Introducing Trigger Support for XML Database Systems *

© Maxim Grinev
maxim@grinev.net

Maria Rekouts
rekouts@ispras.ru

Institute for System Programming of Russian Academy of Sciences

Abstract

There is a growing number of XML database systems of different kinds now on the market. XML DBMS vendors rushed to enrich their products with more flexible and advanced features to make them satisfy the requirements of modern applications. And the time is ripe for the database research community to study the issues involved with extending XML DBMS with capabilities analogous to those that are popular in traditional DBMS, keeping in mind that XML databases now become a widespread means for storing and exchanging information on the Web, and increasingly used in dynamic applications such as e-commerce.

In this paper, being bound for the issues, we provide a definition of triggers for XML based on XQuery and a previously defined update language, and methods to support triggers in XML database systems.

1 Introduction

The concept of triggers itself originated from the early times of relational database systems. Not long after the idea of *integrity constraints* in relational databases appeared, researchers recognized the importance of automatic "reactions" to constraint violations, the idea expanded to the more general concept of triggers [12], also now known as *event-condition-action (ECA) rules*, or *active rules*. In database system, all users depend on the correctness or validity of data. Integrity or validity of data is specified by a set of *semantic constraints* or *assertions*. Triggers may be used as extended constraints to enforce validity.

The uses of triggers also occur in the context of materialization of views. An important aspect of the data definition facility of a database system is the ability to define alternative views of stored data. A view is a data object that is derived from one or more existing data objects. Triggers may be used to materialize such views. In particular, this approach was investigated in the System R [12].

Nowadays active rules are often concerned as a mean to support business logic of e-commerce applications [6]. Since such applications most often utilize XML data, the active rules support is much wanted from the underlying XML database systems.

While elaborating on the triggers for XML we confronted the problems. First of all they concerned with irregular and hierarchical nature of XML data that makes impossible to simply adopt the concept of SQL triggers [13]:

- Processing XML data implies manipulation with arbitrarily large XML fragments. Hence, an update language must provide the possibilities to update a whole subtree of XML documents, i.e. 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. Such possibilities are provided in the Sedna update language [1], concerned in this paper, and in other well-known proposals of update extension of XQuery [7]. In contrast, SQL language supports updates operations targeted to tuples of a given table. Such a "bulk" nature of update primitives causes the question: does a new XML fragment that is inserted contains a data that matches any triggers available in the system by the moment?
- Practically, an update language for XML specifies the data that are to be updated by means of XPath [10] (or XQuery [4] as more complicated way). For triggers, to determine the data associated with given trigger XPath is normally used. So, we have XPath/XQuery expression representing data that are to be updated and XPath expression representing data associated with trigger. The problem is that analyzing these two expressions at compile time it is impossible to know if the trigger needs to be triggered under given update statement, because the result of an XPath expression usually depends on data, but data are not available at compile time. Thus, triggers must be processed at the time when update statement is actually executed (i.e. at run-time). And, as in practice most likely there is an arbitrarily large number of trigger, such evaluation must be carried out in an efficient way.

1.1 Outline

The paper is organized as follows. To give a more illustrative overview of the problem we consider an example in Section 1.2. In Section 2 we give our vision of triggers for XML by providing syntax and noting the main points of semantics of triggers. In Section 3 we provide

* This work was partially supported by the grants of the Russian Foundation for Basic Research No. 04-07-08003 and 05-07-90204.

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems SYRCoDIS, St.-Petersburg, Russia, 2005

methods to support triggers in XML DBMS. Section 4 proposes a brief survey of a related work. Section 5 concludes the paper.

1.2 A First Glance at XML Triggers

This example is borrowed from [14], except some modifications that were brought in to illustrate our methods better.

