Query rewriting for open XML data integration systems

Abstract

This paper presents *OpenXView*, a model for open XML data integration systems, characterized by the *autonomy of users* that publish XML data on a common topic. Autonomy implies *frequent and unpredictable changes* to data and a *high degree of structure heterogeneity*. The OpenXView model provides an original integration schema, based on a hybrid ontology - XML schema structure. We propose solutions for two important problems in such systems: *easy access to data* through a simple query language over the common schema and *simple integration view management* when data changes. This paper focuses on the query rewriting problem in OpenXView, for which existing algorithms are not suitable, and proposes a query translation algorithm.

Keywords: XML, heterogeneous data integration, ontology, query rewriting, local-as-view

1 Introduction

Many companies are now considering storing their data in XML repositories. Hence, the integration and transformation of such data has become increasingly important for applications that need to support their users with querying environments.

We address here the problem of XML data integration in a particular context. First, we are interested in *open* integration systems, where users may freely publish data in the system, in order to share information on common interest topics. A typical example is peer-to-peer [8] communities, initially sharing multimedia files, but currently focusing more and more on structured content, such as XML data. The key characteristic of open integration systems is *user autonomy* in publishing data. *Frequent and unpredictable changes* to data and schemas, as users publish new information is a first consequence of user autonomy. The other important effect of autonomy is *data heterogeneity*, for documents coming from different users, that have independently designed the structure of their documents.

The data integration model we have chosen for solving this XML data integration problem is novel. Usually, the common (target) schema for XML data integration is either a tree-like XML schema, or an ontology. In the former case, the advantage is a low model mismatch, i.e. a good adequacy of the common schema model with source data and with query results (XML data). The drawbacks are a limited semantic expressiveness and some rigidity in data typing at query processing: the system often matches only results that preserve in sources the same relations between the queried elements in the common schema. Ontologies eliminate these drawbacks, but the model mismatch between XML schemas and ontologies complexifies the expression of mappings between sources and the common model.

We propose a model that combines the advantages of ontologies and XML schemas, by defining *a hybrid integration schema*: a simple ontology, where concepts have properties organized in hierarchies (such as in XML schemas), but may be connected through two-way "relatedTo" relationships, more flexible at query processing.

On the source side, users publish XML schemas and documents. We introduced in [1] the notion of *Physical Data View* (PDV), better adapted to data integration than the XML schemas published by the sources. A PDV is a view on a real schema; it has a tree-like structure, gathering access paths to useful nodes in the real schema, and mappings between this tree and the ontology graph. Mappings are expressed through simple two-way, node-to-node correspondences between PDV and ontology nodes. The difference between a published XML schema and a PDV is subtle. On the one hand, even if not mandatory, a PDV may discard useless nodes in the XML schema, by removing subtrees or by replacing a path between two nodes by a single "//" edge. Removing nodes helps improving schema management, storage and query processing. The PDV tree is actually a data guide, a summary of access paths to nodes useful for queries. On the other hand, PDVs produced from source XML schemas, unlike these schemas, provide *a unique way* to translate user visible ontology nodes, by associating with each ontology node *at most one* node in a single PDV. This implies that a published XML schema may produce several PDVs. Each time a schema is published, the system must assist the user to generate PDVs, through semi-automatic procedures. This additional effort at publishing time is largely justified by the effort saved at query rewriting time, when heavy combinatorial computation and possibly wrong rewritings are avoided.

Figure 1 illustrates the difference between PDVs and XML schemas through a simple example. The ontology contains a single concept (Artist) with three properties (name, country, birth date). The published XML schema is a tree containing information about two kinds of artists: film directors and actors. Two PDVs are obtained from this schema, so as to dissociate directors from actors both mapped to concept Artist in the ontology, each one providing

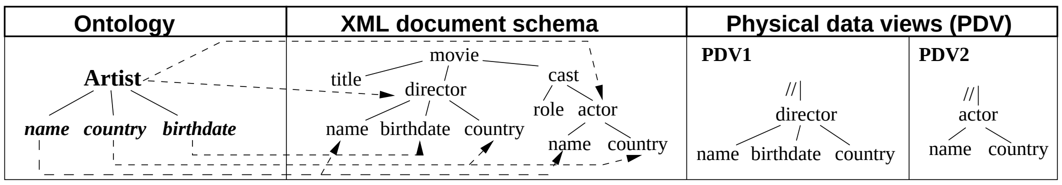


Figure 1: XML document schema vs Physical Data View

a unique translation for the artist (possibly incomplete, e.g. actors lack birth date). Useless nodes are removed from each PDV; this produces a "//" edge above the root: e.g, nodes *movie*, *cast* and *role* removed when creating PDV2.

We present in this paper the *OpenXView* model for open XML data integration. The model aims at *simplified access to data through queries*, combined with *simplified management of the data integration view*.

Users access data by expressing queries over the common ontology structure in *a very simple query language*, based on projections and selections over ontology nodes. Besides the advantage, common to all data integration models, of not requiring knowledge about heterogeneous and changing source schemas, OpenXView avoids also the need of mastering the subtleties of XML query languages. Querying OpenXView asks no more expertise than querying a single relational table. Not only novice users benefit from this simplicity, but also application developers.

