Three Layer Evolution Model for XML Stored in Relational Databases

Andrey Simanovsky

St Petersburg State University
asimanovsky@acm.org

Abstract. XML-relational systems with well defined XML and relational schemas are widely used in industry. In the presence of rapidly changing requirements both schemas of such a model need continuous evolution, which can be performed by a formal framework that describes the evolution of the mutually dependent schema pairs of the system as sequences of elementary schema transformations.
There are a number of common tasks, like relational schema performance tuning, XML syntax clean-up, etc that stand apart from changes of semantics of the schemas. We discuss a framework with three layers of operations: the changes in XML schema only, the semantic changes that affect both schemas, and the changes that affect relational schema only. We show how the layered framework allows to formalize and address the above mentioned tasks. We consider the formal quality of the solutions of the tasks that may be achieved inside the framework.

1 Introduction

A practice of storing XML data in relational databases is widely employed because of the capabilities of the highly-developed RDBMS technologies. Provided that XML documents comply with a particular schema designed for the application domain, a more efficient utilization of these technologies can be achieved. In that case a relational database can use the metadata to organize effective storage, which includes selection of appropriate relational schema. On the other hand, if a choice between possible XML document schemas exists, the schema that is best fit for storing data in relational database should be chosen. Thereby these XML and relational schemas become mutually dependent.

If the product that uses the schemas exists in the environment of rapidly changing requirements, both schemas need to exhibit agile evolution in time. A major part of software products storing XML in relational databases are actually in this situation because they often need to change schemas with every new version of the product. Continuous redesign requires repetitive evolution of data access and data storage algorithms. A formal framework that presents schema changes as a sequence of elementary transformations and ensures invariance of selected properties of the schema and data complying with the schema is an alternative to ad-hoc solutions. Such a model is referred to as a schema evolution model.

Schema changes may be caused by changes in functionality of the product, by system performance requirements, compatibility issues, and other reasons. While it is

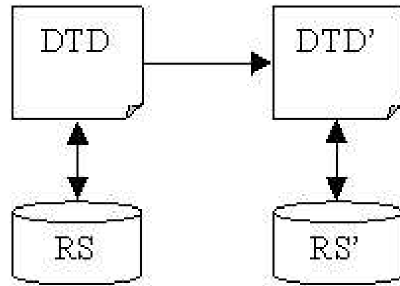

Fig. 1. General XML evolution model applied to XML stored in relational database.

preferable that performance problems are resolved on the relational schema level, functional requirements in the presence of compatibility issues are better expressed in terms of changes in XML schema only.

General XML document evolution models (for example, those found in [5,2], etc) work with XML schema only and cannot express the above mentioned problems. Figure 1[1] demonstrates the application of a general XML document evolution model. DTD denotes an original XML schema; DTD' is a new XML schema; RS is an original relational schema; RS' is a new relational schema. The relational model changes can be expressed only as a result of XML schema evolution; the effects of XML schema changes on relational schema are not incorporated into the model. The following tasks cannot be solved by a general evolution model:

1. Apply changes to the XML schema (e.g. to incorporate XML syntax refinement, or use of schema derivation like [9]) that would not require changing the underlying relational schema (*compatibility*).
2. Ensure that semantic constraints are present in relational schema (to allow XML to SQL queries conversion rather than obtaining XML document) and can be queried directly through SQL rather than XML (*functionality*).
3. Apply changes to the relational schema (e.g., to tune performance) that would not require XML schema changes (*performance*).

We also add another task, *richness*: the changes available in the model should allow to transform a given XML schema complying with a given set of semantic constraints to any XML schema complying with the same constraints. This task is usually solved by general XML evolution models. Resolving it in a specific XML-relational evolution model ensures that it is not less expressive about XML schema, than general XML evolution model.

We present an evolution model that explicitly contains the knowledge of the mutual dependency of XML and relational schemas. The outline of the model is given on Figure 2. S and S' are original and new states of intermediate layer that represents common

[1] We use DTD as XML document schema description throughout the rest of the paper, though most of the reasoning applies to other schema descriptions as well

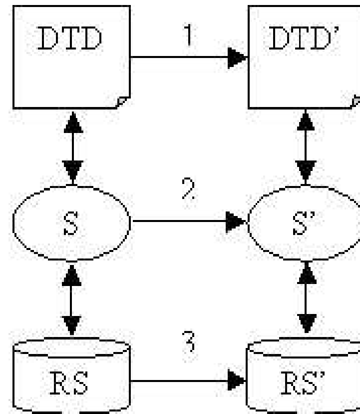

Fig. 2. The proposed evolution model

part of the schemas; intermediate layer contains semantic constraints and is represented in both schemas. Elementary operations of type 2 affect the intermediate layer; operations of type 1 affect only DTD; operations of type 3 affect only relational schema. In this paper we show how the above tasks can be solved in the model by considering the sequences of operations of the 1st, 2nd and 3d types respectively. We demonstrate that operations of the 1st and 2nd types are rich enough to solve the 4th task. We also discuss the quality of the answers to these issues.

