XyView: Universal Relations Revisited

Dan Vodislav
CNAM/CEDRIC
Paris, France
vodislav@cnam.fr

Sophie Cluet
INRIA
Rocquencourt, France
Sophie.Cluet@inria.fr

Grégory Corona
Xyleme
Paris, France
Gregory.Corona@xyleme.com

Imen Sebei
CNAM/CEDRIC
Paris, France
imen.sebei@cnam.fr

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 that can be queried using a QBE-like interface or through simple selection/projection queries. It extends the concept of universal relations 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 explicitely by a unique 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.

Keywords: XML views, heterogeneous data integration, application development tools, universal relation

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 XLM 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, 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 between universal relations and our approach 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 intro-

ducing 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 somehow maps semantically connected heterogeneous 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.

  Somehow similar to 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 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 the related work and explore some future work.

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.

For lack of space and ease of understanding, we show here only two such wires about football (soccer) and in a simplified form. The first considers results from national leagues (e.g., Document 1), and the second results from international inter-country 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.

```
<!-- Document 1: National league result -->
<GameResult>
    <WireHeading> ... </WireHeading>
    <Description> Real Madrid 1 - Valencia 0 </Description>
    <Date> 2004-05-22 </Date>
    <Team>
        <Name> Real Madrid </Name>
        <Scored> 1 </Scored>
        <Scorer><Name> Zidane </Name>
                <Goals> 1 </Goals>
        </Scorer>
    </Team>
    <Team>
        <Name> Valencia </Name>
        <Scored> 0 </Scored>
    </Team>
</GameResult>

<!-- Document 2: Inter-country game -->
<Result Date="2004-03-15">
    <Summary> France 1 - Spain 1 </Summary>
    <Scorers>
        <Scorer Goals="1">
            <Name> Zidane </Name>
            <Country> France </Country>
        </Scorer>
        <Scorer Goals="1">
            <Name> Raul </Name>
            <Country> Spain </Country>
        </Scorer>
    </Scorers>
</Result>
```

```
<!-- Document 3: Sports encyclopedia -->
<Encyclopedia>
  <Football>
    <Player><Name> Zidane </Name>
            <Biography>...</Biography>
    </Player>
    ...
  </Football>
  ...
</Encyclopedia>
```

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 Figures 2, 3 and 4 (issues regarding the typing of results are discussed in Section 4, we assume here that queries return simple strings). First, one must find what documents are needed among the various document types in the system, and what are their underlying structures (documents may be schema-free). Then, one must understand what are the XML elements (and their access paths) involved in each query, and how to combine them to produce the result (e.g. Q_1 simply involves a union while Q_3 involves two joins and a union). And last, but not least, the application queries must be correctly expressed in XQuery.

Q_1: "Games in which Zidane scored more than once"
Q_2: "The biography of Zidane"
Q_3: "Biographies of scorers from games on 2004-09-08"

Figure 1: Sample queries

Programmers of graphical user interfaces are not database experts. They are usually more comfortable with Java, servlets, stylesheets, etc. than with database schemas or XQuery. Yet, they have to program (and modify) many queries to satisfy their customers needs. Our objectives with XyView is to optimize their productivity 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.

Note that this is an old idea, universal relations in the seventies addressed the same problem. The database was viewed as a single relation queried using simple selections and projections.

Yet, there is a crucial difference between XyView and universal relations as they were defined in the seventies. As a matter of fact, XyView

```
union(
  For $doc in collection(NationalURI)
      $var1 in $doc/GameResult,
      $var2 in $var1/Team/Scorer,
      $var3 in $var1/Description
  Where $var2/Name ftcontains 'Zidane' and
        $var2/Goals > 1
  Return string($var3),
  For $doc in collection(InternationalURI)
      $var1 in $doc/Result,
      $var2 in $a//Scorer,
      $var3 in $a/Summary
  Where $var2/Name ftcontains 'Zidane' and
        $var2/@Goals > 1
  Return string($var3) )
```

Figure 2: Query Q1 in XQuery

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**. In the next section, we explain this in more details. But let us first see why defining a universal relation using a query may be problematic.