Simple management of the data integration process is very important for open systems, because of their continuously and unpredictably content changing. Unlike relational integration systems [5], the OpenXView view is not defined by a query, but rather as a set of one-to-one mappings between source and target schema nodes. The advantage of such a mapping-based view is that it can be semi-automatically generated [14, 13] at publishing time and that it is simpler to visualize and to modify through graphical user-friendly tools. Moreover, OpenXView uses a *local-as-view integration model*, in which local sources are defined as views over the global ontology schema. This simplifies change management, as publishing/modifying a source only interacts with the global schema, not with the other sources.

This paper focuses on *the query rewriting problem* in the OpenXView system. Given a simple "select-where" query Q on the ontology, the system translates Q into a query expression Q' that refers only to PDV structures issued from published schemas, and such that answers to Q' are a subset of answers to Q. Q' contains three main query operations: (i) *structured tree-queries* expressed on PDV trees, to filter and get data from documents [1], (ii) *joins*, because the queried elements may not exist all in the same PDV, and (iii) *unions*, because there are several ways to answer the query. Unlike existing models, where joins are explicitly expressed in the query or in mappings, in OpenXView joins are implicit, based on *concept keys* defined in the common ontology. The canonical form of Q' is a union of all the possible joins between PDVs that provide the queried elements. We propose in this paper several algorithms for query rewriting in OpenXView.

The main contributions of this paper are:

- An original model, called OpenXView, for open XML data integration systems, i.e. adapted to heterogeneous and changing XML content. Based on a hybrid common schema (ontology - XML structure), OpenXView provides both easy querying and simple maintainance of the integration view.
- An algorithm for query rewriting in OpenXView, in a context where existing algorithms are not suitable.

The paper is organized as follows. Section 2 presents related work, then Section 3 defines the data model. Section 4 presents the query rewriting algorithm, Section 5 describes experimental tests and Section 6 concludes the paper.

2 Related Work

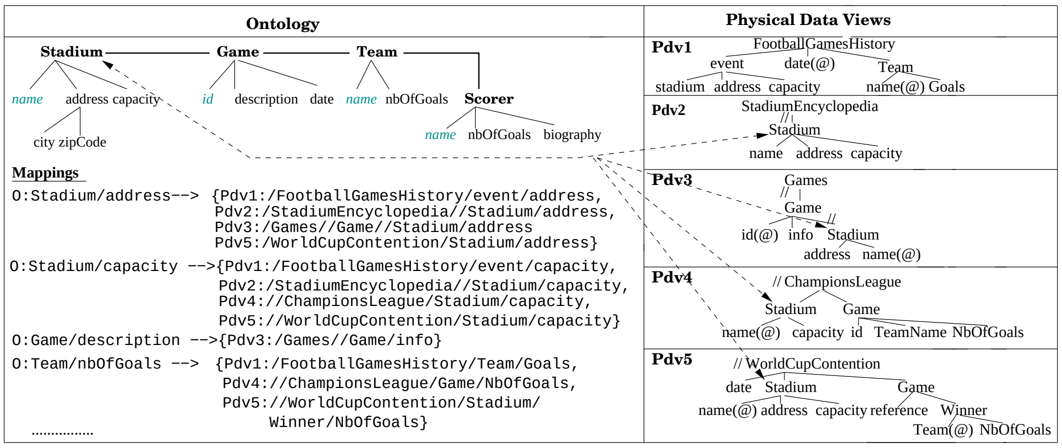
The query translation problem in OpenXView is a particular case of query rewriting using views [3, 5]. PDVs are views over the global ontology, defined through mappings, and the goal is to rewrite the simple "select-where" query on the ontology schema into a union of joins of tree-queries over PDVs.

Many approaches for query rewriting have been studied under different assumptions on the form of the query and the views, most of them for the relational model: query rewriting in description logic [4], recursive queries [11], conjunctive queries and views [5], etc. Significantly less results emerged for semistructured data, most of them focusing on points different from the aspects addressed in our present work: XPath queries and views for caching [15], results restructuring [17], translation to SQL queries [10].

Among query rewriting algorithms, we best compare to the Bucket [5] and MiniCon [12] algorithms. Even though they concern a different model (relational data, explicit joins in queries), the translation process is similar and produces a union of joins. Unlike Bucket/MiniCon, we reduce the number of rewritings to a reasonable subset, by only considering minimal rewritings. Experimental comparison between our algorithm and the Bucket strategy adapted to OpenXView shows better performances for our algorithm if the number of sources is large.

Mixing tree structures and graph ontologies was rarely proposed for the integration model. [2] uses a graph ontology, but extracts trees from it for query processing and [6] introduces more flexibility in tree structures. Query

[1]From PDVs'construction, it follows that a tree-query on a PDV is a tree-query on a the published schema it is constructed from.



```
O:Stadium/address--> {Pdv1:/FootballGamesHistory/event/address,
                      Pdv2:/StadiumEncyclopedia//Stadium/address,
                      Pdv3:/Games//Game//Stadium/address
                      Pdv5:/WorldCupContention/Stadium/address}
O:Stadium/capacity -->{Pdv1:/FootballGamesHistory/event/capacity,
                       Pdv2:/StadiumEncyclopedia//Stadium/capacity,
                       Pdv4://ChampionsLeague/Stadium/capacity,
                       Pdv5://WorldCupContention/Stadium/capacity}
O:Game/description -->{Pdv3:/Games//Game/info}
O:Team/nbOfGoals -->  {Pdv1:/FootballGamesHistory/Team/Goals,
                       Pdv4://ChampionsLeague/Game/NbOfGoals,
                       Pdv5://WorldCupContention/Stadium/
                            Winner/NbOfGoals}
  ................
```

Figure 2: An example of ontology, Physical Data Views (PDV) and mappings

simplification over XML data, similarly to our approach, was studied in [9], where queries use tag names only, and [16], where an annotated XML structure is used to generate query form applications. None is adapted to changing, heterogeneous data.