The rest of the paper is organized as follows. Section 2 gives an overview of related work. Section 3 gives a brief description of the XML-relational evolution model we employ. The subsequent sections contain operations definitions and discuss how the operations address the listed above tasks.

2 Related work

The issue of storing XML documents in relational databases is a well explored area of research. The main trends are structure-driven approach, for example, [10,11], and model-driven approach, for example, "edge approach" from [4] or [17]. The structure-driven algorithms from [11] generate a fixed relational schema from a given XML DTD using either Shared or Hybrid techniques. The generation algorithms of these techniques construct a DTD graph similar to the one discussed in the paper. The XML document order and regular expressions information is lost in the conversion. [14] considers the addition of order information to the schemas generated using the Hybrid algorithm from [10], while [15] explores possibilities of utilizing functional dependencies information in final schema generation. However, all these works do not consider the effects of schema evolution over the algorithms.

Extensive research is done recently in the field of effective querying XML stored in relational database. [11] suggests a general solution for generating queries for arbitrary XML schemas. [6] discusses the use of the knowledge of the XML to SQL mapping algorithm to generate efficient SQL queries from XML queries.

DTD:
```
<!ELEMENT book (booktitle, author)>
<!ELEMENT article(title,author*,contactauthor)>
<!ELEMENT contactauthor authorID IDREF IMPLIED>
<!ELEMENT monograph (title,author,editor?)>
<!ELEMENT editor(monograph*)>
<!ELEMENT editor name CDATA #REQUIRED>
<!ELEMENT author (name,address)>
<!ELEMENT name(firstname?,lastname)>
<!ELEMENT title ANY>
```
Sample document:

```
<monograph>
   <title>Temporal data & relational model</title>
   <author>
      <name>
         <lastname>Date</lastname>
      </name>
      <address>NA</address>
   </author>
   <editor name="Homet">
   </editor>
</monograph>
```

Fig. 3. Sample DTD (from [10]) and XML document sample.

Various database schema evolution models were proposed for object-oriented [8] and relational databases [1]. Recently several works concerning XML appeared ([5,2]). [5] concentrates on versioning and schema derivations approach rather than schema modification. It deals with arbitrary XML schemas. [2] introduces an invariant-based approach to defining XML schema evolution. While these approaches may work well for XML data they cannot be mapped directly to a XML-relational system due to inability to express issues named *compatibility*, *functionality*, and *performance* as was shown in Section 1. On the other hand, [7,8,2] and [5] satisfy *richness* requirement, where the latter two do that for XML. [7,8] can express semantic constraints in a way similar to proposed model. [2] expresses semantic constraints for cycle-free XML schemas.

3 Evolution model overview

In consequent discussion we use as a sample a XML document schema taken from [10]. It is presented on Figure 3.

3.1 Schema S

The essential part of the evolution model is the intermediate schema S. S has a fixed mapping from DTD. Figure 4 shows the schema S for the sample DTD. Dashed rectangles are used to outline the factorized vertices. Filled circles mark the vertices that are

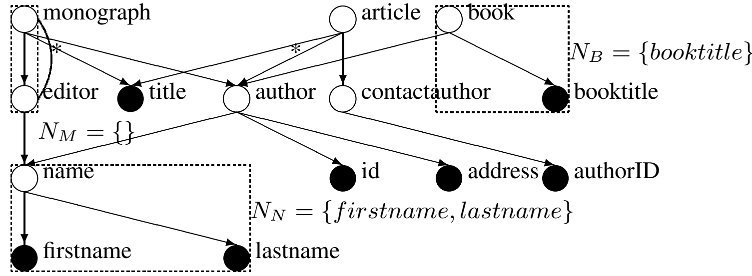

Fig. 4. Mapping sample DTD to S.

included into attribute sets of the factorized vertices. Factorized vertices not shown on the figure: A ($N_A = \{authorID\}$), U ($N_U = \{id, address\}$), T ($N_T = \{title\}$).

