UserMap – an Exploitation of User-Specified XML-to-Relational Mapping Requirements and Related Problems

(Technical Report)

Irena Mlýnková and Jaroslav Pokorný

Charles University
Faculty of Mathematics and Physics
Department of Software Engineering
Malostranske nam. 25
118 00 Prague 1, Czech Republic
{irena.mlynkova,jaroslav.pokorny}@mff.cuni.cz

Abstract. As the XML has become a standard for data representation, it is inevitable to propose and implement techniques for efficient managing of XML data. A natural alternative is to exploit features of (object-)relational database systems, i.e. to rely on their long theoretical and practical history. The main concern of such techniques is the choice of an appropriate XML-to-relational mapping strategy.
In this paper we focus on enhancing of *user-driven* techniques which leave the mapping decisions in hands of users who specify their requirements using schema annotations. We describe our prototype implementation called *UserMap* which is able to exploit the annotations more deeply searching the user-specified "hints" in the rest of the schema and applies an adaptive method on the remaining schema fragments. Using a sample set of supported fixed mapping methods we discuss problems related to query evaluation for storage strategies generated by the system, in particular correction of the candidate set of annotations and related query translation. And finally, we describe the architecture of the whole system.

Keywords: XML-to-relational mapping, user-driven strategy, adaptivity, similarity

1 Introduction

The XML [5] has undoubtedly become a generally acknowledged standard for data representation. This invoked a boom of implementations of W3C recommendations based on various storage strategies from traditional file systems to brand-new *native XML storage strategies*. But currently the most practically used techniques exploit a less efficient but verified and mature technology – (object-)relational database management systems ((O)RDBMS). Although the scientific world has already proven that native XML strategies perform much better, they still lack one important aspect – a reliable and robust implementation verified by years of both theoretical and practical effort.

Thus until the native XML methods "grow up", it is still necessary to improve XML data management in (O)RDBMS.

Currently there is a plenty of existing works concerning database-based[1] XML data management. Almost all the major database vendors more or less support XML and even the SQL standard has been extended by a new part SQL/XML [9] which introduces new XML data type and operations for XML data manipulation. The main concern of the database-based XML techniques is the choice of the way XML data are stored into relations, so-called *XML-to-relational mapping*. On the basis of exploitation or omitting information from XML schema we can distinguish so-called *schema-oblivious* (or *generic*) [7] and *schema-driven* [17] methods. From the point of view of the input data we can distinguish so-called *fixed* methods [7, 17] which store the data purely on the basis of their model and *adaptive* methods [10, 4], where also sample XML documents and XML queries are taken into account. And there are also techniques based on user involvement which can be divided to *user-defined* [2] and *user-driven* [3, 6], where in the former case a user is expected to define both the relational schema and the required mapping, whereas in the latter case a user specifies just local changes of a default mapping.

Both the user-driven and adaptive approaches try to solve the key problem of the fixed methods – the fact that there is no universally suitable fixed method. An illustrative issue is updatability of data, where the efficient storage strategies significantly differ if the feature is required or not. A similar case is exploitation of redundancy which in general leads to a significant space overhead, but it can have reasonable applications, where the need of retrieval efficiency exceeds this disadvantage [3]. And there are also various types of XML data, such as, e.g., data related to Semantic Web, which require special treatment [18].

In this paper we introduce a prototype implementation called *UserMap* which exploits a combination of user-driven and adaptive strategies focusing on two persisting disadvantages of user-driven methods. Firstly, it is the fact that the default mapping strategy is (to our knowledge) always a fixed one. Since the corresponding system must be able to store schema fragments in various ways, an adaptive enhancing of the fixed method seems to be quite natural and suitable. The second shortcoming is weak exploitation of the user-given information. The annotations a user provides can not only be directly applied on particular schema fragments, but can be regarded as "hints" how to store particular XML patterns. We use this information twice again. Firstly, we search for similar patterns in the rest of the schema and store the found fragments in a similar way. And secondly, we exploit the information in the adaptive strategy for not annotated parts of the schema. Hence, *UserMap* proposes new types of annotations, i.e. new storage strategies.

Next, we discuss problems related to the resulting storage strategies. We deal with two key issues – correction of the candidate set of annotations proposed by the system and related query evaluation. In the former case we identify and discuss situations when the proposed annotations are either meaningless or a user interaction and/or a default choice is necessary to choose from multiple possibilities. In the latter case we deal with the interface between various storage strategies and the way the system should

[1] In the rest of the paper the term "database" represents an (O)RDBMS.

cope with redundancy. For this purpose we have selected a sample representative set of annotations using which we illustrate the related issues and open problems. Finally, we describe and discuss the architecture of experimental implementation of the whole system.

The paper is structured as follows: Section 2 overviews the existing related works. In the third section we describe the key ideas of the hybrid user-driven XML-to-relational mapping strategy. Section 4 deals with the problem of correction of the candidate set of annotations proposed by the previously described system and Section 5 analyzes and discusses the key issues related to query evaluation of the resulting relational schema. Section 6 describes the architecture of the system and, finally, Section 7 provides conclusions.

2 Related Work

To our knowledge there are just two representatives of user-driven mapping strategies – mapping definition framework *ShreX* [6] and system *XCacheDB* [3]. As for the annotations both support inlining and outlining of a schema fragment, mapping a fragment to a BLOB column, renaming target tables or columns, and redefining column data types. The former approach furthermore supports the *Edge mapping* [7] strategy and enables to specify the required capturing of the structure of the whole schema (using keys and foreign keys, Interval encoding, or Dewey decimal classification). The latter approach allows a certain degree of redundancy enabling to store the data into both set of tables and a BLOB column. In both the cases the mapping for not annotated parts is fixed and the annotations are applied just directly on the annotated schema fragments.

From the point of view of checking correctness of the resulting mapping strategy and query evaluation paper [6] which introduces system ShreX also proposes definitions of a *correct* and *lossless* mapping. In the former case it means that the mapping produces a valid relational schema in terms of distinct table names, distinct column names within a table, distinct CLOB names, and existence of at least one key in each table. In the latter case lossless mapping is a mapping which is correct and maps each element and attribute of the schema and the sibling order of elements. (Surprisingly, it does not consider XML IDs and IDREFs.) The system is able to check the correctness and losslessness of the annotations and complete possible incompleteness of mapping specifications using default mapping rules. As for the query evaluation ShreX does not support redundancy and thus the choice of the most efficient storage strategy for a particular query is pointless. The interface between storage strategies is solved using a mapping API and a mapping repository which contains information about how each element and attribute is stored, which mapping is used to capture the document structure, and which tables are available in the relational schema. Hence the system is able to get information about storage strategy for any part of the schema and thus both shred the documents and evaluate queries.

System XCacheDB supports only a single strategy for shredding XML data into tables which can be modified by inlining / outlining of a fragment, or storing a fragment to a BLOB column. Hence, the only incorrect combination is concurrent inlining and outlining that can be detected easily. The information about the structure of the current

schema is again stored into the database. Contrary to ShreX, the XCacheDB system allows a kind of redundant annotation intersection enabling to store a schema fragment to a BLOB column and, at the same time, to shred it into a set of tables which needs to be treated in a special way. The proposed enhancing of a classical query evaluator is quite simple but working. It always chooses the query plan with minimal number of joins.

As it is obvious, in both the cases the set of possible situations is somehow simplified. In both the systems the annotations are just directly applied on the annotated fragments without any additional exploitation. In the former case the set of annotation intersections is restricted, whereas in the latter case the set of mapping strategies can be characterized as a set of modifications of a single mapping strategy. But in our case we consider more types of intersections, more complex combinations of mapping strategies, and thus new related problems.

