Efficient Integrity Checking over XML Documents

Daniele Braga[1], Alessandro Campi[1], and Davide Martinenghi[2]

[1] Politecnico di Milano – Dip. di Elettronica e Informazione
p.zza L. da Vinci 32, 20133 Milano, Italy
{braga,campi}@elet.polimi.it

[2] Free University of Bozen/Bolzano – Faculty of Computer Science
p.zza Domenicani, 3, 39100 Bolzano, Italy
martinenghi@inf.unibz.it

Abstract. The need for incremental constraint maintenance within collections of semi-structured documents has been ever increasing in the last years due to the widespread diffusion of XML. This problem is addressed here by adapting to the XML data model some constraint verification techniques known in the context of deductive databases. Our approach allows the declarative specification of constraints as well as their optimization w.r.t. given update patterns. Such optimized constraints are automatically translated into equivalent XQuery expressions in order to avoid illegal updates. This automatic process guarantees an efficient integrity checking that combines the advantages of declarativity with incrementality and early detection of inconsistencies.

1 Introduction

It is well-known that expressing, verifying and automatically enforcing data correctness is a difficult task as well as a pressing need in any data management context. In this respect, XML is no exception; moreover, there is no standard means of specifying generic constraints over large XML document collections. XML Schema offers a rich set of predefined constraints, such as structural, domain and cardinality constraints. However, it lacks full extensibility, as it is not possible to express general integrity requirements in the same way as SQL assertions, typically used to specify business rules at the application level in a declarative way. A large body of research, starting from [21], gave rise to a number of methods for incremental integrity checking within the framework of deductive databases and w.r.t. the relational data model. Indeed, a brute force approach to integrity checking, i.e., verifying the whole database each time data are updated, is unfeasible. This paper addresses this problem in the context of semi-structured data, and namely XML, in order to tackle the difficulties inherent in its hierarchical data model. A suitable formalism for the declarative specification of integrity constraints over XML data is therefore required in order to apply optimization techniques similar to those developed for the relational world. More specifically, we adopt for this purpose a formalism called XPathLog, a logical language inspired by Datalog and defined in [18]. In our approach, the tree structure of XPathLog constraints is mapped to a relational representation (in Datalog) which lends itself well to the above mentioned optimization techniques. The optimization only needs to take place once, at schema design time: it

takes as input a set of constraints and an update pattern and, using the hypothesis that the database is always consistent prior to the update, it produces as output a set of optimized constraints, which are as instantiated as possible. These optimized constraints are finally translated into XQuery expressions that can be matched against the XML document so as to check that the update does not introduce any violation of the constraints. At runtime, the optimized checks are performed instead of the full ones, whenever the updates are recognized as matching the patterns used in the simplification.

In particular, the constraint simplification method we adopt generates optimized constraints that can be tested *before* the execution of an update (and without simulating the updated state), so that inconsistent database states are completely avoided.

2 Constraint verification

Semantic information in databases is typically represented in the form of integrity constraints, which are properties that must always be satisfied for the data to be considered *consistent*. In this respect, database management systems should provide means to automatically verify, in an efficient way, that database updates do not introduce any violation of integrity. A complete check of generic constraints is too costly in any nontrivial case; in view of this, verification of integrity constraints can be rendered more efficient by deriving specialized checks that are easier to execute at each update. Even better performance is achieved if these checks can be tested before illegal updates. Nevertheless, the common practice is still based on *ad hoc* techniques: domain experts hand-code tests in the application program producing the update requests or design triggers within the database management system that respond to certain update actions. However, both methods are prone to errors and little flexibility w.r.t. changes in the schema or design of the database, which motivates the need for automated integrity verification methods.

In order to formalize the notion of consistency, and thus the constraint verification problem, we refer to deductive databases, in which a *database state* is the set of database facts and rules (tuples and views). As semantics of a database state D we take its *standard model*: the truth value of a closed formula F, relative to D, is defined as its valuation in the standard model and denoted $D(F)$.

Definition 1 (Consistency). *A database state D is* consistent *with a set of integrity constraints Γ iff $D(\Gamma) = true$.*

An *update* U is a mapping $U : \mathscr{D} \mapsto \mathscr{D}$, where $\mathscr{D}$ is the space of database states. For convenience, for any database state D, we indicate the state arising after update U as D^U. The constraint verification problem may be formulated as follows. Given a database state D, a set of integrity constraints Γ, such that $D(\Gamma) = true$, and an update U, does $D^U(\Gamma) = true$ hold too? As mentioned, evaluating $D^U(\Gamma)$ may be too expensive, so a suitable reformulation of the problem can be given in the following terms: is there a set of integrity constraints Γ^U such that $D^U(\Gamma) = D(\Gamma^U)$ and Γ^U is easier to evaluate than Γ? In other words, the looked for condition Γ^U should specialize the original Γ, as specific information coming from U is available, and avoid redundant checks by exploiting the fact that $D(\Gamma) = true$. We observe that reasoning about the future database state D^U with a condition (Γ^U) that is tested in the present state D, complies

with the semantics of *deferred* integrity checking (i.e., integrity constraints do *not* have to hold in intermediate transaction states).

