

SOURCE IDENTIFICATION AND QUERY REWRITING IN OPEN XML DATA INTEGRATION SYSTEMS

Francois BOISSON, Michel SCHOLL, Imen SEBEI, Dan VODISLAV
CEDRIC / CNAM Paris
francois.boisson@gmail.com, scholl@cnam.fr, imen.sebei@cnam.fr, vodislav@cnam.fr

ABSTRACT

This paper presents *OpenXView*, a model for open, large scale 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*. OpenXView provides an original integration schema, based on an hybrid ontology - XML schema structure model. We propose solutions for several important problems in such systems: *easy access to data* through a simple query language over the common schema, *simple data integration view management* when data changes and *scalable query rewriting algorithms*. This paper focuses on source identification for query rewriting in OpenXView, i.e. the computation of combinations of sources that can answer a user query. It proposes two algorithms for *minimal source combinations*, scalable with the number of sources. The first one is based on a general branch-and-bound strategy, while the second one, very efficient, is limited to queries whose number of attributes is no more than 8, sufficient in most applications.

Keywords: XML, heterogeneous data integration, ontology, query rewriting, source identification

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 simple querying environments.

We address here the problem of XML data integration in a particular context. First, we are interested in *open* integration systems over *a large number of sources*, 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 (Kolonari et al 2005) 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, which 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 (Halevy et al 2003). 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 for the common schema and for mappings to sources : the system often matches only source structures that preserve the same hierarchical relations between elements as in the common schema. Ontologies eliminate these drawbacks, but the model mismatch between XML schemas and ontologies leads to a more complex expression of mappings between sources and the common model.

We propose a model that combines the advantages of XML schemas and ontologies, 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 “relatedTo” relationships, more flexible at query processing.

On the source side, users publish XML tree-like schemas and documents. We introduced in (Vodislav 2006) 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.

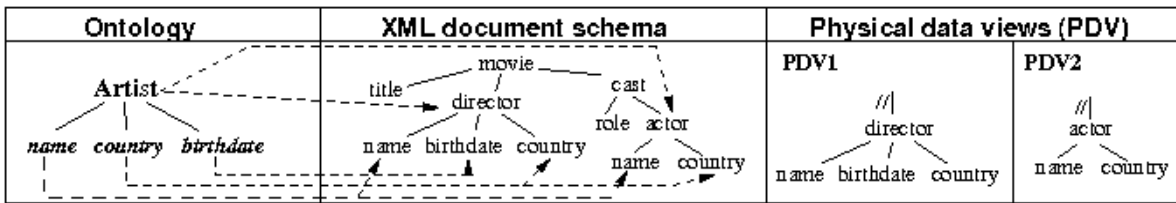


Figure 1: XML document schema versus Physical Data View

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 sub-trees 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 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, which are not necessarily XML database experts.

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 (Halvey 2001), 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 (Rahm and Bernstein 2002, Xiao et al 2004) 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 other sources.

This paper focuses on *the query rewriting problem* in the OpenXView system. Given a simple selection / projection query Q on the ontology, the system translates Q into a query expression Q' that refers only to PDV structures issued from published schemas. Q' contains three main query operations: (i) *structured tree-queries* expressed on PDV trees, to filter and get data from documents¹, (ii) *joins*, because the queried elements may not all exist 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, joins in OpenXView 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.

Our main contributions to the data integration issue are:

- o 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

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

- structure), OpenXView provides easy querying and simple maintenance of the integration view.
- An algorithm for query rewriting in OpenXView, in a context where existing algorithms are not suitable. We focus on the source identification part of query rewriting and propose two algorithms (SI1 and SI2) as main contributions of this paper, that are scalable with the number of sources. The SI2 algorithm, based on the pre-computation of minimal covers, outperforms SI1, but is limited to queries with no more than 8 properties.

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

2. RELATED WORK

Query translation in OpenXView is a particular case of query rewriting using views (Lenzerini 2002, Halvey 2001). PDVs are views over the global ontology, defined through mappings that respect a global-local-as-view (GLAV) schema (Madhavan and Halevy 2003). The goal is to rewrite simple selection/projection queries on the ontology 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 (Beeri et al, 1997), recursive queries (Duschka et al 1997), conjunctive queries and views (Halvey 2001), etc. Significantly fewer results emerged for semistructured data, most of them focusing on points different from the aspects addressed in our present work: results restructuring (Papakonstantinou 1999), translation to SQL queries (Manolescu et al 2002), translation between tree structures and/or ontologies (Halevy et al 2003).

