Views for simplifying access to heterogeneous XML data

Dan Vodislav[1], Sophie Cluet[2], Grégory Corona[3], and Imen Sebei[1]

[1] CNAM/CEDRIC, Paris, France
[2] INRIA, Rocquencourt, France
[3] Xyleme, Paris, France

Abstract. We present *XyView*, a practical solution for fast development of user- (web forms) and machine-oriented applications (web services) over a repository of heterogeneous schema-free XML documents. XyView provides the means to view such a repository as an array, queried using a QBE-like interface or through simple selection/projection queries. Close to the concept of universal relation, it extends it in mainly two ways: (i) the input is not a relational schema but a potentially large set of XML data guides; (ii) the view is not defined explicitly by a query but implicitly by various mappings so as to avoid data loss and duplicates generated by joins. Developed on top of the Xyleme content management system, XyView can easily be adapted to any system supporting XQuery.

1 Introduction

For decades, companies have produced digital data such as notes, contracts, emails, progress reports, minutes, etc. This data constitute a mine of useful information that is largely unexploited. The advent of XML provides the opportunity to change that. Many enterprises are now considering storing their home data in XML repositories so as to be able to query them in a significant way, i.e., with tools more sophisticated than full text search engines. In this paper, we are addressing the problem of querying such repositories. More precisely, we are interested in developing, easily and quickly, a simple query API (web services) or user interfaces (web forms) over these repositories.

An important characteristic of the applications we are considering is that they deal with legacy data that have been mostly produced by human beings using standard text editors. As a result, the data is (i) poorly typed (well formed rather than valid XML) and (ii) highly heterogeneous (although documents have strong semantic connections). These features are particularly challenging since they call for sophisticated tools to ease the application programmer task while at the same time disabling most existing approaches.

The solution we propose borrows from the universal relation paradigm of the seventies [18]: *XyView* provides the means to easily view a set of heterogeneous XML documents as a single array that can be queried through simple selections and projections. Obviously, the context being XML, the array contains XML subtrees and is built using XQuery. But the fundamental differences with classical universal relations are the following:

– The array is not defined by one query but by a specification of how a simple selection-projection user query is to be translated into an XQuery.
  This difference is important. The problem with universal relations is that, unless the database schema has particularly nice properties which is rarely the case, projection operations generate many duplicates that are not always easy to remove. This is due to the join operations entering the definition of the universal relation. Alternatively, the join operations can also be the cause of missing information. This is usually solved by introducing outer-joins but at the cost of having to deal with null values.
  Note that these problems of data loss and duplicates may occur any time a view is defined as a structured query (SQL or XQuery).
  Our approach is not to define the view as a query but rather as a virtual set of queries that are generated on the fly to fit the user current requirements. In this way, we avoid incomplete or verbose answers.
– To deal with the complexity of the input data, we define views in two steps. The first deals with data heterogeneity and maps heterogeneous, but semantically connected documents into a target structure. At run time, this step generates unions. The second step corresponds to a standard view definition where data is aggregated. At run time, this leads to joins.
  Borrowing from a general wrapper-mediator architecture, our view model adds an intermediary level that (i) strongly structures the view by separating unions from joins, and (ii) provides homogeneous XML typing for the universal relation elements.

We implemented XyView as a set of tools on top of the Xyleme [19] XML repository, but it can easily be adapted to any system supporting XQuery. The XyView tools cover the view definition process but also automatic generation of web form applications and web services. Although its expressive power is limited as will be explained in this paper, XyView has proved its worth with several industrial applications.

The rest of the paper is organized as follows. The next section presents an example application scenario that illustrates the problem we are addressing. Section 3 describes the XyView model. Section 4 explores the expressiveness and some more subtle features of the model, then Section 5 describes the XyView system that is built on top of an XML repository. The final sections present related work and explore some future improvements.

2 Example Application Scenario and Motivation

The example that we present here is a drastic simplification of a real life application. A sports news company handles several types of news wires. The wires are well formed XML documents, with no global schema, that have been extracted from text files. These files have been edited by various local correspondents over the years, according to the company (mostly verbal) editing recommendations. The wires have different structures, depending on the sport and the kind of information they contain.

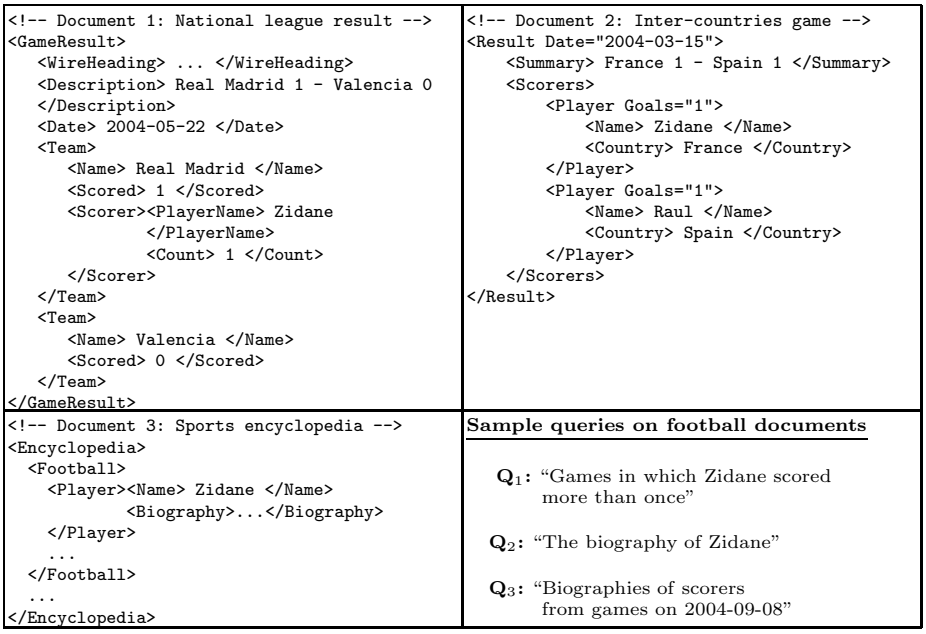

Fig. 1. Examples of documents and queries

For ease of understanding, we show in Figure 1 only two such wires about football (soccer) in a simplified form. The first considers results from national leagues (e.g., Document 1), and the second results from international games (e.g., Document 2). The news company wants to build an application that queries through simple web forms the various football results wires and a sports encyclopedia with detailed information about football players (Document 3).