OpenXView is related to previous work of the authors [1], where the system aimed at query simplification over heterogeneous XML data with few changes. It was advocated in [1], that query rewriting can be strongly simplified by fixing in advance unions and joins in the integration view, which is unrealistic in open systems.

3 The Data Model

Definition 1 *An **OpenXView ontology** is a labeled directed graph, such that:*

- *It contains two kinds of nodes:* concepts, *having unique names, and concept* properties*. Each concept has a set of properties attached, each property belongs to a single concept (no property sharing).*
- *For a given concept, its properties form* a composition tree *whose root is the concept itself, the edges representing "partOf" relationships. Properties are* typed*; types may be* atomic *(integer, date, string, etc.) or* composed *(XML element).*
- *Informally a concept (property) instance is the set of instances of pdv nodes attached (by the ontology-pdv mapping) to this concept (property). A pdv node instance is the set of corresponding paths from root to the node instance in any document satisfying the pdv. A more formal semantics of PDVs is given in [1].*
- *Each concept has* a key*, composed of a subset of its properties - without loss of generality, we consider in the following, keys composed of a single property. The property k that serves as key allows to specify the pdv paths on which a join may occur: if k is mapped to $n1$ in $pdv1$ and $n2$ in $pdv2$, then a join may occur between a document satisfying $pdv1$ and a document satisfying $pdv2$ and whose paths from root to n1 (resp. from root to n2) have same value for the elements $n1$ and $n2$.*
- *Concepts may be connected through symmetric "relatedTo" relationships (edges in the ontology graph), that indicate, at a general level, a semantic relationship between the concepts.*

As an example, Figure 2 presents a simple ontology containing four concepts (*Stadium, Game, Team, Scorer*), each one with several properties connected in "partOf" trees, and a key (in italic). There are "relatedTo" relations between *Stadium* - *Game*, *Game* - *Team* and *Team* - *Scorer*.

Ontology relations express *constraints* to be satisfied on real data, in order to enforce semantics. A "partOf" relation between properties in the ontology constrain associated nodes in a PDV to have a similar "partOf" relation. More precisely, if p_1 `partOf` p_2 in the ontology, and in some PDV e_1 and e_2 are respectively associated with p_1 and p_2, then e_1 *must be an ancestor of* e_2. Our choice is to *check "partOf" relations between properties at publishing time* and consider they must exist for any query, while *relations between concepts, more flexible, are checked at query rewriting*, as described in the following.

Definition 2 ***OpenXView data sources*** *are represented as PDVs (introduced in Section 1).* ***A PDV*** *is defined by* a labeled tree, *and* a mapping *between this tree and the ontology.*

- ***The PDV structure*** *is a tree, where each node has a parent edge labelled with "/" or "//". The PDV tree represents a summary of simple path expressions that lead to elements in an XML document of the data source.*
- ***The mapping*** *between a PDV and the ontology is a set of* relations *between nodes of the ontology (concepts or properties) and nodes of the PDV tree (paths in the documents), such that* ***an ontology node is mapped to at most one node of that PDV****. As explained in Section 1, this restriction guarantees that any set of ontology nodes has at most one translation into a single PDV, thus simplifying query rewriting.*

Figure 2 describes five PDVs, for documents about football games, where only "//" edges are marked (the default is "/"). Nodes annotated with (@) represent attributes. Mappings between PDVs and the ontology nodes are presented here grouped by ontology node, such as needed at query rewriting. Each ontology node has at most one corresponding node in each PDV, e.g. *Stadium/address* is mapped to */FootballGamesHistory/event/address* in PDV1, to */StadiumEncyclopedia//Stadium/address* in PDV2, etc.

Definition 3 ***An OpenXView query*** *Q is a "select-where" query over the ontology nodes:*

Q: `Select` $p_1, ..., p_n$
`Where` $cond_1(p'_1)$ `and`...`and` $cond_m(p'_m)$

where thep_is and p'_js $1 \leq i \leq n$, $1 \leq j \leq m$ are ontology nodes (concepts or properties) and cond_j *are predicates over the node value, compatible with the node type.*

A user query on the ontology is translated into a *union* of queries over PDV structures. Each member of the union is either a tree query over a single PDV if the user query elements can all be found in the PDV, or (the query elements cannot be found all in the same PDV) the member is a *join* of tree queries over several PDVs. Each PDV is involved in such a join as a *tree query pattern*, expressing structural conditions, query projections and selections, for matching and extracting data from XML documents. ***Query translation produces a union of n-ary joins between PDV tree query patterns***. Union expresses all the possible ways to answer the user query (by joining PDVs, or with a single PDV). We call **rewriting** such an n-ary join, i.e. query translation produces a set of rewritings. ***Joins between PDVs are based on concept keys***, such that the same concept instance is chosen in different PDVs to answer the query.