Among query rewriting algorithms, Bucket, Inverse rule (Halvey 2001), MiniCon (Pottinger et al 2001), C&B (Deutsch and Tannen 2003) or (Yu and Popa 2004) are the better known. These algorithms, covering various data models and constraint types are not appropriate for OpenXView, because they do not deal with implicit joins. The related works closest to OpenXView are XML integration systems such as Styx (Amann et al 2002) and Yacob (Sattler et al 2005), that use an ontology as a global schema and implicit joins. Styx uses concept keys for implicit joins, but its query rewriting algorithm does not scale with the number of sources. Yacob uses both explicit and implicit joins, but defines a special (and very expensive) algebraic operator to compute concept extensions through implicit joins, instead of producing XQuery rewritings.

Mixing tree structures and graph ontologies was rarely proposed for the integration model. (Amann et al 2002) uses a graph ontology and extracts trees from it for query processing, (Jagadish et al 2004) introduces more flexibility in tree structures. Query simplification over XML data, was studied in (Li et al 2004), where queries use tag names only, and in (Papakonstantinou et al 2002), 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 (Vodislav et al 2006), where the system aimed at query simplification over heterogeneous XML data in a repository with few changes. It was advocated in (Vodislav et al 2006), 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 labelled graph, whose nodes, called **concepts**, have unique labels representing the concept name.*

- *Each concept has a set of **properties**, organised in a composition hierarchy represented as a labelled tree. The tree edges represent composition (“partOf”) relationships. By convention, the label of the root is the concept name. Properties are not shared between concepts. Properties (also named attributes) are typed; types may be atomic (integer, date, string, etc.) or composed (XML element). Only leaf attributes may have atomic types, all non-leaf attributes are composed.*
- *Each concept has a **key**, composed of a subset of its leaf attributes. Intuitively, each key value identifies an “instance” of the concept in the real data, as defined below.*
- *Each edge in the ontology graph represents a “relatedTo” **relation between concepts**, which implies a relation between the concept instances, as explained below.*

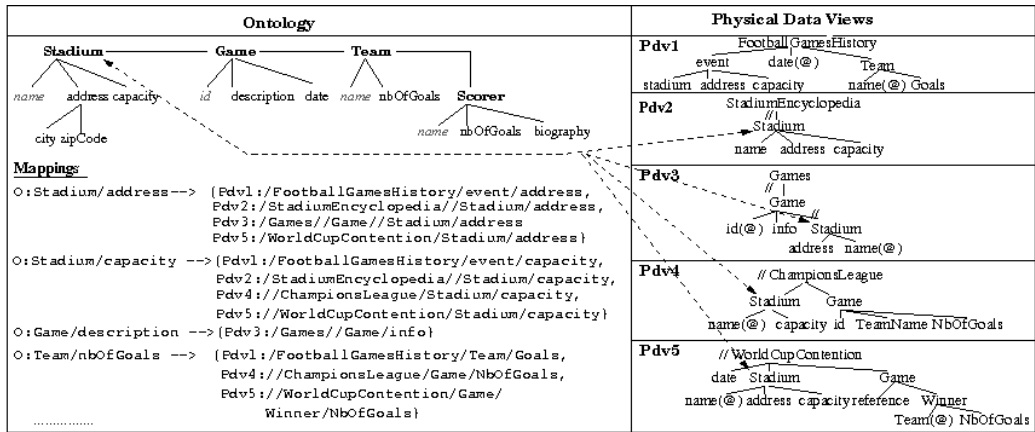


Figure 2: An example of ontology, Physical Data Views (PDV) and mappings

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 constrains 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_2 must be an ancestor of e_1 . 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 labelled tree, and a mapping between this tree and the ontology. For more details on PDVs, see (Vodislav et al 2006)*

- **The PDV structure:** a tree, where each node has a parent edge labelled: “/” 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 property nodes of the ontology 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 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.

Intuitively, a *concept instance* corresponds to a given value for the concept key (a key value identifies an instance). A *concept occurrence* corresponds to the occurrence of a concept instance in a document. There may be several *concept occurrences* for a *concept instance*, across various documents. A concept occurrence is composed of all the elements corresponding to the concept's properties “accompanying” the key element, i.e. respecting the structural constraints imposed by the PDV tree schema wrt the key element. A concept occurrence in a document generally contains only a part of the concept properties. The value of a concept instance containing all the properties is obtained by joining (on the key) several concept occurrences. .