First, the *DTD graph* is built. The DTD is simplified by replacing attributes with leaf tags, inlining entities, replacing regular expressions in tag definitions. After that the tags become the vertices of the DTD graph. If one tag contains another an edge exists between corresponding DTD graph vertices. Each vertex is labeled with the name of the tag. An edge may be optionally marked by an asterisk ("*"). In that case it is a *star-edge*. Figure 4 contains a sample DTD graph. A formal definition of the DTD graph and the process of obtaining it from the DTD can be found in [12].

Figure 4 also shows a factorization of the DTD graph. Each *factorized vertex*, i.e., a vertex of the factorized DTD graph[2] has its own *start-vertex*. Each vertex in a factorized vertex can be reached by a single path from its start-vertex. The path does not contain star-edges.

Some of the leaf vertices of the DTD graph are *attributes* of the factorized vertex, which contains them. The choice of the attributes is up to the schema designer with a condition that it does not violate the semantic invariants of the schema S (semantic constraints are explained below). Each factorized vertex can contain another auxiliary attribute. Its value denotes element types of the beginnings of the edges that start from the vertices of the given factorized vertex and end in the start-vertices of other factorized vertices. For example, `book`, `monograph` or `article` are the values of the auxiliary attribute for the factorized vertex U. The auxiliary attribute is needed for *add/remove upper edge* operation (see Xml-only operations) only and can be omitted if the operation is not used. Thereinafter we omit explicit mention of this attribute.

The factorized DTD graph maps original DTD into intermediate schema S. [12] contains a formal description of the schema S construction process.

The mapping of the relational schema to S is represented by a set of views. Each view is a projection of a composition of equi-joins: $V_i = F_i(X_1, X_2, .., X_N)$, where X_i is the relation of the schema. In the naive case $V_i = \pi_{Vi}(X_i)$. Figure 5 represents a fragment of the script that generates a relational schema for the sample DTD in the

[2] The same term is used to refer to the fragment of the original graph that includes vertices comprising the factorized vertex and edges between them

```
CREATE TABLE U ( id INT NOT NULL, address VARCHAR(50) );
CREATE TABLE N ( id INT NOT NULL, firstname VARCHAR(50), lastname VARCHAR(50)
);
CREATE TABLE T ( id INT NOT NULL, title VARCHAR(50) );
CREATE TABLE B ( id INT NOT NULL, booktitle VARCHAR(50), id_U INT NOT NULL );
CREATE TABLE A ( id INT NOT NULL, authorId VARCHAR(50), id_N INT NOT NULL );
CREATE TABLE M ( id INT NOT NULL, id_U INT NOT NULL, id_N INT NOT NULL, id_T
INT NOT NULL, id_M INT NOT NULL);
ALTER TABLE B ADD CONTRAINT fk_U FOREIGN KEY ( id_U ) REFERENCES U (id);
...
```

Fig. 5. A fragment of the SQL script for the sample for the case $V_i = X_i$.

1. Closure: $\forall t \in V' \; P_e(t), D_e(t) \subseteq V'$
2. Rootedness: $\exists T \in V' \; \forall t \in V' \; T \in PL(t) \wedge P_e(T) = \emptyset$
3. Pointedness: $\exists \perp V' \; \forall t \in V' \; t \in PL(\perp)$
4. Immediate predecessors: $\forall t \in V' \; P(t) = P_e(t) - \cup\alpha_x(PL(x) \cap P_e(t) \cap (PL(t) - DL(t)) - \{x\}, P_e(t)) - \cup\alpha_x(PL(x) \cap P_e(t) \cap (PL(t) \cap DL(t)) - \{x\}, P_e(t)) - \cup\alpha_x((PL(x) - P(x)) \cap (PL(t) - DL(t)) \cap P_e(t), PL(x) \cap DL(x) - \{x\})$
5. Predecessors graph: $\forall t \in V' \; PL(t) = \cup\alpha_x(PL(x), P(t)) \cup \{t\}$
6. Immediate descendants: $\forall t \in V' \; D(t) = D_e(t) - \cup\alpha_x(DL(x) \cap D_e(t) \cap (DL(t) - PL(t)) - \{x\}, D_e(t)) - \cup\alpha_x(DL(x) \cap D_e(t) \cap (DL(t) \cap PL(t)) - \{x\}, D_e(t)) - \cup\alpha_x((DL(x) - D(x)) \cap (DL(t) - PL(t)) \cap D_e(t), DL(x) \cap PL(x) - \{x\})$
7. Descendants graph: $\forall t \in V' \; DL(t) = \cup\alpha_x(DL(x), D(t)) \cup \{t\}$
8. Interface: $\forall t \in V' \; I(t) = N(t) \cup H(t)$
9. Nativeness: $\forall t \in V' \; N(t) = N_e(t) - H(t)$
10. Inheritance: $\forall t \in V' \; H(t) = \cup\alpha_x(I(x), P(t))$

Fig. 6. Invariants set (axioms)

naive case. Edges are represented by integrity constraints on the view representing the end vertex.