3 Hybrid User-Driven XML-to-Relational Mapping

A general idea of fixed schema-driven XML-to-relational mapping methods is to *map* the given XML schema S into a set of relations $R = \{r_1, r_2, ..., r_n\}$ using a mapping strategy s_{rel}. An extreme case is when S is mapped into a single relation resulting in many null values. Other extreme occurs when for each element $e \in S$ a single relation is created resulting in numerous join operations. In user-driven strategies the mapping is influenced by user-defined *annotations* which specify how a particular user wants to store selected schema fragments $F = \{f_1, f_2, ..., f_m\}$. The user usually provides S (i.e. selected fragments) with *annotating attributes* from the predefined set of attributes Ω_A, each of which represents a particular fixed mapping strategy, resulting in an *annotated schema* S'. A classical user-driven strategy then consists of the following steps:

1. S is annotated using Ω_A resulting in S'.
2. Annotated fragments from F are mapped to relations according to appropriate mapping methods.
3. Not annotated fragments of S are mapped to relations using a default fixed mapping strategy s_{def}.

Our method enhances a classical user-driven strategy combining it with the idea of adaptive approaches. We simply add the following steps between the step 1 and 2:

a. For $\forall\ f \in F$ we identify a set $F_f = \{f' \in S \backslash \{f\} : sim(f, f') > T_{sim}\}$, where $sim(f, f')$ expresses the similarity of fragments f and f' and T_{sim} denotes the required minimum similarity threshold.
b. For $\forall\ f \in F$ all fragments in F_f are annotated with annotating attributes of f and added to F.
c. $S \backslash F$ is annotated using an adaptive strategy and the newly annotated fragments are added to F.

The whole mapping process is schematically depicted in Figure 1 where the given schema S with $F = \{f, g\}$ is mapped to a database schema R. If the proposed enhancing, i.e. steps 1.a – 1.c, are included, the system gradually identifies and adds into F new

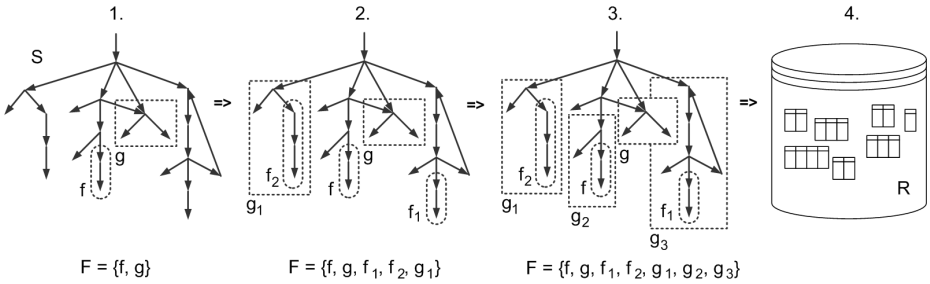

Fig. 1. Schema of the mapping process

annotated fragments f_1, f_2, g_1, g_2, and g_3 which are mapped using user-required mapping strategies. If the enhancing is not included (i.e. in case of a classical user-driven strategy), only fragments f and g are annotated using user-required strategies and the rest of the schema using s_{def}. Thus, the key advantages of the proposed enhancing are the following two:

1. The user is not forced to annotate all schema fragments that have to be stored alternatively, but only those with different structure.
2. The system can reveal structural similarities which are not evident "at first glance" and which could remain hidden to the user.

3.1 Exploitation of Annotations in More Detail

Let us view the given XML schema S as a directed graph $G_S = (V_S, E_S)$ whose nodes correspond to elements, attributes, and operators and edges represent relationships among them. An *annotation* is a function $\alpha : V_S \rightarrow \mathbb{P}(\Omega_A)$ and a node $n \in V_S$ is *annotated* if $\alpha(n) \neq \emptyset$. Each annotated node n uniquely determines an *annotated fragment* f, i.e. a subgraph of G_S consisting of n, all descendants of n, and corresponding edges. As each annotation $\alpha(n)$ determines the mapping strategy for the whole annotated fragment f, we assume that for each $n' \in f : \alpha(n) \subseteq \alpha(n')$, i.e. that the $\alpha(n)$ is "distributed" to all subfragments of f. Hence, $f \sqsubset f'$ denotes f being a proper subfragment of f' and $f \sqsubseteq f'$ denotes $f \sqsubset f' \vee f = f'$.

The main idea of the first enhancing of user-driven techniques remains the same regardless the chosen similarity measure sim, the threshold T_{sim}, or the search algorithm. The choice of sim and T_{sim} influences the precision of the system, whereas the algorithm influences the efficiency of finding the required fragments. We deal with the two issues, and especially avoiding the exhaustive search and tuning of the similarity measure, in [13, 14] in detail and propose a search heuristics called *basic annotation strategy (BAS)* which is able to skip processing of schema fragments which are unlikely to be enough similar.

Conversely, at first glance the user-driven techniques have nothing in common with the adaptive ones. But under a closer investigation we can see that the user-given annotations provide a similar information – they "say" how particular schema fragments

should be stored to enable efficient data querying and processing. Thus we can reuse the user-given information. For this purpose we define an operation *contraction* which enables to omit those schema fragments where we already know the storage strategy and focus on the remaining ones.

Definition 1. *A* contraction *of a schema graph G_S with annotated fragment set F is an operation which replaces each fragment $f \in F$, s.t. $\nexists f' \in F : f \sqsubset f'$, with a single auxiliary node called a* contracted node*. The resulting graph is called a* contracted graph *G_S^{con}.*

The basic idea of the adaptive strategy is as follows: Having a contracted graph G_S^{con} we repeat the BAS algorithm and operation contraction until there is no fragment to annotate. The BAS algorithm is just slightly modified:

- It searches for schema fragments which are not involved in the schema, i.e. it searches among all nodes of the given graph and returns the (eventually empty) set of identified fragments.
- For similarity evaluation we do not take into account contracted nodes.
- The annotations of contracted nodes are always overriding in relation to the newly defined ones.

We denote this modification of BAS as a *contraction-aware annotation strategy (CAS)*. The resulting annotating strategy is called *global annotation strategy (GAS)*.

Hence at this stage we have an XML schema S and a set of schema annotations F consisting of two subsets:

- F_{orig}, i.e. annotations provided by a user and
- F_{adapt}, i.e. annotations denoted by GAS algorithm.

(Note that the F_{adapt} can be empty representing the case of a classical user-driven strategy.) The annotations from set F_{adapt} are considered as possible candidates for annotating, but not all of them should be included in the final storage strategy. Firstly, not all the candidate combinations can be applied on a schema fragment at the same time. And secondly, not all the candidate annotations have to be required by the user. A user can specify final schema fragments, i.e. fragments which should not be influenced by the GAS algorithm, but despite this feature the system can still propose candidates unappropriate for particular application. Thus the natural following step is correction of the set F_{adapt}.

Whenever the set of annotations is corrected, i.e. the final user-approved storage strategy is determined, there remains the open problem of query evaluation. Firstly, the system must cope with the interface between mapping methods, i.e. to be able to process parts of a single query using different storage strategies. And considering the redundancy, it must efficiently determine which of the available storage strategies should be used for evaluation of a particular query.

4 Correction of Candidate Set

The first step related to efficient query evaluation is correction of the candidate set of annotations F_{adapt}. Apart from checking correctness and completeness [6] of schema annotations, we can distinguish three cases corresponding to the following three steps:

1. The system removes cases which are forbidden or meaningless.
2. The system identifies cases where the user can choose from several possibilities.
3. The system accepts further user-specified corrections of proposed annotations which do not correspond to intended future usage.

In the first two cases the system must be able to correctly identify the situations, the last case is rather the question of user-friendly interface.

4.1 Missed Annotation Candidates

We can also identify situations when the system could automatically add more annotation candidates. We discuss them in the following examples, where $f_A, f'_A, f_B, f'_B, f'_{A,B}$ denote schema fragments f and f' annotated using storage strategies A, B, or both.