So, we consider that a view is defined by a query and that a query on a view corresponds to the combination of two queries. Going back to our example, a first interesting exercice is to define the view query that, combined respectively with the three selection-projection queries $\mathbf{Q_1}$-$\mathbf{Q_3}$, would return the queries of Figures 2-4. This query has to provide a full view over the various documents structures. Thus, it would naturally feature (i) the join and union operations of Query $\mathbf{Q_3}$, as well as (ii) a variable for each internal nodes so as to preserve the connexion between the elements belonging to the same subtree. As a consequence:

- The join operations would make it impossible to return the biography of players who are not part of some games, and, alternatively, would lead to returning several biography occurences for most players. Information loss can be solved by introducing outer-joins, but they generate null values and the need to deal with them. As for duplicates, they can be eliminated by introducing **distinct** operations, but (i) it is sometimes very difficult

```
For $doc in collection(EncyclopediaURI)
    $var1 in $doc/Football/Player,
    $var2 in $var1/Name,
    $var3 in $var1/Biography
Where $var2 ftcontains 'Zidane'
Return string($var3)
```

Figure 3: Query Q2 in XQuery

```
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/Name = $var2/Name
   Return string($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//Scorer/Name = $var2/Name
   Return string($var3) )
```

Figure 4: Query Q3 in XQuery

to distinguish between good and bad duplicates and (ii) distinct operations have a cost (notably when the desired order is not that required by the distinct operation).

- In a similar way, but with no apparent reasonnable XQuery solution, variables may be responsible for information loss. E.g., consider a variable on the internal node representing scorers (required if one wants to query both their name and their goals). If it cannot be instanciated, the corresponding parent element would be discarded. As a result, games with no scorer would be discarded from all results.

Finally, the example we gave is rather simple. In real applications, the view designer has to manipulate many more structures. Defining the view query then becomes near to impossible for a non-expert. As for the optimization of the generated nested queries, the chances are that it will be poor.

The next section details the basic XyView model. Then, Section 4 adds some expressive power to this simple model and Section 5 briefly describes the graphical tools that the GUI programmer uses to build his/her simple query environment and to ease the GUI programming.

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 discussed above. Useless joins and variables are not considered at query translation; 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 specification, the queries of Figures 2-4 would be generated at run time by XyView to answer Queries Q_1-Q_3.

This results in a simpler definition and maintainance 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:

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 the logical data views.

Figure 5 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 so-called **"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 soc-