The example in Figure 3 shows a user query formulated on the ontology in Figure 2, asking for the stadium address, capacity and the game description, for games where a team scored more than 3 times. A possible covering of all the query elements, is obtained by a join between $Pdv3$ and $Pdv5$, where $Pdv3$ provides the game description and $Pdv5$ the stadium address, capacity and the team number of goals. The join is based on the key of $Game$. Nevertheless, in order to be **valid**, a rewriting must fulfill an additional condition: *to satisfy the semantic relations between the query concepts*.

Definition 4 *Each "relatedTo" link between two concepts c_1, c_2 appearing in a query Q, results in a* ***query constraint****. Each rewriting of the translation must satisfy this constraint. Several semantics for query constraints are possible in OpenXView; without loss of generality we consider in the following the simple seamnics:* ***if*** *c_1 $relatedTo$ c_2****, some PDV of the rewriting must contain nodes associated with the keys of*** *c_1* ***and*** *c_2. We note the constraint Rel(c_1, c_2).*

The rationale of constraints checking is to restrict the couples of instances for c_1 and c_2 only to those semantically related. Note that PDV semantics [1] guarantees that only the "closest" instances of c_1 and c_2 are returned.

4 Translating user queries

We illustrate the query translation algorithm on the user query example of Figure 3, expressed as:

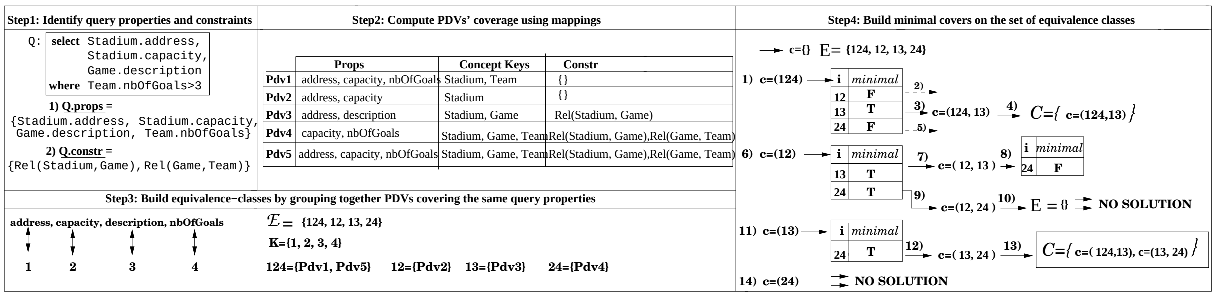
Q : `Select` Stadium.address, Stadium.capacity, Game.description
`Where` Team.nbOfGoals > 3

The translation process has 6 steps, illustrated in Figures 3 and 5. It produces *a set of rewritings* of the original query, each rewriting being a n-ary join between PDV tree patterns, as explained above. The translation steps are:

- Step1: Identify query properties and constraints.
- Step2: Using ontology-to-PDV mappings, compute for each PDV the query properties and constraints it covers.
- Step3: Create equivalence classes by grouping together PDVs that cover the same query properties.
- Step4: Build minimal covers of the query properties, using the set of equivalence classes.
- Step5: For each minimal cover, compute the valid rewritings.
- Step6: For each rewriting, generate an equivalent XQuery, that filters in each document only the closest possible instances of query concepts [1].

Due to lack of space, we focus here on *cover generation* (steps 1-4), the most critical part of query rewriting. Subsequent generation of valid rewritings and of the final XQuery (steps 5-6) are only briefly illustrated on the same example. We propose here ***two important optimizations*** over the naive approach, that would consider each possible subset of PDVs and check whether it covers the query properties. An experimental study presented in the following section evaluates the improvement brought by our algorithm.

1. We reduce the complexity, by transposing the cover problem from PDVs to equivalence classes. For n PDVs (n is continuously growing and may be very large), and a query over k properties, there are at most $2^k - 1$ equivalence classes, where k is small and does not vary in time. As experimentally verified in section5, we believe that in most practical cases the number of non-empty equivalence classes is much smaller.
2. We strongly reduce the number of rewritings, by only considering *minimal covers*, i.e. covers where no PDV/class is redundant, as explained in the following. We propose a complete and effective algorithm for generating minimal covers.


Step1: Identify query properties and constraints
Q:
1) Q.props =
{Stadium.address, Stadium.capacity, Game.description, Team.nbOfGoals}
2) Q.constr =
{Rel(Stadium,Game), Rel(Game,Team)}
Step2: Compute PDVs' coverage using mappings
Step3: Build equivalence−classes by grouping together PDVs covering the same query properties
address, capacity, description, nbOfGoals
1
2
3
4
ℰ = {124, 12, 13, 24}
K={1, 2, 3, 4}
124={Pdv1, Pdv5}
12={Pdv2}
13={Pdv3}
24={Pdv4}
Step4: Build minimal covers on the set of equivalence classes
c={}
E= {124, 12, 13, 24}
1) c=(124)
i
minimal
12 F
13 T
24 F
2)
3)
c=(124, 13)
4)
C={ c=(124,13) }
5)
6) c=(12)
13 T
24 T
7)
c=(12, 13)
8)
24 F
9)
c=(12, 24)
10)
E = {}
NO SOLUTION
11) c=(13)
24 T
12)
c=(13, 24)
13)
C={ c=(124,13), c=(13, 24) }
14) c=(24)
NO SOLUTION

```
Q: select Stadium.address,
          Stadium.capacity,
          Game.description
   where  Team.nbOfGoals>3
```