3 General constraints over semi-structured data

Consistency requirements for XML data are not different from those holding for relational data, and constraint definition and enforcement are expected to become fundamental aspects of XML data management. In current XML specifications, fixed-format structural integrity constraints can already be defined by using XML Schema definitions; they are concerned with type definitions, occurrence cardinalities, unique constraints, and referential integrity. However, a generic constraint definition language for XML, with expressive power comparable to assertions and checks of SQL, is still not present in the XML Schema specification. We deem this a crucial issue, as this lack of expressiveness does not allow one to specify business rules to be directly included in the schema. Moreover, generic mechanisms for constraint enforcement are also lacking. In this paper we cover both aspects.

Our approach moves from a recently proposed adaptation of the framework of deductive databases to the world of semi-structured data. More precisely, we refer to XPathLog [18] as the language for specifying generic XML constraints, which are expressed in terms of queries that must have an empty result.

Even though, in principle, we could write denials in XQuery, a declarative, first-order logic language is closer to what is usually done for relational data [14]; a logical approach leads to cleaner constraint definitions, and the direct mapping from XPathLog to Datalog helps the optimization process.

3.1 XPathLog

XPathLog [18] is an extension of XPath modeled on Datalog. In particular, the XPath language is extended with variable bindings and is embedded into first-order logic to form XPath-Logic; XPathLog is then the Horn fragment of XPath-Logic. Thanks to its logic-based nature, XPathLog is well-suited to querying XML data and providing declarative specifications of integrity constraints.

It uses an *edge-labeled graph* model in which subelements are ordered and attributes are unordered. Path expressions have the form root/axisStep/.../axisStep where root specifies the starting point of the expressions (such as the root of a document or a variable bound to a node) and every axisStep has the form axis::nodetest[qualifier]*. An axis defines a navigation direction in the XML tree: child, attribute, parent, ancestor, descendant, preceding-sibling and following-sibling. All elements satisfying along the chosen axis nodetest are selected, then the qualifier(s) are applied to the selection to further filter it. Axes are abbreviated as usual, e.g. path/nodetest stands for path/child::nodetest and path/@nodetest for path/attribute::nodetest.

XPath-Logic formulas are built as follows. An infinite set of variables is assumed along with a signature of element names, attribute names, function names, constant symbols and predicate names. A *reference expression* is a path expression that may

be extended to bind selected nodes to variables with the construct "$\rightarrow Var$". Reference expressions have the form root/refAxisStep/.../refAxisStep, where the syntax of refAxisStep is as follows:

$$\text{axis::(nodetest}|Var)[\text{qualifier}]^*[\rightarrow Var][\text{qualifier}]^*.$$

XPath-Logic predicates are predicates over reference expressions and atoms and literals are defined as usual. Formulas are thus obtained by combining atoms with connectives ($\wedge$, $\vee$, $\neg$) and with quantified ($\exists$, $\forall$) variables. Clauses are written in the form *Head*$\leftarrow\!\!\sim$*Body* where the head, if present, is an atom and the body a conjunction of literals. In particular, a *denial* is a headless clause; integrity constraints will be written as denials, which indicates that there must be no variable binding satisfying the condition in the denial body for the data to be consistent. Unless otherwise indicated, clause variables (written with capital letters) are implicitly universally quantified.

Aggregates are written with the syntax $\text{agg}(V[G_1,\ldots,G_n];\text{reference-expression})$, where agg is an aggregate (such as Sum, Cnt, etc.), V, if present, is the variable on which the aggregate operation is performed and $G_1,\ldots,G_n$ are the group-by variables. A D subscript (e.g., Cnt_D) indicates that only *distinct* values are considered. Note that V is absent for Cnt and Cnt_D.

3.2 Examples

Consider two documents: `pub.xml` containing a collection of published articles and `rev.xml` containing information on reviewer/paper assignment for all tracks of a given conference. The DTDs are as follows.

```
<!-- pub.xml -->
<!ELEMENT dblp (pub)*>
<!ELEMENT pub (title,aut+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT aut (name)>
<!ELEMENT name (#PCDATA)>
```

```
<!-- rev.xml -->
<!ELEMENT review (track)+>
<!ELEMENT track (name,rev+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT rev (name,sub+)>
<!ELEMENT sub(title,auts+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT auts (name)>
```

Example 1. Consider the following integrity constraint, which imposes the absence of *conflict of interests* in the submission review process (i.e., no one can review papers written by a coauthor or by him/herself):

$$\begin{aligned}\leftarrow\!\!\sim\ & //rev[name/text() \rightarrow R]/sub/auts/name/text() \rightarrow A \\ & \wedge(A = R \vee //pub[aut/name/text() \rightarrow A \wedge aut/name/text() \rightarrow R])\end{aligned}$$

The text() function refers to the text content of the enclosing element. The condition in the body of this constraint indicates that there is a reviewer named R who is assigned a submission whose author has name A and, in turn, either A and R are the same or two authors of a same publication have names A and R, respectively.