The application manipulates documents whose structures are similar, but not necessarily identical, to Documents 1, 2 and 3. Notably, other documents may have more or less information. These three kinds of documents are stored in a single XML content management system in collections whose respective identifiers are NationalURI, InternationalURI and EncyclopediaURI.

The application queries, as those in Figure 1, may concern football results (Q_1), player biographies (Q_2), or both (Q_3).

These apparently simple queries are in fact rather hard to program in XQuery as illustrated by Figure 2 for Query Q_3 (issues regarding the typing of results are discussed in Section 4, we assume here that queries return simple strings).

Our objective with XyView is to optimize the productivity of graphical user interface programmers, who are not database experts, by allowing them to view the database as something as simple as a query form consisting of fields that can be used to filter or extract data. In the meantime, we want to simplify as much as possible the task of creating and maintaining such views.

The main contributions of this paper are the following:

```
union(
   For $doc1 in collection(NationalURI),
       $var1 in $doc1/GameResult,
       $doc2 in collection(EncyclopediaURI),
       $var2 in $doc2/Encyclopedia/Football/Player,
       $var3 in $var2/Biography
   Where $var1/Date = xs:date('2004-09-08') and
         $var1/Team/Scorer/PlayerName = $var2/Name
   Return string($var3),
   For $doc1 in collection(InternationalURI),
       $var1 in $doc1/Result,
       $doc2 in collection(EncyclopediaURI),
       $var2 in $doc2/Encyclopedia/Football/Player,
       $var3 in $var2/Biography
   Where $var1/@Date = xs:date('2004-09-08') and
         $var1//Player/Name = $var2/Name
   Return string($var3) )
```

Fig. 2. Query Q_3 expressed in XQuery

- A view model (XyView) that provides a universal relation-like access to heterogeneous, schema-free XML data, freeing the user from manipulating complex schemas and query languages.
- A method for avoiding the drawbacks of query-based views: joins in the view definition produce duplicates or data loss; expressing the view query for heterogeneous data is difficult; queries on the view produce nested queries, that are harder to optimize. Instead, views are defined by simple mappings and join conditions, easy to create and to maintain with graphical tools. A simple, effective and scalable query translation algorithm produces equivalent XQuery, with no useless duplicates, no data loss and no nesting.
- A view structure model, organized on several levels, adapted to heterogeneous, schema-free XML data.
- A set of tools for creating views, for rapid development and for automatic generation of web forms and web services.

3 The XyView model

In XyView, views are defined by a set of mappings and join conditions that specify how a simple selection-projection user query is translated into an XQuery. This approach overcomes the problems of query-based views, as explained later. The view definition is equivalent to a virtual set of flat and easily optimizable queries that are generated on the fly to fit the user current requirements. Notably, given the appropriate view specification, the query of Figure 2 would be generated at run time by XyView to answer Query Q_3.

This results in a simpler definition and maintenance of views, using intuitive graphical editors. Given the complexity of view queries caused by data heterogeneity, this is a crucial advantage for the view designer.

Also, in order to cope with heterogeneous data, XyView adds an intermediary level in the view definition process. To the physical and view schemas, we add logical schemas whose purpose is to provide homogeneity to semantically related data. More precisely:

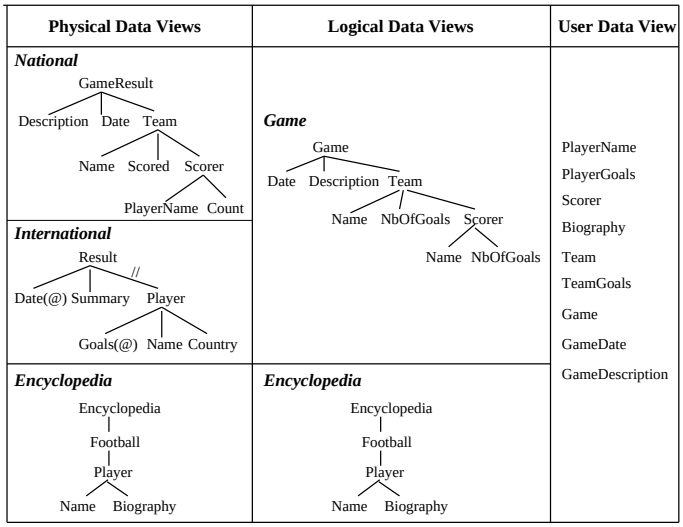

Fig. 3. From Trees to Table

1. The first level deals with schema-free data, by defining *physical data views* that summarize XML access paths to useful information in documents.
2. The second level deals with heterogeneity, by defining integrated *logical data views* over *unions* of physical data views with similar contents.
3. The third level defines the *user data view* as *joins* between logical data views.

Figure 3 illustrates this three level definition. It is built using the sample data introduced in Section 2. On the right handside is the **user data view**. It consists of a set of **"concepts"** that the user wants to query. Concepts are *typed* by the view designer. For instance *PlayerGoals* and *TeamGoals* are integers, *GameDate* is of type date, the other concepts are considered as XML strings (or elements, as will be explained in the next section).

As is the case with universal relations, the query language supported at this level consists of selections and projections. For instance, Query Q_3, that returns biographies of scorers from games on 2004-09-08 consists of a selection on *GameDate* = *2004-09-08* and a projection on *Biography*.

On the left handside of the figure, are the **physical data views (PDV)**. They represent the data as it is stored in the repository. In the example, there are three physical data views (*National*, *International* and *Encyclopedia*), the first two representing respectively local and international soccer games results, the other a sport encyclopedia. The trees are **data summaries**, i.e. trees gathering useful access paths to data elements in the XML documents. Similar to Lore data guides [8], they are generated by the system to cope with the fact that many documents are simply well-formed and do not come with a schema.