3.2 Semantic constraints

The semantic invariants for the evolution of S are defined as a set of axioms similar to the axiom sets used for the type lattices in OO databases in [7,8]. The axioms are expressed as equations between the evolution model sets. Figure 6 contains the axioms. The principal difference as compared to the evolution model sets used for type lattices is that the absence of cyclic dependencies between sets is not required. Instead, a more complicated lattice over a product of vertices and information about the calculated sets of the model is used. An iterative algorithm ([16]) that calculates the evolution model sets on the complicated lattice is guaranteed to stop and return always the same results [12].

The evolution model sets are defined as follows. The hierarchy on the factorized vertices is defined by the sets of *immediate predecessors* $P(t)$ for each vertex t. Immediate predecessors are vertices that explicitly include vertex t. *Essential predecessors* $P_e(t)$ are explicitly specified by the schema designer sets of vertices. It is required that $P(t) \subseteq P_e(t)$. *Vertex subhierarchy* $PL(t)$ is a sub-graph of a factorized DTD graph, which vertex set consists of vertices, from which t is reachable. *Native attributes* $N(t)$ are a set of attributes defined in vertex t. *Inherited attributes* $H(t)$ are the union of attributes of all its predecessors. Essential attributes $N_e(t)$ are explicitly specified. It is required that $N(t) \subseteq N_e(t)$. *Interface* $I(t)$ is the union of its inherited and native attributes. The *reverse hierarchy* is defined by the sets of *immediate descendants* $D(t)$ for each vertex t. Immediate descendants are vertices that explicitly are included into vertex t. Vertex *reverse subhierarchy* $DL(t)$ is a sub-graph of a factorized DTD graph, which vertex set consists of vertices reachable from t. *Essential descendants* $D_e(t)$ are explicitly specified sets of vertices. It is required that $D(t) \subseteq D_e(t)$.

4 Elementary operations taxonomy

In the following sections the operations of the evolution model are discussed. Each operation over the schemas belongs to one of the following classes:

- *Xml-only* operations (represented by the arrow marked with “1” on Figure 2) — the operations that are applied to and affect only XML schema.
- *S-operations* (represented by the arrow marked with “2” on Figure 2) — the operations over the common part.
- *Relational-only* operations (represented by the arrow marked with “3” on Figure 2) — the operations that are applied to and affect only relational schema.

4.1 Xml-only operations definition

Xml-only operations are divided into two groups: a) affecting DTD graph and b) not affecting DTD graph.

The operations of type 1a, i.e., those affecting DTD graph, are the following:

- *move the vertex up/down*; replaces a pair of elements $<!ELEMENTA(B,..)>$ and $<!ELEMENTB(C,..)>$ from one factorized vertex with a pair $<!ELEMENTA(B,C,..)>$ and $<!ELEMENTB(..)>$ where B is not an attribute[3], and vice versa (see Figure 7a);
- *add/remove leaf vertex*; replaces an element $<!ELEMENTA(..)>$ where A is not an attribute with a pair $<!ELEMENTA(B)>$ and $<!ELEMENTB>$ where new element B is not an attribute, and vice versa (see Figure 7b);

[3] of a factorized vertex of the model S; the same applies to other uses of “attribute” in the operations definitions

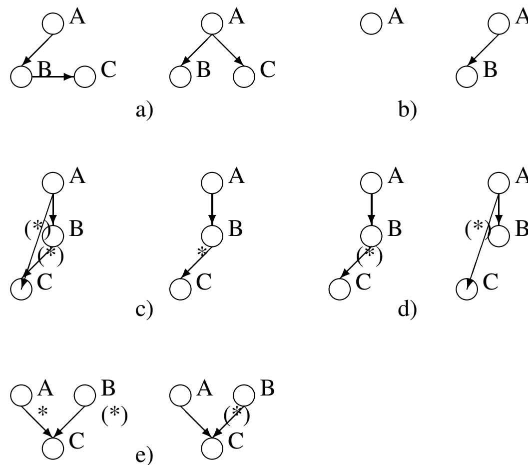

Fig. 7. DTD graph operations: a) move the vertex up/down; b) add/remove leaf vertex; c) add/remove upper edge; d) move edge; e) mark/unmark edge.