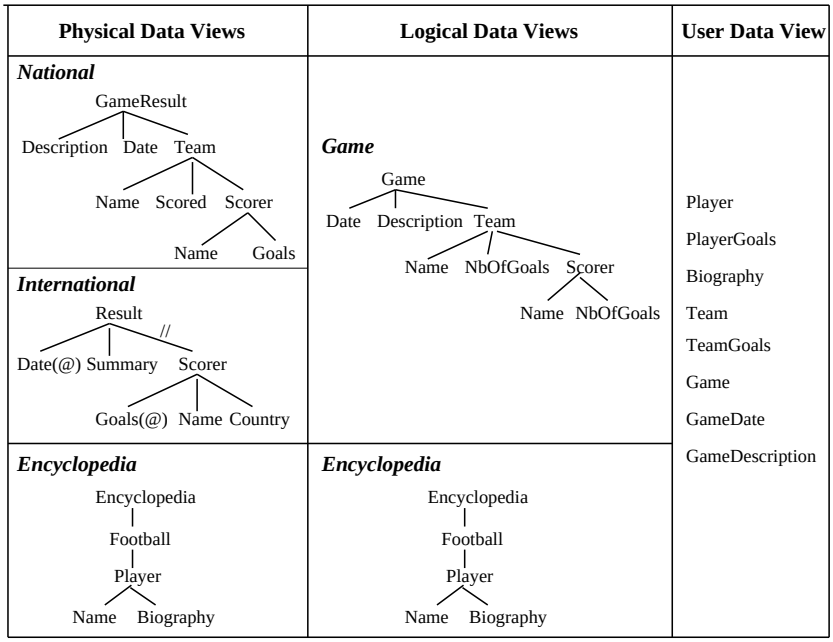

Figure 5: From Trees to Table

cer 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 DTD or XML 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 given set of documents. In both cases, we use an incremental algorithm that takes all the XML paths in documents and extends the data summaries (initially restricted to the root elements) with eventually 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. This is what we have done in the example. E.g;, the subtree *WireHeading* 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 *Scorer*, because it is useless and removing it introduces no ambiguity; the edge leading to *Scorer* is marked with //. This simplification process can greatly ease the view design process, by keeping only useful access paths from possibly cumbersome document structures. In the example, document structures are simple and the difference between the two types of soccer games result is light. In real life, structures are often much more complex.

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 with each level. Next, we further detail how user queries are translated into queries against the repository.

3.1 From Physical to Logical Data Views

Before we detail the way one goes from physical to logical data views, let us say a little more about physical and logical data views.

A physical data view consists of a data summary 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. A cluster can be queried in XQuery as a collection of documents, by using the *fn:collection* function on the cluster URI. Since a PDV is defined over a set of clusters, querying a PDV implies a union operation over these clusters. For the sake of clarity, in the following examples we consider a single cluster for each PDV.

A logical data view is an annotated data summary. The annotations represent the correspondence (mappings) between physical and logical data views. This is illustrated in Figure 6 for LDV *Game* and the *National* and *International* PDVs. 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. If one considers that a node in a tree can easily be identified by 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]. Although an LDV node can be mapped to several PDV nodes, we impose the following *restriction*: a LDV node can be mapped to *at most one node in the same PDV*. This restriction eliminates any ambiguity in going from LDV to PDV for query processing. The consequence is that any query on the LDV is translated on the PDV in a unique (and easy to compute) way. The restriction transforms the correspondence between LDVs and PDVs into a *DTD-to-DTD mapping*, following the approaches compared in [4]. The advantages are precision (no risk of incorrect combinations of PDV nodes at query translation) and fast translation. It is always possible to respect the restriction by carefully choosing in each PDV at most one node with the same

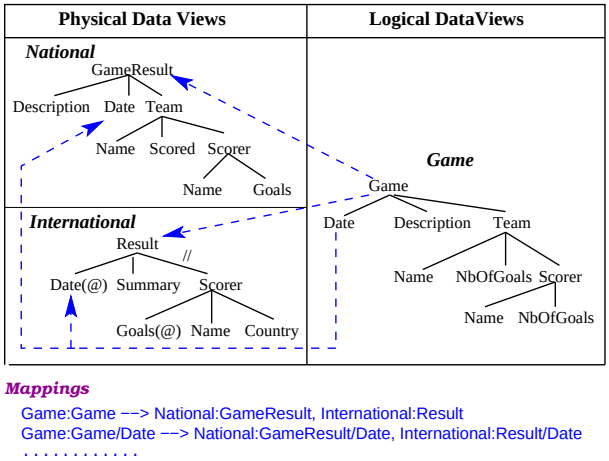

Figure 6: From physical to logical data views

meaning; document structures not respecting the restriction can be split into several 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.
- Correspondences based on node-to-node mappings are easier to maintain than query-based ones.
- In Section 4, we will see that such mappings can easily be extended in order to support a richer semantics.

A query against a logical data view is translated straightforwardly into a union of queries against its corresponding physical data views. As will be explained in Section 3.3, each member of the union is obtained by transforming paths from the LDV query into the corresponding paths in each PDV.

