Active XQuery

Angela Bonifati Daniele Braga Alessandro Campi Stefano Ceri
Dipartimento di Elettronica e Informazione, Politecnico di Milano
Piazza Leonardo da Vinci 32, I-20133 Milano, Italy
bonifati/braga/campi/ceri@elet.polimi.it

Abstract

Besides being adopted as the new interchange format for the Internet, XML is finding increasing acceptance as a native data repository language. In order to make XML repositories fully equipped with data management capabilities, suitable query and update languages are being developed. However, once the user is allowed to perform updates, it is perceivably necessary to guarantee the correctness of his/her updates, especially if document validity or semantic constraints are violated. We address this problem by exploiting the well-grounded concept of active rules.

In this paper, we propose Active XQuery, an active language for XML repositories that is based on a previously defined XQuery update model. In particular, we present the syntax and semantics of our language, aiming at emulating the trigger definition and execution model of SQL3. An active extension of XQuery arises nontrivial problems, related to the need of interleaving updates and triggers. These problems have led us to define an algorithm for update reformulation and to devise a compact semantics. In conclusion, the paper presents an architecture for rapid prototyping, and hints optimization and research issues.

1. Introduction

XQuery has emerged as the W3C-proposed standard query language for XML, and time is ripe for the database research community to study the issues involved with extending XQuery with advanced reactive capabilities.

Due to its nonprocedural nature, SQL has gained tremendous popularity for developing data-intensive applications. Relational vendors have a still growing commercial interest in supporting active features [CCW00], therefore in designing *Active XQuery* we tried to adhere to the spirit and practice of trigger definition and execution model of the SQL3 standard.

Our work capitalizes on previous efforts on update languages for XML and semi-structured data [TI*01, AQ*97]. Unfortunately, the W3C has not yet proposed an XQuery syntax for updates, although such extension is strongly needed. Meanwhile, proprietary solutions for updates of XML documents have been envisioned by tool vendors, both supporting native XML repositories [Tamino, eXcelon] or relational-based implementations [Oracle, IBM, Microsoft]. For this paper, we adopted the XQuery update extension proposed by [TI*01], that is simple and essential. Quite luckily, such language is based on few, fundamental abstractions; therefore our work will be easily adapted to a forthcoming standardization of XQuery updates based on similar abstractions.

Compared to relational updates, XQuery updates can be seen as *bulk* statements, since they may involve arbitrarily large fragments of documents, which are inserted or dropped by means of a single statement. These may trigger active rules which monitor events relative to *internal* portions of such fragments. Thus, the main difficulty in extending the notion of triggers from the relational domain to hierarchical data is indeed due to the different granularity of update events and rule events. To overcome this difficulty we have defined an algorithm that expands bulk statements into a collection of equivalent statements, each one relative to a smaller fragment, so as to guarantee that any trigger defined for the document will be correctly considered. Each of these statements is in turn a self-standing XQuery update.

The paper provides the following important contributions:

1. We propose an active extension to the W3C-proposed standard XQuery language, adapting the SQL3 notions of BEFORE vs AFTER triggers as well as the notion of ROW vs STATEMENT level granularity. W.r.t. SQL3, we review the definition of conflict set, trigger execution context and order of execution, in order to suitably adapt to the hierarchical nature of XML data.

2. We propose an algorithm for transforming bulk statements into expanded statements centered on smaller fragments; after this transformation, a procedure correctly invokes triggers and expanded updates.

3. We propose an architecture for the rapid prototyping of rules on top of an XQuery engine supporting updates, outline optimization options, and point to research problems.

1.1 Background

Querying XML documents based on their semantic content has been extensively studied within the database and semi-structured data communities and, ultimately, within the W3C. Once established, query languages have a natural extension in supporting content-based updates or in extracting views of XML documents. As exemplars of update languages for semi-structured data and for XML data respectively, we consider Lorel and XQuery. Lorel [AQ*97] allows the creation and modification of new atomic and complex objects, the creation and deletion of database names, and the bulk loading of a database. XQuery [XQ01] is the W3C proposal of a standard query language, and has been extended to support updates as a result of a research work [TI*01]. XQuery update operations include deletion, insertion, replacement and renaming of XML data.

Active rules to enforce the correctness of update operations and to automatically maintain views of data has been extensively studied in database systems [CCW00]. Many research projects provided substantial contributions to the the field of active databases (among others, Starburst [Wi96], Hipac [DBC96], Reach [BBK*92], Sentinel [CAM93], and IDEA [CF97]); we adopted SQL3 [CKM99] as the guide for our language definition for two reasons. First, we feel that SQL3 execution model is the most used in commercial systems. Second, we have found that, upon the extensions that we envision, this model is suitable for XQuery.

Although until today we could not focus on XQuery (a recent W3C standard), we already dedicated some of our previous work to active XML rules. In [BCP00] we have proposed active extensions of Lorel and XSLT in order to propose the use of active rules for the implementation of e-services, such as personalized delivery of information and push technology. Push technology, applied to a distributed environment, has been also discussed in [BCP01], which shows how standard DOM events and the SOAP interchange format can be used to implement a mechanism for dispatching rules from the rule repository to the XML repositories spread over Internet.

Other references address the use of reactive components for building e-commerce applications. Among them, a view specification language (in the OQL style) equipped with active capabilities has been defined in [AC*99]. The actors involved in an electronic commerce application might need different views of the repository data, and these are encoded through a set of activity specifications, methods and triggers. Enhanced mechanisms for notification, access control and logging/tracing of user activities are provided. Here active rules are application-specific and use a set of proprietary method calls, defined within the views.