- *add/remove upper edge*; replaces a pair of elements $<!ELEMENTA(B,C*,..)>$ and $<!ELEMENTB(C*,..)>$ where C is a start vertex and both stars are optional with a pair $<!ELEMENTA(B,..)>$ and $<!ELEMENTB(C*,..)>$, and vice versa (see Figure 7c)[4];
- *move edge*; replaces a pair of elements $<!ELEMENTA(B,..)>$ and $<!ELEMENTB(C*,..)>$ where C is a start vertex and a star after C is optional with a pair $<!ELEMENTA(B,C*,..)>$ and $<!ELEMENTB(..)>$ where B is not an attribute, and vice versa (see Figure 7d);
- *mark/unmark edge*; replaces a an element $<!ELEMENTA(C*,..)>$ with an element $<!ELEMENTA(C,..)>$ if exists element $<!ELEMENTB(C*,..)>$ from another factorized vertex with an optional star after C, and vice versa (see Figure 7e).

4.2 Compatibility task solution

In this section it will be shown that it is possible to convert a given DTD that maps to a schema S into any arbitrary DTD that maps into the same schema S by applying a sequence of Xml-only operations. This would mean that the solution provided by the model is a complete solution: the model allows to obtain any XML with the given intermediate schema S, any other changes would necessarily lead to changes in relational schema[5].

A fragment of the DTD graph contained in one factorized vertex is a tree, e.g. `name`, `firstname` and `lastname` and edges between them from the sample DTD (see

[4] Note that semantics are not violated if the auxiliary attribute that was mentioned in Section 3 is present

[5] Consequent changes in relational schema may compensate each other, but the model allows to obtain the same resulting schema without changing relational schema at all.

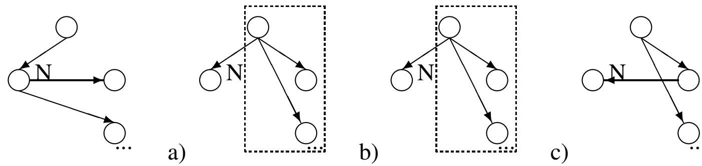

Fig. 8. Operations application for the proof of Statement 1: a) one or more *move vertex* operations; b) Induction proposition is applied (dashed rectangle shows the range of application); c) one or more *move vertex* operations.

Figure 4) form a tree of the factorized vertex U. The *move vertex up/down* operation changes edges in this tree.

Statement 4.1. The *move vertex up/down* operation allows that any given set of vertices of a factorized vertex with a fixed start-vertex can be organized into any tree.
The proof can be done by induction by the number of vertices in a factorized vertex. Provided that vertices are organized into some tree we try to evolve to the target tree. Figure 8 shows the operations sequence. The full proof of this and subsequent statements of this subsection can be found in [13].

Statement 4.2. The *move vertex up/down* and *add/remove leaf vertex* allow to organize vertices of a factorized vertex with the given start-vertex and attribute set into any tree[6].

Statement 4.3. *Add/remove upper edge*, *move edge*, and *mark/unmark edge* allow to obtain any set of outgoing edges that connect the vertices of a given factorized vertex with a fixed set of start-vertices.

The above three statements have the following obvious corollary, which forms the next statement.

Statement 4.4. The operations of the type 1a enable to transform a given DTD graph to any other DTD graph that maps to the same schema S.
Statement 4 implies that, as soon as there is a list of operations of type 1b that allow to transform a DTD having a given DTD graph to any other DTD with the same DTD graph, are available the compatibility task can be completely solved within the model. The following operations compose the necessary operations of the type 1b list:

- inline/de-inline entity;
- convert leaf tag to attribute and vice versa;
- simplify/de-simplify regular expression;
- change processing instruction node;
- add/remove comment or namespace node.

[6] Note that *add/remove leaf vertex* operation alone is not enough because of the presence of attributes

4.3 S-operations

There are eight operations defined for the S-model. They are similar to operations used in OO databases evolution model in [8]. Two additional operations, merge and split vertex, are added; they are not found in OO prototypes. Here these operations are expressed in terms of the schema S only due to the lack of space. An example of their application in the sample is given. More details on S-operations can be found in [12].