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 is illustrated in Figure 7. Note that

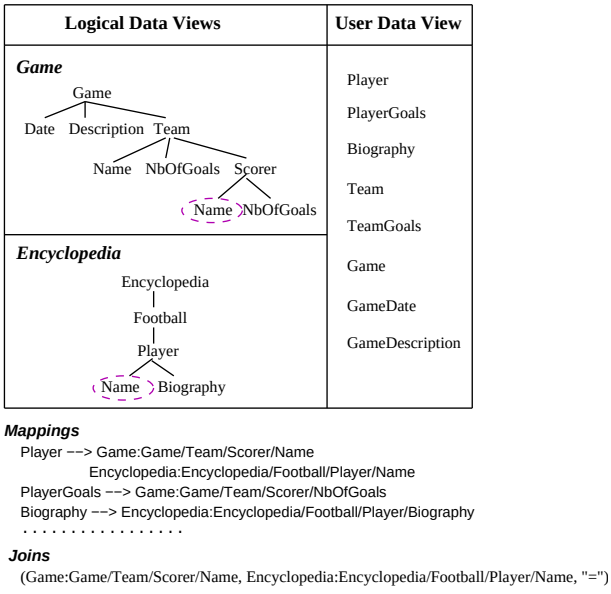

Figure 7: From logical to user data view

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.

A query against a user data view consists of a set of selection predicates and a set of projected concepts. As a simple example, consider Query Q_3 whose corresponding user query is given on the left side of Figure 8.

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

3.3 More on 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` $cond_1(c_{c1})$ `and` ... `and` $cond_m(c_{cm})$

where c_i *and* c_{cj}, $i = 1, ..., n$, $j = 1, ..., m$, *are user data view concepts and* $cond_j$ *are predicates over a single concept, based on a predefined set of operators ('=', 'contains', '>', etc).*

Figure 8 shows on the left side the user query Q_3 and illustrates step by step its translation into XQuery. The translation algorithm for a user query `Q` consists of *five steps*:

1. Identify LDVs and joins involved in user query `Q`;
2. Produce a tree representation of `Q` by adding query annotations to 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:

$\mathbf{C}_Q = \{c_1, ..., c_n\} \bigcup \{c_{c1}, ..., c_{cm}\}$

$\mathbf{L}_Q = L_{Qstrict} \bigcup L_{Qjoin}$
$L_{Qstrict}$ contains all LDVs having at least a non-join node mapped to some concept in C_Q
L_{Qjoin} contains all LDVs not in $L_{Qstrict}$ necessary to connect by chained joins LDVs from $L_{Qstrict}$

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

In the example (Figure 8, 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.

Note that the definition of L_Q is such that it discards useless joins.