1.2 Outline

The paper is organized as follows. Section 2 proposes motivational examples and provides a bird's-eye view of XQuery triggers. Section 3 presents the syntax of the language. Section 4 defines the semantics by describing the expansion algorithm, the execution model and the system architecture. Section 5 outlines open research issues.

2 A bird's-eye view of XQuery triggers

We give an intuitive overview of our proposal at work through a simple example. Let us assume a scenario based on the following Lib.xml document, that belongs to the XML repository of a university library::

```
<Library>
  ...
  <Shelf nr="45">
    <Book id="AO97">
      <Author> J. Acute </Author>
      <Author> J. Obtuse </Author>
      <Title> Triangle Inequalities </Title>
    </Book>
    <Book id="So98">
      <Author> A. Sound </Author>
      <Title> Automated Reasoning </Title>
    </Book>
    ...
  </Shelf>
  ...
</Library>
```

An example of update to the library is the bulk insertion of a whole shelf (nr. 45) into the document by means of the following XQuery update statement (s_0). The new library content is extracted from a collection of new shelves, located in a separate document (within the repository). In order to insert fragment `$frag` the language requires to envelop the actual `INSERT` operation into an external `UPDATE` clause, targeted to a variable that is bound to the element that will *contain* the fragment (node `$target`). Recursively nested update statements (and therefore `UPDATE` clauses) are allowed within the curly brackets.

```
s0:
FOR $target IN document("Lib.xml")/Library,
    $frag   IN document("New.xml")/Shelves/Shelf
WHERE $frag/@nr="45"
UPDATE $target  {  INSERT $frag  }
```

In our scenario, the library automatically maintains an index with a list of all authors, keeping pointers (IDREFs) to

the library entries. The index is part of the library document, whose complete DTD is:

```
<!ELEMENT Lib (Shelf+, AuthorIndex)>
<!ELEMENT Shelf (Book*)>
<!ATTLIST Shelf  nr  ID  #REQUIRED>
<!ELEMENT Book (Author+, Title)>
<!ATTLIST Book  id  ID  #REQUIRED>
<!ELEMENT Author (#PCDATA)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT AuthorIndex (AuthorEntry*)>
<!ELEMENT AuthorEntry (Name, PubsCount)>
<!ATTLIST AuthorEntry  uni   CDATA   #IMPLIED
                       pubs  IDREFS  #IMPLIED>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT PubsCount (#PCDATA)>
```

The following XML excerpt demonstrates the author index:

```
<AuthorIndex>
 ...
 <AuthorEntry uni="PoliMi" pubs=".. AO97 ..">
  <Name> J. Acute </Name>
  <PubsCount> ... </PubsCount>
 </AuthorEntry>
 ...
 <AuthorEntry uni="Princeton" pubs=".. So98 ..">
  <Name> A. Sound </Name>
  <PubsCount> ... </PubsCount>
 </AuthorEntry>
 ...
</AuthorIndex>
```

Triggers are responsible to guarantee referential integrity among the authors' publications ('pubs' attribute) and the books in the library. In particular, we want to guarantee the following properties:

1. **No dangling references**: *deletion of a 'Book' element causes all its authors (listed in the index) to lose "dangling" references to that publication.*
2. **Automatic indexing**: *insertion of a 'Book' element causes a new reference to be inserted in all index entries that represent new book's authors.* This may require new 'AuthorEntry' elements to be added to the index.

Automatic deletion of dangling pointers is performed by trigger `NoDangle`, that updates 'AuthorEntry' elements removing from their 'pubs' attributes all references[1] to the deleted book (identified by keyword `OLD_NODE`):

```
CREATE TRIGGER NoDangle
AFTER DELETE OF document("Lib.xml")//Book
FOR EACH NODE
DO ( FOR
  $AutIndex IN document("Lib.xml")//AuthorIndex,
  $MatchAut IN $AutIndex/AuthorEntry
                      [Name = OLD_NODE/Author],
  $DangRef  IN $MatchAut/ref(pubs, OLD_NODE/@id)
UPDATE $AutIndex { DELETE $DangRef } )
```

[1]Note that bindings to a single IDREF within an IDREFS attribute are declared according to the syntax extension proposed in [TI*01].

Two other triggers perform the insertion of new references and new 'AuthorEntry' elements. If one of the authors of the new book is not yet in the list, the higher-prioritized trigger `AddNewEntry` inserts a new "empty" 'AuthorEntry' element. Thus, low-prioritized trigger `AddNewReference` can assume that the index already contains entries for all the incoming authors.

```
CREATE TRIGGER AddNewEntry
AFTER INSERT OF document("Lib.xml")//Book
FOR EACH NODE
LET $AuthorsNotInList := (
  FOR $n IN NEW_NODE/Author
  WHERE empty(//AuthorIndex/AuthorEntry[Name=$n])
  RETURN $n )
WHEN ( not( empty($AuthorsNotInList )) )
DO ( FOR $ai IN document("Lib.xml")//AuthorIndex,
          $NewAuthor IN $AuthorsNotInList
     UPDATE $ai
     { INSERT <AuthorEntry>
                <Name> {$NewAuthor/text()} </Name>
                 <PubsCount> 0 </PubsCount>
                </AuthorEntry> } )

CREATE TRIGGER AddNewReference
WITH PRIORITY -10
AFTER INSERT OF document("Lib.xml")//Book
FOR EACH NODE
DO ( FOR $ai IN document("Lib.xml")//AuthorIndex,
          $a  IN $ai/AuthorEntry[Name=$a]
     UPDATE $a
     { INSERT new_ref(pubs, NEW_NODE/@id)} )
```