	Props	Concept Keys	Constr
Pdv1	address, capacity, nbOfGoals	Stadium, Team	{}
Pdv2	address, capacity	Stadium	{}
Pdv3	address, description	Stadium, Game	Rel(Stadium, Game)
Pdv4	capacity, nbOfGoals	Stadium, Game, Team	Rel(Stadium, Game),Rel(Game, Team)
Pdv5	address, capacity, nbOfGoals	Stadium, Game, Team	Rel(Stadium, Game),Rel(Game, Team)

Figure 3: Steps for translating query Q

4.1 Step1: Identify query properties and constraints

The following elements are identified in the query (Figure 3 step 1):

- *Properties* utilized in the query: $Q.props$ = $\{$ Stadium.address, Stadium.capacity, Game.description, Team.nbOfGoals$\}$.
- *Constraints* involved in the query, coming from "relatedTo" relations between concepts utilized in the query: $Q.constr$ = $\{$Rel(Stadium, Game), Rel(Game, Team)$\}$.

4.2 Step2: Compute for each PDV the query properties and constraints it covers

Definition 5 ***PDV coverage****: Given a PDV pdv and an ontology node p, let c be the concept of p (if p is a concept, then c=p). We say that pdv* ***covers property*** *p if there exists some node in pdv mapped to p* ***and*** *there exists some node in pdv mapped to the key of c. We say that pdv* ***covers a constraint*** *cst = $Rel(c_1, c_2)$ if there exist nodes in pdv mapped to the keys of both concepts c_1 and c_2.*

Notation: Prop(pdv) is the set of properties covered by PDV pdv (properties of the ontology for which there is a mapping with the nodes of pdv). For an equivalence class $cl \in \mathcal{E}$, PDV(cl) is the set of PDVs in the equivalence class and Prop(cl) is the set of query properties covered by each PDV in cl.

Notation: We index $Q.props$ by integers in $K = \{1, ..., k\}$, where k is the size of $Q.props$. E.g., in Figure 3 step 3, 3 denotes property Game.description. The set $Q.props$ can be represented by K. The set of covered properties in an equivalence class, a subset of K, is denoted , for simplicity, by the ordered sequence of its elements, i.e. $\{$1, 2, 4$\}$ is denoted by 124. In Figure 3 step 3, there are 4 equivalence classes: 124, 12, 13 and 24. Class 124 covers properties 1, 2 and 4 and contains Pdv1 and Pdv5.

To build the set of equivalence classes, a single traversal of the set of PDVs is necessary, each PDV being placed in the equivalence class that corresponds to its property coverage. In our example, query translation only handles 4 equivalence classes. More generally, as previously argued, query translation is expected to reduce the large number of PDVs to a very small number of equivalence classes.

4.3 Step4: Build minimal covers of the set of query properties

Definition 6 ***Cover****: Let $\mathcal{E}$ be the set of equivalence classes of PDVs over the set of query properties K. A cover c of K with elements of $\mathcal{E}$ is a subset of $\mathcal{E}$, such that $\bigcup_{cl \in c} Prop(cl) = K$ (classes in c cover together all the properties in K). We note Prop(c)= $\bigcup_{cl \in c} Prop(cl)$. A partial cover c' is a subset of $\mathcal{E}$ such that Prop(c$'$) $\subset K$ but Prop(c$'$) $\neq K$.*

e.g. for the example in Figure 3, c=$\{$124, 12$\}$ is just a partial cover, because it does not cover 3, while $\{$124, 12, 13$\}$ and $\{$124, 13$\}$ are covers of K.

Definition 7 ***Minimal cover:*** *A cover c is minimal if the removal of any member of c produces a partial cover. A partial cover c is minimal if the removal of any member produces a partial cover that covers less properties than c.*

In the example below, c=$\{$124, 12, 13$\}$ is not minimal, because c without member 12 is still a cover, while $\{$124, 13$\}$ is a minimal cover.

The problem of finding a minimal set cover is a well known NP-complete combinatorial problem [7]. We now give an algorithm called MC (Minimal Cover) (see Figure 4), that computes all the minimal covers.

4.3.1 MC-algorithm

The recursive algorithm ***MC*** takes as input parameters:

- c: a sequence of equivalence classes representing a minimal partial cover, to be completed to a minimal cover.
- E: the set of equivalence classes not yet considered in building c. E is structured as an ordered list.
- MP: the multiset of properties covered by classes in c, incrementally updated by the algorithm. As a matter of fact, MP is structured so as to keep for each property the number of classes covering it, e.g. if c=(12, 13), then MP=$\{$1:2, 2:1, 3:1$\}$.

```
MC($c$, $E$, $MP$)
 output : set of covers
Begin
 If ($MP \overset{s}{=} K$) %–$c$ is a cover
    return($\{c\}$)
 End if
 $\mathcal{C}$=$\emptyset$
 While $E \neq \emptyset$ repeat
    $i$=pop($E$)
    If minimal($c$, $i$, $MP$)
       $\mathcal{C}$=$\mathcal{C} \cup MC(c+i$, copy($E$), $MP \cup Prop(i))$
    End If
 End While
 return $\mathcal{C}$
End$MC$
```

```
minimal($c$, $i$, $MP$)
 output : boolean
Begin
 If $Prop(i) \subset MP$
    return(false)
 End If
 $NewMProp$=$MP \cup Prop(i)$
 For each $m \in c$ repeat
    If ($newMProp - Prop(m)$)$\overset{s}{=}NewMProp$
       return(false)
    End If
 End For
 return true
End $minimal$
```