Example 2. Consider a conference policy imposing that a reviewer involved in three or more tracks cannot review more than 10 papers. This is expressed as follows:

$$\begin{aligned}\leftarrow\!\!\sim\ & \text{Cnt}_\text{D}\{[R];//track[/rev/name/\text{text}() \rightarrow R]\} \geq 3\wedge \\ & \text{Cnt}_\text{D}\{[R];//rev[/name/\text{text}() \rightarrow R]/sub\} \geq 10\end{aligned}$$

4 Mapping XML constraints to the relational data model

In order to apply our simplification framework to XML constraints, as will be described in Section 5, schemata, update patterns, and constraints need to be mapped from the XML domain to the relational model. Note that these mappings take place statically and thus do not affect runtime performance.

4.1 Mapping of the schema and of update statements

The problem of representing XML data in relations was considered, e.g., in [25]. Our approach is targeted to deductive databases: each node type is mapped to a corresponding predicate. The first three attributes of all predicates respectively represent, for each XML item: its (unique) node identifier, its position and the node identifier of its parent node. It is worth noting that the second attribute is crucial, as the XML data model is ordered. Whenever a parent-child relationship within a DTD is a one-to-one correspondence (or an optional inclusion), a more compact form is possible, because a new predicate for the child node is not necessary: the attributes of the child may be equivalently represented within the predicate that corresponds to the parent (allowing null values in case of optional child nodes). The documents of the previous section map to the relational schema

```
pub(Id,Pos,IdParent_{dblp},Title)          aut(Id,Pos,IdParent_{pub},Name)
track(Id,Pos,IdParent_{review},Name)       rev(Id,Pos,IdParent_{track},Name)
sub(Id,Pos,IdParent_{rev},Title)           auts(Id,Pos,IdParent_{sub},Name)
```

where *Id*, *Pos* and $IdParent_{tagname}$ preserve the hierarchy of the documents and where the PCDATA content of the `name` and `title` node types is systematically embedded into the container nodes, so as to reduce the number of predicates.

As already mentioned, mapping a hierarchical ordered structure to a flat unordered data model forces the exposition of information that is typically hidden within XML repositories, such as the order of the sub-nodes of a given node and unique node identifiers. The root nodes of the documents (`dblp` and `review`) are not represented as predicates, as they have no local attributes but only subelements; however, such nodes are referenced in the database as values for the IdParent_{dblp} and IdParent_{review} attributes respectively, within the representation of their child nodes. Publications map to the *pub* predicate, authors in `pub.xml` map to *aut*, while authors in `rev.xml` map to *auts*, and so on, with predicates corresponding to tagnames. Last, `names` and `titles` map to attributes within the predicates corresponding to their containers.

Data mapping criteria influence update mapping. We express updates with the XUpdate language [13], but other formalisms that allows the specification of insertions of data fragments would also apply. Consider the following statement:

```
<xupdate:modifications version="1.0" xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:insert-after select="/review/track[2]/rev[5]/sub[6]" >
    <xupdate:element name="sub">
      <title> Taming Web Services </title> <auts> <name> Jack </name> </auts>
    </xupdate:element> </xupdate:insert-after> </xupdate:modifications>
```

In the corresponding relational model, this update statement corresponds to adding $\{\ sub(id_s, 7, id_r,$ "Taming Web Services"$),\ auts(id_a, 2, id_s,$ "Jack"$)\ \}$ where id_a and id_s represent the identifiers that are to be associated to the new nodes and id_r is the identifier associated to the target `rev` element. Their value is immaterial to the semantics of the update, provided that a mechanism to impose their uniqueness is available. On the other hand, the actual value of id_r depends on the dataset and needs to be retrieved by interpreting the `select` clause of the XUpdate statement. Namely, id_r is the identifier for the fifth (`reviewer`) child of the second (`track`) node, in turn contained into the root (`review`) node of the document `rev.xml`. Positions (7 and 2 in the second argument of both predicates) are also derived by parsing the update statement: 7 is determined as the successor of 6, according to the `insert-after` semantics of the update; 2 is due to the ordering, since the `auts` comes after the `title` element. Finally, note that the *same* value id_s occurs both as the first argument of $sub()$ and the third argument of $auts()$, since the latter represents a subelement of the former.

4.2 Mapping of integrity constraints

The last step in the mapping from XML to the framework of deductive databases is to compile denials into Datalog. We express constraints as Datalog denials: clauses with an empty head (understood as *false*), whose body indicates not holding conditions. Input to this phase are the schemata (XML and relational) and an XPathLog denial in a normal form without disjunctions [3]. All p.e. in XPathLog generate chains of conditions over the predicates corresponding to the node types traversed by the path expression. to the traversed node types. Containment in terms of parent-child relationship translates to correspondences between variables in the first position of the container and in the third position of the contained item.

Quite straightforwardly, XPathLog denial expressing that the author of the "Duckburg tales" cannot be Goofy and its mapping (anonymous variables are indicated with an underscore):

$$\leftsquigarrow //pub[title = \text{“}Duckburg\ tales''\text{”}]/aut/name \to N \wedge N = \text{“}Goofy''$$

$$\leftarrow pub(I_p, _, _, \text{“}Duckburg\ tales''\text{”}) \wedge aut(_, _, I_p, N) \wedge N = \text{“}Goofy''.$$

The fact that the XML data model is ordered impacts the translation. Either the `position()` function is used in the original denial or a filter is used that contains an expression returning an integer. In both cases, the second argument in the relational predicate is associated to a variable that is matched against a suitable comparison expression (containing the variable associated to the `position()` function or directly to the expression that returns the value).