Finally, triggers `IncrementCounter` and `DecrementCounter` maintain a counter of authors' publications (we only show `IncrementCounter` for brevity).

```
CREATE TRIGGER IncrementCounter
AFTER INSERT OF //new_ref(pubs)
FOR EACH NODE
LET $Counter := NEW_NODE/../PubsCount
DO ( FOR $AuthorEntry IN NEW_NODE/..
     UPDATE $AuthorEntry
     { REPLACE $Counter WITH $Counter + 1 } )
```

These triggers demonstrate that the execution of the action part of a trigger can cause the activation of other triggers (`AddNewReference` triggers `IncrementCounter`).

3 Syntax of Active XQuery

An XQuery trigger consists of four components: the *triggering operation*, the *triggering granularity*, the *trigger condition* and the *trigger action*. Consistent with the terminology of [WC96], a trigger is *triggered* when one of its triggering operations occur, it is being *considered* when its condition is under evaluation, it is *executed* when its action is performed. When the trigger consideration starts, it is also *de-triggered*.

The syntax of an XQuery trigger is the following:

```
CREATE TRIGGER Trigger-Name
[WITH PRIORITY Signed-Integer-Number]
(BEFORE | AFTER)
  (INSERT | DELETE | REPLACE | RENAME)+
    OF XPathExpression ( , XPathExpression)*
[FOR EACH (NODE | STATEMENT)]
[XQuery-Let-Clause]
[WHEN XQuery-Where-Clause]
DO (XQuery-UpdateOp | ExternalOp)
```

- The `CREATE TRIGGER` clause is used to define a new XQuery trigger, with the specified name.
- Rules can be prioritized in an absolute ordering, expressed with an optional `WITH PRIORITY` clause, which admits as argument any signed integer number. If this clause is omitted, the default priority is zero.
- The `BEFORE/AFTER` clause expresses the triggering time relative to the operation.
- Each trigger is associated with a set of update operations (insert, delete, rename, replace), adopted from the update extension of XQuery [TI*01].
- The operation is relative to elements that match an XPath expression (specified after the `OF` keyword), i.e. a step-by-step path descending the hierarchy of documents (according to [XPa99] and its update-related extensions[2]). One or more predicates (XPath *filters*) are allowed in the steps to eliminate nodes that fail to satisfy given conditions. Once evaluated on document instances, the XPath expressions result into sequences of nodes, possibly belonging to different documents.
- The optional clause `FOR EACH NODE/STATEMENT` expresses the trigger granularity. A *statement-level* trigger executes once for each set of nodes extracted by evaluating the XPath expressions mentioned above, while a *node-level* trigger executes once for each of those nodes. Based on the trigger granularity, it is possible to mention in the trigger the transition variables:
  - If the trigger is node-level, variables `OLD_NODE` and `NEW_NODE` denote the affected XML element in its before and after state.
  - If the trigger is statement-level, variables `OLD_NODES` and `NEW_NODES` denote the sequence of affected XML elements in their before and after state.
- An optional *XQuery-Let-Clause* is used to define XQuery variables whose scope covers both the condition and the action of the trigger. This clause extends the 'REFERENCING' clause of SQL3, because it can be used to redefine transition variables.
- The `WHEN` clause represents the trigger condition, and can be an arbitrarily complex XQuery where clause. If omitted, a trigger condition that specifies `WHEN TRUE` is implicit.
- The action is expressed by means of the `DO` clause, and it can contain accomplished through the invocation of an arbitrarily complex update operation. In addition, a generic *ExternalOp* syntax indicates the possibility of extending the XQuery trigger language with support to external operations, permitting, e.g., to send mail or to invoke SOAP procedures; such extensions are outside the scope of this paper.

For a complete syntax of XQuery refer to [XQ01]. For the syntax of the update language, refer to [TI*01].

4 Semantics of Active XQuery

4.1 Intuitive semantics

In our view, the intuitive semantics of XQuery triggers should be as close as possible to the semantics of SQL3 triggers, as discussed in [CKM99]. Accordingly, each XQuery operation should be computed in the context of a recursive procedure, such that:

- At the time of execution of an update, the set of affected nodes is computed (leading to the binding of transition variables).
- A given update statement is preceded by `BEFORE` triggers and followed by `AFTER` triggers[3]; statement-level and node-level triggers may interleave, and are considered in priority order.
- If a given trigger executes an operation and this in turn causes some triggering, the trigger execution context is suspended, and a new procedure is recursively invoked; recursion depth of recursion is limited by some given threshold, which is system specific.

However, such intuitive semantics cannot be immediately replicated for XQuery, due to the hierarchical structure of

[2] The additional keyword `ref`, introduced in [TI*01], can be used to denote a single IDREF within an attribute of type IDREFS.

[3] In order to avoid nondeterministic and/or nonmonotonic behavior, `BEFORE` triggers may be subject to limitations in their actions; such limitations are still a subject of studies in the SQL standards [CKM99] and are typically implemented in relational systems by suitable exceptions (e.g., Oracle's "mutating table exception", see [WC96]).

XML and the "bulk" nature of update primitives. According to the update language of [TI*01], the insertion of "content" may refer to an arbitrarily large XML fragment, and likewise the deletion of a node may cause the dropping of an arbitrarily large XML fragment. Note that the SQL language supports instead updates operations targeted to tuples of a given table. Therefore, a precise description of the semantics of XQuery triggers requires to be combined with a management strategy for bulk updates.