Situation A – Unidentified Annotated Subfragment Let us consider the situation depicted in Figure 2 where schema S is provided with a set of annotated fragments $F_{orig} = \{f_A, f_B\}$, where $f_B \sqsubset f_A$, whereas the GAS algorithm identified a set of fragments $F_{adapt} = \{f'_A\}$ (as depicted by schema S'). The question is whether it is necessary to add also fragment f'_B, where $f'_B \sqsubset f'_A$ (as depicted by schema S'').

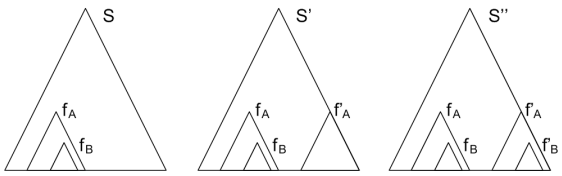

Fig. 2. Unidentified annotated subfragment

If we analyze the situation, the answer is obvious. If $sim(f_B, f'_B) > T_{sim}$, the algorithm would add f'_B to F_{adapt} too. Thus if f'_B is not annotated, its similarity is not high enough and thus it should not be added to F_{adapt}.

Situation B – Unidentified Annotated Superfragment The second situation is depicted in Figure 3. Schema S is again provided with a set of annotated fragments $F_{orig} = \{f_A, f_B\}$, $f_B \sqsubset f_A$, but the GAS algorithm identified a set of fragments $F_{adapt} = \{f'_B\}$ (as depicted by schema S'). In this case we do not discuss whether to add also fragment f'_A, where $f'_B \sqsubset f'_A$, because we can apply the same reason as in case of Situation A. The question is whether f' should be annotated using both A and B, i.e. $F_{adapt} = \{f'_{A,B}\}$ (as depicted by schema S'').

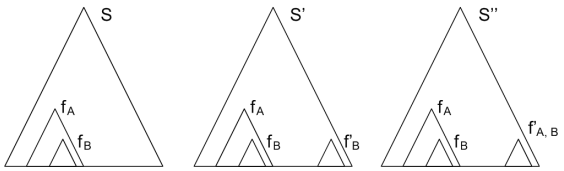

Fig. 3. Unidentified annotated superfragment

In this case we cannot simply state which of the possibilities should be chosen, because for both we can find good reasons. Thus this is the case for user interaction and/or a default setting.

4.2 Sample Set of Annotations

For demonstration of further open issues related to correction of candidate set of annotations as well as query evaluation, we have chosen particular types of fixed mapping methods which represent typical and verified storage strategies. The whole set of supported annotating attributes, their values, and corresponding mapping strategies is listed in Table 1.

Attribute	**Value**	**Function**
`INOUT`	`inline`, `outline`	The fragment is inlined or outlined to/from parent table.
`GENERIC`	`edge`, `attribute`, `universal`	The fragment is stored using schema-oblivious Edge, Attribute, or Universal strategy [7].
`SCHEMA`	`basic`, `shared`, `hybrid`	The fragment is stored using schema-driven Basic, Shared, or Hybrid strategy [17].
`TOCLOB`	`true`	The fragment is stored to a CLOB column.
`INTERVAL`	`true`	The fragment is indexed using the Interval encoding [19].

Table 1. Supported schema annotations

Similarly to the existing works we support inlining and outlining of a schema fragment to/from parent table or its storing to a single CLOB column. As for the "classical" mapping methods we support a set of schema-oblivious storage strategies – the Edge, Attribute, and Universal mapping [7] – and a set of schema-driven storage strategies – the Basic, Shared, and Hybrid algorithm [17]. Last but not least, we support a kind of numbering schema which speeds up processing of particular queries – the Interval encoding [19].

Naturally, the set of supported mapping strategies could be much wider and involve more representatives of the existing reasonable mapping strategies. But our aim was to

choose well-known representatives of particular approaches which enable to illustrate various situations.

4.3 Annotation Intersection

As the annotated fragments can intersect as well as a single fragment can be annotated using multiple storage strategies, we have defined three types annotation intersection assuming that the system is provided with both the set of annotations and types of their mutual intersection.

Definition 2. *Intersecting annotations are* redundant *if the corresponding mapping strategies are applied on the common schema fragment separately.*

Definition 3. *Intersecting annotations are* overriding *if only one of the corresponding mapping strategies is applied on the common schema fragment.*

Definition 4. *Intersecting annotations are* influencing *if the corresponding mapping strategies are combined resulting in one composite storage strategy applied on the common schema fragment.*

Redundant annotations can be exploited, e.g., when a user wants to store XHTML fragments both in a single CLOB column (for fast retrieval of the whole fragment) and, at the same time, into a set of tables (to enable querying particular items). An example of overriding annotations can occur when a user specifies a general mapping strategy for the whole schema S and then annotates fragments which should be stored alternatively. Naturally, in this case the strategy which is applied on the common schema fragment is always the one specified for its root element. The last mentioned type of annotations can be used in a situation when a user specifies, e.g., the 4NF decomposition for a particular schema fragment and, at the same time, an additional numbering schema which speeds up processing of particular types of queries. In this case the numbering schema is regarded as a supplemental index over the data stored in relations of 4NF decomposition, i.e. the data are not stored redundantly as in the first case.

Apart from the allowed types of intersection, there are also cases when a particular combination of annotations is senseless. For instance consider the situation depicted in Figure 4, where schema S contains two annotated fragments f_{TOCLOB} and f_{SCHEMA}, whereas $f_{SCHEMA} \sqsubset f_{TOCLOB}$ and the `TOCLOB` annotation overrides all the previously specified strategies. As it is obvious, such combination of annotations is useless, since there is no point in shredding a part of a schema fragment stored in a CLOB column into a set of tables. The situation also depicts that the order of composition of annotations is important. Obviously, the opposite order, i.e. if $f_{TOCLOB} \sqsubset f_{SCHEMA}$, is reasonable and can result in both redundant and overriding intersection.

Another question is for which subsets of the supported schema annotations the intersection type should be specified. Consider the situation depicted in Figure 5, where schema S contains two annotated fragments f_A and $f_{B,C}$, whereas $f_{B,C} \sqsubseteq f_A$. At this situation we naturally need to know the result of intersection of all the three annotations together. And with the above finding, it can also differ depending on the order of their

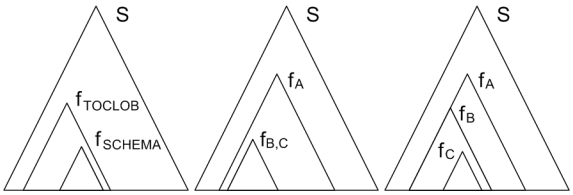

Fig. 4. Forbidden intersection annotations

Fig. 5. Intersection of of multiple annotations I.

Fig. 6. Intersection of multiple annotations II.

mutual composition. Therefore, we should theoretically specify the result of intersection of all possible subsets of Ω_A and all respective orders. But, in fact, as there are pairs of annotations or their orders which are forbidden, the number of such specifications significantly decreases. And, in addition, such specifications need to be stated for the whole system only once, not for each mapping task separately.

We demonstrate both the situations for the sample set of annotations in the following sections.

Intersections of Pairs of Annotations The specification of allowed types of intersections for pairs of annotations is relatively simple. Four our sample set of annotations they are listed in Tables 2 and 3, where $\emptyset$ represents no effect of intersection, $\times$ represents forbidden intersection, and ✓represents allowed intersection. Each field of the table represents the result of applying the mapping strategy in the row on the mapping strategy in the column.