Example 3. The XPathLog constraint of example 1, is translated into the following couple of Datalog denials (due to the presence of a disjunction).

$$\begin{aligned}\Gamma = \{ &\leftarrow rev(I_r, _, _, R) \wedge sub(I_s, _, I_r, _) \wedge auts(_, _, I_s, R),\\ &\leftarrow rev(I_r, _, _, R) \wedge sub(I_s, _, I_r, _) \wedge auts(_, _, I_s, A)\\ &\quad \wedge aut(_, _, I_p, R) \wedge aut(_, _, I_p, A)\}\end{aligned}$$

[3] A default rewriting allows one to reduce to such normal form any denial expressed with disjunctions, so that we can restrict to this case without loss of generality.

5 Simplification of integrity constraints

Several methods for optimized and incremental constraint checking in deductive databases, known as *simplification* methods, were produced since the landmark contribution by Nicolas [21]. Simplification in this context means to derive specialized versions of the integrity constraints w.r.t. given update patterns, employing the hypothesis that the database is initially consistent. In the following, we briefly describe the approach of [16]. To illustrate the framework, we limit our attention to tuple insertions, consistently with the fact that XML documents typically grow. An update transaction is expressed as a set of ground atoms representing the tuples that will be added to the database. Placeholders for constants, called *parameters* (written in boldface: **a**, **b**, ...), allow one to indicate update *patterns*. For example, the notation $\{p(\mathbf{a}), q(\mathbf{a})\}$, where **a** is a parameter, refers to the class of update transactions that add the same tuple to both unary relation p and unary relation q. The first step in the simplification process is to introduce a syntactic transformation After that translates a set of denials Γ referring to the updated database state into another set Σ that holds in the present state if and only if Γ holds after the update.

Definition 2. *Let Γ be a set of denials and U an update. The notation $\mathsf{After}^U(\Gamma)$ refers to a copy of Γ in which all atoms of the form $p(\vec{t})$ have been simultaneously replaced by $(p(\vec{t}) \vee \vec{t} = \vec{a}_1 \vee \cdots \vee \vec{t} = \vec{a}_n)$, where $p(\vec{a}_1), \ldots, p(\vec{a}_n)$ are all additions on p in U, $\vec{t}$ is a sequence of terms and $\vec{a}_1, \ldots, \vec{a}_n$ are sequences of constants or parameters (we assume that the result of this transformation is always given as a set of denials which can be produced by using, e.g., De Morgan's laws).*

Example 4. Consider a relation $p(ISSN, TITLE)$ and let $U = \{p(\mathbf{i}, \mathbf{t})\}$ be the addition of a publication with title **t** and ISSN number **i** and $\phi = \leftarrow p(X,Y) \wedge p(X,Z) \wedge Y \neq Z$ the denial imposing uniqueness of ISSN. $\mathsf{After}^U(\{\phi\})$ is as follows:

$$
\begin{aligned}
&\{ \leftarrow [p(X,Y) \vee (X = \mathbf{i} \wedge Y = \mathbf{t})] \wedge [p(X,Z) \vee (X = \mathbf{i} \wedge Z = \mathbf{t})] \wedge\ Y \neq Z\} \\
\equiv &\{ \leftarrow p(X,Y) \wedge p(X,Z) \wedge Y \neq Z, \\
&\leftarrow p(X,Y) \wedge X = \mathbf{i} \wedge Z = \mathbf{t} \wedge Y \neq Z, \\
&\leftarrow X = \mathbf{i} \wedge Y = \mathbf{t} \wedge p(X,Z) \wedge Y \neq Z, \\
&\leftarrow X = \mathbf{i} \wedge Y = \mathbf{t} \wedge X = \mathbf{i} \wedge Z = \mathbf{t} \wedge Y \neq Z\}.
\end{aligned}
$$

Clearly, After's output is not in any "normalized" form, as it may contain redundant denials and sub-formulas (such as, e.g., $a = a$). Moreover, assuming that the original denials hold in the current database state can be used to achieve further simplification. For this purpose, a transformation $\mathsf{Optimize}_\Delta(\Gamma)$ is defined that exploits a given set of denials Δ consisting of trusted hypotheses to simplify the input set Γ. The proposed implementation [17] is described in [16] in terms of sound rewrite rules, whose application reduces denials in size and number and instantiates them as much as possible. For reasons of space, we refrain from giving a complete list of the rewrite rules in the Optimize operator and we describe its behavior as follows.

Given a set of denials Γ, a denial $\phi \in \Gamma$ is removed if it can be proved redundant from $\Gamma \setminus \{\phi\}$; ϕ is replaced by a denial ψ that can be proved from Γ if ψ subsumes ϕ; equalities involving variables are eliminated as needed. The resulting procedure is

terminating, as it is based on resolution proofs restricted in size. The operators After and Optimize can be assembled to define a procedure for simplification of integrity constraints.

Definition 3. *For an update U and two sets of denials Γ and Δ, we define* $\mathsf{Simp}^U_\Delta(\Gamma) = \mathsf{Optimize}_{\Gamma\cup\Delta}(\mathsf{After}^U(\Gamma))$.