4.2 Update expansion

Our strategy with bulk updates consists of decomposing each original XQuery bulk update s_0 into a sequence S of smaller granularity updates, such that the change to each XML element involved in the update is addressed by a self-standing update operation of S. This strategy requires the definition of two separate mechanisms, one for expanding updates and one for executing them, where the latter includes the composition of updates with triggers. Note that statement expansion requires accessing the affected XML data; this is obvious in the case of bulk deletions (when the specific elements to be deleted need to be first accessed), but occurs as well with bulk insertions.

An alternative semantics would transform s_0 into another single update statement, consisting of nested "smaller" updates, such that each element of the XML content affected by the original bulk statement would be explicitly inserted or deleted by a "small" update operation. In this case, composition of triggers and updates occurs as well, but triggers are invoked within the bulk update execution.

After a careful trade-off analysis [Bon01] we have excluded such alternative, because it requires a tight integration between the trigger engine and the XQuery optimizer, while the decomposition-based model (that we adopted) leads to a good separation between the two system components. Thus, our approach can be easily supported "on top" of an existing XQuery optimizer, in the same way as a trigger engine can be easily supported on a relational storage system [Wi96].

Other advantages characterize our approach:

- With the decomposition-based model, matching triggering events to update operations is trivial: triggering occurs when the trigger's update operation is executed.
- The composition of triggers with updates is also rather simple, and takes place by means of a recursively defined statement execution procedure which is very similar to the one discussed in 4.1.
- As with SQL3, the set of nodes affected by an update is computed by the decomposer, before performing the update, i.e. according to the semantics of the original user-level update primitive.

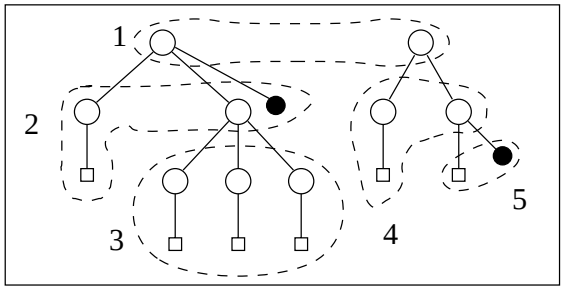

Figure 1. Visit of an XML fragment

However, we are aware of disadvantages as well:

- An inevitable one is that any decomposition of a bulk statement into a sequence of statements leads to exposing intermediate states - the ones left by a prefix of the decomposition sequence; therefore, the decomposition strategy affects the *semantics* of triggers, as this in turn is order-dependent. This is inevitable: order dependence characterizes even relational systems and is amplified by a hierarchical structure which can be processed in many ways. Some degree of order dependency, however, is present in both approaches.
- The second disadvantage is that having multiple independent and autonomous statements might lead to repeated executions of the same tree traversals in subsequent statement executions. In practice, this disadvantage is minor, because transparent caching mechanisms should make such traversals very efficient; moreover, simple optimizations could be done, such as preserving pointers to already computed nodes from one statement execution to the next one.

In designing the expansion strategy, we use a visit of the hierarchical structures which mimics the "natural" order of update propagation, in which inserts proceed top-down, and deletions proceed bottom-up. Such a visit strategy is described, in a simple case of bulk insert, in Figure 1. The adopted notation displays attributes as black circles, elements as empty circles and PCDATA content as empty boxes; dashed lines indicate the nodes that are treated together; fragments are visited in a mixed breadth-depth order, indicated by numbers. In the first step of the algorithm, the first-level elements are visited and grouped into a common update statement: root nodes need to be treated separately, since they lack a common ancestor.

4.3 Expansion Algorithm

In the following we detail the expansion process for `INSERT` statements. `DELETE` statements expansion is analogous (and omitted for brevity), while `REPLACE` statements can be rewritten in terms of `INSERT` statements immediately followed by suitable `DELETE` statements. Instead, `RENAME` operations perform mere name changes and do not require further expansion.

Essentially, expansion proceeds with an intermixed depth-first and breadth-first visit of the involved fragments. In particular, all the operations relative to the insertion of XML nodes with a common father are enveloped in the same update statement. Among these nodes (that can be attributes, elements or PCDATA content), attributes are inserted first, then elements and PCDATA content are inserted in their proper order. Elements are inserted as empty couples of tags if they contain a complex structure (and thus require further expansion), otherwise they solely have PCDATA content, and they are inserted together with such content.

The function $buildContentOfFirstUpdate$ addresses the construction of the first expanded statement, targeted to the same variable as the user update operation (referenced in the external `UPDATE` clause). It takes the bulk statement ST as argument and outputs one update statement that inserts all the roots of the involved fragments. Thereafter, the algorithm starts from each complex root node and expands its complex sub-elements. This expansion is entrusted to the function $expandNode$, which is then recursively invoked on all complex sub-elements of these elements.

Algorithm 4.1 Bulk Statement Expansion. *Given an arbitrary bulk XQuery update statement ST, the algorithm returns UL, an ordered list of XQuery expanded update statements and directives.*

In a preprocessing phase, the algorithm computes the needed data structures. Precisely, variable V represents the argument of the user update clause; variable S contains the bindings to the involved fragments; $FClause$ and $WClause$ are the for and where clauses of ST; $XFClause$ is an ad-hoc clause that is repeatedly used in the construction of the results; cur_path is a variable that is assigned to the current path, as soon as it is available by the visit of the fragments; variable $name$ is used to name the generated statements and match them with their corresponding directives.