Definition 3: *An OpenXView query Q is a conjunctive selection/projection query over ontology properties:*

$$Q: \text{Select } p_1, \dots, p_n$$

$$\text{Where } \text{cond}_1(p'_1) \text{ and } \dots \text{ and } \text{cond}_m(p'_m)$$

where the p_i and p'_j $1 \leq i \leq n$, $1 \leq j \leq m$ are concept properties and cond_j are predicates over the node value, compatible with the node type.

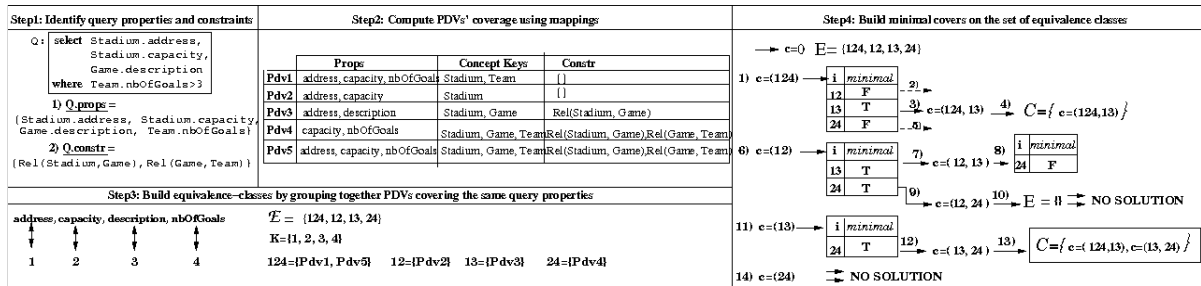


Figure 3: Steps for translating query Q

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 all the query elements can be found in the PDV), or (if 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 implicit (not expressed in the query or in mappings) and based on concept keys**, such that occurrences of 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 of 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 fulfil 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** $Rel(c_1, c_2)$ that must be satisfied by each rewriting of the query. Several semantics for query constraints are possible in *OpenXView*; without loss of generality we consider the following one: **if c_1 relatedTo c_2 , some PDV of the rewriting must contain nodes associated with the keys of both c_1 and 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. The chosen constraint semantics allows couples of instances of c_1 and c_2 that occur in a same document of PDV and that respect the PDV structural relations. Note that PDV semantics (Vodislav et al 2006) guarantees that only the “closest” instances of c_1 and c_2 in each document 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 7. 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:

- o Step 1: Identify query properties and constraints.
- o Step 2: Using ontology-to-PDV mappings, compute for each PDV the query properties and constraints it covers.
- o Step 3: Create equivalence classes by grouping together PDVs that cover the same query properties.
- o Step 4: Build minimal covers of the query properties, using the set of equivalence classes.
- o Step 5: For each minimal cover, compute the valid rewritings.
- o Step 6: For each rewriting, generate an equivalent XQuery that filters in each document only the closest possible instances of query concepts (Vodislav 2006).

Due to lack of space, we only focus in this paper on *source identification* through cover generation (steps 1-4). It is the most critical part of query rewriting. Source identification consists in finding all the combinations of sources that may answer the query. Subsequent generation of valid rewritings and of the final XQuery (steps 5-6) is only briefly illustrated on the same example. We propose here **two important optimisations** 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 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 Section 5, 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 two effective algorithms for generating minimal covers.

4.1. Step 1: Identify query properties and constraints

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

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

4.2. Steps 2 and 3: Compute PDV coverage and create equivalence classes

Definition 5: PDV coverage. Given a PDV pdv and an ontology property p , let c be the concept of p . We say that pdv **covers property** p if there exists some node in pdv mapped to p **and** it 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 . We note $Prop(pdV)$ and $Rel(pdV)$ respectively the set of query properties and query constraints covered by pdv .

The equivalence relation between PDVs is based on coverage, i.e. $pdv1 \equiv pdv2$ iff $Prop(pdV1) = Prop(pdV2)$. For a query Q of size k , the maximum number of equivalence classes is $2^k - 1$, independently of the number n of PDVs.

Notation: For an equivalence class cl , $PDV(cl)$ 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 $K = \{1, \dots, k\}$, where k is the size of $Q.props$. E.g., in Figure 3 Step 3, “3” denotes property *Game/description*. In the following $Q.props$ is 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 Pdv_1 and Pdv_5 .

To build the set of equivalence classes, a sequential scan 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. Step 4: 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 $\cup_{cl \in c} Prop(cl) = K$ (classes in c cover together all the properties in K). We note $Prop(c) = \cup_{cl \in c} Prop(cl)$. 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 fewer 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 (Karp 1972). For a given query with k properties, $|C_k|$, the maximal number of minimal covers, is exponential in k^2 .