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

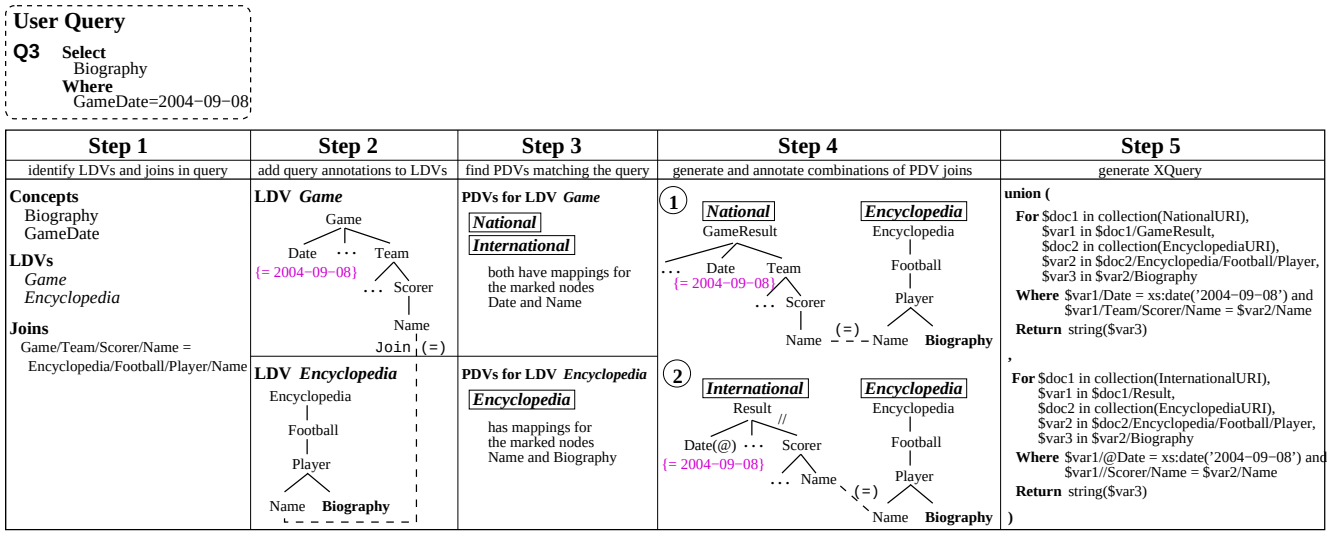

Figure 8: Steps for translating Query Q_3

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

- isProjected, *a boolean that is true iff the node is mapped to a projected concept* (c_1, ..., c_n);
- condSet, *a set of condition predicates composed of predicates* cond_j *of* `Q` *over a concept mapped to the node;*
- isJoined, *a boolean that is 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 8, Step 3 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 marked and 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 is one of the two semantics implemented by XyView, the *strict matching*. It requires that all marked nodes of the LDV have a correspondence in the PDV. The other semantics, *relaxed matching*, allows selection and join nodes without correspondence. For simplicity, we consider strict matching in the rest of the section.

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 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* $\in \text{Comb}_Q$, 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* $\in \text{Comb}_Q$, the set J_{comb} of joins on the PDVs 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 8, Step 4.

Step 5

For each *comb* $\in \text{Comb}_Q$, representing a join between PDVs, a *For-Where-Return* query is gener-

ated. The final XQuery is obtained by unioning all these join queries.

The algorithm for generating a *For-Where-Return* query from a combination is described in Figure 9. The *For / Where / Return* clauses are concatenations of the clauses generated for each individual PDV. To the *Where* clause, one must also concatenate the join conditions from J_{comb}.

```
ForWhereReturn(comb, J_comb) → String
  for each p_i ∈ comb repeat
      forClause_i = For(p_i)
      whereClause_i = Where(p_i)
      returnClause_i = Return(p_i)
  end for
  forClause = concat(forClause_1, ...)
  whereClause = concat(whereClause_1, ..., Join(J_comb))
  returnClause = concat(returnClause_1, ...)
  return concat('For ', forClause,
       ' Where ', whereClause, ' Return ', returnClause)
end ForWhereReturn
```

Figure 9: For-Where-Return algorithm

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 least common ancestor in the PDV tree.*

Rule 2 ensures the closest possible context for the XML elements addressed by the query, i.e. those corresponding to marked nodes in the PDV. For instance, it ensures that in query Q_3 the date and the scorer name belong to the same game. Considering the annotated PDV *National* in the first combination in Figure 8, 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 least common ancestor.

The algorithm for generating the *For* clause for a single PDV is presented in Figure 10. It adds some new query annotations to tree nodes:

- *variable*, a string (possibly null) containing the name of the variable generated for the node;
- *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 name of the variable generated for the documents that match the PDV;
- *varNodeList*, the list of variable nodes in the PDV, following the order of variables in the *For* clause.

```
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
```

Figure 10: "For" clause generation for a PDV

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 a different string at each call. 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 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 least common ancestor 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 last common ancestor.

