critierion	mark
r1@sw.com	128	quality	3
r1@sw.com	128	relevance	4
r2@ew.com	128	quality	5

ReviewMark

Fig. 1. A database instance

This representation provides a sound support for SQL querying. When the data of interest is distributed in several tables, a *join* must be performed. A join associates tuples that share common values for a subset of their attributes. In practice, a join is almost always based on a matching between the primary key of a table and the corresponding foreign key of some other one. Here is for instance the SQL query that retrieves the title and authors' name of papers submitted in 1951.

```
select title, firstName, lastName
from Paper, Author, Person
where Author.email = Person.email
and Paper.idPaper = Author.idPaper
and year = 1951
```

Note that, although the language itself is simple, expressing queries requires both a good knowledge of the specific schema and of the general design principles of relational databases. For instance associations between tuples are not symmetrically represented (nothing indicates that a *Person* authored a paper). Note also that the language is set-oriented: the result of a query is a relation, i.e., a set of tuples. There is no mean to create more involved structures.

Documents

The MyREVIEW system, as many others of its kind, produces documents built from information extracted from the database. Each user interaction entails the production of an HTML page, and several other kinds of documents are also required to be produced. Consider for instance a report on the reviewers' evaluations regarding a given paper. This report exists in the following formats:

- an HTML page;
- a mail message, sent 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 1.

```
...
Paper: Do computer think?
Author: Alan Turing
Abstract: ...

<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}
...
```

Finally the notification mail is another variant.

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:
...

This last example of mail notification requires slightly more expressive primitives. Indeed the layout depends in that case of some attributes values, namely the final status of the paper (the column storing this status is not shown in the example). Clearly some parts of the notification message change whether the paper is accepted or refused, although the list of reviews is still present.

There is no way to produce such documents with pure SQL. On the other hand, even if the formats of these documents differ greatly, they all rely on the same data skeleton: a paper (level 1) with all its reviews (level 2), and each review in turn comes with its marks (level 3). Specifying this hierarchical organization is, conceptually, extremely simple: we just have to follow the links between a paper and its reviews, then between a review and its marks. This navigation is driven by the database content, as well as the decisions made occasionally to follow or not some links (for instance the choice of the fragments associated respectively to the 'accepted' or 'refused' status).

Our claim is that most publishing programs are limited to these data-driven mechanisms. In the next section we propose a language that expresses this navigation. Moreover we show that this corresponds, in terms of embedded SQL, to a simple class of programs which can be precisely characterized.

3 The model

As mentioned in the Introduction, we define our language over a view of the relational database that represents a *data graph*. The graph is virtual, and must be materialized on demand whenever a query evaluation needs to access some of its parts, but it serves as a support to expressing queries with our language, named DOCQL.

Overview

The following diagram summarizes the model. A DOCQL query posed over the virtual graph allows to obtain the result. We would have otherwise to write an embedded SQL program, followed by a transformation of the result. A DOCQL is much more simpler, and can be rewritten and evaluated as a (restricted kind of) program. The present section describes our model. The next one describe how we can rewrite a query in embedded SQL.

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 finite labeled graph (V, E) with the following structure.

1. $V \subseteq \mathcal{T} \cup \mathcal{R}$;
2. each edge in E is of one of the following cases;
 - (a) $r \xrightarrow{a} \tau$, with $r \in \mathcal{R}, a \in \mathcal{A}, \tau \in \mathcal{T}$;
 - (b) $r \xrightarrow{s} s$, with $r, s \in \mathcal{R}$; and in this case there is also an edge $s \xrightarrow{s} r$;
3. for all $r \in \mathcal{R}, a \in \mathcal{A}$, there exists at most one $\tau \in \mathcal{T}$ such that $r \xrightarrow{a} \tau$;
4. for all r, u, v there exists only one occurrence of $r \xrightarrow{u} v$ (no multiple edges).

Now, let \mathcal{S} 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 $S = (V, E)$ be a schema. An instance $I = (V_I, E_I)$ of S is a mapping from S to connected labeled graphs defined by a pair of mappings (I_1, I_2) as follows $(V_I = I_1(V), E_I = I_2(E))$:

1. for each $r \in V \cap \mathcal{R}$, $I_1(r) \subseteq \mathcal{S}$;
2. for each $\tau \in V \cap \mathcal{T}$, $I_1(\tau) \subseteq [\tau]$;
3. for all $r \in V \cap \mathcal{R}, \alpha \in \mathcal{A} \cup \mathcal{R}, \beta \in \mathcal{R} \cup \mathcal{T}$, $I_2(r \xrightarrow{\alpha} \beta) \subseteq \{x \xrightarrow{\alpha} y \mid x \in I_1(r), y \in I_1(\beta)\}$;
4. for all $x \in V_I \cap \mathcal{S}, a \in \mathcal{A}$, there exists at most one $\tau \in \mathcal{T}$, and one $y \in [\tau]$, such that $x \xrightarrow{a} y \in E_I$.