Figure 4: Minimal Cover (MC) algorithm and minimality test

The first call of ***MC*** uses $c = (\ ), E = \mathcal{E}, MP = \emptyset$. The output of ***MC*** is the set of covers obtained by extending c with classes in E. The algorithm realizes the following actions:

1. A call to ***MC*** returns $\{c\}$ when c covers K (c is a cover). The cover test ($MP \overset{s}{=} K$) verifies whether MP and K are equal as sets, i.e. they have the same elements, no matter the multiplicity of MP.
2. If c is partial, one tries to extend c with each class i in E. Candidate classes are extracted in the order of E, using the pop function that returns E's first element and discards it from the list. If the new partial cover is minimal (the code of the $minimal$ boolean function is given in Figure 4), a recursive call of ***MC*** computes minimal covers for the new partial cover $c + i$, whose cover multiset is $MP \bigcup Prop(i)$, with the remaining classes (a copy of E). The set of equivalence classes E transmitted to each recursive call only contains classes *after* candidate i in the order of $\mathcal{E}$. This ensures that cover sequences produced by the algorithm respect the order of $\mathcal{E}$, and guarantees that *no cover is produced twice*.
3. Boolean function $minimal$ (Figure 4) takes 3 parameters: c - a minimal partial cover, i - the candidate class for extending c, MP - the multiset of properties covered by c, and returns `true` iff $c + i$ is a minimal partial cover. The algorithm, linear in the size of c, realizes the following actions:
   (a) It tests whether i is not redundant ($Prop(i) \subset MP$). If i is redundant, it returns false.
   (b) If i is not redundant, we check for each member m of c whether $c + i$ is redundant: $NewMprop$=$MP \cup Prop(i)$ is the multiset of properties for $c+i$. m is redundant if $NewMprop - Prop(m) \overset{s}{=} NewMProp$, i.e. when removing m from $c + i$, no element disappears from $NewMprop$. Then $c + i$ is minimal only if no m is redundant.

In the example in Figure 3 step 4, $\mathcal{E}$ ={124, 12, 13, 24}. In the first call of ***MC***, c=$\emptyset$, the first member of $\mathcal{E}$: 124, produces a minimal partial cover (124), to be extended through a recursive call, with c=(124), E={12, 13, 24}. The first candidate to extend c is i=12, but $c + i$ is not minimal, because i is redundant. 12 is removed from E and the next candidate is i=13, that leads to a minimal partial cover (124, 13), to be extended with c=(124, 13), E={24}. But c covers K, so (124, 13) is a minimal cover solution, and so on until all minimal covers are found.

Lemma 1 **MC** *generates* ***all*** *minimal covers and* ***only*** *minimal covers, without duplicates.*

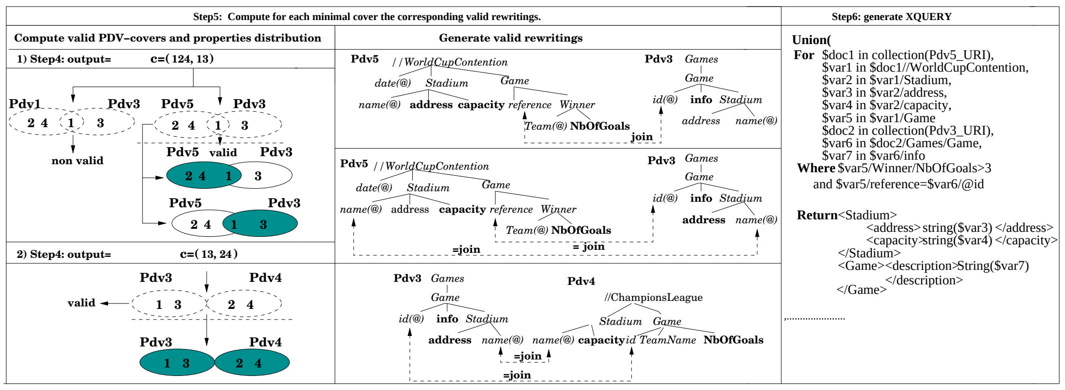
Sketch of proof: Minimality being checked for each cover, ***MC*** generates only minimal covers. Duplicates are avoided by generating only sequences that respect the order of $\mathcal{E}$. Since each call to ***MC*** extends the given partial cover to distinct partial covers, by induction one easily shows that final covers are also distinct. Completeness is also straightforward. Any minimal cover can be represented as a sequence of classes respecting the order of $\mathcal{E}$: c=(cl_1, ..., cl_n). It is easy to show that each prefix of c is a minimal partial cover. Then c is produced by successive calls to ***MC*** that produce minimal partial covers (cl_1), (cl_1, cl_2), ...