We end this section by briefly describing algorithms for the other clauses. The algorithm for the *Where* clause produces a *conjunctive* condition. It takes all the PDV nodes with $condSet \neq \emptyset$ and generates a condition predicate on the node, for each element of *condSet*. The node is identified by the path from its first variable ancestor. A similar algorithm is used to generate join conditions in the *Where* clause; the difference is that join predicates concern two nodes, not one.

Note that the types of the view concepts are used to generate well-typed constants in the *Where* clause. For instance, because concept *GameDate* is of type date, condition `GameDate = 2004-09-08` in Query Q_3 is translated into ... `= xs:date('2004-09-08')`. Note that the previous expression may produce an error if the element value type is not compatible with the constant type. In reality we use a dedicated comparison function that checks also type compatibility (returning *false* if types are not compatible). For the sake of clarity, we use the simple form of conditions in this paper.

The *Return* clause describes the query result, using the variables of projected PDV nodes. The *Return* clause determines the XML type of the result. Several choices are possible in XyView; they are discussed in the next section. Here, we transform the biography elements to return simple strings.

The final XQuery corresponding to Q_3 is shown in Figure 8, Step 5.

4 Deeper inside XyView

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 the nodes dependency) 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. 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 unecessary joins. In the transformation from physical to logical data views, joins are replaced by unions.

Note that if a projected concept has an atomic type other than string, a cast operator is applied to the flatten string value. For instance, when concept *PlayerGoals*, of type integer, is projected, the *Return* clause has the form:

```
Return xs:integer(string($var)).
```

To avoid errors produced by cast operations, we use in reality a dedicated function that checks if the conversion is possible and if not returns the input string value. For the sake of clarity, we use only the cast operator here.

We now address two issues concerning (i) the possibility to return trees rather than flat results and (ii) the addition of some expressive power.

4.1 Tree Results

There are three possibilities to return tree rather than flat results. The first and simplest one consists in typing results according to the PDVs, i.e., returning trees as they are stored in the repository. In other words, the trees are flattened from their roots to the projected nodes, the projected nodes are returned as they are. Note that this solution leads to heterogeneous results.

For instance, consider the example of Query Q_1, modified to ask for all the game information (concept *Game*), instead of only the game de-

scription (concept *GameDescription*). This query would be translated as follows:

```
union(
For $doc1 in collection(NationalURI),
    $var1 in $doc1/GameResult,
    $var2 in $var1/Team/Scorer
Where $var2/Name ftcontains 'Zidane' and
      $var2/Goals > 1
Return $var1,
For $doc1 in collection(InternationalURI),
    $var1 in $doc1/Result,
    $var2 in $var1//Scorer
Where $var2/Name ftcontains 'Zidane' and
      $var2/@Goals > 1
Return $var1)
```

Note that, in that case, the result of the query is heterogeneous, featuring games 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.

This is what the second and third solutions are about. The second solution provides the means to type results according to the logical data views. 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. The query becomes:

```
union(
For $doc1 in collection(NationalURI),
    $var1 in $doc1/GameResult,
    $var2 in $var1/Team/Scorer,
    $var3 in $var1/Date,
    $var4 in $var1/Description,
    $var5 in $var1/Team,
    $var6 in $var5/Name,
    $var7 in $var5/Scored,
    $var8 in $var5/Scorer,
    $var9 in $var8/Name,
    $var10 in $var8/Goals
Where $var2/Name ftcontains 'Zidane' and
      $var2/Goals > 1
Return
  <Game>
    <Date>xs:date(string($var3))</Date>
    <Description>string($var4)</Description>
    <Team>
      <Name>string($var6)</Name>
      <NbOfGoals>xs:integer(string($var7))</NbOfGoals>
      <Scorer>
        <Name>string($var9)</Name>
        <NbOfGoals>xs:integer(string($var10))</NbOfGoals>
      </Scorer>
    </Team>
  </Game>,
For $doc1 in collection(InternationalURI),
    $var1 in $doc1/Result,
    $var2 in $var1//Scorer,
    $var3 in $var1/Summary,
    $var4 in $var1//Scorer,
    $var5 in $var4/Country,
    $var6 in $var4/Name
Where $var2/Name ftcontains 'Zidane' and
      $var2/@Goals > 1
Return
  <Game>
    <Date>xs:date(string($var1/@Date))</Date>
    <Description>string($var3)</Description>
    <Team>
      <Name>string($var5)</Name>
      <Scorer>
        <Name>string($var6)</Name>
        <NbOfGoals>
          xs:integer(string($var4/@Goals))
        </NbOfGoals>
      </Scorer>
    </Team>
  </Game>)
```