In Xyleme, these summaries are generated at loading time, there is one summary per distinct root element. XyView also provides a tool to extract these summaries from a set of documents. In both cases, we use an incremental al-

gorithm that takes all the XML paths in documents and extends the existing data summary paths with new subpaths. Note that the algorithm does not care about data types. It is the view designer who associates types to view concepts. More will be said on this topic in the sequel.

When designing the view, one can edit data summaries to remove branches that are useless for the application or to create shortcuts in long branches by using a descendant (//) connection between two nodes. E.g., the subtree *Wire-Heading* has been removed from the structure of Document 1, while the structure of Document 3 only features the *Football* element, other sports having been discarded. Also, the *Biography* element is not detailed in the PDV, because its internal structure is not useful for the application. PDV *International* contains an example of shortcut: element *Scorers* has been discarded from the path to *Player*, because it is useless and removing it introduces no ambiguity; the edge leading to *Player* is marked with //. This simplification eases the view design process, by keeping only useful access paths from possibly cumbersome document structures. Also, // shortcuts significantly improve query processing of the final XQuery, by reducing the number of structural conditions to check.

In the center of the figure, we have gotten rid of the soccer games results heterogeneity by introducing so-called **logical data views (LDV)**. Logical data view *Game* unifies in a single structure game results from documents described by PDVs *National* and *International*. Note that the second LDV (*Encyclopedia*) is a duplication of the corresponding PDV. In real life, we do not duplicate data views, we did it here for the sake of clarity.

We now illustrate how one goes first from physical to logical then to user data views and the queries that are associated to each level. Next, we further detail how user queries are translated into queries against the repository.

3.1 From Physical to Logical Data Views

A physical data view consists of a data summary tree and a set of so-called *clusters* in which we find documents conforming to the summary (there may be documents conforming to other summaries as well). A cluster is the unit in which we store documents and provides an entry point in the repository. It is queried in XQuery as a collection of documents, by using the *fn:collection* function on the cluster URI. For the sake of clarity, in the following examples we consider a single cluster for each PDV.

Semantics: a PDV P is a view over a collection of documents $Coll(P)$ (the cluster), whose schema is a data summary tree $Tree(P)$. For each p node of $Tree(P)$, let $path(p)$ be the path from the root of $Tree(p)$ to p. *The interpretation* of p is the set of XML elements that match $path(p)$ in some document of $Coll(P)$, i.e. the result of the XQuery expression $Coll(P)/path(p)$.

$Eval(p) = XQuery(Coll(P)/path(p))$

Notations: for x and y nodes in the same tree, $LCA(x, y)$ denotes their lowest common ancestor and $ancestor(x, y)$ is true if x is ancestor of y.

The interpretation of a tuple $(p_1, ..., p_k)$, $p_i \in Tree(P)$ is:

$Eval(p_1, ..., p_k) = \{(e_1, ..., e_k) \mid \exists doc \in Coll(P), \forall i \in \{1, ..., k\},$
$e_i \in Eval(p_i) \bigcap Elem(doc), \forall j, l \in \{1, ..., k\},$
$\exists e_{jl} \in Eval(LCA(p_j, p_l)), ancestor(e_{jl}, LCA(e_j, e_l)) \bigcap Elem(doc)\},$
where $Elem(doc)$ is the set of XML elements of document *doc*.

The meaning is that a tuple's elements must belong to the same document and must be the "closest" possible. Closeness is expressed wrt the PDV schema: any two tuple elements e_j, e_l must be at least as close as the corresponding PDV nodes in the schema p_j, p_l, i.e. e_j and e_l must have a common ancestor at the level of $LCA(p_j, p_l)$.

We will show in Section 3.3 how the query translation algorithm guarantees closeness by introducing XQuery variables for the LCA nodes.

A logical data view is an annotated data summary. The annotations represent the correspondence (mappings) between physical and logical data views. This is illustrated on the left side of Figure 4 for LDV *Game* and PDVs *National* and *International*. Note that to each node in the LDV data summary is associated the set of corresponding nodes in the physical data views. To keep the figure readable, only mappings for LDV nodes *Game* and *Date* are illustrated.

Mappings between LDVs and PDVs are based on correspondences between LDV and PDV tree nodes. By identifying a node in a tree with its path from the root, one can note that this approach to representing correspondence between trees is close to the *path-to-path mappings* used in [4]. The *restriction* we add - an LDV node can be mapped to *at most one node in the same PDV* - makes sure that translation from LDV to PDV in query processing is unique. A PDV not respecting the restriction can always be split into several "correct" PDVs.

Compared to classical query-based methods to define correspondences between schemas, the simplicity of our node-to-node mappings approach provides several advantages.

- In many cases, mappings can be semi-automatically generated by relying on the semantics carried by a sequence of tags (see [15, 17]).
- The process of creating these mappings can easily be supported by a graphical interface and they are easier to maintain than query-based mappings.
- Such node-to-node mappings are easy to reverse, therefore the view model can be seen both as *global-as-view* (providing easy query translation) and *local-as-view* (providing easy update).
- In Section 4, we will see that such mappings can easily be extended in order to support a richer semantics.

Semantics: a LDV L is a view over a set of PDVs $PDV(L)$, whose schema is a tree $Tree(L)$. The interpretation of a tuple $(l_1, ..., l_k)$, $l_i \in Tree(L)$ is:

$Eval(l_1, ..., l_k) = \bigcup_{P \in PDV(L)} Eval(p_1, ..., p_k)$, $p_i \in P$ is mapped to l_i

This union semantics is defined in two variants: *strict matching*, where only PDVs containing mappings to all the l_i are considered, and *relaxed matching*, where all the PDVs are considered, but only incomplete tuples, based on existing mappings, are built in each PDV.

The algorithm in Section 3.3 translates straightforwardly a query against a LDV into a union of queries against its corresponding PDVs by transforming paths from the LDV query into the corresponding paths in each PDV.