Theorem 1 ([16]). Simp *terminates on any input and, for any two set of denials Γ, Δ and update U,* $\mathsf{Simp}^U_\Delta(\Gamma)$ *holds in a database state D consistent with Δ iff Γ holds in D^U.*

We use $\mathsf{Simp}^U(\Gamma)$ as a shorthand for $\mathsf{Simp}^U_\Gamma(\Gamma)$.

Example 5. [4 cont.] The first denial in $\mathsf{After}^U(\{\phi\})$ is the same as ϕ and is thus redundant; the last one is a tautology; both the second and third reduce to the same denial; therefore the resulting simplification is $\mathsf{Simp}^U(\{\phi\}) = \{\leftarrow p(\mathbf{i}, Y) \wedge Y \neq \mathbf{t}\}$, which indicates that, upon insertion of a new publication, there must not already exist another publication with the same ISSN and a different title.

5.1 Examples

We now consider some examples based on the relational schema of documents `pub.xml` and `rev.xml` given in section 4.

Example 6. [1 continued] Let us consider constraint Γ from example 3 imposing the absence of conflict of interests in the submission review process. An update of interest is, e.g., the insertion of a new submission to the attention of a reviewer.

For instance, a submission with a single author complies with the pattern
$U = \{sub(\mathbf{i}_s, \mathbf{p}_s, \mathbf{i}_r, \mathbf{t}), auts(\mathbf{i}_a, \mathbf{p}_a, \mathbf{i}_s, \mathbf{n})\}$,

where the parameter ($\mathbf{i}_s$) is the same in both added tuples. The fact that $\mathbf{i}_s$ and $\mathbf{i}_a$ are new node identifiers can be expressed as a set of extra hypotheses to be exploited in the constraint simplification process:

$\Delta = \{\leftarrow sub(\mathbf{i}_s, _, _, _), \leftarrow auts(_, _, \mathbf{i}_s, _), \leftarrow auts(\mathbf{i}_a, _, _, _)\}$.
The simplified integrity check w.r.t. update U and constraint Γ is given by
$\mathsf{Simp}^U_\Delta(\Gamma)$: $\{\leftarrow rev(\mathbf{i}_r, _, _, \mathbf{n}), \leftarrow rev(\mathbf{i}_r, _, _, R) \wedge aut(_, _, I_p, \mathbf{n}) \wedge aut(_, _, I_p, R)\}$.

The first denial requires that the added author of the submission ($\mathbf{n}$) is not the same person as the assigned reviewer ($\mathbf{i}_r$). The second denial imposes that the assigned reviewer is not a coauthor of the added author $\mathbf{n}$. These conditions are clearly much cheaper to evaluate than the original constraints Γ, as they are instantiated to specific values and involve fewer relations.

Example 7. Consider the denial $\phi = \leftarrow rev(I_r, _, _, _) \wedge \mathsf{Cnt}_\mathsf{D}(sub(_, _, \underline{I_r}, _)) > 4$ imposing a maximum of 4 reviews per reviewer per track. The simplified integrity check of ϕ w.r.t. update U from example 6 is $\mathsf{Simp}^U_\Delta(\{\phi\}) = \{\leftarrow rev(\mathbf{i}_r, _, _, _) \wedge \mathsf{Cnt}_\mathsf{D}(sub(_, _, \mathbf{i}_r, _)) > 3\}$, which checks that the specific reviewer $\mathbf{i}_r$ is not already assigned 3 different reviews in that track.

6 Translation into XQuery

The simplified constraints obtained with the technique described in the previous section are useful only if they can be checked before the corresponding update, so as to prevent the execution of statements that would violate integrity. Under the hypothesis that the dataset is stored into an XML repository capable of executing XQuery statements, the simplified constraints need to be translated into suitable equivalent XQuery expressions in order to be checked. This section discusses the translation of Datalog denials into XQuery. We exemplify the translation process using the (non-simplified) set of constraints Γ defined in example 3. For brevity, we only show the translation of the second denial.

The first step is the expansion of the Datalog denial. It consists in replacing every constant in a database predicate (or variable already appearing elsewhere in database predicates) by a new variable and adding the equality between the new variable and the replaced item. This process is applied to all positions, but the first and the third one, which refer to element and parent identifiers and thus keeps information on the parent-child relationship of the XML nodes. In our case, the expansion is:

$$\leftarrow rev(I_r, B, C, R) \wedge sub(I_s, D, I_r, E) \wedge auts(F, G, I_s, A)$$
$$\wedge aut(H, I, I_p, J) \wedge aut(K, L, I_p, M) \wedge J = R \wedge M = A$$

The atoms in the denial must be sorted so that, if a variable referring to the parent of a node also occurs as the id of another node, then the occurrence as an id comes first. Here, no such rearrangement is needed. Then, for each atom $p(Id, Pos, Par, D_1, \ldots, D_n)$ where D_1, ..., D_n are the values of tags d1, ..., dn, resp., we do as follows. If the definition of `$Par` has not yet been created, then we generate `$Id in //p` and `$Par in $Id/..`; otherwise we just generate `$Id in $Par/p`. This is followed by `$Pos in $Id/position()`, `$D1 in $Id/d1/text(), ..., $Dn in $Id/dn/text()`.