AddAttribute. Attribute a (the one being added) of vertex t is included into $N_e(t)$. The sets N_e, N, H are recalculated for the vertex t and vertices reachable from t. Schema designer (or an external algorithm) may include this attribute into N_e sets of some successors of vertex t.

RemoveAttribute. Attribute a (the one being deleted) of vertex t is excluded from $N_e(t)$, the sets N_e , N, H are recalculated for the vertex t and vertices reachable from t. Note, if t is in $P(s)$ or a is in $N_e(s)$, according to the axiom set, attribute a is included into $N(s)$.

AddEdge. Let edge $(t; s)$ is added. s is added into $P_e(t)$ and t is added into $D_e(s)$. Expressions dependent on $P_e(t)$ and $D_e(s)$ are recalculated. Note, that s is added into $P(t)$ if there is no other path from s to t (same for $D(s)$).

RemoveEdge. This operation is complex, it may cause generation of new edges in the schema in order to keep from violating the axiom set. New edges will start in predecessors of the end vertex of a deleted edge and will end in its successors. Provided that we remove edge $(t; s)$, s is deleted from $P_e(t)$. All expressions dependent on $P_e(t)$ are recalculated. If axioms are violated then the operation is rejected by the system. (Alternatively, the vertex t may be included into root vertex T, the change can be achieved in two stages: adding T into $P_e(t)$, RemoveEdge for the edge $(s; t)$, and if s is in $P_e(t)$ s is added into the graph. The same may apply to vertex s and the stop vertex of the hierarchy). Analogous operations are performed with $D_e(s)$.

AddVertex. Provided vertex t is added. t is included into hierarchy. The sets $P_e(t)$ and $D_e(s)$ are to be defined by schema designer (or an external algorithm) to generate incoming edges of t. t is added into $P_e(s)$ to satisfy axioms. In the simplest case, $P_e(t) = T$.

RemoveVertex. It is a complex modification. First, the vertex t is to be removed with the edges starting and ending in it. Second, a number of edges from its predecessors to its successors may be added. Third, attributes of vertex t, that are essential for its successors should migrate properly. The operation is implemented in three stages: applying RemoveAttribute to all attributes of t, applying RemoveEdge to all outgoing edges, and applying RemoveEdge to all incoming edges.

MergeVertex. Vertices being merged must be connected by an edge. Merged vertex P_e , D_e and N_e sets are unions of P_e , D_e and N_e sets of vertices being merged with exclusion of themselves. In P_e sets of reachable and D_e set of reaching vertices occurrences of the vertices being merged are replaced with occurrence of the merged one.

SplitVertex. P_e , D_e and N_e sets of split vertex are separated into two disjoint sets each. In P_e and N_e sets of reachable vertices occurrences of split vertex are replaced with one or both of the created without violating inclusion of $P(t)$ in $P_e(t)$ and $D(t)$ in $D_e(t)$ properties.

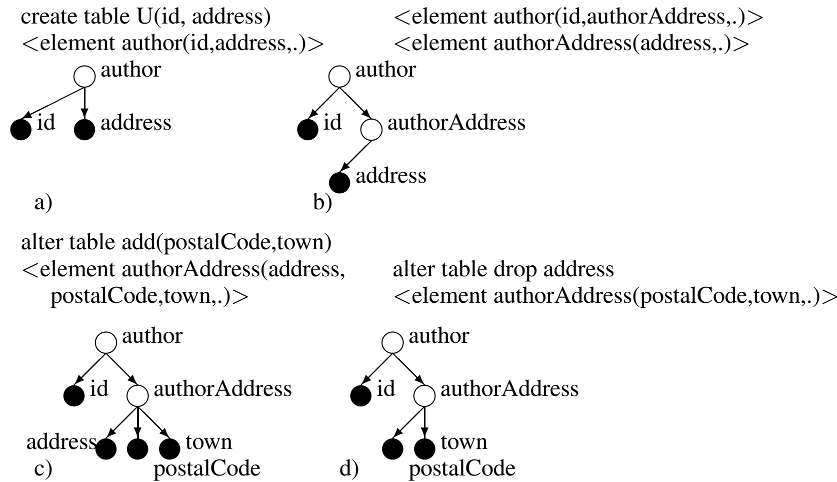

Fig. 9. Updating address information: a) initial fragment; b) add leaf vertex and move vertex down; S and relational schema are not changed; c) add attribute postalCode to U; add attribute town to U; plus two move vertex down to move postalCode and town under authorAddress; both schemas are changed; d) remove address attribute from author.

