Online Appendix to "PATAXÓ: a framework to allow updates through XML views"

VANESSA P. BRAGANHOLO
COPPE, Universidade Federal do Rio de Janeiro, Brazil
and
SUSAN B. DAVIDSON
CIS, University of Pennsylvania
and
CARLOS A. HEUSER
Instituto de Informática, Universidade Federal do Rio Grande do Sul, Brazil

---

A. THEOREM PROOFS

In this section, we present the proofs of theorems 6.5, 6.7, 6.8 and 6.9.

THEOREM 6.5 *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then:* evalRel(eval(qt, d)) $\subseteq$ relOuterUnion(map(split(qt)), d).

PROOF. The $\subseteq$ operation needs the two multi-sets being compared to be union compatible. By definition, the schema of *evalRel* is the *evaluation schema* S, which is composed of all leaf node names in qt. The execution of *map(split(qt), d)* results in a set of relational views $\{V_1, ..., V_n\}$. Each view V_i is a schema composed of names of leaf nodes in qt_i (which is produced by *split(qt)*). By definition of *split*,

---

This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.

Author's address: Vanessa P. Braganholo, Programa de Sistemas, COPPE/UFRJ, Caixa Postal 68511, CEP 21941-972. Rio de Janeiro - RJ, Brazil. E-mail {vanessa@cos.ufrj.br}. Susan B. Davidson, 572 Levine North, Department of Computer and Information Science, University of Pennsylvania, Levine Hall, 3330 Walnut Street, Philadelphia-PA, USA, 19104-6389. E-mail: {susan@cis.upenn.edu}. Carlos A. Heuser. UFRGS/Informática, Caixa Postal 15064, CEP 91.501-970, Porto Alegre-RS, Brazil. E-mail: {heuser@inf.ufrgs.br}
Research supported in part by CNPq, Capes (BEX 1123-02/5) and FAPERJ Brazil, the Fulbright Program, as well as NSF IIS 0415810. This research was conducted while Vanessa was at UFRGS, and the paper was written and reviewed while she was at COPPE/UFRJ.

each split tree qt_i contains a single τ_N node n_i: the subtrees rooted at τ_N nodes different from n_i are deleted from qt_i. However, nodes deleted in qt_i are preserved in qt_j, so that each node n in qt is in at least one of the $qt_1, ...qt_n$. Consequently, the schema of $V_1 \bigcup ... \bigcup V_n$ equals S.

Assume t is in *evalRel(eval(qt, d))*, but not in *relOuterUnion(map(split(qt)), d)*. Let x be the XML view resulting from *eval(qt, d)*. Since t is in *evalRel(eval(qt, d))*, it was constructed by taking values from the leaf nodes in a given path p. The path p starts in a node n which is the deepest node of type τ_N or τ_T in a given subtree and goes up to the root of x. If n is of type τ_N, and V_i is the view corresponding to n, then t is in *evalV(V_i,d)*, and consequently, t is in *relOuterUnion(map(split(qt)), d)*, a contradiction. If n is of type τ_T, then the node that originated n in the query tree has at least one node of type τ_N in its subtree. Assume $V_j, ..., V_k$ are the relational views corresponding to those τ_N nodes. Consequently, t is in $V_j \bigcup ... \bigcup V_k$, and thus in *relOuterUnion(map(split(qt)), d)*, a contradiction. □

THEOREM 6.7 *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then every tuple t in* relOuterUnion(map(split(qt)), d) − evalRel(eval (qt, d)) $\subseteq$ stubs(x).

PROOF. Tuples in *relOuterUnion(map(split(qt)), d)* that are not in *evalRel(eval(qt, d))* are those resulting from left outer joins with no match in a given relational view $V_i \in$ *map(split(qt), d)*. Since *stubs(x)* contains tuples that has *null*s in attributes related to descendant nodes, and a LEFT JOIN always keeps information of the ancestor, then:
relOuterUnion(map(split(qt)), d) − *evalRel(eval(qt, d))* $\subseteq$ *stubs(x)*. □

THEOREM 6.8 *Given a query tree qt defined over database $\mathcal{D}$, then for any instance d of $\mathcal{D}$ and correct update u over qt,* evalRel(apply(x, u)) $\subseteq v'_1 \bigcup ... \bigcup v'_n$, *where* $\bigcup$ *denotes outer union.*

PROOF. Since the update u does not change the view schema, and the application of an update U_{ij} over view v_i also does not change v_i's schema, by Theorem 6.5 we have that *evalRel(apply(x, u))* and $v'_1 \bigcup ... \bigcup v'_n$ have the same schema (are union compatible).