Then we build an XQuery boolean expression (returning `true` in case of violation) by prefixing the definitions with the `some` keyword and by suffixing them with the `satisfies` keyword followed by all the remaining conditions in the denial separated by `and`. This is a well-formed XQuery expression. Here we have:

```
some $Ir in //rev, $C in $Ir/.., $B in $Ir/position(),    $R in $Ir/name/text(),
     $Is in $Ir/sub,             $D in $Is/position(),    $E in $Is/title/text(),
     $F in $Is/auts,             $G in $F/position(),     $A in $F/name/text(),
     $H in //aut,                $Ip in $H/..,            $I in $H/position(),
     $J in $H/name/text(),       $K in $Ip/aut,           $L in $K/position(),
     $M in $K/name/text()
satisfies $J = $R and $M = $A
```

Such expression can be optimized by eliminating definitions of variables which are never used, unless they refer to node identifiers. Such variables are to be retained because they express an existential condition on the element they are bound to. Variables referring to the position of an element are to be retained only if used in other parts of the denial. In the example, we can therefore eliminate the definitions of variables `$B`, `$C`, `$D`, `$E`, `$G`, `$I`, `$L`. If a variable is used only once outside its definition, its occurrence is replaced with its definition. Here, e.g., the definition of `$Is` is removed and `$Is` is replaced by `$Ir/sub` in the definition of `$F`, obtaining `$F in $Ir/sub/auts`.

Variables occurring in the `satisfies` part are replaced by their definition. Here we obtain the following query.

```
some      $Ir in //rev, $H in //aut
satisfies $H/name/text()=$Ir/name/text()
      and $H/../aut/name/text()=$Ir/sub/auts/name/text()
```

The translation of the simplified version $\mathsf{Simp}^U_\Delta(\Gamma)$ is made along the same lines. Again, we only consider the simplified version of the second constraint (the denial $\leftarrow\ rev(\mathbf{i}_r,_,_,R) \wedge aut(_,_,I_p,\mathbf{n}) \wedge aut(_,_,I_p,R)$). Now, a parameter can occur in the first or third position of an atom. In such case, the parameter must be replaced by a suitable representation of the element it refers to. Here we obtain:

```
some      $D in //aut
satisfies $D/name/text()=%n
      and $D/../aut/name/text()= /review/track[%i]/rev[%j]/name/text()
```

where `/review/track[%i]/rev[%j]` conveniently represents $\mathbf{i}_r$. Similarly, `\%n` corresponds to $\mathbf{n}$. The placeholders `%i`, `%j` and `%n` will be known at update time and replaced in the query.

The general strategy described above needs to be modified in the presence of aggregates. Aggregates apply to sequences of nodes; therefore, the most suitable constructs to define such sequences are `let` clauses. In particular, there is a `let` clause for each aggregate. This does not affect generality, as variables bound in the `let` clauses correspond to the aggregate's target path expression possibly defined starting from variables already bound in the `for` clauses above. The expression is wrapped inside an `exists(...)` construct in order to obtain a boolean result; for this purpose, an empty `<idle/>` tag is returned if the condition is verified. Again, integrity is violated if the query returns `true`. Constraint $\leftarrow rev(I_r,_,_,_) \wedge \mathsf{Cnt}_{\mathsf{D}}(sub(_,_,\underline{I_r},_)) > 4$, shown in example 7, is mapped to XQuery as shown below.

```
exists( for $Ir in //rev  let $D := $R/sub  where count($D) > 4  return <idle/> )
```

The other constraints in the examples can be translated according to the same strategy.

7 Evaluation

We now present some experiments conducted on a series of XML datasets matching the DTD presented in section 2, varying in size from 32 to 256 MB, on the examples described in order to evaluate the performance of our approach. Figures 1(a), 1(b) refer to the integrity constraints of examples 1, 2, respectively. The data were generated remapping data from the DBLP repository [15] into the schema of our running examples. Our tests were run on a machine with a 3.4 GHz processor, 1 GB of RAM and 140 GB of hard disk, using eXist [8] as XQuery engine. Execution times are indicated in milliseconds and represent the average of the measured times of 200 attempts for each experiment (plus 50 additional operations that were used as a "warm-up" procedure and thus not measured). The size of the documents is indicated in MB on the x-axis. Each figure corresponds to one of the running examples and reports three curves representing respectively the time needed (i) to verify the original constraint (diamonds), (ii) to