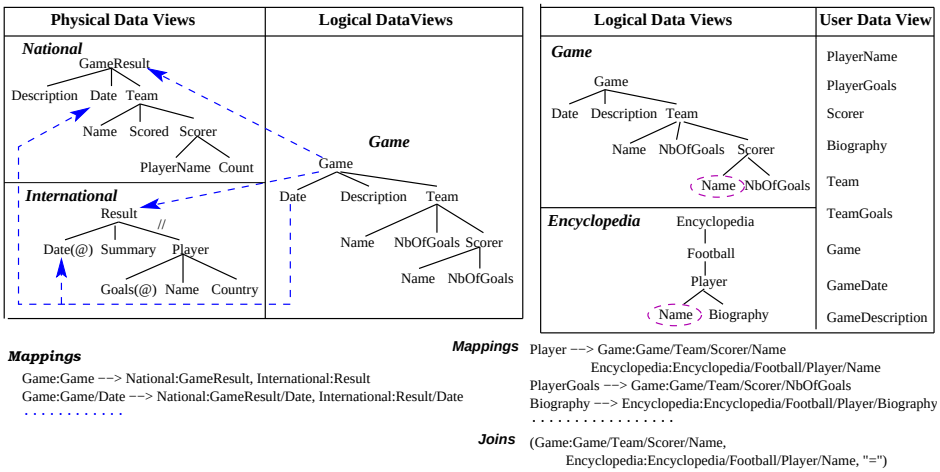

Fig. 4. From physical to logical data view, from logical to user data view

3.2 From Logical to User Data Views

A user data view consists of a set of typed concepts, their correspondence with nodes in the logical data views and a set of predicates that are used to join the logical data views (in the example, a single join predicate is defined). This case is illustrated on the right side of Figure 4. Each concept has at least a mapping to some LDV node, concept *Player* being the only one mapped to both LDVs. The join predicate specifies the joined LDV nodes and the join operator ('=' in our example). If several join predicates connect two LDVs, the global join condition is the conjunction of the individual predicates.

The semantics of a user data view is described by the query translation algorithm below. Roughly speaking, the interpretation of a tuple of concepts is a n-ary join of partial tuples of LDV nodes, found in LDVs through mappings.

We now explain in details the translation algorithm from user queries to physical queries, via logical queries.

3.3 Translating User Queries

Let us now consider Query Q_3 as an example to illustrate the translation algorithms. It involves a join between the two LDVs in order to return the biographies of scorers from games played on 2004-09-08.

Definition 1. *A user query in XyView has the form*

`Q: Select` c_1, ..., c_n
`Where` $cond_1(c'_1)$ `and` ... `and` $cond_m(c'_m)$

where c_i *and* c'_j, $i = 1, ..., n$, $j = 1, ..., m$, *are user view concepts and* $cond_j$ *are predicates over a single concept.*

Figure 5 illustrates the translation of Query Q_3 into XQuery. The translation algorithm for a user query `Q` consists of *five steps*:

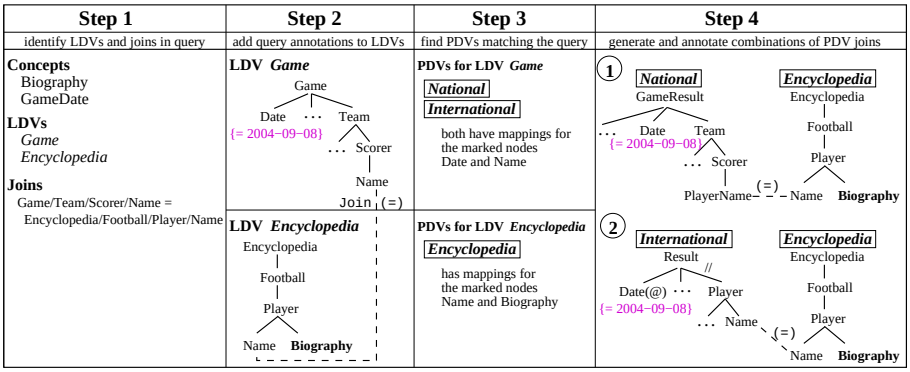

Fig. 5. First steps for translating Query Q_3: `Select` *Biography* `Where` *GameDate*=2004-09-08

1. Identify LDVs and joins involved in user query `Q`;
2. Produce a tree representation of `Q` based on the LDV trees;
3. For each LDV annotated tree, find the subset of PDV trees that match `Q`;
4. Generate all combinations of joins between PDVs;
5. Generate the final XQuery by unioning the combinations of step 4.

Step 1 One identifies the sets of concepts (C_Q), LDVs (L_Q) and joins (J_Q) involved in `Q`. They are the following:

$\mathbf{C}_Q = \{c_1, ..., c_n\} \bigcup \{c'_1, ..., c'_m\}$

$\mathbf{L}_Q$ basically contains only LDVs involved in the query, i.e. having nodes mapped to some concept in C_Q, **in order to discard useless joins**. Several semantics are implemented in XyView for L_Q. One possibility is *to remove redundant LDVs from* L_Q, i.e. those contributing with data already provided by other LDVs. Another possibility is *to add LDVs that appear along some join path between LDVs in the initial* L_Q.

$\mathbf{J}_Q = \{j=((l_1, path_1), (l_2, path_2), op) \mid j$ is a join, $l_i \in L_Q$, $path_i$ is a path in l_i, i=1,2, *op* is the join predicate$\}$, joins between members of L_Q.

In the example (Figure 5, Step 1),

C_{Q3}=$\{$`Biography`, `GameDate`$\}$

L_{Q3}=$\{$*Game*, *Encyclopedia*$\}$, because *Game* has a node mapped to concept `GameDate` and *Encyclopedia* has a node mapped to concept `Biography`

J_{Q3}=$\{$((*Game*, Game/Team/Scorer/Name), (*Encyclopedia*, Encyclopedia/Football/Player/Name), '=')$\}$ includes the only existing join, because L_{Q3} contains both joined LDVs.

Step 2 One adds query annotations to nodes of LDV trees from L_Q.