Consequently, one expects this step to be expensive, even if it drastically reduces the number of covers (compared to the total number of minimal and non-minimal covers). We now present two algorithms for source identification, called SI1 and SI2 that compute minimal covers of the set of query properties.

4.3.1. SI1 Algorithm

This algorithm (Figure 4), recursively generates all minimal covers of K with equivalence classes in \mathcal{E} . The recursive branch-and-bound algorithm **MC** takes as input parameters:

1. c : a sequence of equivalence classes representing a minimal partial cover, to be completed to a minimal cover.
2. E : the set of equivalence classes not yet considered in building c . E is structured as an ordered list.
3. 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 $MP = \{1:2, 2:1, 3:1\}$.

<pre> SI1(\mathcal{E}, K) output: set of minimal covers begin $c_0 = ()$ $E_0 = \mathcal{E}$ $MP_0 = \emptyset$ return MC(c_0, E_0, MP_0, K) end SI1 </pre>	<pre> MC(c, E, MP, K) output: set of minimal covers begin if ($MP =_s K$) % c is a cover return ($\{c\}$) end if $C = \emptyset$ while $E \neq \emptyset$ repeat $i = \text{pop}(E)$ if minimal(c, i, MP) $C = C \cup \text{MC}(c+i, \text{copy}(E), MP \cup \text{Prop}(i))$ end if end while return C end MC </pre>	<pre> minimal(c, i, MP) output: boolean begin if $\text{Prop}(i) \subset MP$ return(false) end if $\text{NewMPProp} = MP \cup \text{Prop}(i)$ for each $m \in c$ repeat if ($\text{newMPProp} - \text{Prop}(m) =_s \text{NewMPProp}$) return(false) end if end for return true end minimal </pre>
--	--	--

Figure 4: SI1 algorithm using a branch-and-bound Minimal Cover (**MC**) algorithm with minimality test

The first call of **MC** uses $c = ()$, $E = \mathcal{E}$, $MP = \emptyset$. The output of **MC** is a set of minimal 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 ($MP =_s K$) 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 $MP \cup \text{Prop}(i)$, with the remaining classes (a copy of E). The set of equivalence classes E transmitted to each recursive call only

$$^2 |C_k| = \sum_{l=1}^{l=k} y(k, l), \text{ where } y(k, l) \text{ is the number of minimal covers of } \{1, \dots, k\} \text{ with } l \text{ members :}$$

$$y(k, l) = \frac{1}{l!} \sum_{m=l}^{\min(k, 2^l-1)} C_{m-l}^{2^l-1} m! s(k, m), \text{ where } s(k, m) \text{ is a Stirling number of the second kind.}$$

Example: $|C_4| = 49$, $|C_6| = 6424$, $|C_8| = 3732406$.

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 $NewMprop - Prop(m) =_s NewMprop$, 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 through a recursive call 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} : $c = (cl_1, \dots, cl_n)$. 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) , ...

4.3.2. SI2 Algorithm

In contrast to algorithm SI1, the novel algorithm we present in this section supposes the existence of the following structure: $MSC_k(A_s, c)$ a pre-computed relational table describing all the minimal covers of the set $\{1, 2, \dots, k\}$. Tuple $[A_s, c]$ in MSC_k states that the subset $A_s \in \mathcal{P}(\{1, 2, \dots, k\}) - \{\emptyset\}$ is a member of minimal cover c . As aforementioned, the number of minimal covers $|C_k|$ of $\{1, \dots, k\}$ increases very rapidly with k .

We assume in the following that this table holds in memory. This implies an upper bound on k which, with our implementation, is ≤ 8 . Because of space limitations we do not detail the off line construction of $MSC_k(A_s, c)$: it is pre-computed, independently of any query, by an algorithm similar to algorithm SI1.

We also consider a table $EC_Q(A_s, S)$, containing the equivalence classes computed in Step 3. It associates with each equivalence class (identified by the set A_s of covered properties), the set S of PDVs in that equivalence class. Given MSC_k and EC_Q , the following relational expression calculates C_Q the set of minimal covers of Query Q , which is composed only of members appearing in EC_Q (equivalence classes):

- $R(A_s) = \Pi_{A_s}(MSC_k(A_s, c)) - \Pi_{A_s}(EC_Q(A_s, S))$
- $S(c) = \Pi_c(MSC_k(A_s, c)) \triangleright \triangleleft R(A_s)$
- $C_Q(c) = \Pi_c(MSC_k(A_s, c)) - S(c)$

```

SI2(members, MSC_k)
  output : valid, all bits initially set to 1
begin
  for each A_s in P({1,..,k})-{\emptyset} loop
    if members(A_s)=false
      valid = valid and not MSC_k(A_s)
    end if
  end for
End SI2

```

Figure 5: SI2 algorithm utilizing pre-computed minimal set covers

$R(A_s)$ is the set of "bad" members (property subsets), where a bad member is a member of a minimal cover that is not in EC_Q . Then $S(c)$ is the set of bad minimal covers i.e. covers which hold at least one member in $R(A_s)$. Therefore, $C_Q(c)$ is the set of "good" minimal covers i.e. made of members appearing in EC_Q .