Let us assume a scenario based on the following `lib.xml` document, that belongs to an XML database of a university library:

```
<library>
 ...
 <shelf nr="45">
  <book_count>2</book_count>
  <book id="AO97">
   <author>J. Acute</author>
   <author>J. Obtuse</author>
   <title>Triangle Inequalities </title>
   <year>1973</year>
  </book>
  <book id="So98">
   <author>A. Sound</author>
   <title>Automated Reasoning</title>
   <year>1990</year>
  </book>
 </shelf>
 ...
 <box nr="02">
  <book id="XW89">
   <author>Michel Foucault</author>
   <title>The Order of Things</title>
   <year>1966</year>
  </book>
  ...
 </box>
 ...
 <authorIndex>
  <authorEntry>J. Acute</authorEntry>
  <authorEntry>J. Obtuse</authorEntry>
  <authorEntry>Michel Foucault</authorEntry>
  ...
 </authorIndex>
</library>
```

The library automatically maintains an index of all authors whose books are published before 1980. The index is a part of the library document, whose complete DTD is:

```
<!ELEMENT lib (shelf+, box+, authorIndex)>
<!ELEMENT shelf (book_count, book*)>
<!ATTLIST shelf nr #REQUIRED>
<!ELEMENT book_count (#PCDATA)>
<!ELEMENT book (author+, title, year)>
<!ATTLIST book id #REQUIRED>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authorIndex (authorEntry*)>
<!ELEMENT authroEntry (name)>
<!ELEMENT name (#PCDATA)>
```

Suppose triggers are responsible to guarantee the correctness of the index. We do not provide full set of triggers needed to guarantee the correctness as it can involve a reader into an unnecessary complexity. Let us consider only two of them:

Trigger tr1 is triggered when any book that was published before 1980 is replaced. If a new book is published before 1980 and there are no authors of this book in the author' index then new author entry for this author is inserted into the index.

```
CREATE TRIGGER tr1
AFTER REPLACE
OF document("lib.xml")//book[year<1980]
FOR EACH NODE
LET $AuthorNotInList :=
    (if (NEW_NODE/year<1980)
     then for $n in NEW_NODE/author
          where empty(document("lib.xml")
                /library/authorIndex/
                authorEntry[name=$n])
          return $n
     else () )
WHEN (not(empty($AuthorNotInList)))
DO (UPDATE
    insert
     for $NewAuthor in $AuthorNotInList
     return <authorEntry>
             <name>$NewAuthor</name>
            <authorEntry>
    into document("lib.xml")/library/authorIndex )
```

Trigger tr2 is triggered when any book that was published before 1980 is deleted. If there are no other books that were published before 1980 by this author then his/her entry is deleted from the index.

```
CREATE TRIGGER tr2
AFTER DELETE OF document("lib.xml")//book[year<1980]
                                      /author
FOR EACH NODE
WHEN (empty(document("lib.xml")//
            book[year<1980]
            [author/text()=OLD_NODE/text()]))
DO (UPDATE
    delete document("lib.xml")//
           authorEntry[name=OLD_NODE/text()] )
```

Trigger tr3 has event `UPDATE-CONTENT` which means that it is triggered when any descendant of element `shelf` with attribute `nr` equal 45 is updated. Then the number of books on the shelf is re-counted.

```
CREATE TRIGGER tr3
AFTER UPDATE-CONTENT OF document("lib.xml")/library
                        /shelf[@nr=45]
FOR EACH NODE
DO (UPDATE
    replace document("lib.xml")//shelf[@nr=45]
                                /book_count
    as $x
    with
    <book_count>
     {count(document("lib.xml")//shelf[@nr=45]/book)}
    </book_count>
    )
```

An example of update to the library is the replacement of book which id is "AO97" from the shelf number 45 with another book which year of publishing is before 1980. Here is corresponding update statement `s0`:

```
UPDATE
replace document("lib.xml")/library/shelf[@nr=45]/
                                    book[@id="AO97"]
as $b
with <book id="{$b/@id}">
      <author>S. Kuznetsov</author>
      <title>Database Systems</title>
      <year>1979</year>
     </book>
```

This replace statement causes the consideration of all of the three triggers: tr1 is associated exactly with the same node that is replaced and the triggering operation