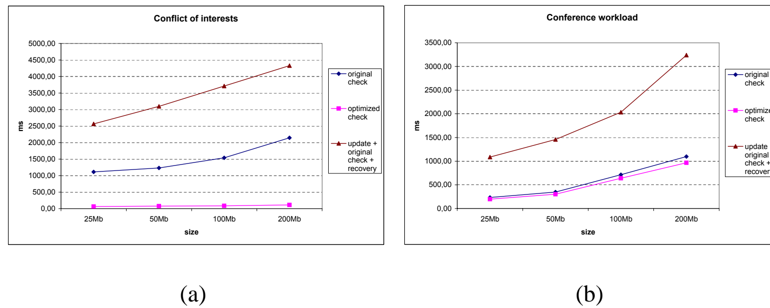
verify the optimized constraint (squares), and (iii) to execute an update, verify the original constraint, and undo the update (triangles). We observe that we do not have to take into account the time spent to produce the optimized constraints, nor the cost of mapping schemata and constraints to the relational model, as in our framework these are generated at schema design time and thus do not interfere with run time performance[4]. The curves with diamonds and squares are used to compare integrity checking in the non-simplified and, resp., simplified case, when the update is legal. The execution time needed to perform the update is not included, as this is identical (and unavoidable) in both the optimized and un-optimized case. The curve with triangles includes both the update execution time and the time needed to rollback the update, which is necessary when the update is illegal; when the update is illegal, we then compare the curve with triangles to the curve with squares. Rollbacks, needed since constraints are checked after an update, were simulated by performing a compensating action to re-construct the state prior to the update. The interpretation of these results is twofold, as we must consider two possible scenarios.

The update is legal: in the un-optimized framework the update is executed first and the full constraint is then checked against the updated database (showing that the update is legal); on the other hand, with the optimized strategy of our approach, the simplified constraint is checked first and the update is performed afterwards, as it is possible to check properties of the future database state in the present state (see Section 5).

The update is illegal: in the un-optimized framework execution is as in the previous case, but this time the check shows that there is some inconsistency and, finally, a compensative action is performed. On the contrary, with our optimized strategy, the simplified constraint is checked first, which reports an integrity violation w.r.t. the proposed update; therefore the update statement is *not* executed.

From the experimental results shown in figures 1(a) and 1(b) we observe two features. The comparison between the performance of the optimized and un-optimized checks shows that the optimized version is always more efficient than the original one. In some cases, as shown in figure 1(a), the difference is remarkable, since the simplified version contains specific values coming from the concrete update statement which allow one to filter the values on which complex computations are applied. Further improvement is due to the elimination of a join condition in the optimized query. In other cases the improvement is not as evident because introduction of filters does not completely eliminate the complexity of evaluation of subsequent steps, such as the calculation of

[4] The only activity to be performed at runtime is the matching of the actual update with a suitable known pattern, so as to apply the right optimized constraint. In our framework, we consider the case in which such recognition is trivially achieved and its cost is negligible, either because the patterns are very simple or because the user declares which pattern is in use while performing the update itself, choosing among a set of patterns published at schema design time. Otherwise, efficient representations of patterns and ad-hoc matching techniques should be investigated, so as to minimize this cost, which should of course be considered in the runtime evaluation. Unrecognized updates can either be processed w.r.t. the full integrity check or undergo a runtime simplification, but this case was not considered in our experiments. Nevertheless, we point out that the cost of the simplification itself is not dramatic: for instance, the simplified constraints of examples 1 and 6 were generated in less than 50 ms. Further details on the complexity analysis and the evaluation of the simplification procedure are in [5].



(a) (b)

Fig. 1. Conflict of interests (a) and Conference workload (b)

aggregate operations (figure 1(b)). The gain of early detection of inconsistency, which is a distinctive feature of our approach, is unquestionable in the case of illegal updates. This is prominently apparent in the cases considered in figures 1(a) and 1(b), since, as is well-known, the modification of XML documents is an expensive task.

8 Related work

Integrity checking is often regarded as an instance of materialized view maintenance: integrity constraints are defined as views that must always remain empty for the database to be consistent. The database literature is rich in methods that deal with relational view/integrity maintenance; insightful discussions are in [11] and [7].

A large body of research in the field has also been produced by the logic programming and artificial intelligence communities, starting from [21]. Logic-based methods that produce *simplified* integrity tests can be classified according to different criteria, e.g., whether these tests can be checked before or only after the update, whether updates can be compound or only singleton, whether the tests are necessary and sufficient or only sufficient conditions for consistency, whether the language includes aggregates. Some of these methods are surveyed in [19]. In this respect, the choice of the simplification method of [16] seems ideal, as it matches all the above criteria, and is further motivated by the availability of an implementation.

An attempt to adapt view maintenance techniques to the semi-structured data model has been made in [26] and in [22]. Incremental approaches have been proposed with respect to validation of structural constraints in [1], as well as to key and foreign key constraints in [4], where the validating algorithm parses the document with SAX and constructs an index of standard XML keys in one pass, with the help of suitable automata which recognize the context, the target, and the paths of such keys. Later, [24] addressed incremental validation in the context of streaming data under memory limitation. DTDs are considered as grammars and condition are provided on such grammars for the recognition of their languages to be performed by finite state automata instead of pushdown automata. Here the focus is again on validation w.r.t. DTD-like structural constraints only, and constraints upon values or involving aggregates are not addressed.

An attempt to simplification of general integrity constraints for XML has been made in [2], where, however, constraints are specified in a procedural fashion with an extension of XML Schema that includes loops with embedded assertions.

We are not aware of other works addressing validation w.r.t. general constraints for XML. However, integrity constraint simplification can be reduced to query containment if the constraints can be viewed as queries. Relevant works to this end are [23, 20].