4.4 Steps 5 and 6: Compute valid rewritings and generate XQuery

These two steps are illustrated (see Figure 5) on the example query. First, minimal covers of equivalence classes, produced at step 4, are transformed into minimal covers of PDVs. This means that for each minimal cover c produced by ***MC***, we replace equivalence classes in c with any possible PDV in each class, i.e. we compute the cartesian product of c's classes. The result is a set of minimal *PDV-covers*, among which we only keep ***valid*** PDV-covers. The validity test checks whether the PDV-cover contains all the necessary elements to correctly join the PDVs. In our case, the test only concerns query constraints: for each $Rel(c_1, c_2)$, there must exist a PDV covering it (fusion joins are always possible, since a PDV that covers a property also covers the property's concept key).

Consider the example in Figure 5, for cover c =(124, 13). Class 124 contains $\{Pdv_1, Pdv_5\}$ and 13 contains $\{Pdv_3\}$, so we obtain two PDV-cover candidates: (Pdv_1, Pdv_3) and (Pdv_5, Pdv_3). The first one is not valid, since it does not cover constraint $Rel(Game, Team)$, while the second one is valid.

Unfortunately, a valid PDV-cover is not enough to produce a query rewriting. One must decide, for each query property, what PDV in the PDV-cover will produce it. E.g., in Figure 5, for PDV-cover (Pdv_5, Pdv_3), property


Step5: Compute for each minimal cover the corresponding valid rewritings.
Compute valid PDV–covers and properties distribution
Generate valid rewritings
Step6: generate XQUERY
1) Step4: output= c=(124, 13)
Pdv1 Pdv3 Pdv5 Pdv3
2 4 1 3
2 4 1 3
non valid
Pdv5 valid Pdv3
2 4 1 3
Pdv5 Pdv3
2 4 1 3
2) Step4: output= c=(13, 24)
Pdv3 Pdv4
valid
1 3 2 4
Pdv3 Pdv4
1 3 2 4
Pdv5 //WorldCupContention
date(@) Stadium Game
name(@) address capacity reference Winner
Team(@) NbOfGoals
join
Pdv3 Games
Game
id(@) info Stadium
address name(@)
=join
= join
Pdv4
//ChampionsLeague
Stadium Game
name(@) capacity id TeamName NbOfGoals
=join

```
Union(
For  $doc1 in collection(Pdv5_URI),
     $var1 in $doc1//WorldCupContention,
     $var2 in $var1/Stadium,
     $var3 in $var2/address,
     $var4 in $var2/capacity,
     $var5 in $var1/Game
     $doc2 in collection(Pdv3_URI),
     $var6 in $doc2/Games/Game,
     $var7 in $var6/info
 Where $var5/Winner/NbOfGoals>3
    and $var5/reference=$var6/@id

 Return<Stadium>
              <address> string($var3) </address>
              <capacity>string($var4) </capacity>
        </Stadium>
        <Game><description>String($var7)
                  </description>
        </Game>

,......................
```

Figure 5: Steps for translating query Q (continuation)

1 (Stadium.address) can be produced by either Pdv_5, or Pdv_3. Step5 decides, for each PDV-cover, what are the possible ***property distributions***. Due to lack of space, this problem is out of the scope of this paper. In Figure 5, there are 3 possible property distributions: $\{(Pdv_5 : 241, Pdv_3 : 3), (Pdv_5 : 24, Pdv_3 : 13), (Pdv_3 : 13, Pdv_4 : 24)\}$.

For each property distribution, a ***query rewriting*** is performed by considering for each PDV its tree-query and by adding join conditions. In Figure 5, there are three query rewritings, one for each property distribution. E.g., the rewriting for $(Pdv_5 : 24, Pdv_3 : 13)$ needs two join conditions: (i) a fusion join on the key of Stadium, to merge Stadium address and capacity, and (ii) a join on the key of Game, connecting the instance of Game that provides the game description (*info* in Pdv_3), to the instance of Game that covers $Rel(Game, Team)$ in Pdv_5.

Finally, each rewriting produces a ***For-Where-Return XQuery***. Figure 5 shows such an XQuery for the first rewriting. Finally the union of all the *For-Where-Return* queries is taken.

5 Experiments

We did a first experiment to evaluate the performance of the ***MC*** algorithm on a set of synthetic PDV coverages. We used an ontology of 20 concepts, 10 properties/concept, each concept being related to one (25%), two (50%) or three (25%) other concepts. A uniform distribution of PDVs was generated such that each PDV covers randomly 2 or 3 concepts, 3 properties of each of these concepts. Queries were randomly generated with k = 3, 4 or 6 properties, chosen as part of two related concepts in the ontology. Each measurement is the average of results for 50 random queries.

The left part of Figure 6 displays the number of equivalence classes vs the number n of PDVs, for queries of sizes k = 3, 4 and 6. For each k, the maximal number of equivalence classes ($2^k - 1$) is represented as an horizontal line. When n is large (larger than 1000) and k is small (less or equal to 6) we significantly reduce the complexity of query rewriting by first grouping the PDVs into a small number (at most $2^k - 1$) of equivalence classes as previously argued. Furthermore the experimental results confirm that the actual number of classes is much smaller in practical cases. Even for a very large number of PDVs (n=1000), the average number of classes is 5 instead of 7 (k=3), 9 instead of 15 (k=4) and 20 instead of 63 (k=6). The exponential growth of the number of classes with the query size is significantly reduced in practice.