Insertions, Suppose t is a tuple in *evalRel(apply(x, u))*, resulting from a insertion of a subtree in x. Assume t is not in $v'_1 \bigcup ... \bigcup v'_n$, and that update U_{ij} is the translation of u.

Consider a tuple t' which was inserted by update U_{ij} in v_i. Since U_{ij} is the translation of u, t' has the values of one of the subtrees that were inserted in x by u, and also the values of x that were above the update point ref of u. As a consequence, $t = t'$ and t is in $v'_1 \bigcup ... \bigcup v'_n$, a contradiction.

The same applies for the insertion of a more complex subtree. It will generate several tuples t_1, ..., t_n to appear in *evalRel(apply(x, u))*. Each of these tuples will be inserted in the relational views by a set of updates $U_{ij}, ..., U_{kl}$. So *evalRel(apply(x, u))* $\subseteq v'_1 \bigcup ... \bigcup v'_n$ holds for insertions.

Modifications. Suppose t is a tuple in *evalRel(apply(x, u))*, resulting from a modification of a leaf value in x. Assume t is not in $v'_1 \bigcup ... \bigcup v'_n$, and that update

U_{ij} is the translation of u.

Consider a tuple t' which was modified by update U_{ij} in v_i. Since U_{ij} is the translation of u, t' had a single attribute modified - the one that was updated in x. As a consequence, $t = t'$ and t is in $v'_1 \bigcup ... \bigcup v'_n$, a contradiction.

The same applies for modifications that affect more than one leaf in x, that is, when ref in u evaluates to more than one update point. For every node affected by the modification, will be generated one modification U_{ij}. Since by Theorem 6.5 all tuples in *evalRel(x)* are in $v_1 \bigcup ... \bigcup v_n$, then *evalRel(apply(x, u))* $\subseteq v'_1 \bigcup ... \bigcup v'_n$.

Deletions. Following the inverse reasoning for insertions, every subtree deleted from x makes a tuple disappear from *evalRel(apply(x, u))*s. Analogously, the translation U_{ij} of u will make that tuple disappear from $v'_1 \bigcup ... \bigcup v'_n$, so *evalRel(apply(x, u))* $\subseteq v'_1 \bigcup ... \bigcup v'_n$ holds. □

THEOREM 6.9 *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then* $v'_1 \bigcup ... \bigcup v'_n - \text{evalRel}(\text{apply}(x, u)) \subseteq \text{stubs}(\text{apply}(x, u))$.

PROOF. Insertions of incomplete subtrees or deletions of incomplete subtrees may cause tuples to be filled in with nulls because of the LEFT JOINS in some v'_i. These tuples, however, will be in *stubs*. The reasoning is the same as in proof of Theorem 6.7. □

B. ALGORITHMS TO TRANSLATE UPDATES

This section presents algorithms to translate insertions, deletions and modifications from the XML view to updates over the corresponding relational views. Such algorithms are mentioned in Section 6.2.

B.1 Insertions

```
translateInsert(V, qt, ref, Δ)
//Inserts Δ in the XML view V using ref as insertion point. Δ must be inserted under every node
  resulting from the evaluation of ref in V. qt is the query tree.
//Assumes that view(n) returns the name of the rel. view associated with node n

Let p be the unqualified portion of ref concatenated with the root of Δ
Let m be the node resulting from the evaluation of p against qt
Let N be the set of nodes resulting from the evaluation of ref in V
for each n in N
   if abstract_type(m) = τN
      generateInsertSQL(view(m), root(Δ), n, V)
   else Let X be the set of nodes of abstract type τN in Δ
     for each x in X
       generateInsertSQL(view(x), x, n, V)
     end for
   end if
end for

generateInsertSQL(RelView, r, InsertionPoint, V)
//Inserts the subtree rooted at r into RelView
sql = "INSERT INTO" + RelView + getAttribuelView)
sql = sql + " VALUES ("
for i = 0 to getTotalNumberAttributes(RelView) - 1
   att = getAttribute(RelView, i)
   if att is a child n of r
      sql = sql + getValue(n)
   else Find att in V, starting from InsertionPoint examining the leaf nodes
         until V's root is found
```

```
      Let the node found be m
      sql = sql + getValue(m)
   end if
   if i < getTotalNumberAttributes(RelView) - 1
      sql = sql + ", "
   else sql = sql + ")"
   end if
enf for
```