	INOUT	GENERIC	SCHEMA	TOCLOB	INTERVAL
INOUT	$\emptyset$	×	×	×	×
GENERIC	×	$\emptyset$	✓	×	×
SCHEMA	×	✓	$\emptyset$	×	×
TOCLOB	×	✓	✓	$\emptyset$	×
INTERVAL	×	×	×	×	$\emptyset$

Table 2. Overriding and redundant intersection

	INOUT	GENERIC	SCHEMA	TOCLOB	INTERVAL
INOUT	$\emptyset$	✓	✓	×	×
GENERIC	×	$\emptyset$	×	×	×
SCHEMA	×	×	$\emptyset$	×	×
TOCLOB	×	×	×	$\emptyset$	×
INTERVAL	×	✓	✓	×	$\emptyset$

Table 3. Influencing intersection

As can be seen from the tables, the amount of reasonable and thus allowed combinations of mapping methods is relatively low in comparison with the theoretically possible options. Firstly, we assume that the combination of two identical annotations results in an empty operation, i.e. it has no effect as depicted at diagonals. And, in addition, in our case the results for overriding and redundant annotations are identical and thus listed in

one common table. Another two obvious cases are `INOUT` and `INTERVAL` annotations which are supposed to influence any other method. Therefore, they can occur only in case of influencing intersections and only in one particular order of composition. Naturally, they can be applied only on methods which shred a schema fragment into a set of tables. Considering the `TOCLOB` annotation, it can be also applied on a method which shreds a schema fragment into one or more tables, since the whole fragment is then viewed as a single attribute. As for the composition orders, as depicted in Figure 4, the `TOCLOB` annotation can be applied only on a mapping strategy, but not vice versa. Note that from another point of view the `TOCLOB` annotation can be regarded as influencing, rather than overriding. Similarly to the `INOUT` annotation it influences the given storage strategy treating a schema fragment as a single attribute. But it is only a question of semantics.

Last but not least, note that if the order of composition of mapping strategies is not obvious (e.g. if a single fragment is annotated using multiple strategies), we take as the result union of both the possible orders.

Intersections of Multiple Annotations As for the intersection of multiple annotations together we need to distinguish several cases. We demonstrate them using the example depicted in Figure 6, where schema S contains three annotated fragments f_A, f_B, and f_C, whereas $f_C \sqsubset f_B \sqsubset f_A$. The question is what will be the result of annotation for their intersection.

As for the first situation let us assume that the intersection of f_A and f_B is overriding. Then the situation transforms to the case of intersection of two methods (i.e. f_B and f_C) as defined in the previous section. The second situation occurs when the intersection of f_A and f_B is redundant, meaning that the common schema fragment is stored using both the strategies A and B. Then the situation of intersection with strategy C transforms to union of separate intersections of two pairs of methods (i.e. f_A, f_C and f_B, f_C). Or, also in this case the user can specify on which of the two strategies A and B should strategy C be applied. The third situation involves the last case when the intersection of f_A and f_B is influencing. In this case the resulting intersection must be defined for all the possible cases. For our sample set of annotations, the solution again corresponds to union of separate intersections of two pairs of methods (whereas one of them always results in forbidden intersection). But, in general, the result can lead to a brand new one method and therefore a new set of rules for intersecting.

Note that also in this case if the order of composition of mapping strategies is not evident, we again take as the result union of all the possible orders. And similar discussion can be done for larger sets of annotations as well.

4.4 Examples of Schema Annotations

The annotations supported by system *UserMap* (see Table 1) are expressed using attributes from a name space called *usermap* and can be associated with element definitions of XSDs. Similarly to the paper [6] they could be associated also with attributes, attribute groups, element groups, etc. But we restrain to elements for simplicity.

Example 1 – Exploitation of CLOBs The exploitation of CLOBs enables to speed up reconstruction of schema fragments. It is useful especially in cases when the user knows that particular schema fragment is rather document-oriented and will be retrieved as a whole. Consider the example in Figure 7 where a fragment of XSD of the *Internet Movie Database (IMDb)*[2] contains information about actors. Each actor has name consisting of the first name and the last name and filmography, i.e. a list of movies each consisting of a title and a year. For better lucidity the element names are underlined and schema annotations are in boldface.

```
<xs:element name="Actor"
            usermap:SCHEMA="hybrid">
 <xs:complexType>
  <xs:sequence>

   <xs:element name="Name"
               usermap:TOCLOB="true">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="FirstName"
                  type="xs:string"/>
      <xs:element name="LastName"
                  type="xs:string"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>

   ...
```

```
<xs:element name="Filmography">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="Movie"
               maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="Title"
                  type="xs:string"
                  usermap:INOUT="outline"/>
      <xs:element name="Year"
                  type="xs:int"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>

 </xs:sequence>
</xs:complexType>
</xs:element>
```

Fig. 7. Exploitation of CLOBs – XML schema

According to the annotations, the whole fragment, i.e. element `Actor`, should be stored using the Hybrid algorithm, as specified by the `SCHEMA="hybrid"` annotation, its subelement `Name` should be stored into a CLOB column (`TOCLOB="true"`), and subelement `Title` should be outlined to a separate table (`INOUT="outline"`).

As depicted in Tables 2 and 3 the intersection of `SCHEMA` and `TOCLOB` annotations can be either overriding or redundant. Firstly, let us consider the overriding case. The resulting relational schema is depicted in Figure 8 (b).

As it is expectable the resulting relational schema corresponds to the result of classical Hybrid algorithm (depicted in Figure 8 (a)) except for two cases. Firstly, the element `Name` is treated as an element having a text content and stored into a single CLOB column. And secondly, the element `Title` is stored into a separate table, although the classical Hybrid algorithm would inline it to table `Movie` too.

Example 2 – Redundant Mapping Strategies Let us again consider the XSD example in Figure 7, but this time assuming that the intersection of `SCHEMA` and `TOCLOB` annotations is redundant. The resulting relational schema is depicted in Figure 8 (c). In this case the element `Name` is stored twice, using both the strategies, i.e. into two

[2] `http://www.imdb.com/`

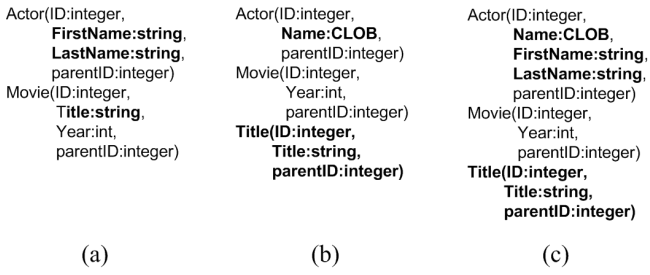

Fig. 8. Exploitation of CLOBs – relational schemes

columns corresponding to classical Hybrid algorithm and, at the same time, into one CLOB column.

Note that a similar type of storage strategy can be defined also using the system XCacheDB [3] which enables both redundant and overriding mapping to a CLOB column. The main difference is that the system supports only one particular type of shredding into a set of tables (which can be modified by inlining and outlining).

Example 3 – Influencing Mapping Strategies Last but not least, consider an example of influencing intersection of mapping strategies. In example depicted in Figure 9 we use the same fragment of IMDb XSD, but with different annotations. This time the element `Actor` should be stored using schema-oblivious Edge mapping (`GENERIC="edge"`), whereas queries over its subelement `Filmography` are enhanced using the Interval encoding (`INTERVAL="true"`).

```
<xs:element name="Actor"
            usermap:GENERIC="edge">
 <xs:complexType>
  <xs:sequence>

   <xs:element name="Name">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="FirstName"
                  type="xs:string"/>
      <xs:element name="LastName"
                  type="xs:string"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>

   ...
```

```
<xs:element name="Filmography"
            usermap:INTERVAL="true">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="Movie"
               maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="Title"
                  type="xs:string"/>
      <xs:element name="Year"
                  type="xs:int"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>

 </xs:sequence>
</xs:complexType>
</xs:element>
```

Fig. 9. Influencing mapping strategies – XML schema

Figure 10 depicts both the classical Edge mapping (a) and the result of the strategy specified by the annotations (b). In the former case all the edges are stored into a single table `Edge`. In the latter case edges of subelement `Filmography` are stored into a separate table `EdgeFilmography` having additional columns (`intervalStart` and `intervalEnd`) for storing values of Interval encoding. Note that the influencing enables to skip the column `order`, since the Interval encoding involves total ordering.