The right part of Figure 6 compares the complexity of ***MC***, measured in number of minimality tests, with an alternative strategy, used by the Bucket algorithm [5], who first generates the sets of sources that cover the query, then tests for validity. In our case, this strategy corresponds to first generating covers, as being generated by ***MC*** but without minimality tests, then testing for minimality. Note that this strategy already improves Bucket, since it already avoids some non-minimal covers (a cover is never extended with new elements). ***MC*** checks each partial cover for minimality; on the one hand, this avoids as soon as possible covers that will never be minimal, on the other hand it may check for minimality, partial covers that do not lead to a cover. We counted and displayed the number of minimality tests for both strategies, as a function of the number of equivalence classes, for k=4. Beyond a threshold (8 equivalence classes in our case), ***MC*** behaves much better than the improved Bucket strategy, for which the number of tests grows fast. The third curve accounts for the real number of minimal covers, as an indication of the efficiency (tests count/results count) of both algorithms.

6 Conclusion

We introduced in this paper a model for open XML data integration systems based on a novel hybrid ontology-XML schema structure. We proposed an algorithm for query rewriting. A first experimental evaluation showed that it performs better than the adaptation of the Bucket algorithm [5]. As part of our future work we intend to improve this

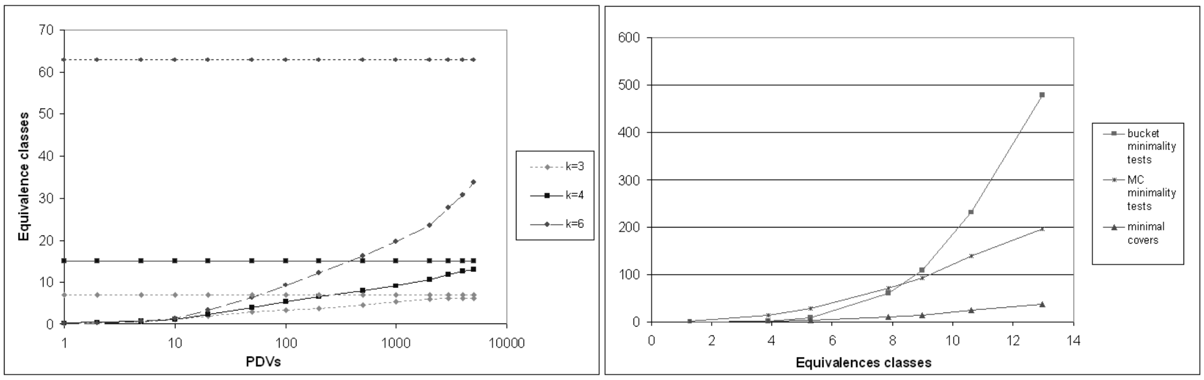


Figure 6: Measures on (a) number of equivalence classes and (b) number of minimality tests

algorithm and conduct a more comprehensive performance evaluation.

References

[1] Previous work of the authors.

[2] B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Querying xml sources using an ontology-based mediator. *CoopIS/DOA/ODBASE*, pages 429–448, 2002.

[3] D. Calvanese, D. Lembo, and M. Lenzerini. Survey on methods for query rewriting and query answering using views. *Technical Report D2I*, 2001.

[4] C.Beeri, A. Y.Levy, and M.-C. Rousset. Rewriting queries using views in description logic. *PODS*, 1997.

[5] A. Halvey. Answering queries using views: A survey. *The VLDB Journal*, pages 270–294, 2001.

[6] H.V.Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. Colorful xml: One hierarchy isn't enough. *Proc. SIGMOD*, 2004.

[7] R. Karp. Reducibility among combinatorial problems. *In R. Miller and J. Thatcher, editors, Complexity of Computer Communications*, pages 85–103, 1972.

[8] G. Koloniari and E. Pitoura. Peer-to-peer management of xml data: issues and research challenges. *ACM SIGMOD Record*, 2005.

[9] Y. Li, C. Yu, and H. Jagadish. Schema free xquery. *VLDB*, 2004.

[10] I. Manolescu, D. Florescu, and D. Kossmann. Answering xml queries on heterogeneous data sources. *VLDB*, 2002.

[11] O.M.Duschka and M.R.Genesereth. Answering recursive queries using views. *PODS*, 1997.

[12] R. Pottinger and A. Halevey. Minicon: A scalable algorithm for answering queries using views. *The VLDB Journal*, pages 182–198, 2001.

[13] R.J.Miller, L. Haas, and M.A.Hernandez. Schema mapping as query discovery. *VLDB*, 2000.

[14] L. Xiao, L. Zhang, G. Huang, and B. Shi. Automatic mappings from xml documents to ontologies. *Proceedings of the fourth international Conference on Computer and information technology*, 2004.

[15] W. Xu and Z. M. Ozsoyoglu. Rewriting xpath queries using materialized views. *VLDB*, 2005.

[16] Y.Papakonstantinou, M.Petropoulos, and V.Vassalos. Qursed: Querying and reporting semistructured data. *Proc. SIGMOD*, 2002.

[17] Y.Papakonstantinou and V.Vassalos. Rewriting queries using semistructured views. *SIGMOD*, 1999.