

Data-Driven Publication of Relational Databases *

Sonia Guéhis
LAMSADE UMR CNRS 7024
Université Paris-Dauphine
sonia.guehis@dauphine.fr

Philippe Rigaux
LAMSADE UMR CNRS 7024
Université Paris-Dauphine
philippe.rigaux@dauphine.fr

Emmanuel Waller
LRI UMR CNRS 8623
Université Paris-Sud Orsay
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 data graph which supports navigation operators. Applications can express the data they need by navigating in the graph. These operations are provided by a declarative query language over virtual graphs, named DOCQL. 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. 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.

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 web sites, as well as many other publication areas (XML publication, mail messages, \LaTeX files, 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. We aim at defining a very simple, direct and concise language to meet these 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.

The rest of the paper develops our motivation, and describes the syntax and semantics of the language. A DOCQL query can be easily represented in a formalism which gives rise to rewriting and optimization techniques: due to space limitations the interested reader is referred to the extended version (<http://www.lamsade.dauphine.fr/rigaux>). DOCQL is implemented and used in MYREVIEW (<http://myreview.lri.fr>), a widely used conference management system which strongly resorts to the production of various documents.

Related work

DOCQL constitutes a direct approach to data publishing from a relational database which tries to avoid the burden of repetitive programming tasks. 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 [11] or UnQL [3]. An alternative would be to generate an XML document from the relational instance (using, say, the XSQL utility of Oracle [7]), and then to transform this document with an XSLT stylesheet. Some recent research works attempt at enabling a composition of XML export and XML publishing languages [8, 6]. The composition algorithm of SilkRoute [8] for instance avoids a full materialization. Several other significant papers deal with *XML publishing*, i.e., exporting existing relational data in an XML view [4, 9, 5, 2]. Although

*Research supported by the CRIT GVD project.

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.

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.

Relational database publishing

Figure 1 shows a sample of the MYREVIEW 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 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.

The MYREVIEW system, as many others of its kind, produces documents built from information extracted from the database. 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 L^AT_EX document whose output can be printed by the PC chair. Here is for instance a (simplified) HTML version of the report, produced from the instance of Figure 1.

```
...
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 L^AT_EX 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.

The notification mail produces the reviews on a paper, following the general layout of the above documents. 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.

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.

A part of the data graph is shown in Figure 2 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

idPaper	title	year	accepted
128	Do computer think?	1951	Y

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	critierion	mark
r1@sw.com	128	quality	3
r1@sw.com	128	relevance	4
r2@ew.com	128	quality	5

ReviewMark

Figure 1. A database instance

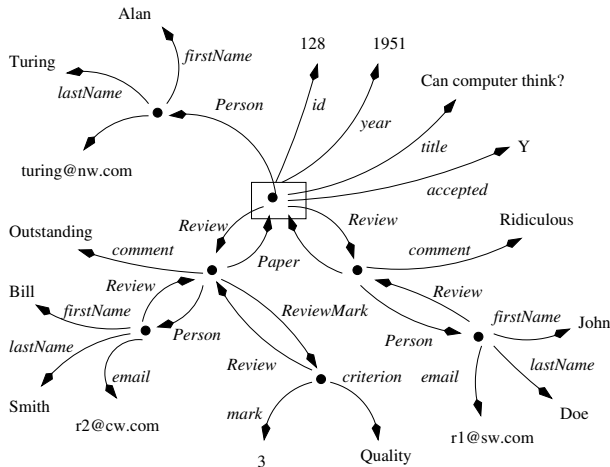


Figure 2. A part of the virtual graph

are usually distinguished in a relational database, namely tuple-to-attribute and tuple-to-tuple.

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 the moment.

```
@paper[idPaper=128]{
  This paper, entitled @title, written by
  @person.firstName @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. 2 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.

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 3 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 $q(\mathcal{G}_I)$.

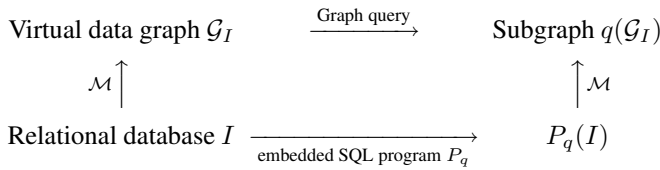


Figure 3. Evaluation of a DocQL query q

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.

3 The model

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 $r \xrightarrow{\lambda(e)} s$, with $r, s \in \mathcal{R}$, there exists an edge $e' \in E$ of the form $s \xrightarrow{\lambda(e')}$, called the reverse edge of e ;

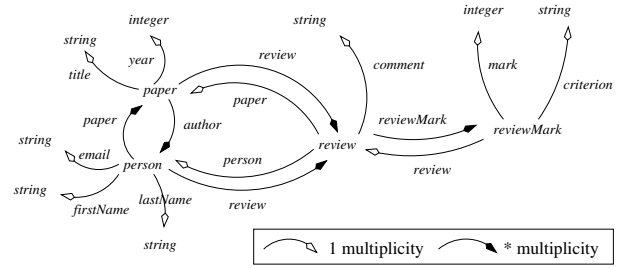


Figure 4. The schema of the data graph

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 4 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 4 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:

1. for each $v \in V$
 - $\{ V_I(v) \subset \mathcal{I} \text{ if } v \in \mathcal{R} \text{ (tuple - to - tuple)}$
 - $\{ V_I(v) \subset [v] \text{ if } v \in \mathcal{T} \text{ (tuple - to - value)}$
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 db in V_I such that, for each $r \in V \cap \mathcal{R}$ and for each $v \in V_I(r)$, $db \xrightarrow{r} v \in E_I$.

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.

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 [10]. 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. The general form of a path expression is $l_1[p_1].l_2[p_2].\dots.l_n[p_n]$. A path expression is *valued* if its last label is an attribute name. It is *valid* if the path denoted by $l_1.l_2.\dots.l_n$ 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.

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 $q = db$, $E_I(q, v) = \emptyset$ and $V_I(q, v) = \{db\}$.
2. if $q = l$, where l is a label, $E_I(q, v) = \{e \in E_I \mid e \text{ is of the form } v \xrightarrow{l} v'\}$, 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

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

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 2. 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 “Doe”. 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 $V_I(q, v) = \{v_1, \dots, v_k\}, k \geq 0$, then

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

As a special case, the semantics of a query r is $[r(\mathcal{G}_I, db)]$. 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.

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 2.

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

```
@db.Person{@firstName @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`. 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 '@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 2 is:

```
Comment for 'Do computer think?': Outstanding
Comment for 'Do 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.

Acknowledgements. We are grateful to C. du Mouza and D. Gross-Amblard for their useful comments.

References

- [1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] Oracle Corp. *XML, XSLT and Oracle8i*. http://technet.oracle.com/tech/xml/xsql_servlet/.
- [8] 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.
- [9] 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.
- [10] The XPath language recommendation (1.0). World Wide Web Consortium, 1999. <http://www.w3.org/TR/xpath>.
- [11] The Extensible Stylesheet Language Family (XSL). World Wide Web Consortium, 1999. <http://www.w3.org/Style/XSL>.