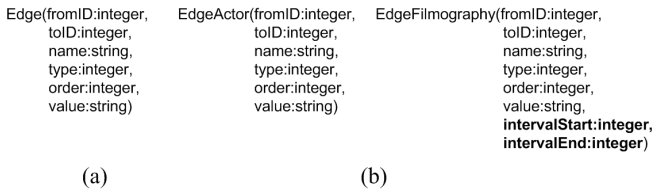

Fig. 10. Influencing mapping strategies – relational schemes

The closest example can be found in case of system ShreX [6] which supports annotation `structurescheme` specifying how the structure of the whole schema is captured, i.e. using keys and foreign keys, Interval encoding, or Dewey decimal classification. This feature can be considered as a special type of influencing mapping, though its purpose is slightly different.

Auxiliary Columns As the above described examples of relational schemes are illustrative, they only represent the characteristics of the mapping strategies. In fact, when the schemes are generated automatically, we cannot use directly, e.g., the element and attribute names for table and column names of the schema. In addition, each of the data tables contains also auxiliary information. The most important ones are a unique ID of the respective XML document (`docID`) and a unique ID of each record (`recordID`). The former one enables to distinguish where the data originate from, but it is in the following examples omitted for simplicity. The latter one is exploited in two cases for similar reason. Since each of the storage strategies can have structurally different tables we use this uniform ID to enable their joining, as well as for uniform resulting value of the SQL query. We illustrate its usage in the following text.

5 Query Evaluation

The basic idea of XML query evaluation in (O)RDBMS-based storage strategies is relatively simple. An XML query posed over the data stored in the database is *translated* to a set of SQL queries (which is usually a singleton) and the resulting set of tuples is transformed to an XML document. We speak about *reconstruction* of XML fragments. As it is analyzed in detail in [12], the amount of works which focus on efficient XML-to-SQL query translation is enormous (the analysis considers about 40 papers), can be classified according to various purposes, and focus on various aspects of the problem.

Two key metrics for query evaluation are *functionality*, i.e. the variety of types of supported XML queries, and *performance*, i.e. the efficiency of evaluation of the query.

The main idea of our proposed system is to enable a user to create a hybrid XML-to-relational storage strategy, i.e. a relational schema which consists of multiple subschemes having different structure. Assuming that each of the subschemes can (and usually does) require a different XML-to-SQL query translation algorithm, we focus on the problem of interface between the storage strategies, i.e. the problem of evaluation of parts of a single XML query using various storage strategies. Second issue is related to redundancy that can occur in case of intersection of annotations, in particular redundant and influencing ones, where a single schema fragment is stored using two or more strategies. Therefore a natural assumption is that the system can estimate the cost of query evaluation using all possible strategies and choose the optimal one. For this purpose we again exploit our sample set of annotations (see Table 1) and using simple examples we illustrate the related issues.

5.1 Interface between Schema Annotations

Let us consider the three types of annotation intersections separately and discuss their difference from the point of view of query evaluation. In case of overriding intersection of strategies A and B, the interface must allow joining (one or more) tables of strategy A with (one or more) tables of strategy B. In case of redundant intersection of strategies A and B the situation is similar but, in addition, the interface must enable to use any of the strategies. The influencing annotation intersection requires a brief discussion: Under a closer investigation we can see that there are two types of annotations, i.e. mapping strategies, that can influence another one. Each of them is represented by one of the annotations of the sample set. The `INOUT` annotation enables to modify the structure of the resulting storage strategy, i.e. the amount of tables and/or columns. Therefore it is processed before the schema is mapped to relations and having the information about the structure it does not need to be taken into account later. We call these annotations *early binding*. (Note that the `TOBLOB` annotation can be viewed as a kind of early binding annotation as well.) On the other hand, the `INTERVAL` annotation enhances a given storage strategy with additional information which is exploited as late as a query is evaluated. We call these annotations *late binding*.

Structural Tables Since the resulting storage strategy is not fixed and can be influenced by many factors, we need to store the information about the structure of each mapped XML schema. Similarly to papers [6] [3] we store the information into supplemental tables. This simple idea enables to parse the schema annotations only once and not every time a document is shredded into relations or a query is posed, as well as it enables to make the processing to be independent on the way the mapping is specified.

In particular for the sample set of annotation strategies we use the following tables:

`xmlSchemaTable(uri,schemaID)`

contains information about XML schemes for which there exists a mapping in the repository, i.e. URI of the XML schema (`uri`) and ID of the schema (`schemaID`) unique within all the stored schemes.

`xmlDocTable(schemaID,url,docID)`

that contains information about XML documents stored in a database schema created for particular XML schemes, i.e. URL of the XML document (`url`) and ID of the document (`docID`) unique within all the documents valid against a schema (`schemaID`) stored in the repository.

`xmlAttrTable(schemaID,attrID,mapID,xmlName,paramID)`

contains information about storage strategies for particular attributes in the schema, i.e. ID of the attribute (`attrID`) unique within a schema (`schemaID`), ID of the storage strategy for the attribute (`mapID`), name of the attribute (`xmlName`), and ID of additional parameters (`paramID`) corresponding to the respective storage strategy, or null if there is no such information needed.

Note that the attributes could be treated similarly to elements with text content, i.e. different storage strategies could be defined for an element and for its attributes. But for simplicity the experimental implementation assumes that the storage strategy for attributes is always denoted by the storage strategy for corresponding element.

`xmlElemTable(schemaID,elemID,mapID,xmlName,paramID)`

contains information about storage strategies for particular elements in the schema, i.e. ID of the element (`elemID`) unique within a schema (`schemaID`), ID of the storage strategy for the element (`mapID`), name of the element (`xmlName`), and ID of additional parameters (`paramID`) corresponding to the respective storage strategy, or null if there is no such information needed.

`xmlElemAttrTable(schemaID,elemID,attrID)`

contains information about relationship between an element (`elemID`) and its attribute (`attrID`) in a schema (`schemaID`).

Note that this information could be stored into the `xmlAttrTable` as well, but assuming that XSDs enable to specify global attributes and attribute groups which can be referenced from several elements, the relationship can be in general M:N.

`xmlElemElemTable(schemaID,elemID,subElemID)`

contains information about M:N relationship between an element (`elemID`) and its subelements (`subElemID`) in a schema (`schemaID`).

Last but not least, there remains the structure of tables containing the additional parameters necessary for particular mapping strategies. Since each of the strategies can require different information we have outlined the parameters from tables describing elements and attributes and we store it separately, though the relationship between the tables is 1:1.

If we consider the sample set of annotations, we obviously do not need any other information for the `TOCLOB` and `INOUT` annotations themselves, since both of them are early binding annotations which influence the amount of tables of another strategy.

As for the GENERIC annotation, in case of pure Edge and Universal mapping the target schema is fixed regardless the source data. But in case of Attribute mapping which requires a separate table for each distinct element or attribute name in the schema, or when inlining or outlining is applied on any of the three cases, we need to know the name of table where the element / attribute is stored.

In case of SCHEMA annotation the situation is particularly complicated since the resulting relational schema is given by the structure of the source XML schema. And, in addition, TOCLOB or INOUT annotations can be applied on any of the three algorithms and influence the structure as well. Therefore, for each element and attribute we need to store the information where and how it is stored. For this purpose we use table

```
xmlBSHTable(mapID,mapType,tableName,columnName)
```

that contains mapping type (mapType) of the element / attribute, where n denotes numeric data type, s denotes string data type, and e denotes element content (of an element), name of table for storing the element / attribute (tableName), and name of column for storing the element / attribute (columnName).

Last but not least, there remains the late binding INTERVAL annotation. As we have mentioned it is considered as an additional index that can speed up processing of particular approaches. Hence for element and attributes enhanced with this index we need to know the names of columns where the corresponding values are stored.