B.2 Modifications

<u>translateModify(V, qt,ref,Δ)</u>

```
Let p be the unqualified portion of ref
Let m be the node resulting from the evaluation of p against qt
if abstract_type(m) = τ_N
   r = m
else Let r be the ancestor of m whose abstract type is τ_T , τ_G or τ_N
end if
if abstract_type(r) = τ_N
   generateModifySQL(view(r), Δ, ref)
else Let X be the set of nodes with abstract type τ_N under r
   for each x in X
      generateModifySQL(view(x), Δ, ref)
   end for
end if
```

<u>generateModifySQL(RelView, Δ, ref)</u>

```
sql = "UPDATE " + RelView + " SET "
Let t be the terminal node in ref
sql = sql + t + "=" + Δ
for each filter f in ref
   if f is the first filter in ref
      sql = sql + " WHERE " + f
   else sql = sql + " AND " + f
   end if
end for
```

B.3 Deletions

<u>translateDelete(V, qt,ref)</u>

```
//Deletes the subtree rooted at ref from V

Let p be the unqualified portion of ref concatenated with the root of Δ
Let m be the node resulting from the evaluation of p against qt
if abstract_type(m) = τ_N
   generateDeleteSQL(view(m), ref)
else Let X be the set of nodes of abstract type τ_N under m
   for each x in X
      generateDeleteSQL(view(x), ref)
   end for
end if
```

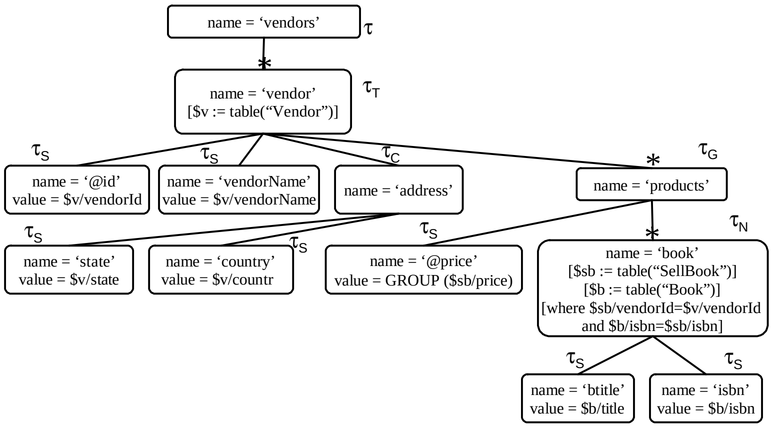
<u>generateDeleteSQL(RelView, ref)</u>

```
sql = "DELETE FROM " + RelView
for each filter f in ref
   if f is the first filter in ref
      sql = sql + + " WHERE " + f
   else sql = sql + " AND " + f
   end if
end for
```

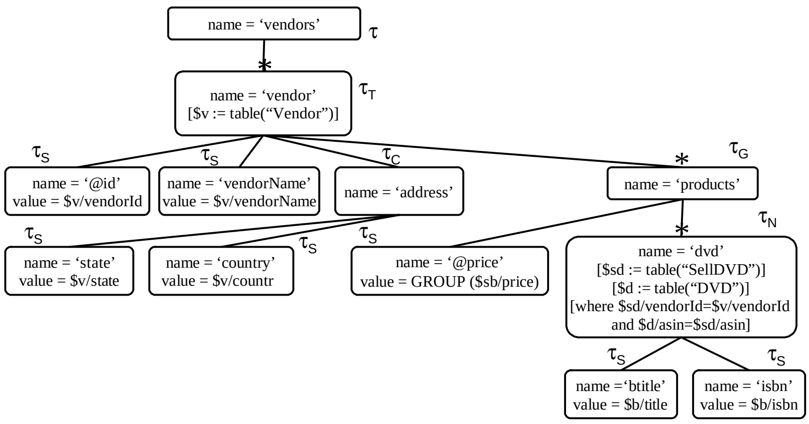
C. PARTITIONED QUERY TREES

In this section, we present the partitioned query trees corresponding to the application of algorithm *split* to the query tree of Figure 11.

Partitioned query tree for $\tau_N(book)$:



Partitioned query tree for $\tau_N(dvd)$:



Received April 2005 Revised October 2005 Accepted April 2006