If r is a relation name, $I_1(r)$ is the set of nodes in r . Any access to the database must be through relation names, as in the relational model, and is called *entry points*. Given a relational database, there exists a one-to-one mapping between the relational schemas and instances and the graph schemas and instances. Moreover, this mapping is effective, that is, there exists an algorithm computing it.

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 node denoted by a path. Our paths expressions correspond to a restriction of the XPath language [15]. In its simplest form, a path is a sequence of edges pairwise connected in a graph schema S . A path is *valued* if its last edge is an attribute name. A *predicate* is any Boolean combination of atomic formulas of the form $p \theta value$ where p is a valued path. The general form of a path is:

$$e_1[p_1].e_2[p_2].\dots.e_l[p_l]$$

where e_i are adges and p_i are (optional) predicates. A path is interpreted with respect to a given node in the graph as follows.

Definition 3 (Path interpretation) Let $p = a_1[p_1].a_2[p_2].\dots.a_l[p_l]$, $l \geq 0$, be a path in S , I an instance of S , and v a node in I . The interpretation of p over I from v is defined inductively as follows. Let $v.a_1[p_1]$ denote the set $\{m \mid v \xrightarrow{a_1} m \wedge p_1 \text{ is true}\}$.

1. $p(I, v) = \bigcup_{m \in v.a_1[p_1]} q(I, m)$, where $q = a_2[p_2].\dots.a_l[p_l]$;
2. $\epsilon(I, m) = m$ (where ϵ denotes the empty path).

If a_1 is a database entry point, the path is *absolute* and it is useless to provide v . Now, let A be a finite alphabet. Rules are defined as follows.

Definition 4 (Rules) A rule is a pair (p, b) or a 3-tuple (p, b, e) , where p is a path, and b and e are finite sequences of words and rules over A , called the body of the rule.

A *query* is simply a rule $r(p, b)$ such that p is an absolute path (i.e., it can be evaluated from an entry point of the database).

Definition 5 (Rules semantics) Let $r = (p, b)$ (or (p, b, e)) be a rule, with $b = u_0 r_1 u_1 \dots r_n u_n$, $n \geq 0$, and $p(I, v) = \{v_1, \dots, v_k\}$, $k \geq 0$. Let I be an instance of S , and v a node in I . The semantics $[r(I, v)]$ of r over I from v is defined inductively as follows:

1. $[r(I, v)] = [b(I, v_1)] \dots [b(I, v_k)]$;
and if $k = 0$:
(a) if $r = (p, b)$, then $[r(I, v)] = \epsilon$;
(b) if $r = (p, b, e)$, then $[r(I, v)] = [e(I, v)]$;
2. for each node m , $[b(I, m)] = u_0[r_1(I, m)]u_1 \dots [r_n(I, m)]u_n$.

As a special case, the semantics of a query r is $[r(I)]$. This 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 nodes returned by the paths of the step.

Example

Figure 3 shows a DOCQL query over our sample database (we assume an additional attribute *accepted* in table *Paper*). The query produces a document which summarizes the status of a paper, along with its reviews. The concrete syntax for rules is of the form `@path{body}`, and ‘.’ denotes the empty path ϵ .

The evaluation of this query over the instance of Fig. 2 can be described as follows. First one accesses the paper whose id is 128 (framed by the box in the figure). From this node the paths `title`, `author.person.firstName` and `author.person.lastName` lead to the appropriate values. 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 node. The evaluation of this rule can be described similarly.

```

@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}
    {The paper is going to be rejected}

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

```

Fig. 3. A sample query

4 Query evaluation

Since the graph is a view of the underlying relational database, we must translate any DOCQL query toward an embedded SQL program in order to materialize an appropriate part of the graph. This materialization can then be used as a support for our navigation operators (i.e., paths). The materialized part should be minimal with respect to the query needs, and the embedded SQL program should be as efficient as possible. We describe informally in this section a simple evaluation strategy. A precise study and comparison with alternative evaluations is left for future work.