Definition 2. *The following query node annotations are defined for LDV nodes:*

- isProjected*, a boolean, true iff the node is mapped to a projected concept;*
- condSet*, a set of condition predicates composed of predicates* $cond_j$ *of* `Q` *over a concept mapped to the node;*

– isJoined, *a boolean, true iff the node occurs in* J_Q.

Definition 3. *A LDV tree node is called* a marked node *if* isProjected = *true or* condSet$\neq \emptyset$ *or* isJoined = *true.*

Figure 5, Step 2 shows query annotations added to LDV trees for Q_3. The projected node, **Biography** (*isProjected*=true), is in bold font; join nodes (*isJoined*=true) are connected through a dashed line; and the selection node (*condSet* $\neq \emptyset$) is annotated with the set of condition predicates. We removed nodes that are not involved in the query.

Step 3 For each $l \in L_Q$, the set of PDVs matching `Q` is

$$\mathbf{P}_{Q,l} = \{p \mid p \text{ is a PDV}, \forall\ n \text{ marked node of } l \Rightarrow \exists\ n' \text{ of } p \text{ mapped to } n\}$$

This corresponds to the *strict matching* semantics presented in Section 3.1. In the example, we obtain $P_{Q3,Game}$={*National*, *International*} and $P_{Q3,Encyclopedia}$={*Encyclopedia*}, because all the marked nodes in Step 2 are mapped into all the corresponding PDVs. This is not always the case; suppose that game dates are lacking from PDV *National*, in that case *National* must be removed from $P_{Q3,Game}$, because *Date* is a marked node in LDV *Game*.

Step 4 One computes all the combinations $Comb_Q$ of joins between PDVs found at Step 3. Note that joins may be n-ary (in opposition to binary), this may occur when the schema has more than two LDVs.

For each *comb* $\in Comb_Q$, the PDV nodes get the same query annotation as the LDV nodes to which they are mapped. A PDV node mapped to no LDV node gets *isProjected*=false, *condSet*=$\emptyset$ and *isJoined*=false.

Also, for each *comb* $\in Comb_Q$, the set J_{comb} of join conditions is obtained from J_Q by replacing the LDV nodes by the corresponding PDV nodes.

In our example, there are two such combinations, shown in Figure 5, Step 4.

Step 5 For each *comb* $\in Comb_Q$, representing a join between PDVs, a *For-Where-Return* query is generated. The final XQuery is obtained by unioning all these join queries. For our example, the final result is presented in Figure 2.

The algorithm for generating a *For-Where-Return* query creates *For/Where/Return* clauses as concatenations of the clauses generated for each individual PDV. To the *Where* clause, one must also concatenate the join conditions from J_{comb}.

Let us describe now the algorithms **For**, **Where** and **Return** that generate the corresponding clauses for a single annotated PDV.

The *For* clause defines variables and access paths to queried data in the PDV. Variable generation respects the following rules:

Rule 1 *A variable is defined for each projected node in the PDV.*

Rule 2 *For any two marked nodes of a PDV, there is a variable definition for their lowest common ancestor in the PDV tree.*

Rule 2 ensures the closeness semantics for the XML elements addressed by the query, i.e. those corresponding to marked nodes. For instance, it ensures that in query Q_3 the date and the scorer name belong to the same game.

```
ForClause(pdv) → String
    pdv.variable = GenerateVar()
    pdv.varNodeList =
          VariableGen(pdv.root)
    forClause = concat(pdv.variable,
          ' in collection(',
          pdv.collectionURI, ')' )
    for each n_i ∈ pdv.varNodeList repeat
       forClause = concat(forClause,
          ',', n_i.variable, ' in ',
          AncestorVarAndPath(n_i, pdv))
    end for
    return forClause
end ForClause
```

```
VariableGen(n) → NodeList
    for each n_i ∈ n.children repeat
       varNodeList_i = VariableGen(n_i)
    end for
    childrenVarNodeList = concat(varNodeList_1, ...)
    if n.isProjected then
       n.variable = GenerateVar()
       n.markedAncestor = true
    else maChildren = NbMarkedAncestor(n.children)
       n.markedAncestor = n.condSet ≠ ∅ or
          n.isJoined or maChildren > 0
       if maChildren > 1 then
          n.variable = GenerateVar()
       else n.variable = null
       end if
    end if
    if n.variable ≠ null then
       return concatList([n], childrenVarNodeList)
    else return childrenVarNodeList
    end if
end VariableGen
```

Fig. 6. "For" clause generation for a PDV

Considering the annotated PDV *National* in the first combination in Figure 5, Step 4, there are only two marked nodes *Date* and *Name*, none of them projected. Then, the only variable element defined on national games is that of *GameResult*, their lowest common ancestor.

The algorithm for generating the *For* clause for a single PDV is presented in Figure 6. It adds some new query annotations to tree nodes:

- *variable*, the name of the variable generated for the node, if any;
- *markedAncestor*, a boolean, true iff the node's subtree contains at least a marked node.

It also adds the following annotation for each PDV:

- *variable*, the variable name for documents that match the PDV;
- *varNodeList*, the list of variable nodes in the PDV, following the order of variables in the *For* clause.

The *ForClause* function uses the *VariableGen* function to obtain the ordered list of variable nodes in the PDV. The *For* clause starts by defining the document variable iterating in the collection associated to the PDV. All variable names are generated by calls to function *GenerateVar*, that returns unique variable names. The rest of the *For* clause defines variables for each variable node. The *AncestorVarAndPath* function searches for the first variable ancestor of the node, then returns this variable concatenated with the path from this ancestor to the node. If no variable ancestor exists, one uses the PDV variable and the path from the root to the node.

The *VariableGen* function builds the list of variable nodes in the subtree of the parameter node n, but also annotates with *variable* and *markedAncestor* each node in the subtree. First, it recursively builds the variable node lists for each child of n, then concatenates these lists. Then, it must decide if n is a variable node or not; if not, the result is the concatenated list from children, else n is added in front of this list. This step produces a consistent order for the *For* clause, because a node is always placed before its descendants.