In practice, MSC_k is implemented as a bitmap with 2^k-1 lines (for all possible A_s) and $|C_k|$ columns (for all minimal covers): $MSC_k(A_s, c)$ is set to 1 if cover c has for a member A_s . Figure 6 illustrates the use of this bitmap for finding the minimal covers for the example query Q in Figure 3. The algorithm relies on two additional boolean vectors:

- *members*: with size 2^k-1 , where $members(A_s)$ is set to 0 if the set of attributes A_s is in $R(A_s)$, i.e. it is a bad member. Note that *members* is constructed with no additional cost when building $EC_Q(A_s, S)$.
- *valid*: with size the number of minimal set covers $|C_k|$ where $valid(c)$ is set to 1 if minimal set cover c is a good one, i.e. it belongs to $C_Q(c)$. All bits of *valid* are initialized to 1.

For Query Q , of size 4, we use MSC_4 , which has $2^4-1=15$ lines and $|C_4|=49$ columns. Initially, all *valid* elements are set to 1. The *member* vector on the left indicates that only equivalence classes 124, 24, 13 and 12 exist for Q . For each bad member (line in the bitmap where *member*=0), the minimal covers containing that member are invalidated in *valid*. At the end, as expected, only minimal covers $c=(124, 13)$ and $c=(13, 24)$ are valid for the query.

The size of the bitmap $|C_k|$ drastically increases with k . Therefore k must be reasonably small not only because MSC_k must hold in memory, but also because of the complexity of the bitmap operations is $O(|C_k|)$, which grows very fast. For the amount of RAM memory available in today computers, the bound is $k \leq 8$, which is reasonable for the applications we have in mind.

The complexity of algorithm SI_2 is then $O(2^k) \times |C_k|$. However since k is small, the bitmap operation is fast, as confirmed by the experiments presented in Section 5.

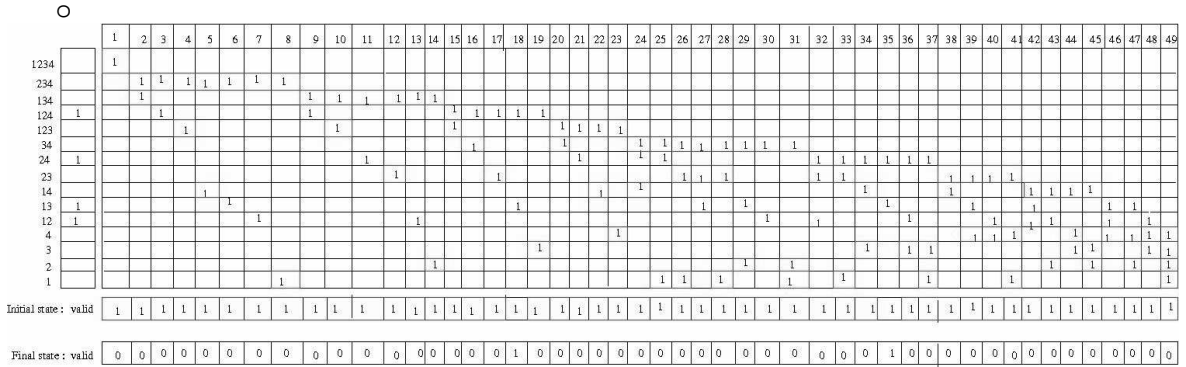


Figure 6: SI_2 bitmap structure, for the example query

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

These two steps are illustrated (Figure 7) 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 at Step 4, 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 (joins are always possible, since a PDV that covers a property also covers the property's concept key).

Consider the example in Figure 7, for $c=(124, 13)$. 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.

However, a valid PDV-cover is not enough to produce a query rewriting. One must decide, for each query property, what PDV in the cover will provide it. E.g., in Figure 7, for PDV-cover (Pdv_5, Pdv_3) , property 1 (*Stadium/address*) can be produced by either Pdv_5 , or Pdv_3 . Step 5 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 7, there are three 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 7, 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 join on *Stadium*'s key, to merge *Stadium* address and capacity, and (ii) a join on *Game*'s key, connecting instances of *Game* that provide game description (*info* in Pdv_3), to instances of *Game* that cover $Rel(Game, Team)$ in Pdv_5 .