A DOCQL query can be naturally represented as a tree of rules. Each node in the tree is a rule body, and each edge is labelled by a path. Our strategy consists of the following steps:

- use rewriting rules to put the query tree in normal form;
- generate the SQL queries corresponding to edges in the normalized query;
- generate the embedded SQL program for the whole query tree.

We illustrate these steps upon the example of Fig. 3. The normal form (NF) is defined as follows:

1. all paths are exclusively composed of relation names;
2. if b is a body, and p_1, p_2 two paths rooted at b , then p_1 is not a prefix of p_2 , and conversely;
3. all the rules remaining in bodies are of the form $a[*pred*]$, where a is an attribute name and $pred$ a (possibly empty) predicate;
4. each body is uniquely identified by a binding variable $\$v$.

Figure 4 shows a rule tree in NF for our example. We omit for the time being the rule @accepted.

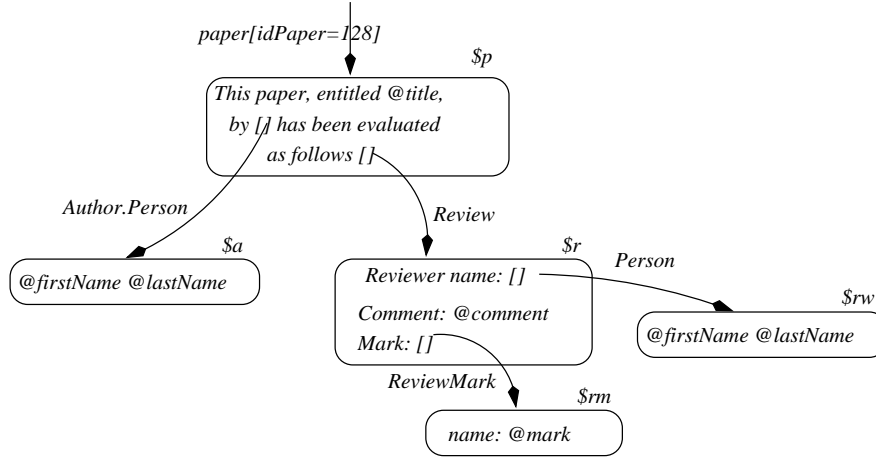


Fig. 4. The normal form for the example of Fig. 3

Since the paths `author.person.firstName` and `author.person.lastName` in the query have a common prefix, the two rules have been merged. Thanks to some simple rewritings (not presented here), the normal form is equivalent to the original one, and complies to the constraints defined above. Each body is associated to a unique label (resp. $\$p$, $\$a$, $\$r$, $\$rm$ and $\$rw$ on the figure).

Consider now the following rule:

```
@accepted['Y']{Congratulations, the paper is accepted}
                {Sorry the paper is rejected}
```

The rule is of the form $r(p, b, e)$, where p is, by definition of the normal form, a path of length 1 composed of an attribute name with a predicate. If there exists at least one instantiation of p (that is, if `accepted` equals to 'Y'), then b must be evaluated, otherwise e . The choice of evaluating either b or e is made at runtime, and has some implications on the global evaluation strategy of the whole DOCQL query, because b and e might correspond to quite different query trees (for instance b could list the accommodations of the conferences, whereas e could list some alternatives submission opportunities for the rejected paper). Observe that, b and e are in normal form as well.

Intuitively, each path in a NF query can be evaluated by an SQL query, whereas each rule in a body is an attribute of the binding variable. In general (the root of the query tree being a special case), a path $P = e_1[p_1].e_2[p_2].\dots.e_n[p_n]$ is the label of an edge (b_1, b_2) in the normalized query tree. Let $\$v_1$ and $\$v_2$ be the binding variables of the bodies b_1, b_2 , respectively. At runtime, given an instance of $\$v_1$, we search all the possible instantiations of v_2 such that $v_1.r_1.r_2.\dots.r_n.v_2$ is a path satisfying p_1, p_2, \dots, p_n in the data graph. These instantiations can be obtained by the following SQL query:

```
select * into $v2
```

```

from $v1 Join l1 Join l2 ... Join ln // --- ? MAnu : r1, ... rn ?
where p1 and p2 and .. and pn

```

where Join denotes (by a small abuse of notation) the join operation which implements the semantics of an edge in the data graph. Let's consider again the example of Fig 3. The paths in a normalized query are thus materialised in SQL as follows:

```

- Paper[idPaper=128]: SELECT * FROM Paper WHERE idPaper=128
- Author.Person: SELECT * FROM Author WHERE idPaper=$p.idPaper
- Review: SELECT * FROM Review WHERE idPaper=$p.idPaper
- Person: SELECT * FROM Reviewer WHERE email=$r.email
- ReviewMark: SELECT * FROM ReviewMark WHERE idPaper=$r.idPaper AND
  email=$r.email