Rules 1 and 2 are used to decide if n is a variable node. This is true either if n is projected, or if it has at least 2 children being *markedAncestor*. In the latter case, it is easy to demonstrate that n is the LCA of marked nodes from the subtrees of these children. Function *NbMarkedAncestor* returns the number of nodes being *markedAncestor* in the parameter list. Also, n is itself *markedAncestor* if it is projected or if it has at least one child being *markedAncestor*.

Note that **the algorithm does not generate useless variables**, only marked nodes (i.e. needed in the user query) are connected through variables on the LCA.

The algorithm for the *Where* clause produces a *conjunctive* condition. For each PDV node with $condSet \neq \emptyset$, it generates a condition predicate for each element of *condSet*. The node is identified by the path from its first variable ancestor. Note that well-typed constants are generated, by using the types of the view concepts. A similar algorithm is used to generate join conditions.

The *Return* clause describes the query result, using the variables of projected PDV nodes. Several choices for the XML type of the result are possible in XyView; they are discussed in the next section.

4 Deeper inside XyView

4.1 Duplicates and data loss

So far, we have presented views as providing a flat, relational-like, representation of arbitrary XML trees. The flattening is performed by accessing nodes through path expressions (preserving closeness through variables on the lowest common ancestor) and applying the XQuery **string()** operation on the projected nodes. The transformation from logical to user data views then corresponds to a simple sequence of join operations between results of path expressions followed by projection/map operations. In the transformation from physical to logical data views, joins are replaced by unions. The main difference with a standard view mechanism is that the view query is not defined *a priori* but rather in an opportunistic way, depending on the user query, so as to avoid duplicates and information loss that would be generated by unnecessary joins and variables.

Therefore, XyView differs from any standard view mechanism relying on query composition: **a XyView view is not defined by a query and is not equivalent to a query**. To see why query-based views may be problematic, let us have a closer look to this possibility.

If we consider that a XyView view is defined by a query, this query has to provide a full view over the various documents structures for all the view concepts. Thus, it would naturally feature (i) all the possible join and union operations in the view, as well as (ii) variables for internal nodes so as to preserve closeness of all concept elements belonging to the same subtree. A good candidate for the view query is *the user query that projects all the view concepts*, i.e. its translation through the previous algorithm.

A query on this view will face the following problems:

- **Data loss**: the join operations would make it impossible to return e.g., the biography of players who are not part of some games. The problem is that

the join with LDV *Game*, that is part of the view query definition, is useless when querying player biographies. Data loss can be solved by introducing outer-joins, but they generate null values and the need to deal with them.
In a similar way, but with no apparent reasonable XQuery solution, useless variables may be responsible for data loss. E.g., the view query contains a variable on the internal node representing scorers (required to connect scorer name and goals). If it cannot be instantiated, the corresponding parent element would be discarded, i.e. games with no scorer would be discarded from all results.

– **Duplicates**: useless joins would also lead to unnecessary duplicates, e.g. by returning several biography occurrences for most players (one for each occurrence as a scorer). The same is true for useless variables, coming from view concepts not addressed in the query. A result being produced for each new binding to the tuple of all the variables, useless variables produce duplicates for the useful variables in the result.
Duplicates can be eliminated through **distinct** operations, but (i) it is sometimes very difficult to distinguish between good (existing in data) and bad duplicates, and (ii) distinct operations have a cost (notably when the desired order is not that required by the distinct operation).

4.2 View customization

More expressive power is added to XyView through *LDV node annotations*, which allow ***customizing*** query translation, notably by (i) typing results using tree structures and transformation functions rather than returning flat results and (ii) adding selections to the view.

Tree results can be obtained in XyView in several ways. The simplest one consists in typing results according to the PDVs, i.e., returning trees as they are stored in the repository. Note that this solution leads to heterogeneous results.

For instance, consider a new query, close to Query Q_3, modified to ask *scorers* (name and number of goals) from games on 2004-09-08.

Q_3': "Scorers from games on 2004-09-08"

This query would be translated as follows:

```
union(
For $doc1 in collection(NationalURI),
    $var1 in $doc1/GameResult, $var2 in $var1/Team/Scorer
Where $var1/Date = xs:date('2004-09-08')
Return $var2,
For $doc1 in collection(InternationalURI),
    $var1 in $doc1/Result, $var2 in $var1//Player
Where $var1/@Date = xs:date('2004-09-08')
Return $var2)
```

Note that, in that case, the result of the query is heterogeneous, featuring scorers as they are stored in *National* and *International* PDVs. This solution is well adapted for *ad hoc* queries expressed by users who want to see the data as it has been produced. Also, it is an interesting semantics for the view designer who, in the preliminary phase, wants to get some information about data types. However, if the results are to be fed to an application or if the end users are not aware of the data as it is stored, we need to provide an alternative.

The second solution provides the means to type results according to the LDVs. This can be performed in a simple way by associating to each leaf node the full text (or typed atomic value) corresponding to the physical nodes to which they are mapped. It is then simple to re-construct the elements as they are defined in the logical data view. Query Q_3' becomes:

```
union(
For $doc1 in collection(NationalURI),
    $var1 in $doc1/GameResult, $var2 in $var1/Team/Scorer,
    $var3 in $var2/PlayerName, $var4 in $var2/Count
Where $var1/Date = xs:date('2004-09-08')
Return <Scorer>
         <Name>string($var3)</Name>
         <NbOfGoals>xs:integer(string($var4))</NbOfGoals>
       </Scorer>,
For $doc1 in collection(InternationalURI),
    $var1 in $doc1/Result, $var2 in $var1//Player,
    $var3 in $var2/Name
Where $var1/@Date = xs:date('2004-09-08')
Return <Scorer>
         <Name>string($var3)</Name>
         <NbOfGoals>xs:integer(string($var2/@Goals))</NbOfGoals>
       </Scorer>)
```

Note that new variable definitions are generated in the *For* clause, in order to access PDV nodes necessary to build the LDV subtree for the scorer. Results of both unioned queries have the same type, given by the LDV. Note also that the LDV subtree may not have all the subelements if some of them are not mapped into the PDV.