In the sample we consider a revision of author address information. First, the author-address tag is added through the add leaf vertex Xml-only operation. Next, the move vertex down Xml-only operation is used. Then attributes with postal code, town and local address are added, and the old attribute with address information is removed. Changes in schemas in the process of transformation are shown on Figure 9.

4.4 Functionality and richness tasks solution

The functionality task obviously has a complete solution, which is enforced by the design of the model. That is, since the schema S is expressed in both, relational and XML, terms and only schema S contains semantics, necessarily, all semantics is contained in well defined way in relational schema and can be queried directly through SQL.

The complete solution for the richness task means that any XML schema satisfying given semantic requirements may be obtained in the model. That enables solutions of schema transformation and other common problems inside the model.

Statement 4.5. The operations of the 2nd type allow to transform any given schema S to any other schema S having the same P_e, D_e and N_e sets.
It is trivial corollary of completeness of the schema S model. The proof of the latter can be found in [12].

Statement 5 enables us to provide a complete solution for the *richness* task. Indeed, an arbitrary XML schema complying with a given set of semantic constraints can be obtained from the given one complying with the constraints by consequent application of operations of the 2nd and 1st types. Statement 5 ensures that the schema S of the new

a) CREATE TABLE U(id INT NOT NULL, postalCode NUMBER, town VARCHAR(50));
V_u := U
b) CREATE TABLE U1(postalCode NUMBER);
INSERT INTO U1 (SELECT postalCode from V_u GROUP BY postalCode);
V_u := SELECT id, postalCode, town, localAddress FROM U JOIN U1 ON U.postalCode = U1.postalCode
c) ALTER TABLE U1 ADD(town VARCHAR(50));
UPDATE U1 SET U1.town = U.town FROM V_u WHERE U1.postalCode = U.postalCode;
V_u := SELECT id, postalCode, town, localAddress FROM U JOIN U1 ON U.postalCode = U1.postalCode AND U.town = U1.town
d) ALTER TABLE U DROP town;
V_u := SELECT id, postalCode, town, localAddress FROM U JOIN U1 ON U.postalCode = U1.postalCode

Fig. 10. Moving town information into a separate relation: a) original relation; b) add duplicate attribute postalCode (with creation of new relation U1); c) add duplicate attribute town; d) remove duplicate attribute town.

XML schema is obtained. The solution of *compatibility* problem enables to transform the intermediate result into the target XML schema.

4.5 Relational-only operations

The relational-only operations consist of one operation: *remove/add duplicate attribute in another relation*. It replaces a pair of relations $X_1(A, ..)$ and $X_2(A, ..)$ with a pair of relations $X_1(..)$ (X_1 may be dropped if empty; in the reverse operation X_1 may be created) and $X_2(A, ..)$ where A is an attribute of one of the factorized vertices, and vice versa. The operation also affects the formulae of the mapping to schema S. Namely, the equi-joins of X_1 and X_2 (if any) used in the definitions of V_i receive additional attribute A to the joined attributes set. Thus, the operation changes but does not break the form of the formulae that define the views.

In the sample we normalize the relations supposing that a town can be derived from a postal code. First, the postal code is duplicated in a new relation U1. Then the town information is moved from the relation U, which represents author information, to U1. Changes in S and relational schema in the process of transformation are shown on Figure 10.

4.6 Performance task solution

The complete solution of the performance task ensures that we are able to perform arbitrary relational schema transformations as far as we do not need to change the XML schema. This task is similar to compatibility task with the schemas exchanging places. The following statement ensures that *performance* task is completely solved.

Statement 4.6. Operations of the 3d type allow to transform a set of relations with the given sets of attributes and views values to any other set of relations with the same set of attributes and views values, if each attribute is used in the definition of at least one

view.
Proof. Provided we have an initial set of relations we try to evolve this set to a target set of relations. Each relation includes a set of attributes as its header (which is part of its relational type [3]). When an operation is applied one of the relation headers loses or obtains an attribute (namely, the header of X_1 in the notation of the definition). It is obvious that starting from the original set of relations we can obtain a relations set which relations will have the same headers as the headers of the target set. We will show that in case that the headers sets are identical the relation sets will be identical as well.
We apply induction by the number of attributes. Note that the proof will disregard operations — we only show that constraints in the form of view values and relation headers define the values of the relations.
If we have one attribute (the same works for none attributes but the case of none attributes can be omitted) there is at least one view with the header that includes all (that is, one) attributes. All relations are necessarily projections of this view and since the view value does not change, the relations have the same values.
Now suppose we have N attributes in relations. Consider the selections on a fixed value of Nth attribute of all relations and views. In the formulae that define the views selection commutes[7] with other operations and the formulae may be transformed to make them have the form as required by the statement. The induction proposition can be applied to these selections. Since the selections on every value of the Nth attribute are equal, the original relations are equal (we use the fact that the real attribute domains that are really used are all finite).
We have shown that relation headers set determines the relations values, on the other hand, target schema headers set can be obtained by the operations of the 3rd type. That means that the target relations set can be obtained, QED.