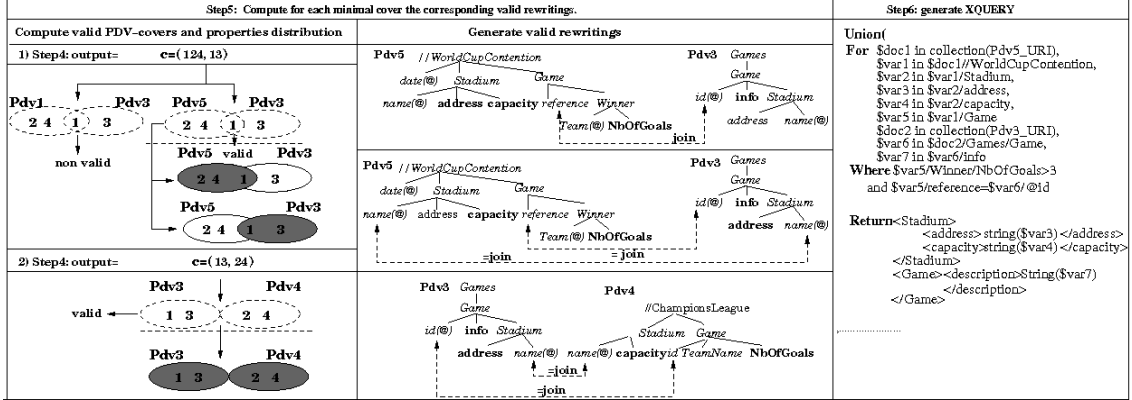


Figure 7: Steps for translating query Q (continuation)

Note that many improvements can be brought to this simple algorithm, by introducing equivalence and containment relations between rewritings in order to discard part of them, by delaying the moment when class-covers are transformed into PDV-covers, etc. All these improvements will be addressed in future work.

Finally, each rewriting produces a *For-Where-Return XQuery*. Figure 7 shows such an XQuery for the first rewriting. Finally, the union of all the *For-Where-Return* queries is taken.

5. EXPERIMENTS

The algorithms and experiments were implemented in Java (JDK1.5.0-06), on a PC with 512 MBytes of RAM and a P4 3 GHz CPU, running Windows XP. We used a synthetic ontology with 30 concepts of 10 properties each, and a number of sources ranging from $N = 100$ to $N = 10000$. Mapping of properties to sources (source indexing) respects *locality*, i.e. sources provide properties in several concepts instead of being randomly mapped to the ontology properties. More precisely, we used the following repartition: 25% of the sources cover 2 concepts, 50% 3 concepts and 25% 4 concepts, with 50% of the concept's attributes (randomly chosen) in each case. Queries were randomly generated with $k = 2, 4, 6$ attributes randomly chosen within two concepts. The results presented here are robust wrt changes in the repartitions and parameters. Because of space limitations we present only the results for the above synthetic set and queries.

Our performance evaluation focuses on the total time for running each of Step 3 (equivalence classes building) and Step 4 (minimal covers computation), concerning the source identification part of the query rewriting algorithm. Each time measure is the average over 50 random queries with the same value of k and N . We successively made three experiments: the first one evaluates Step 3 which is common to both SI1 and SI2 algorithms of Step 4. The second one compares the performance of both algorithms for Step 4 and the third experiment studies the predominance of one step over the other depending on the algorithm. Because of space limitations we only present the most illustrative figures for the general behavior of our algorithms.

Step 3: Figure 8.a displays the number of equivalence classes (number of tuples of $EC_{\mathcal{O}}$) vs the number of sources, for queries of size $k = 6$. Some nodes are labelled by the time taken to perform Step 3. The experimental results confirm that the number of classes (which theoretically can reach $2^k - 1$), is much smaller in practice (for $N = 4000$, a very large number of sources, the average number of classes is 33). Anyway, this number is very small wrt N for all values of N . Recall that Step 4 takes as input *classes* and not the sources themselves. Step 1 reduces the source identification problem to the scanning in Step 4 of an expected small number of classes. In contrast, naive algorithms such as Bucket (Halevy 2001) compute combinations

directly from the sources. As expected, the time growth is linear in N . Experiments for other values of k confirmed the same behavior and the linearity with k .