Both typing solutions above can be performed automatically by activating the appropriate translation option. Still, there are some cases where we want to achieve more sophisticated typing. For instance, we may want to add some PCDATA or attributes to the internal nodes. To do this, the view designer further annotates the nodes of the data summaries with ***transformation functions***.

Consider for instance the previous example, but in which we want to add an attribute called *source* that gives the URI of the source document. Suppose that the document URI can be obtained by applying the **element2URI(*element*)** function to some element of that document. Node *Scorer* in the LDV must be annotated as follows:

```
Return: <Scorer source=element2URI($$)> $1 $2 </Scorer>
```

As in Yacc, we use $$ to signify the current node (*Scorer*), $1 and $2 to represent its first (*Name*) and second (*NbOfGoals*) children. Typing of children $1 and $2 is recursively done following the same method. For instance, this solution allows selecting in the result only part of the node's subelements. Also, we use # to represent user input, i.e. the list of constant values in the user query coming from conditions on the current node, and may be used as an argument in transformation functions.

Note that producing flat string results, or PDV subtrees, or LDV subtrees is equivalent, respectively, to the following annotations for the *Scorer* LDV node:

```
Return: string($$)              //flat
Return: $$                      //PDV subtree
Return: <Scorer> $1 $2 </Scorer>  //LDV subtree
```

Selections to the view may also be specified through annotations. Suppose that we want to discard from our view all the games before 2000. This can be simply done through a new type of node annotation: selection predicates. In the example, the following annotation must be added to the *Date* node in the LDV:

```
Where: $$ >= xs:date('2000-01-01')
```

If the user query concerns some LDV, all the selection predicates of that LDV are added to the conjunctive *Where* clause of the generated XQuery.

These modifications are easily added to the algorithm detailed in the previous section. However, note that, even with these additions, the view mechanism is far from supporting all the features of XQuery. Notably, XyView does not provide grouping/nesting, sorting or disjunctive join predicates. Some of the missing features can be supported by the client program, using e.g., stylesheets. In any case, there is a necessary tradeoff between ease of use and expressive power. So far, the tool has proven useful for most applications.

5 The XyView system

XyView has been implemented as a set of tools for rapid development of web applications over the Xyleme XML repository. Yet, XyView is not dependent on Xyleme and can be easily adapted to any content management system that supports XQuery. The XyView system is composed of the following modules:

- *A view editor* that enables visual creation and modification of XyView views.
- *A run-time environment* that provides a simple API for using XyView views in user- (web forms) or machine-oriented (web services) web application.
- *A web-form application generator* that provides a graphical environment for creating simple web-form applications over the Xyleme repository.

The view editor (upper-left window in Figure 7) is a graphical tool enabling simple and intuitive creation of each view component: PDVs (using data summary extractors), LDVs, concepts, mappings, joins, etc. Views are saved in a persistent form, as a set of XML files.

The run-time environment provides a simple Java API for using XyView views in programs. The main functionalities provided by the XyView API are: (i) creating/modifying a view, (ii) loading/saving a view from/in its persistent form, (iii) building user queries against the view, (iv) translating a user query into an equivalent XQuery.

Note that XyView simply translates the user query into XQuery and does not interfere afterwards in the communication between the application and the XML repository. This architecture has the advantage of minimizing the dependency between XyView and the underlying XML content management system, allowing easy adaptation of XyView to any system supporting XQuery.

The web-form application generator (upper-right window in Figure 7) enables complete development of simple applications for end-users that query the Xyleme repository through a web-form interface. It provides a graphical interface that helps the application programmer to choose a XyView view, then to formulate

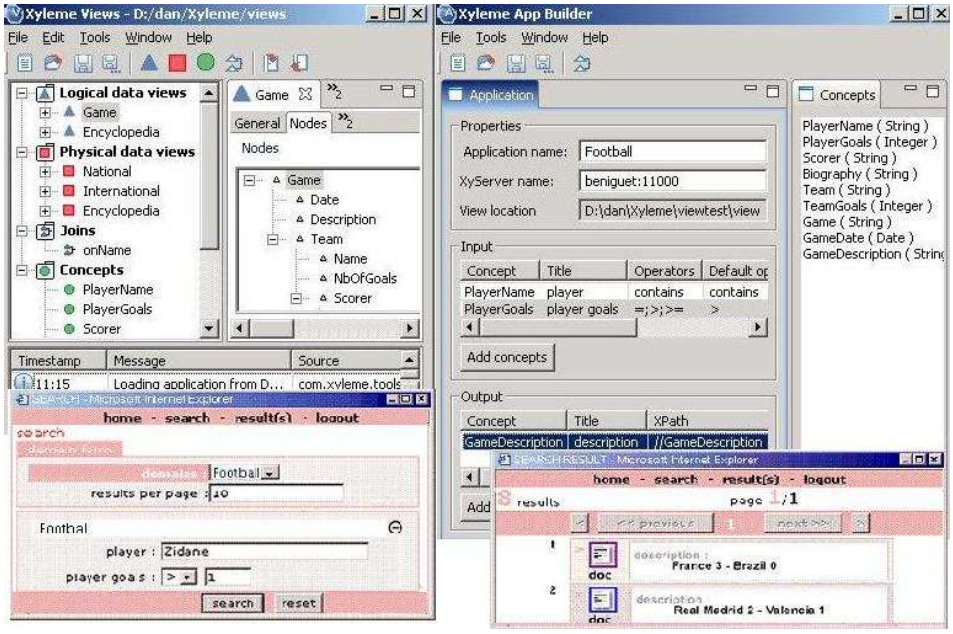

Fig. 7. XyView editor and web-form generator

queries on the view concepts. The web-form built in Figure 7 is based on the Football example view; it asks user input for conditions on concepts *PlayerName* and *PlayerGoals* and displays concept *GameDescription*. The system automatically generates the HTML query form (bottom-left window) and the application servlets producing the query report (bottom-right window).