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 game. Results of both unioned queries have the same type, given by the logical data view. Note also that the team's number of goals is not returned by the second query, because PDV *National* has no node mapped to the corresponding LDV element.

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. This is what the third solution is about. With this solution, the application programmer further annotates the nodes of the data summaries with transformation functions.

Consider for instance the previous example, but in which we want the returned games to keep only the date and the description. We also 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 function **element2URI(*element*)** to some element of that document. To get this behavior, node *Game* in the LDV must be annotated as follows:

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

As in Yacc, we use $$ to signify the current node (*Game*), $1 and $2 to represent its first (*Date*) and second (*Description*) children. Typing

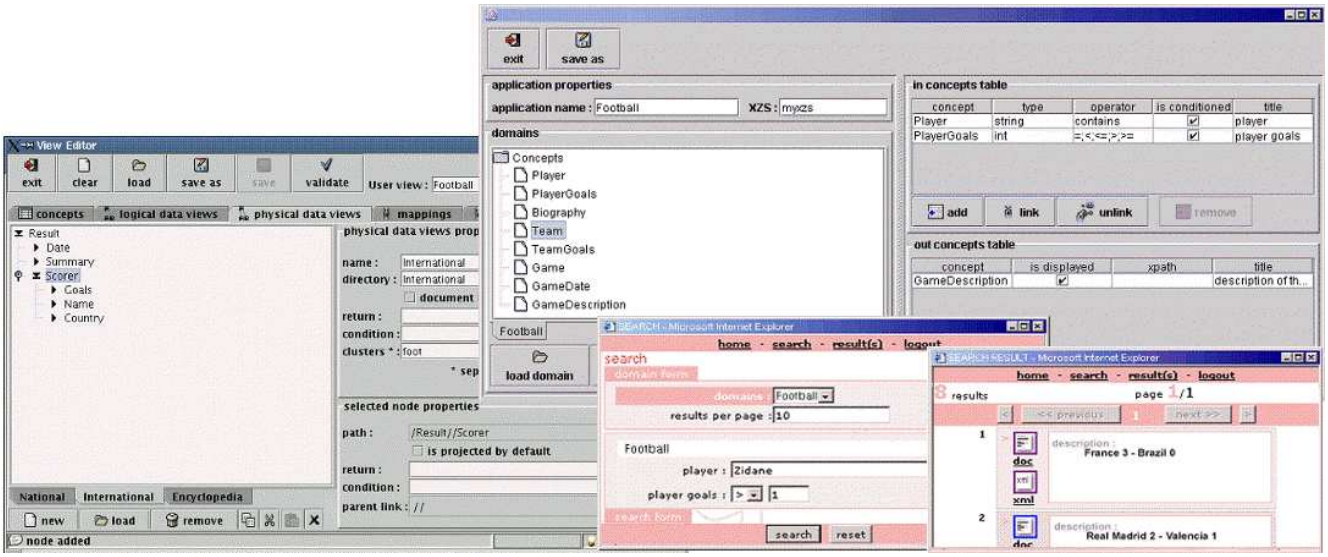

Figure 11: XyView view editor and XyGen web-form application generator

of children \$1 and \$2 is recursively done following the same method. Note that producing only flat string results is equivalent to the following annotation for all LDV nodes:

```
Return: string($$)
```

4.2 More Features

The possibility to annotate the nodes can be used for other purposes than typing, e.g. to apply any external function to the node value. Besides \$\$ and \$$i$, other special symbols may be used in annotations, e.g. # to represent user input, i.e. the list of constant values in the user query coming from conditions on the current node.