This version of the algorithm takes care of the expansion of insert statements, and accepts for simplicity only flat user statements. Possible nested user updates can be treated via previous reduction to a list of flat statements.

```
begin
  V = getUpdateVariable(ST)
  S = bindInvolvedFragments(ST)
  FClause = getFORClause(ST)
  WClause = getWHEREClause(ST)
  XFClause = FClause + ", $curFrag IN " + V +
             "/*[empty(" + V + "/*[AFTER $curFrag])]"
  cur_path = "$curFrag"
  name = buildUniqueName(cur_path)
  UL = "EvalBefore(" + name + ") " + "Name:" + name + " "
  UL += FClause + WClause
  UL += "UPDATE " + V + "{ "
  UL += buildContentOfFirstUpdate(ST) + "} "
  for each fragment in S, consider again its root node N:
      if ( N is complex and requires further expansion )
        then UL += expandNode(N, cur_path, XFClause, WClause)
      UL += "EvalAfter(" + name + ") "
  return UL
end;

FUNCTION expandNode(Node N, String cur_path,
                    String XFClause, String WClause)
RETURNS OUT, an ordered list of update statements and directives
begin
  name = buildUniqueName(cur_path)
  OUT = "EvalBefore(" + name + ") " + "Name:" + name + " "
  if ( cur_path="$curFrag" )
    then OUT += XFClause + WClause +
                "UPDATE $curFrag { "
    else OUT += XFClause + ", $cur_node IN " + cur_path +
                WClause + "UPDATE $cur_node { "
  for each attribute A of N
    OUT += "INSERT new_attribute( "
    OUT += A.name + ", " + A.value + ") "
  for each subelement C of N
    if ( C is an XML-Element )
      then if ( C has only PCDATA content )
             then OUT += "INSERT " + buildTagWithPCDATA(C)
             else OUT += "INSERT " + buildEmptyTag(C)
      else OUT += "INSERT " + C.content
  OUT += "} "
  for each subelement C of N:
    if ( C is complex and requires further expansion )
      then cur_path += "/*[" + position of C + "]"
           OUT += expandNode(C, cur_path, XFClause, WClause)
  OUT += "EvalAfter(" + name + ") "
  return OUT
end;
```

Moreover, the algorithm interleaves the statements with special directives to the rule engine that enable the construction of conflict sets. In particular, the directive $EvalBefore$ contains the name of the statement that it precedes, and the directive $EvalAfter$ contains the name of the statement that it follows. The positions of these directives within the list of statements reflect the intrinsic semantics of the original statement. If we consider an `INSERT` operation, the $EvalBefore$ directives precede each expanded statement, while $EvalAfter$ directives solely follow the expanded statement of the leaf portions of the fragment. The $EvalBefore$ directives that refer to the remaining (non-leaf) portions are postponed by recursion and follow the directive of the last leaf of the fragment, as shown below.

Example 4.2 *We consider the expansion of the bulk statement s_0 (from section 2), that inserts an entire shelf in the document Lib.xml. We need to expand s_0 into smaller self-standing update statements, in order to make the defined triggers sensitive to the insertion of the children of the 'Shelf' element. The expansion algorithm outputs the following sequence:*

```
EvalBefore(s1)

Name:s1
FOR $x IN document("Lib.xml")/Library,
 $frag IN document("New.xml")/Shelves/Shelf[@nr="45"]
UPDATE $x
{  INSERT <Shelf/>  }

EvalBefore(s2)

Name:s2
FOR $x IN document("Lib.xml")/Library,
 $frag IN document("New.xml")/Shelves/Shelf[@nr="45"],
 $curfragment IN $x/*[empty($x/*[AFTER $curfragment])]
UPDATE $curfragment
{  INSERT new_attribute(nr, "45")
   INSERT <Book/>
   INSERT <Book/>  }

EvalBefore(s3)

Name:s3
FOR $x IN document("Lib.xml")/Library,
 $frag IN document("New.xml")/Shelves/Shelf[@nr="45"],
 $curfragment IN $x/*[empty($x/*[AFTER $curfragment])],
 $cur_node IN $curfragment/*[1]
UPDATE $cur_node
{  INSERT new_attribute(id, "AO97")
   INSERT <Author> J. Acute </Author>
   INSERT <Author> J. Obtuse </Author>
   INSERT <Title> Triangle Inequalities </Title>  }

EvalAfter(s3)

EvalBefore(s4)

Name:s4
FOR $x IN document("Lib.xml")/Library,
 $frag IN document("New.xml")/Shelves/Shelf[@nr="45"],
 $curfragment IN $x/*[empty($x/*[AFTER $curfragment])],
 $cur_node IN $curfragment/*[2]
UPDATE $cur_node
{  INSERT new_attribute(id, "So98")
   INSERT <Author> A. Sound </Author>
   INSERT <Title> Automated Reasoning </Title>  }

EvalAfter(s4)

EvalAfter(s2)

EvalAfter(s1)
```

The order of evaluation of triggers corresponds to the intuitive order that could be expected by a user unaware of the decomposition process. This can be appreciated if one considers the hierarchy $s_1 < s_2 < \{s_3, s_4\}$ and the order of execution of before and after triggers. For instance, the "`AFTER INSERT`" triggers triggered by s_2 (referred to by the directive `EvalAfter(s2)`) see the side effects not only of statements s_3 and s_4, but also of all trigger executions caused by them.