An example of content of structural tables for the three relational schemes (a), (b), (c) in Figure 8 is depicted in Figure 11. Since there are no attributes in the sample schema, we only deal with structural tables related to elements. The processing of attributes would be very similar. (We do not use particular IDs of the 1:1 relationship for simplicity; the related tables are mentioned in separate rows.)

As it is obvious, the tables contain all the information necessary for both document shredding and query evaluation. For each element of the sample schema we know its storage strategy and related details, i.e. the way it is stored and the target table and/or column. Note the way we treat redundant intersection of annotations. In this case we create two instances of the redundant fragment, each having its own ID and type of storage strategy.

5.2 Document Shredding

Having the information from structural tables, the process of document shredding is relatively simple. For instance, if we consider the relational schemes in Figure 8, for storing an element we generally need two information – ID of its parent element and constructors of its subelements that are mapped to the same table. Therefore the process can be described as a recursive creation of constructor for the current element from constructors of its subelements (and attributes). During the top-down progress the ID of the current element e is propagated to be stored in parentID columns of its subelements $e_1, e_2, ..., e_k$ mapped to tables. During the bottom-up return from the recursion its subelements $e_{k+1}, e_{k+2}, ..., e_n$ mapped to simple types hand over their constructors and column names. Then element e creates its own constructor.

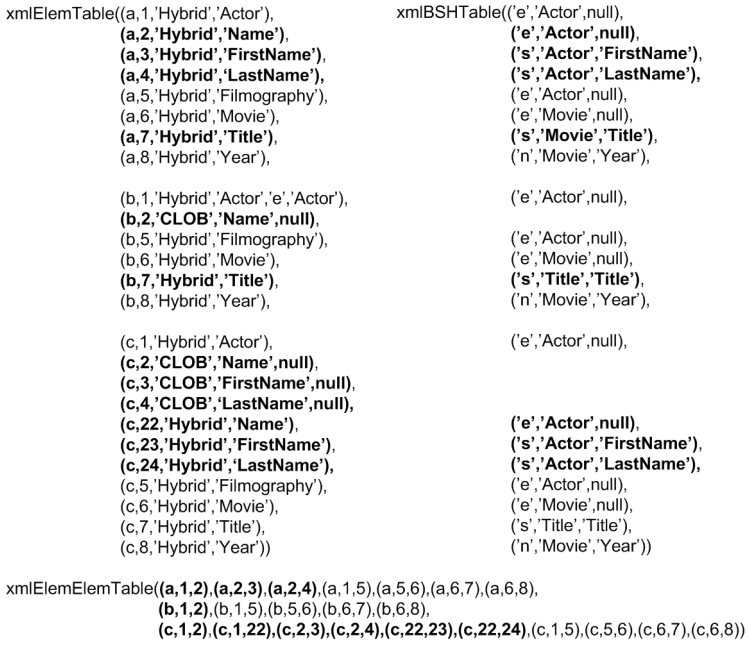

Fig. 11. Exploitation of CLOBs – structural tables

For instance, consider the element `Movie` in relational schema (b) in Figure 8. Being provided with parent ID, the shredding process first identifies that the element `Movie` has element content. Therefore it generates its ID and recursively processes its subelements. The subelement `Title` stores itself to own table, but the subelement `Year` returns the constructor of the integer value and name of the corresponding column. Hence the constructor of the `Movie` element consists of three items – `ID`, `Year`, and `parentID`.

5.3 Query Translation

Similarly to the process of document shredding, with the information from structural tables the query evaluation is quite straightforward. Using the following examples we illustrate the related key ideas. Assuming that wild-card queries are usually converted into union of several simple path queries with predicates, one for each satisfying wild-card substitution [11], we consider only examples of simple-path queries.

Example 1 – Early Binding Annotations Let us consider the example of query

```
/Actor/Filmography/Movie[Year=2007]/Title
```

and relational schema depicted in Figure 8 (b). Firstly, the table where element `Actor` is stored, is added to the `FROM` clause. Seeing that element `Filmography` is mapped

to the same table, this step has no effect. Then, since the element `Movie` is mapped to its own table, the table `Movie` is added to the `WHERE` clause and joined using IDs. The processing of predicate `Year=2007` first requires analysis of the query on the left-hand side. Since element `Year` is mapped to column of the `Movie` table, it does not cause new joins, but only adding the condition to the `WHERE` clause. The `Title` element is again detected to be stored in a separate table resulting in another join. Finally, the `SELECT` clause is provided with the reference to its record ID, resulting in the following query:

```
SELECT t.recordID
FROM Actor a, Movie m, Title t
WHERE m.parentID = a.ID AND
      m.Year = 2000 AND
      t.parentID = m.ID
```

(Note that using the knowledge of semantics of the XML schema, i.e. on the basis of analysis of structural tables, this query could be further optimized, in particular, table `Actor` can be omitted.)

From the point of view of evaluation of a single query using several storage strategies, the described example of query evaluation copes with an early binding influencing annotation `INOUT` applied on annotation `SCHEMA`. Thus the translation approach is similar to classical Hybrid algorithm, except for slightly different structure which is captured using the structural tables. A similar effect would occur in case of overriding annotation `TOCLOB`.

Example 2 – Structurally Different Tables Now, let us consider the same query but a situation, where element `Actor` is associated with annotation `SCHEMA="hybrid"` and element `Movie` with annotation `GENERIC="edge"` as depicted in Figure 12, whereas the intersection is overriding.

```
<xs:element name="Actor"
            usermap:SCHEMA="hybrid">
 <xs:complexType>
  <xs:sequence>

   <xs:element name="Name">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="FirstName"
                  type="xs:string"/>
      <xs:element name="LastName"
                  type="xs:string"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>

   ...
```

```
<xs:element name="Filmography">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="Movie"
               maxOccurs="unbounded"
               usermap:GENERIC="edge">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="Title"
                  type="xs:string"/>
      <xs:element name="Year"
                  type="xs:int"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>

 </xs:sequence>
</xs:complexType>
</xs:element>
```

Fig. 12. Join of structurally different tables – XML schema

Hence, we need to join structurally different tables whose IDs have quite different meaning. As for the first part of the query `/Actor/Filmography` the translation remains the same as in the previous example. As for the second part of the query `/Movie[Year=2007]/Title`, we need to join the `Edge` table three times, i.e. for `Movie` and `Title` elements and for the predicate `Year=2007` resulting in a query:

```
SELECT t.recordID
FROM Edge m, Edge y, Edge t
WHERE m.toID = y.fromId AND
      y.value = 2000 AND
      m.toID = t.fromId
```

Finally, for the purpose of joining the two tables, i.e. `Actor` and `Edge`, we need to specify the interface between them. In our particular case we exploit the auxiliary `recordID` column of both the tables and information from structural table

```
xmlInterTable(anotID,parentID)
```

which contains pairs of parent-child relationships between `recordID` of an annotated element (`anotID`) and `recordID` of its parent element (`parentID`). Then the resulting query translation is as follows:

```
SELECT t.recordID
FROM Actor a, xmlInterTable i,
     Edge m, Edge y, Edge t
WHERE a.recordID = i.parentID AND m.recordID = i.anotID AND
      m.toID = y.fromId AND
      y.value = 2000 AND
      m.toID = t.fromId
```

The example depicts that a join with `xmlInterTable` is added every time the query "passes borders" of two mapping strategies. The obvious exception is the case of early binding influencing annotations. Naturally, more complex mapping strategies could require another information about their mutual interface, but for our particular sample set this information is sufficient.

5.4 Exploitation of Redundancy

The above described algorithm of query evaluation assumes that there is always one possible way it can be performed. Naturally each SQL query can have multiple query plans, each having its cost depending on the order tables are joined, selectivity of `WHERE` conditions, usage of `ORDER BY` clauses, etc. But in this section we deal with the set of distinct mapping strategies that "cover" the query.