Another important use of annotations is to express selections to the view. 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 web application, user- (web forms) or machine-oriented (web services).
- *A web-form application generator* that provides a graphical environment for designing and generating simple web-form applications over the Xyleme repository.

The view editor (left-bottom window in Figure 11) provides a graphical interface for creating and modifying XyView views. Each view level

is edited in a separate tab, physical and logical data views as annotated trees, the user data view as a list of typed concepts. The editor also provides a data summary extractor. In Figure 11, the current tab corresponds to the editing of physical data view *International*. Additional tabs allow mappings and join editing. Once the view definition is completed, the in-memory view representation is saved in a persistent form, as a set of XML files.

The run-time environment provides a simple API for querying XyView views in Java applications. The XyView API covers all the actions illustrated in Figure 12, which shows the typical workflow of an application that uses XyView views:

1. Ask XyView to load in memory a given view.
2. Send to XyView a user query against the view.
3. Receive from XyView the XQuery translation of the user query.
4. Use the XQuery string returned by XyView to query the XML repository.
5. Receive and process the XML results.

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

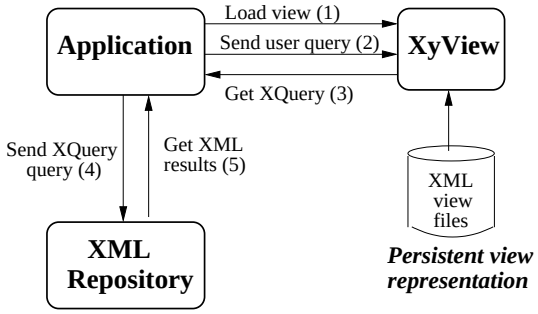

Figure 12: Flow of actions in XyView applications

The XyGen web-form application generator enables complete development of simple applications for end-users that query the Xyleme repository through a web-form interface. The right side of Figure 11 shows an example of web-form application development that corresponds to Query Q_1. The generator provides a graphical interface (window on top of the figure) that helps the application programmer to choose a XyView view (left side of the window), then to formulate queries on the view concepts (conditions and projected concepts on the right side of the window). Then, XyGen automatically generates:

- An HTML web-form for each user query (bottom-middle window) and the corresponding result report (bottom-right window).
- Servlets for each web-form, that handle communication with XyView and with Xyleme, following the flow presented in Figure 12. Servlets are automatically installed on the target web server (*myxzs* in the example).

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. The 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

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, meaning-

fulness 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] comes probably the closest 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 webform 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] is a commercial product providing 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 somehow 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 performances.

7 Conclusions and future work

We have presented a view model and system that allow the fast development of user- (web forms) or machine- (web services) oriented applications on top of heterogenous collections of XML documents. The model builds a universal relation-like view and relies on a three-level representation of data: physical, logical (to deal with heterogeneity) and user (flat view). The system relies heavily on an algorithm that translates a user query into an XQuery against the stored data. It comes with a set of tools to generate run-time programs. The core of the system can easily be ported on any content management system supporting XQuery.

Although far from complete, the view mechanism has proven its efficiency with several industrial applications. From our experience, its main limitation concerns its inability to deal with grouping and aggregate functions, required by some applications (mainly financial). So far, these features are supported at the client level which forbids the use of interesting optimization techniques (especially, when the underlying system is distributed) and implies some hard work for the programmer.

The addition of grouping and aggregates without destroying the much appreciated simplicity of the system is rather challenging. We are working on some applications to understand what is really needed and how we can restrict the expressive power of XQuery in this domain so as to provide a reasonable tradeoff.

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, C. Botev, and J. Shanmugasundaram. 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*.