4.4 Description of trigger execution model

Given the update decomposition algorithm, we can now define the trigger execution model precisely, thus giving an operational semantics of the combined execution of updates and triggers. As observed, our query execution model is inspired to the SQL3 execution model, however adapted to the hierarchical nature of XML, and exploiting the expansion of statements.

Assume the XQuery engine is starting the execution of a generic bulk update statement S, which has been submitted by the user. The following algorithm defines this semantics operationally.

```
PROCEDURE EXECUTE_STATEMENT(Statement S)
1 Call EXPAND_STATEMENT(S) and store the
  returned structures (RF and SIL)
2 For each item Ii in SIL, if it is
2.1 an 'EvalBefore' instruction, call
    COMPUTE_BEFORE_CONFLICT_SET(Sn, RF),
    where Sn is the statement related to Ii
2.2 an 'EvalAfter' instruction, call
    COMPUTE_AFTER_CONFLICT_SET(Sn, RF),
    where Sn is the statement related to Ii
2.3 an update statement, execute Ii,
    updating the XML repository

PROCEDURE EXPAND_STATEMENT(Statement S)
RETURNS FragmentSequence RF,
        StatementInstructionList SIL
1 Retrieve RF, the set of fragments that are
  relevant for the execution of S
2 Expand S into SIL by visiting RF with the
  expansion algorithm
3 Return SIL and RF (NEW_RF and/or OLD_RF)

PROCEDURE COMPUTE_BEFORE_CONFLICT_SET
     (Statement Sn, FragmentSequence RF)
1 Compute BT, the set of eligible
  BEFORE triggers activated by Sn
2 Order all computed triggers according to
  their global ordering
3 For each trigger T in BT, PROCESS_TRIGGER(T)

PROCEDURE COMPUTE_AFTER_CONFLICT_SET
          (Statement Sn, FragmentSequence RF)
1 Compute AT, the set of eligible AFTER
  triggers activated by Sn
2 Order all computed triggers according
  to their global ordering
3 For each trigger T in AT, PROCESS_TRIGGER(T)

PROCEDURE PROCESS_TRIGGER(trigger T)
1 Calculate pointers corresponding to
  NEW_NODE(S) and OLD_NODE(S).
2 If any, valuate the Let clause of T and bind
  the new variables.
3 Evaluate the condition C of T
4 If C evaluates to TRUE, EXECUTE_STATEMENT(A)
  (A being the action of T)
```

In the former algorithm `EXECUTE_STATEMENT` is invoked on `S`. The expansion algorithm 4.1 retrieves the set

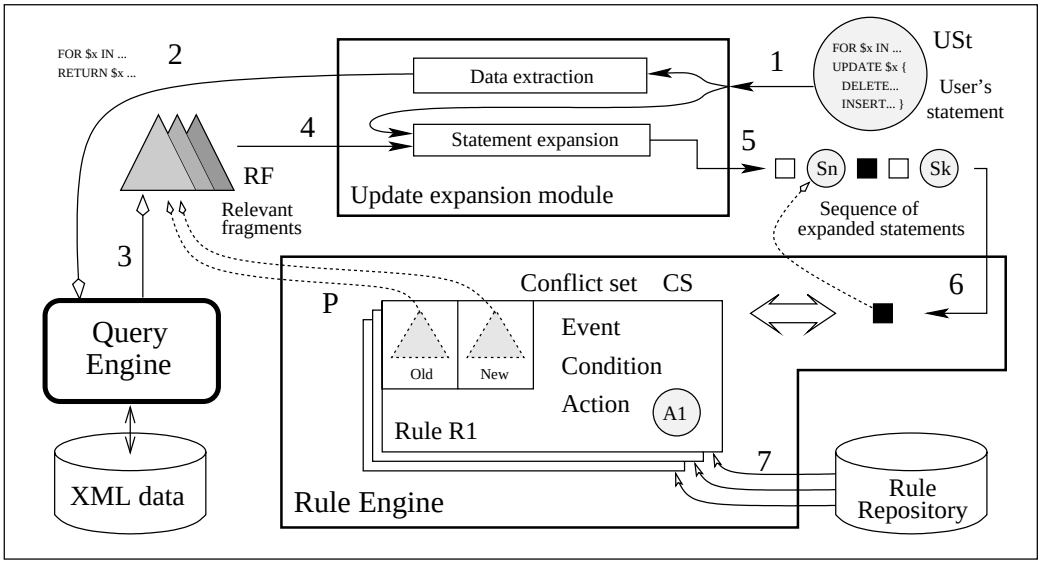

Figure 2. System architecture

of relevant fragments (RF, involved by S), and produces a sequence of statements and directives (SIL). SIL is generated according to the mixed depth-breadth order visit of the fragments, and this is accomplished by calling procedure EXPAND_STATEMENT. The algorithm needs to communicate with the query engine in order to inspect XML data and to build RF, which constitutes a separate structure. More precisely, REPLACE operations will yield both NEW_RF and OLD_RF structures, deletions will produce only OLD_RF structures and insertions only NEW_RF structures. These structures are accessed by rules in order to bind transition variables, as explained below. The last task of EXPAND_STATEMENT is to return SIL and to rename RF as NEW_RF and/or OLD_RF (depending on the type of original statement S).

Once the EXPAND_STATEMENT has been completed, each item I_i in SIL is one of the following mutually exclusive cases. If the item is an $EvalBefore$ directive, then the procedure COMPUTE_BEFORE_CONFLICT_SET is invoked; if it is an $EvalAfter$ directive, then the procedure COMPUTE_AFTER_CONFLICT_SET is invoked; otherwise, it is an expanded statement, ready to be executed. Both COMPUTE_BEFORE_CONFLICT_SET and COMPUTE_AFTER_CONFLICT_SET receive as parameters the statement S_n referred by the directive and RF.