```

Each binding variable of a body b is used as a parameter for the paths of the descendants of b . Clearly, each rule r in a NF query can therefore be implemented as a function that (i) finds all the instantiations of the path, (ii) binds each path instance to a variable v_r , and (iii) calls the functions of the children of r , using each binding of v_r as a parameter. Here is for instance the function that implements the first rule of Figure 4.

```

Rule-Paper()
begin
  for each ($p IN SELECT * FROM Paper WHERE idPaper=128)
  loop
    Call Rule-Author ($p)
    Call Rule-Review ($p)
  end loop
end

```

The structure of the evaluation program (a composition of function calls) is quite representative of common SQL programming practices. Essentially it consists of several nested cursors which perform a depth-first traversal of the database. A striking feature of this programming pattern is that the data accessed by a program form a tree in our virtual data graph. The depth of the tree is determined by the database schema (because, roughly speaking, it is the number of tables that must be accessed), but the breadth is conditioned by the instance of the database. The key point is that, whereas both aspects are handled quite differently with a classical programming language, they are treated uniformly in our approach which does not distinguish the schema from the instance.

Any DOCQL query can be evaluated by a restricted embedded SQL program which consists only of the following operators: SQL, a LOOP construct which binds successively all the tuples in a result set, and an IF construct which evaluates only the truth value of boolean expressions over binding variables. This language enables "data-driven" SQL programming and corresponds in practice to the functionality required for database publishing. The DOCQL language provides a much easier and simpler way of expressing such programs. An interesting (and open) question is whether any program in this language can be expressed in DOCQL. This is part of future work.

5 Related work

There exists few proposals of declarative languages explicitly devoted to relational database publishing. A close work, at least in its motivations, is SuperSQL [14] which extends SQL to insert tuples in textual fragments. The language remains limited by the absence of recursion or nesting. Another important and relevant work is Strudel [11], a system which attempts to define web sites declaratively as a tree of queries on semi-structured data. Several rich ideas can be found, including navigation operators. 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 execution model is similar to that of the XSLT language [16] 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. The main issues are languages for specifying views [12], verification of a view validity with respect to a DTD [3] and optimization techniques [9, 2, 6].

There exists a subtle 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. An important problem in this context is the *composition* of XML queries over the XML view, with SQL queries defining the view (see, for instance, SilkRoute [12]). Here, we do not have to compose queries. Rather, we translate navigation operators in SQL extended with (limited) loops and tests.

Commercial RDBMS provide tools for exporting in XML some parts of the database. 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 is not satisfactory because it 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.

A well established concept for XML publishing, both in research prototypes and commercial products, is the specification of the exported data as a tree of co-related SQL queries. Basically this is an abstraction of nested cursors over result sets, each defined with respect to its parent(s). Query trees are investigated in [4, 3, 5, 2, 9, 6, 10]. 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. The next step consists in applying XSLT programs to such virtual DOMs, in order to import in the relational realm the benefits of the XSLT standard [13].

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 requires repetitive programming patterns and the manipulation of several languages.

DOCQL enables a data-driven language which avoids to rely on classical programming primitives by relying only on the schema and current instance of the database, uniformly represented in a graph. We defined the main characteristics of the language (syntax, operational semantics and a simple evaluation strategy). 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.

A promising aspect of this work is that it can easily be adapted to other contexts. An important target is the ability to evaluate XSLT programs over a relational database. XSLT is a widely used language which is designed for XML documents. Using XSLT in a relational setting involves an intermediate step (a conversion to XML) which raises several technical difficulties. Since the execution model of DOCQL is very close from that of XSLT, we can envisage to define a subset of the latter that evaluates directly on the underlying relational database and avoids the XML conversion phase. This requires the definition of an appropriate fragment of XPath and XSLT programming constructs, in order to preserve the evaluation capabilities of our framework.

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. 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.
10. 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.
11. M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the Boat with Strudel: Experiences with a Web-Site Management System. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 414–425, 1998.
12. 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.
13. 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.
14. 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.
15. The XPath language recommendation (1.0). World Wide Web Consortium, 1999. <http://www.w3.org/TR/xpath>.
16. The Extensible Stylesheet Language Family (XSL). World Wide Web Consortium, 1999. <http://www.w3.org/Style/XSL>.