5 Conclusion

We considered evolution model as a way to design XML-relational systems in the environment with rapidly changing requirements. We defined *compatibility*, *functionality*, *performance*, and *richness* tasks that an XML-relational model should be able to solve. XML-only schema evolution models are not expressive enough to address all these tasks. The tasks need simultaneous consideration of both mutually dependent schemas.

We have suggested an XML-relational evolution model that allows to perform XML- or relational-only changes. We have shown that it ensures that semantic constraints can be obtained from relational schema, consequently, the model does not require obtaining an original XML document to evaluate an XML query. The model is rich enough to evolve a schema to any given XML schema. Thus, the employed model, contrary to XML-only models, is able to answer the stated requirements.

Several directions for future research exist. Intermediate schemas of the employed XML-relational model may be enriched with entities that will represent regular expressions and namespaces of the XML schema, or indices of the relational schema. Another

[7] The proof could avoid using this fact as well as the finiteness of domains, but it would require more space.

interesting direction of research is to modify the model to allow a larger range of mappings from XML to S-schemas. For example, the model can be enlarged to allow an edge-table approach [4] to be employed for storing part of XML data.

References

1. Chien-Tsai, L. et al: Database schema evolution through the specification and maintenance of changes on entities and relationships. Entity-Relationship Approach - ER'94, Business Modelling and Re-Engineering, 13th International Conference on the Entity-Relationship Approach, Manchester, U.K., December 13-16, 1994, Proceedings (1994) 881:132–151
2. Coox, S.: Axiomatization of the evolution of xml database schema. Programming (2003) 29(3):140–146
3. Date, C. J. et al: Temporal data and relational model (2002)
4. Florescu, D., Kossman, D.: A performance evaluation of alternative mapping schemes for storing xml data in a relational database. technical report 3684 (1999)
5. Hong, S., Kramer, D., Rundensteiner, E. A.: Xem: Xml evolution management http://www.citeseer.ist.psu.edu/su02xem.html
6. Krishnamurthy, R., Kaushik, R., Naughton, J.: Efficient xml-to-sql query translation: Where to add the intelligence? http://www.vldb04.org/protected/ eProceedings/contents/pdf/RS4P3.PDF
7. Peters, R. J., Ozsu, M. T.: Tigukat: a uniform behavioral objectbase management system. The VLDB Journal (1995) 4(3):445–492
8. Peters, R. J., Ozsu, M. T.: An axiomatic model of dynamic schema evolution in objectbase systems. ACM Transactions on Database Systems (1997) 22(1):75–114
9. Qi, H., Ling, T. W.: Extending and inferring functional dependencies in schema transformation. In CIKM '04: Proceedings of the Thirteenth ACM conference on Information and knowledge management, ACM Press (2004) 12–21
10. Shanmugasundaram, J. et al: Relational databases for querying xml documents: Limitations and opportunities. In Proc. VLDB Edinburgh, Scotland (1999) 302–314
11. Shanmugasundaram, J. et al: A general technique for querying xml documents using a relational database system. SIGMOD Record (2001) (3):20–26
12. Simanovsky, A.: Evolution of schema of xml-documents stored in a relational database. In proceedings of 6th Baltic DBIS Conf. (2004) 192–204
13. Simanovsky, A.: Simultaneous evolution of mutually dependent xml and relational schemas. In Proc. SYRCoDIS St Petersburg, Russia (2005)
14. Tatarinov, I. et al: Storing and querying ordered XML using a relational database system (2002)
15. Wang, Q. et al: Mapping xml documents to relations in the presence of functional dependencies. Journal of Software (2003) (7):1275–1281
16. Yamamoto, M. et al: Formalization of graph search algorithms and its applications. In proceedings of Theorem Proving in Higher Order Logics (TPHOLs'98), LNCS, Springer-Verlag (1998) 1479:479–496
17. Yoshikawa, M., Amagasa, T.: Xrel: a path-based approach to storage and retrieval of xml documents using relational databases. ACM Transactions on Internet Technology (2001) 1(1):110–141