Procedures COMPUTE_BEFORE_CONFLICT_SET and COMPUTE_AFTER_CONFLICT_SET respectively calculate BT and AT, the sets of triggered rules. This defines the conflict sets pertaining to statement S_n, which include both statement level and node level triggers, in priority order. For each trigger T in these conflict sets, PROCESS_TRIGGER is executed with T as parameter. Local pointers are calculated in order to bind OLD_NODE(S) and NEW_NODE(S) transition variables (they may have been used in a LET clause of T). When the condition evaluates to true, the action of T is executed by invoking EXECUTE_STATEMENT and recursively iterating the execution. When a new statement is executed, the current trigger execution context is suspended, and the entire process restarts.

During the processing of bulk statements, data is maintained in two places: the XML repository and the separate global fragments NEW_RF and/or OLD_RF, which are pointed to by local pointers (so we have a useful structure sharing that avoids redundance). The XML repository is persistent, while the trigger execution contexts and the global fragments are stored only until the execution of the bulk statement is completed.

4.5 System architecture

Figure 2 summarizes all the components of our proposed architecture. The update expansion module is responsible for the transformation of bulk statements [1]. It consists of two layers that operate in a cascading sequence.

The data extraction layer first instantiates a query upon the query engine [2]. This query is directly drawn from the user update; precisely, the ForClause, LetClause and WhereClause of the update are directly replicated in the query. Then, the query accesses the XML repositories to retrieve the relevant fragments RF [3] that are taken as inputs [4] by the statement expansion layer. This layer is responsible for the effective instantiation of the expansion algorithm

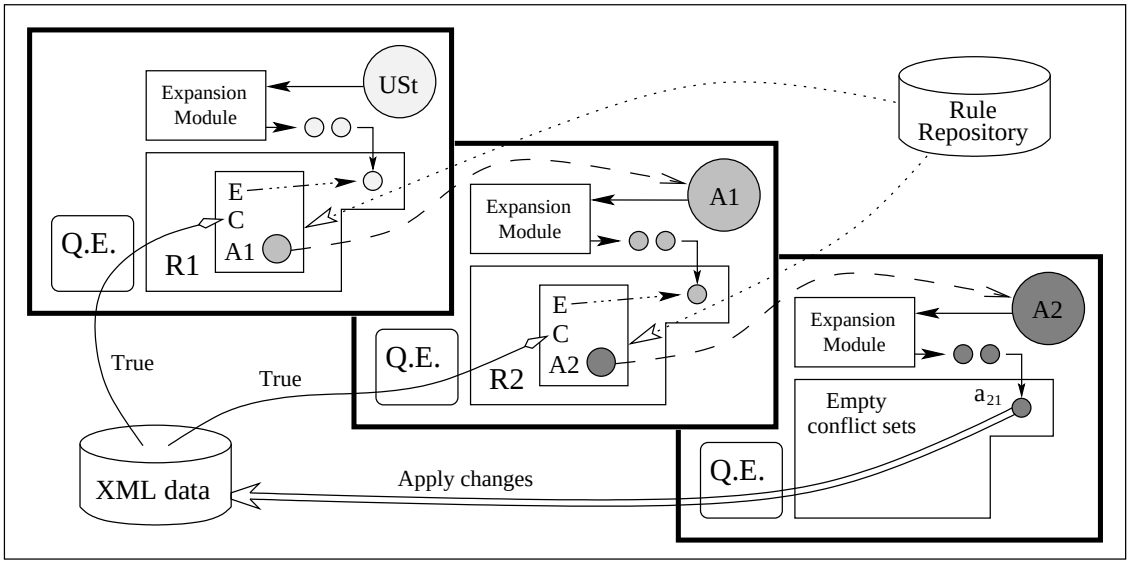

Figure 3. Recursive rule execution

over the user update. The result is the list of statements and directives [5](displayed as circles and boxes in the figure) that have been given as outputs by the algorithm.

Directives [6] are commands directly issued to the rule engine; they schedule the times at which the the conflict sets are calculated. The rule engine searches the rule repository [7] for rules addressing the nodes on which the referred statement S_n operates and whose triggering event matches the operation type. A rule might have more than one activation, since many similar nodes can be affected by S_n, and each rule instance is provided with the two pointers `OLD_NODE(S)` and `NEW_NODE(S)`, that represent the old/new fragment(s) (one of them might be set to null).

All triggered rules are collected in a conflict set (CS), where they are ordered with respect to their priority. Note that node level and statement level rules are mixed in CS, and that CS contains rules addressing nodes with different tagnames as well, since S_n affects all subelements and attributes of a single node. Only the actions of those rules with true condition are performed.

Statement A1 can be a bulk statement itself, and thus it must be processed by a recursive replication of the abstract machine described in figure 2. If we imagine a scenario in which the execution of a rule's action causes another [cascading] rule to activate, we can represent the resulting stack of execution contexts like in figure 3. Here statement USt triggers rule R1, the action A1 of R1 is itself expanded, it triggers rule R2 and an update is eventually performed, since statement a_{21} causes no rule activation (the conflict sets related to a_{21} are empty).

5 Open Issues

Our study let us foresee several optimization and research issues, briefly discussed below.