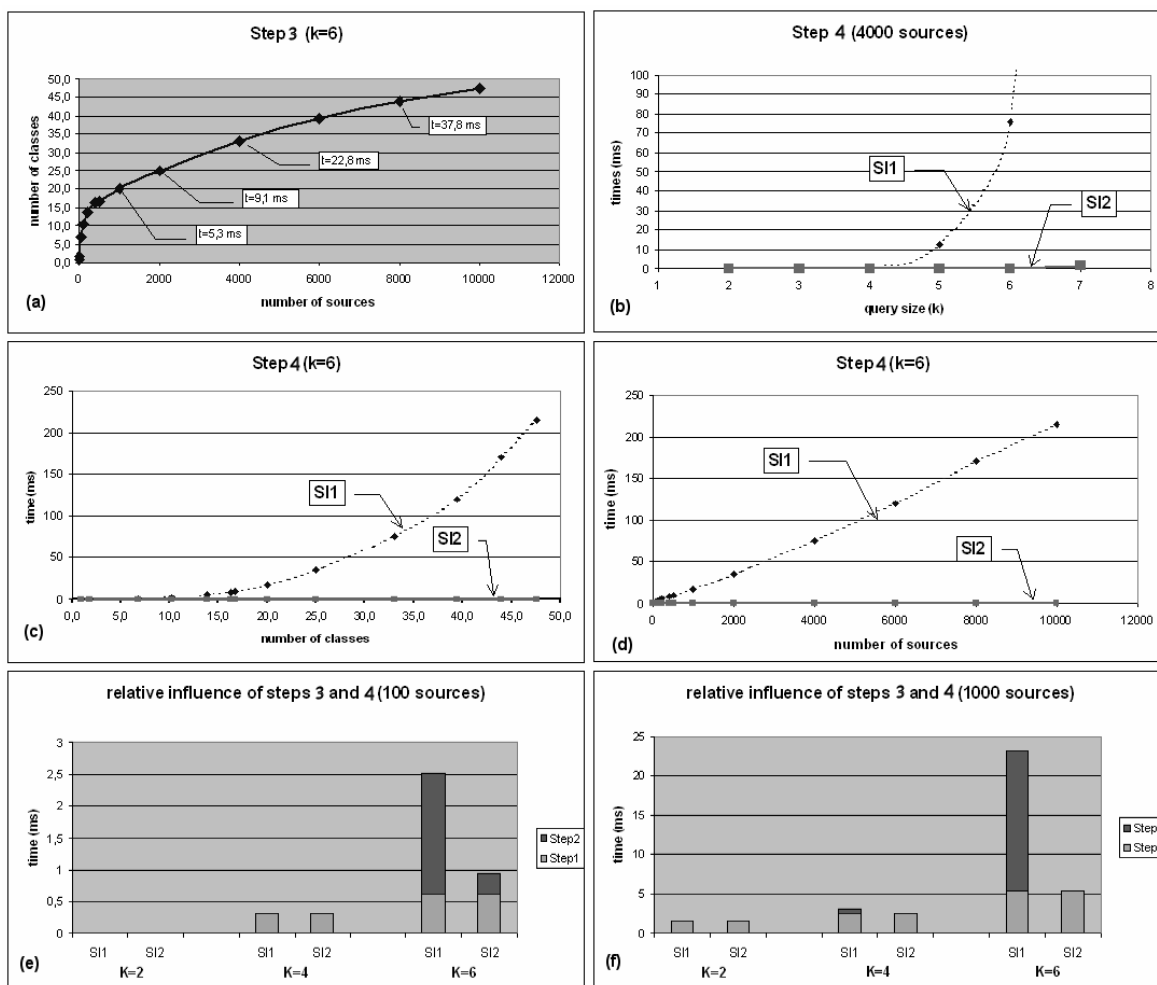


Figure 8: Experimental results

Step 4: We successively study the impact of k and of N on the time performance of Step 4. Figure 8.b displays the time for performing Step 4 with both algorithms vs the number of attributes k , for 4000 sources. While with SI2 it is almost negligible for all values of k (≤ 8), with SI1 it increases exponentially. Figure 8.c and Figure 8.d display the time for running Step 4 respectively versus the number of sources (Figure 8.c) and the number of classes (Figure 8.d). They confirm that algorithm SI2 drastically outperforms algorithm SI1. Time for running Step 4 with algorithm SI2 is negligible or at least less than 1 ms even for a large number of sources (classes). In contrast, algorithm SI1 performance degrades exponentially with the number of classes (Figure 8.c). Note however the quasi-linearity with N , because of the slow growth of the number of classes with N (Figure 8.d).