There are several proposals and studies of constraint specification languages for XML by now. In [9] a unified constraint model (UCM) is proposed, which captures in a single framework the main features of o-o schemata and XML DTDs. UCM builds on the W3C XML query algebra and focuses on trading expressivity of the constraint language with simplicity of reasoning about the properties of the constraints. UCM leverages key/foreign key constraints and the XML type system, for expressing a restricted class of constraints whose consistency is proved decidable. This work addresses core algorithms for enforcing a particular class of constraints within a query engine, while our work relies on the availability of a query engine and addresses the simplification of constraints of arbitrary complexity (as long as they are expressible in XPathLog).

The XUpdate language, which was used for the experimental evaluation, is described in [13]. A discussion on update languages for XML is in [27].

As for XML-relational mappings, there exist several approaches to the problem of representing semi-structured data in relations [25, 3, 6, 10]. For a survey, see [12].

9 Conclusion and future work

In this paper we presented a technique enabling efficient constraint maintenance for XML datasets. We described the scenario in which integrity constraints are declaratively expressed in XPathLog, an intuitive logical language. These constraints are translated into Datalog denials that apply to an equivalent relational representation of the same data. Such denials are then simplified w.r.t. given update patterns so as to produce optimized consistency checks that are finally mapped into XQuery expressions that can be evaluated against the original XML document.

Besides the possibility to declaratively specify constraints, the main benefits of our approach are as follows. Firstly, the ability to produce optimized constraints typically allows a much faster integrity checking. Secondly, performance is further improved by completely avoiding the execution of illegal updates: the optimized check is executed first and the update is performed only if it does not violate integrity.

In this paper we focused on updates whose contents are specified extensionally, as in the XUpdate language. More complex updates may be specified with a rule-based language such as XPathLog, i.e., intensionally in terms of other queries. Yet, introducing such updates would not increase complexity, as these are already dealt with by the relational simplification framework of section 5 and can be translated from XPathLog to Datalog as indicated in section 4.

Several future directions are possible to improve the proposed method. We are studying the feasibility of a trigger-based view/integrity maintenance approach for XML that would combine active behavior with constraint simplification. Further lines of investigation include integrating visual query specification to allow the intuitive specification of constraints: domain experts lacking specific competencies in logic would be provided with the ability to design constraints to be further processed with our approach.

Acknowledgements: D. Martinenghi is supported by EU's IST project FP6-7603 ("TONES").

References

1. A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.
2. M. Benedikt, G. Bruns, J. Gibson, R. Kuss, and A. Ng. Automated Update Management for XML Integrity Constraints. In *Inf. Proc. of PLAN-X Workshop*, 2002.
3. P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. *ICDE*, 64–75, 2002.
4. Y. Chen, S. B. Davidson, and Y. Zheng. Xkvalidator: a constraint validator for XML. In *CIKM*, 446–452, New York, NY, USA, 2002. ACM Press.
5. H. Christiansen and D. Martinenghi. On simplification of database integrity constraints. *Fundamenta Informaticae*, 71(4):371–417, 2006.
6. A. Deutsch, M. Fernandez, and D. Suciu. Storing semi-struct-ured data with STORED. *SIGMOD*, 431–442, 1999.
7. G. Dong and J. Su. Incremental Maintenance of Recursive Views Using Relational Calculus/SQL. *SIGMOD Record*, 29(1):44–51, 2000.
8. eXist. Open source native xml database. `http://exist.sourceforge.net`.
9. W. Fan, G. M. Kuper, and J. Siméon. A unified constraint model for XML. *Computer Networks*, 39(5):489–505, 2002.
10. D. Florescu and D. Kossman. Storing and Querying XML Data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.
11. A. Gupta and I. S. Mumick (eds.). *Materialized views: techniques, implementations, and applications*. MIT Press, 1999.
12. R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-SQL query translation literature: The state of the art and open problems. *XSym*, 1–18, 2003.
13. A. Laux and L. Matin. XUpdate working draft. Technical report, `http://www.xmldb.org/xupdate`, 2000.
14. A. Levy and Y. Sagiv. Constraints and redundancy in datalog. In *PODS*, 67–80, New York, NY, USA, 1992.
15. M. Ley. Digital Bibliography & Library Project. `http://dblp.uni-trier.de/`.
16. D. Martinenghi. Simplification of integrity constraints with aggregates and arithmetic built-ins. In *Flexible Query-Answering Systems*, 348–361, 2004.
17. D. Martinenghi. A simplification procedure for integrity constraints. `http://www.ruc.dk/~dm/spic`, 2004.
18. W. May. XPath-Logic and XPathLog: a logic-programming-style XML data manipulation language. *TPLP*, 4(3):239–287, 2004.
19. E. Mayol and E. Teniente. A Survey of Current Methods for Integrity Constraint Maintenance and View Updating. In *ER Workshops*, 62–73, 1999.
20. F. Neven and T. Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In *ICDT*, 315–329, 2003.
21. J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.
22. A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental maintenance of path expression views. In *SIGMOD*, 2005.
23. T. Schwentick. XPath query containment. *SIGMOD Record*, 33(1):101–109, 2004.
24. L. Segoufin and V. Vianu. Validating Streaming XML Documents. In *PODS*, 53–64, 2002.
25. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. DeWitt, J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, 302–314, 1999.
26. D. Suciu. Query Decomposition and View Maintenance for Query Languages for Unstructured Data. In *VLDB*, 227–238, 1996.
27. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, 2001.