Consider again the same sample query and the annotated schema in Figure 12, where element `Actor` is associated with annotation `SCHEMA="hybrid"` and element `Movie` with annotation `GENERIC="edge"`, whereas the intersection is now redundant. Then we have two possibilities how to evaluate the query – either using

purely the Hybrid mapping or using both the Hybrid and Edge mapping and their interface. In the former case the query would require joining of two tables – `Actor` and `Movie` (the classical Hybrid algorithm inlines the element `Title`). The query translation of the later case was discussed above and involves joining of five tables – table `Actor` of the Hybrid mapping, three `Edge` tables from the Edge mapping, and table `xmlInterTable` carrying the interface information between the two strategies. If we use a simple cost metric which considers purely the amount of join operations necessary for query evaluation, the former translation strategy is naturally better choice.

In general, there can exist a plenty of possibilities how to evaluate a query Q. For this purpose we first analyze the structural tables and search for all the possible sequences of strategies using which Q can be evaluated and we build and auxiliary *evaluation graph* G^{eval}. Consider the sample situation in Figure 13.

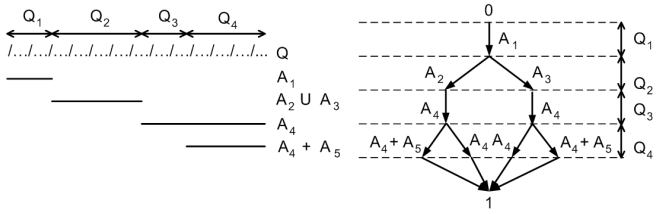

Fig. 13. Example of evaluation graph G^{eval}

The figure schematically depicts that query Q is divided into fourth parts Q_1, Q_2, Q_3, and Q_4, determined by four annotations, i.e. mapping strategies it "traverses". Part Q_1 can be evaluated only using strategy A_1. Part Q_2 can be evaluated either using strategy A_2 or A_3 meaning that the intersection of the two annotations is redundant (denoted by the union sign), but they override annotation A_1. As for the part Q_3, the respective strategy is again only A_4 which overrides the previous two strategies A_2 and A_3. And finally, part Q_4 can be evaluated using both A_4 or influencing intersection of A_4 and A_5 (denoted by the plus sign).

On the right-hand side of the figure is depicted corresponding evaluation graph G^{eval} whose edges correspond to storage strategies and nodes to interfaces among them. The graph also contains two auxiliary nodes 0 and 1 which represent the beginning and end of the query and respective auxiliary edges.

The construction of G^{eval} is relatively simple:

1. The auxiliary node 0 is created.
2. Starting with the set of storage strategies $A = \{A_1, A_2, ..., A_k\}$ for the root element of query Q, respective k outgoing edges of node 0 are created.
3. Each edge and corresponding storage strategy A_i is processed recursively: Traversing the query Q and structural tables we search for each interface of A_i and A_j, s.t. $A_i \neq A_j$.
   (a) In case of redundant intersection of A_i and A_j, for both A_i and A_j new outgoing edges are created.

(b) In case of late binding influencing intersection of A_i and A_j, for both A_i and $A_i + A_j$ new outgoing edges are created.
(c) In case of overriding intersection of A_i and A_j, a new outgoing edge for A_j is created.

4. The auxiliary node 1 and respective edges connecting all leaves of the graph with node 1 are created.

For the purpose of searching for the best evaluation sequence of storage strategies, each edge $e \in G^{eval}$ is assigned its *length* which expresses the cost $cost_{eval}(Q_i, A_j)$ of evaluating of a query part Q_i using a strategy A_j and cost $cost_{inter}(A_{prev}, A_j)$ of the interface between strategy A_{prev} used for evaluation of Q_{i-1} and current strategy A_j.

Definition 5. Length of edge $e = \langle v_x, v_y \rangle$ *of evaluation graph* G^{eval} *is defined as follows:*

$$length(e) = \begin{cases} cost_{eval}(Q_i, A_j) & v_x = 0 \\ cost_{eval}(Q_i, A_j) + & \\ cost_{inter}(A_{prev}, A_j) & v_x \neq 0 \\ 0 & v_y = 1 \end{cases}$$

Now, having a graph G^{eval} and corresponding lengths of its edges, the problem of finding the optimal evaluation sequence of strategies transforms to the shortest path problem, i.e. searching the shortest path from node 0 to 1, which can be solved, e.g., using the classical Dijkstra's algorithm.

Reconstruction of XML Fragments Similarly to query evaluation also in case of reconstruction of resulting XML fragments there can occur multiple ways of retrieval of the relevant data. Consider the situation depicted in Figure 9, where element `Actor` is associated with annotation `GENERIC="edge"` and element `Filmography` with late binding influencing annotation `INTERVAL="true"`, and query

```
/Actor/Filmography/Movie[Year=2007]
```

whose SQL translation returns a set $R = \{r_1, r_2, ..., r_k\}$ of values of `recordID` column of records from table `EdgeFilmography` which fulfill the query. The required XML result is a set of elements `Movie`, each containing subelements `Title` and `Year` with corresponding values. With regard to the specified annotations we have two possibilities how to retrieve corresponding information – the Edge mapping or the Interval encoding.

In the former case we process each `recordID` $r_i \in R$; $i = 1, 2, ..., k$ separately: Firstly, we create a new `Movie` node of DOM tree T_{DOM} of the XML result. Then we select the set of all its subelements (and attributes) from the `EdgeFilmography` table. Subelements having a text content are added to the result, i.e. for each element a corresponding node in T_{DOM} is created. Subelements having element content are processed recursively. In general, for the purpose of the reconstruction we need to perform $O(k \cdot n)$ select queries from the `EdgeFilmography` table, where n is the maximum number of non-leaf nodes of XML fragments rooted at element `Movie`.

In the latter case, i.e. when exploiting the Interval encoding, the retrieval of the relevant information is much easier and faster. The Interval encoding enables to retrieve information of the whole XML fragment at once and totally ordered. Having the set $R = \{r_1, r_2, ..., r_k\}$ we need a single query which contains a single join of two tables `EdgeFilmography`:

```
SELECT *
FROM EdgeFilmography m, EdgeFilmography e
WHERE m.recordID IN (r1, r2, ..., rk) AND
      m.intervalStart <= e.intervalStart AND
      e.IntervalEnd   <= m.intervalEnd
ORDER BY e.intervalStart
```

Thus, also in case of document reconstruction we need to choose the most efficient way of retrieval of the data. For this purpose we can use the same approach as in case of query evaluation. The only difference is, that the costs of the strategies can differ. As for our sample set of annotations the most striking example is the `TOCLOB` storage strategy, where in case of query evaluation it has a high cost assuming that it requires preprocessing of the CLOB content, whereas in case of reconstruction its cost is low.

6 Architecture of System *UserMap*

As depicted in Figure 14 the architecture of experimental system *UserMap* consists of several modules which can be divided according to phases of processing they belong to. The particular modularity is given not only by logical partitioning of the system, but it is also influenced by the needs of experiments and corresponding ability to omit various modules [13].

Phase I. Preparation Being given an annotated XSD schema S the system first checks its validity using the *Xerces Java parser* [16] and builds its DOM tree T_S [1]. Then, for easier processing, the DOM tree is transformed into a *DOM graph* G_S [15], i.e. a graph representation of XSD schema similar to classical DTD graph [17] extended for XML Schema constructs. For the same purpose the *graph builder* extracts the set of annotated fragments F_{orig}.

Phase II. Searching for Annotation Candidates At this phase of the processing the *BAS module* performs the BAS algorithm and then the *GAS module* performs the GAS algorithm being given the schema S and the set of user-specified annotations F_{orig}. The former module identifies the set of annotation candidates F_{BAS}, the latter one the set of annotation candidates F_{GAS}. The system enables to skip any of the approaches, to compare their results, and thus to compare the resulting storage strategies. The resulting set of annotation candidates $F_{adapt} = F_{BAS} \cup F_{GAS}$. Since the *similarity evaluator* is a separate module evaluating similarity of two given schema fragments f and g, it can be easily replaced with any other method.