Step 3 / Step 4: Last we study the relative importance of Step 3 wrt that of Step 4. Time versus k is plotted for 100 sources (Figure 8.e) and for 1000 sources (Figure 8.f). The first important observation is that with algorithm SI2, when the number of sources is large, the time for running Step 4 is negligible and therefore the source identification performance is that of Step 3. As a matter of fact it was observed that as expected, the latter time is linear in k and N . Although we were not able to find sub-linear algorithms, Step 3 still deserves some improvements. The second observation is that with algorithm SI1, the larger k , the larger the predominance of Step 4 duration over Step 3, even for a relatively small number of sources: while with $k = 4$ Step 4 is almost negligible, with $k = 6$, Step 4 already lasts four times longer than Step 3. Since Step 4

duration is exponential with k , it is to be expected that with larger values of k , the performance of source identification with algorithm SI1 is that of Step 4, which is exponential with k . Therefore, for this algorithm as well, values of k larger than 8 are unpractical.

In conclusion, we assessed the impact on performance of the organization of sources into equivalence classes depending on the query (Step 3) and demonstrated the significant improvement on performance brought by algorithm SI2 wrt to algorithm SI1. SI2 performs well even with a large number of sources.

6. CONCLUSION

We introduced in this paper OpenXView, a model for open, large scale XML data integration systems, based on a novel hybrid ontology-XML schema structure. OpenXView uses a GLAV integration schema, well adapted to changing, open systems. We proposed a general algorithm for query rewriting, but focused on the source identification steps, that represents a critical part of query rewriting. We defined a general method for reducing the complexity of query rewriting in large scale data integration systems, based on the classification of sources in equivalence classes wrt the user query. We proposed two algorithms for source identification, based on the computation of minimal covers of the query properties with source classes: SI1, using a branch-and-bound strategy, and SI2, much more efficient, using pre-computed minimal covers. Both algorithms are limited to queries of reasonable size (number of attributes less or equal to 8). Experiments confirm that our algorithms are scalable with the number of sources and that SI2 largely outperforms SI1.

As part of our future work we intend to improve the query rewriting algorithm, in order to reduce the number of rewritings by studying equivalence and containment conditions, and by exploring ranking techniques. Another research direction concerns the optimization of query processing in OpenXView on top of various data integration environments: peer-to-peer systems, heterogeneous repositories, etc.

REFERENCES

- Amann et al, 2002. Querying xml sources using an ontology-based mediator. *Proc. CoopIS*, pp. 429-448.
- Beeri et al, 1997. Rewriting queries using views in description logic. *Proc. PODS*.
- Deutsch and Tannen, 2003. Reformulation of XML queries and constraints, *Proc. ICDT*
- Duschka et al, 1997. Answering recursive queries using views. *Proc. PODS*.
- Halevy et al, 2003. Piazza: Data management infrastructure for semantic web applications, *Proc. WWW*.
- Halvey, 2001. Answering queries using views: A survey. *The VLDB Journal*, pp. 270-294.
- Jagadish et al, 2004. Colorful XML: One hierarchy isn't enough. *Proc. SIGMOD*.
- Karp, 1972. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Communications*, pp. 85-103.
- Koloniari et al, 2005. P2P management of XML data: issues and research challenges. *ACM SIGMOD Record*.
- Lenzerini, 2002. Data Integration: A Theoretical Perspective. *Proc. PODS*.
- Li et al, 2004. Schema free XQuery. *Proc. VLDB*.
- Madhavan and Halevy, 2003. Composing mappings among data sources. *Proc VLDB*.
- Manolescu et al, 2002. Answering XML queries on heterogeneous data sources. *Proc. VLDB*.
- Papakonstantinou et al, 1999. Rewriting queries using semistructured views. *Proc. SIGMOD*.
- Papakonstantinou. et al, 2002. Qured: Querying and reporting semistructured data. *Proc. SIGMOD*.
- Pottinger et al, 2001. Minicon: A scalable algorithm for answering queries using views. *VLDB Journal*, 10(2-3).
- Rahm and Bernstein, 2001. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4), 334-350.
- Sattler et al, 2005. Concept-based querying in mediator systems. *VLDB Journal*, 14(1), 97-111.
- Vodislav. et al, 2006, Views for simplifying access to heterogeneous XML data, *Proc. CoopIS*.
- Xiao. et al, 2004. Automatic mappings from XML documents to ontologies. *Proc. IEEE CIT*, 321-355.
- Yu and Popa, 2004. Constraint-based XML query rewriting for data integration, *Proc. SIGMOD*.