- **Schema-driven optimization.** The expansion of bulk statements can benefit of the availability of the documents schema, either expressed as a DTD or in terms of XML Schema. In particular, it is possible to avoid expanding the updates relative to those parts of a document that do not activate rules. Such an optimization is relevant within most XML repositories, as few of their element types correspond to distinctive real-world entities and are therefore targeted to triggers.
- **Internal optimizations.** Specialized indexing techniques can be devised in order to optimize the rule engine for repeated executions of the same query over different fragments, e.g. by adding data structures which point to "sibling" nodes. Such data structures can be set up by the expansion module, as it is aware of both the queries and the fragments. Other indexes may link the nodes of the XML repository to rules in the rule repository, in order to facilitate the extraction of the rules which are triggered by given operations.
- **Limitations of expressive power.** As with the SQL3 standard, the class of "legal" `BEFORE` triggers remains to be defined [CKM99]. A simple solution can be excluding from their actions any update operation or lookup into transition variables, but this in practice limits the expressive power of `BEFORE` triggers to error signalling. In many cases, however, such lookups

and updates do not cause any inconsistency; therefore, the class of "legal" triggers - maybe relative to given XML schemas or documents - remains to be defined.

- **Definition of illegal executions.** Similarly, certain classes of executions are illegal, for instance when the update performed by a trigger affects the data which have caused the triggering, thus yielding to contradictory situations. Such illegal executions should be identified, and execution-time "traps" should be instrumented in order to detect them.
- **Compile-time trigger analysis.** Trigger analysis, applied to a given XML repository and rule set, could be used to detect anomalous behavior, such as the lack of termination or confluence [WC96]; conversely, trigger analysis could be used to "validate" triggers, thus excluding that a given rule set could lead to any illegal execution. These techniques are defined for relational triggers but need to be extended for XQuery triggers.

6 Conclusions

In this paper, we have defined an extension of XQuery which enables trigger definition and management. We aimed at compatibility with SQL3, in spite of some difficulties due to "bulk" updates and to the hierarchical nature of XML; this goal has been achieved by defining a rule execution scheme based on the preliminary expansion of user-provided update statements. The main advantage produced by this approach is the clean separation between the rule engine and the query optimizer, yielding to a modular architecture in which the query optimizer can be plugged in. We plan to develop a prototype of such architecture as soon as the W3C will endorse an extension of the XQuery language supporting updates; we also hope that this article will contribute to the discussion on query language standards for XML within the W3C.

References

[AC*99] S. Abiteboul, C. Cluet, L. Mignet, B. Amann, T. Milo and A. Eyal. Active Views for Electronic Commerce. In *Proc. of the 25th VLDB*, pages 138–149, Edinburgh, Scotland, September 1999.

[AQ*97] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. Wiener. The Lorel Query Language for Semistructured Data. In *International Journal on Digital Libraries*, 1(1):66-88, April 1997.

[BBK*92] A. P. Buchmann, H. Branding, T. Kudrass, and J. Zimmermann. Reach: A Real-Time, Active and Heterogeneous Mediator System. In *IEEE Bullettin on Data Engineering*, 15(1-4):44-47, December 1992.

[BCP00] A.Bonifati, S. Ceri and S. Paraboschi. Active Rules for XML: A New Paradigm for E-Services. In *Proc. of TES Workshop*, VLDB 2000, El Cairo, Egypt, September 2000.

[BCP01] A.Bonifati, S. Ceri and S. Paraboschi. Pushing Reactive Services to XML Repositories using Active Rules. In *Proc. of WWW10*, Hong Kong, China, March 2001.

[Bon01] A.Bonifati. Reactive Services For XML Repositories. PhD Thesis, Dipartimento di Elettronica e Informazione, Politecnico di Milano, December 2001.

[CAM93] S. Chakravarthy, E. Anwar, and L. Maugis. Design and implementation of active capability for an object-oriented database. Technical Report UF-CIS-TR-93-001, University of Florida, January 1993.

[CCW00] S. Ceri, R.J. Cochrane and J. Widom. Practical Applications of Triggers and Constraints: Successes and Lingering Issues. Invited Paper. In *Proc. of the 26th VLDB*, El Cairo, Egypt, September 2000.

[CF97] S. Ceri and P. Fraternali. The IDEA Methodology. *Series on Database Systems and Applications*, Addison Wesley Publisher Ltd., May 1997.

[CKM99] R. Cochrane, K. G. Kulkarni and N. Mendonça Mattos. Active Database Features in SQL3. In N. Paton (ed.) *Active Rules in Database Systems*, pages 197-219, Springer-Verlag, 1999.

[DBC96] U. Dayal, A. P. Buchmann and S. Chakravarthy. The HiPAC Project. In *Active Database Systems*, Morgan Kauffmann, pages 177-205, 1996.

[TI*01] I. Tatarinov, Z. G. Ives, A. Y. Halevy, D. S Weld. Updating XML. In *Proc. of SIGMOD 2001*, Santa Barbara, California, May 2001.

[WC96] J. Widom and S. Ceri (editors). Active Database Systems. Morgan Kauffmann Publishers, San Francisco (CA), 1996.

[Wi96] J. Widom. The Starburst Active Database Rule System. In *IEEE TKDE*, 8(4):583-595, August 1996.

[XPa99] XML Path Language (XPath) Specification. W3C Recommendation, 16 November 1999, http://www.w3.org/TR/xpath.

[XQ01] XQuery 1.0: An XML Query Language. W3C Working Draft, last release 7 June 2001, http://www.w3.org/TR/xquery/.