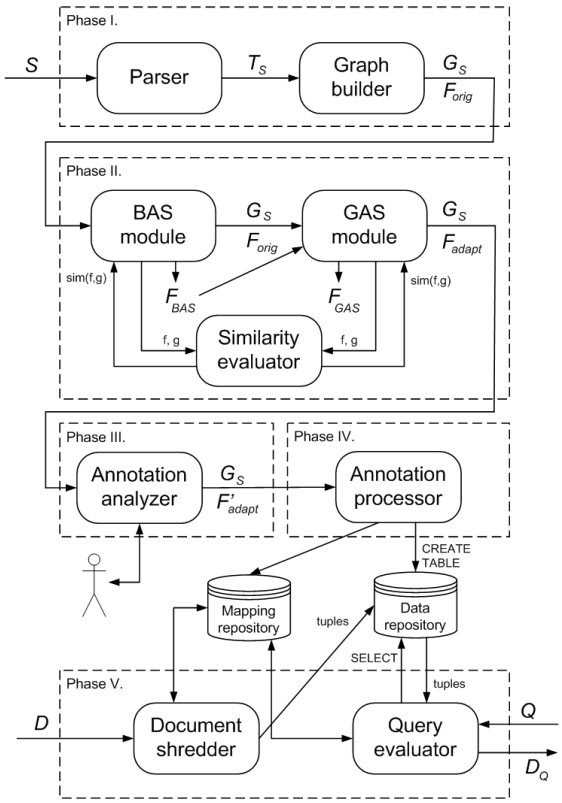

Fig. 14. Architecture of the system

Phase III. Correction of Candidate Set At this phase the candidate set of annotations F_{adapt} needs to be corrected. As described in Section 4 there are two types of correction – automatic and user-specified – resulting in set of corrected annotations F'_{adapt}. In the former case the *annotation analyzer* automatically searches and removes the forbidden annotations, in the latter case a user interaction is required. In the experimental implementation of the system the user interaction is omitted and a default possibility is always applied. But as mentioned in the Conclusion, the very next enhancement of the system will focus on user interaction, user-friendliness, and appropriate GUI. Then also the incorrect situations can be identified and solved using the user interaction as well.

Phase IV. XML-to-Relational Mapping At this phase the *annotation processor* parses the set of corrected annotations and maps the corresponding schema fragments into the *data repository* using the respective approaches. As described in Section 5.1, at the same time the system stores the information about schema mapping into supplemental structural tables in the *mapping repository*.

Phase V. Document Shredding and Query Evaluation At this phase the system is ready for the intended application. It involves two operations:

- shredding a document D valid against XML schema S into corresponding tables and
- evaluation of query Q posed over the schema S which returns document D_Q containing corresponding results.

The *document shredder* reads the XML document using a SAX parser and on the basis of the information from mapping repository generates an SQL script for storing appropriate tuples into the data repository. The *query evaluator* firstly identifies the most efficient evaluation sequence (as described in Section 5.4) and then, also using information from mapping repository, translates the XML query into an SQL query. Similarly, the resulting tuples are then transformed to corresponding XML fragments using the most efficient reconstruction sequence.

7 Conclusion

As can be seen, there are several interesting issues related to the relatively simple idea of hybrid user-driven mapping strategy. We have outlined the key components of the whole system and using simple examples illustrated the related problems. As it is obvious, most of the approaches (such as, e.g., efficient query translation, cost estimation of the queries, etc.) can be significantly optimized, since a detailed research has already been done in these areas.

The very next step of our future work is an elaborate implementation of the proposed system with the emphasis on all the "side" aspects of the proposal including the omitted user-friendly interface which is definitely and important requirement for a system based on user interaction. During the research experimental and prototype implementations of the most important modules of the system were created for the purpose of evaluation or verification of important algorithms. But at this stage we intend to implement the system as a complete and robust application. And this plan is also closely related to the mentioned open issue of optimization of the query evaluator. Similarly to paper [8] we intend to exploit a cost evaluator which is able to dynamically adapt the statistics to the newly coming data and, at the same time, to conform to the assumption of multiple storage strategies used within a single schema.

Acknowledgement

This work was supported by the National Programme of Research (Information Society Project number 1ET100300419).

References

1. *Document Object Model (DOM).* W3C, 2005.

2. S. Amer-Yahia. *Storage Techniques and Mapping Schemas for XML*. Technical Report TD-5P4L7B, AT&T Labs-Research, 2003.
3. A. Balmin and Y. Papakonstantinou. Storing and Querying XML Data Using Denormalized Relational Databases. *The VLDB Journal*, 14(1):30–49, 2005.
4. P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *ICDE '02: Proc. of the 18th Int. Conf. on Data Engineering*, pages 64–75, Washington, DC, USA, 2002. IEEE Computer Society.
5. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C, 2006.
6. F. Du, S. Amer-Yahia, and J. Freire. ShreX: Managing XML Documents in Relational Databases. In *VLDB '04: Proc. of the 30th Int. Conf. on Very Large Data Bases*, pages 1297–1300, Toronto, ON, Canada, 2004. Morgan Kaufmann Publishers Inc.
7. D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.
8. J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Simeon. StatiX: Making XML Count. In *SIGMOD '02: Proc. of the 21st ACM SIGMOD Int. Conf. on Management of Data*, pages 181–192, Madison, Wisconsin, USA, 2002. ACM Press.
9. ISO/IEC 9075-14:2003. *Part 14: XML-Related Specifications (SQL/XML)*. International Organization for Standardization, 2006.
10. M. Klettke and H. Meyer. XML and Object-Relational Database Systems – Enhancing Structural Mappings Based on Statistics. In *Selected papers from the 3rd Int. Workshop WebDB '00 on The World Wide Web and Databases*, pages 151–170, London, UK, 2001. Springer-Verlag.
11. R. Krishnamurthy, V. Chakaravarthy, and J. Naughton. On the Difficulty of Finding Optimal Relational Decompositions for XML Workloads: A Complexity Theoretic Perspective. In *ICDT '03: Proc. of the 9th Int. Conf. on Database Theory*, pages 270–284, Siena, Italy, 2003. Springer.
12. R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In *XSym '03: Proc. of the 1st Int. XML Database Symp.*, volume 2824, pages 1–18, Berlin, Germany, 2003. Springer.
13. I. Mlynkova. A Journey towards More Efficient Processing of XML Data in (O)RDBMS. In *CIT '07: Proc. of the 7th IEEE Int. Conf.e on Computer and Information Technology*, Fukushima, Japan, 2007. IEEE Computer Society. (to appear).
14. I. Mlynkova. *UserMap – an Enhancing of User-Driven XML-to-Relational Mapping Strategies*. Technical report 2007/3. Charles University, Prague, Czech Republic, 2007.
15. I. Mlynkova and J. Pokorny. From XML Schema to Object-Relational Database – an XML Schema-Driven Mapping Algorithm. In *ICWI '04: Proc. of the 3rd IADIS Int. Conf. WWW/Internet*, pages 115–122, Madrid, Spain, 2004. International Association for Development of the Information Society.
16. The Apache XML Project. *Xerces Java Parser 1.4.4*. The Apache Software Foundation, 2005.
17. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB '99: Proc. of the 25th Int. Conf. on Very Large Data Bases*, pages 302–314, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
18. J. Yaghob and F. Zavoral. Semantic Web Infrastructure using DataPile. In *The 2006 IEEE/WIC/ACM Int. Conf. on Web Intelligence and Intelligent Agent Technology*, pages 630–633, Los Alamitos, California, 2006. IEEE Computer Society Press.
19. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Trans. Inter. Tech.*, 1(1):110–141, 2001.