Several applications were developed with XyView on top of Xyleme, to integrate more or less heterogeneous, semi-structured data sources, covering domains such as news publishing, financial reports, press archives, etc. Examples used in this paper are summaries of one such application involving about 50 PDVs and 11 LDVs (an encyclopedia, 10 different sports and an average of 5 kinds of wires for each). The original documents were well annotated ASCII files transformed into XML documents using a dedicated tool.

6 Related work and conclusion

Various approaches for simplifying query formulation over XML data were proposed. Systems like XQBE [1] and Xing [6] use visual specification of XML queries based on tree patterns. But even if it is simpler to express queries graphically than in XQuery, the user must handle XML structures, express joins, etc. Other systems allow writing queries with minimal knowledge about the structure of documents: keyword search in XML data [5, 12, 9] or tag and keyword search [14]. Such systems are not adapted for application development over heterogeneous XML documents, because of their limited expressive power (e.g. no joins) and lack of precision and/or meaningfulness.

XyView's approach of adapting the universal relation paradigm [18] to simplify query formulation fits well the needs of both end user and application development. Querying XyView views is very simple, it guarantees precision, meaningfulness of results and minor processing overhead. The price to pay is the view designer's effort to create and maintain the view. But the XyView model is not query-based and rather borrows from mediator-like [2, 13, 4, 7] or P2P [10] XML data integration systems, to define views through basic one-to-one mappings, like those used in [4, 7]. This allows the use of graphical tools, which greatly simplifies the view designer's task.

An alternative approach is to shred XML in relations, physically (like many RDBMS today) or virtually ([11]), then to create a relational view on top. This solution may work efficiently for homogeneous XML documents, with no structural variation and when XML is really stored in tables. Our application context is more general; we build views over heterogeneous and schema-free XML, stored in any system supporting XQuery.

Among the tools for rapid development of web applications over XML data, Qursed [16] is close to our application development context. Qursed enables rapid development of user-oriented applications over XML data, based on web query forms and reports. Its main module is a visual editor, which roughly takes an HTML query form (input for the user), a report template (output for the user) and an XML Schema describing the data. The programmer defines mappings between input query fields and XML data, then between XML data and report output. Qursed is similar to our XyGen web-form application generator, but can produce more sophisticated output reports. Yet, Qursed is not appropriate for heterogeneous, schema-free XML data. It needs XML Schema for data and can handle a single document schema in the same application. Also, Qursed is designed for user-oriented applications, but not to program web services.

In the same category of tools, BEA Liquid Data [3] provides an advanced environment for data integration and web application development. It overcomes the limitations of Qursed by defining data views over several schemas connected through joins. Unions are also possible, but the method to define them is unnatural, based on a cloning of data view elements. Beyond the fact that this complex tool focuses on specialized programmers, its support for heterogeneous schema-free XML documents has several limitations: (i) data sources must provide a schema, (ii) views are defined by queries, with all the problems of useless joins and variables, (iii) one cannot reasonably mix in the same data view several joins and unions, etc. Even if the latter problem can be bypassed by chaining several data views, this results in bad query processing performance.

Through its simple programming interface that removes the need to work with XQuery and XML schemas, XyView increases the productivity of programmers who implement query interfaces on top of a heterogeneous, schema-free XML repository. The view designer's task is highly simplified by the intuitive representation of views (set of mappings), manipulated through graphical editors. The query translation algorithm is simple, effective and scalable, it avoids useless duplicates, data loss and unnecessary nesting. Tools for Java programming

and for automatic generation of web-form applications complete the XyView environment.

Although the tool does not provide the full expressive power of XQuery, it has proven sufficient for many industrial applications. Furthermore, the possibility to customize the query generation algorithm by adding functions to the view specification opens interesting perspectives in terms of expressive power. We illustrated this by considering typing and selections, we plan to further explore this mechanism to add some needed functionalities such as aggregation.

References

1. E. Augurusa, D. Braga, A. Campi, and S. Ceri. Design and Implementation of a Graphical Interface to XQuery. *Proceedings ACM Symposium on Applied Computing*, pages 1163 – 1167, 2003.
2. C. K. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-Based Information Mediation with MIX. *Proceedings SIGMOD*, 1999.
3. BEA Liquid Data. *http://www.bea.com.*
4. S. Cluet, P. Veltri, and D. Vodislav. Views in a large scale XML repository. *Proceedings of the 27th VLDB Conference*, pages 271–280, 2001.
5. S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. *Proceedings VLDB*, 2003.
6. M. Erwig. Xing: A Visual XML Query Language. *Journal of Visual Languages and Computing*, pages 5–45, Februray 2003.
7. I. Fundulaki, B. Amann, C. Beeri, M. Scholl, and A.-M. Vercoustre. STYX: Connecting the XML Web to the World of Semantics. *Proceedings EDBT*, pages 759–761, 2002.
8. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. *Proceedings of the 23rd VLDB Conference*, pages 436–445, 1997.
9. L. Guo, F. Shao, J. Shanmugasundaram, and C. Botev. XRANK : Ranked keyword search over XML documents. *Proceedings SIGMOD*, 2003.
10. A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. *Proceedings WWW*, 2003.
11. A. Halverson, V. Josifovski, G. Lohman, H. Pirahesh, and M. Mörschel. ROX: Relational over XML. *Proceedings VLDB*, 2004.
12. V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. *Proceedings ICDE*, 2003.
13. Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML query engine for network-bound data. *The VLDB Journal*, 2:380–402, December 2002.
14. Y. Li, C. Yu, and H. Jagadish. Schema-Free XQuery. *Proceedings VLDB*, 2004.
15. J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. *Proceedings VLDB*, pages 49–58, 2001.
16. Y. Papakonstantinou, M. Petropoulos, and V. Vassalos. QURSED: Querying and Reporting Semistructured Data. *Proc. SIGMOD*, 2002.
17. C. Reynaud, J.-P. Sirot, and D. Vodislav. Semantic Integration of XML Heterogeneous Data Sources. *Proceedings IDEAS*, pages 199–208, 2001.
18. J. D. Ullman. Universal Relation Interfaces for Database Systems. *Proceedings IFIP*, 1983.
19. Xyleme. *http://www.xyleme.com.*