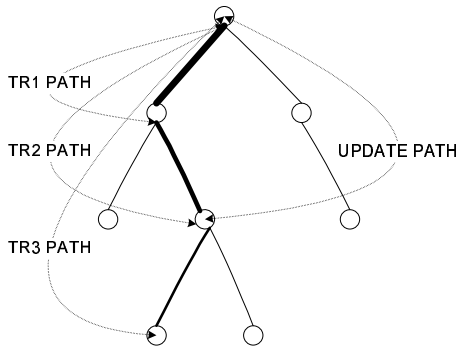

Figure 1: Triggering operations of the triggers triggered by the given update operation with given update/trigger-path length

is REPLACE; tr2 is associated with the data that is descendant of the replaced node and its triggering operation is DELETE; tr3 is associated with a node that is an ancestor of the replaced node and its triggering operation is UPDATE-CONTENT. In this paper we answer the question which of the triggers are triggered by the given update statement and how are the triggered triggers executed along with the update statement.

The paper provides the following important contributions:

- We define triggers for XML by providing its syntax and semantics that is close to SQL triggers semantics except the issues involved with the problems described above.
- We provide universal method to support such triggers in XML DBMS.
- We provide method to support triggers efficiently under the restriction: in update statements, data to be updated is specified in XPath (not an arbitrarily XQuery expression). The method represents combined execution of update statement and eligible triggers and uses previously built merged execution plan.

2 XML Triggers Definition

An XML trigger consists of four components: the *triggering operation*, the *triggering granularity*, the *trigger condition* and the *trigger action*. Consistent with the terminology of [15], a trigger is *triggered* when one of its triggering operations occur, it is being *considered* when its action is performed. When the trigger consideration starts, it is also *de-triggered*. The syntax of a trigger definition is the following:

```
CREATE TRIGGER trigger-name
(BEFORE|AFTER)
 (INSERT|DELETE|REPLACE|UPDATE-CONTENT)+
  OF XPathExpression (,XPathExpression)*
[FOR EACH (NODE|STATEMENT)]
[XQuery-Let-Clause]
[WHEN XQuery-Clause]
DO XQuery-UpdateOp
```

- The CREATE TRIGGER clause is used to define a new XML trigger, with the specified name.
- The BEFORE/AFTER clause expresses the triggering time relative to the update statement.
- Each trigger is associated with a set of update operations (INSERT, DELETE, REPLACE), adopted from the update extension of XQuery [1], and it can be also associated with UPDATE-CONTENT that is not a member of [1]. UPDATE-CONTENT is introduced by us. Trigger with this triggering operation associated with some node is triggered, when any descendants of this node are updated.
- 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. One or more predicates (XPath filters) are allowed in the steps to eliminate nodes that fail to satisfy given conditions.
- 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.
- The WHEN clause represents the trigger condition, and can be an arbitrarily expression legal in the 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 can be an arbitrary complex update operation.

For a complete syntax of XQuery refer to [4]. For the syntax of the update language, refer to [10].

Below in the paper under update path we mean an XQuery/XPath expression that is the part of an update statement and specifies data that are to be updated. Under trigger path we mean an XPath expression that specifies the data associated with the given trigger (XPath expression appearing after OF in a trigger definition).

Figure 1 illustrates different cases of positional relationship between nodes addressed by update path and nodes addressed be trigger path.

Figure 2 provides the triggering operations. For each cell of the table only triggers with triggering operations from this cell can be triggered by the specified update operation with given trigger path vs. update path length.

update vs. trigger path length		update operation		
		Delete	Insert	Replace
	length(tr_pth)>length(up_pth)	Delete	Delete	Delete, Insert
	length(tr_pth)=length(up_pth)	Delete	Insert	Delete, Replace
	length(up_pth)-length(tr_pth) = 1	Update-content	Update-content	Update-content, Insert
	length(up_pth)-length(tr_pth) >1	Update-content	Update-content	Update-Content

Figure 2: Triggering operations of the triggers triggered by the given update operation with given update/trigger path length

3 XML Trigger Support Methods

3.1 Database System Requirements

Traditionally, to achieve maximal efficiency, a trigger support method is tightly bound with the architecture of a particular database system and cannot be regarded in isolation from the architecture. Methods proposed in this paper were initially designed to efficiently support triggers in Sedna XML database system [3]. As a consequence, the methods are based on features intrinsic to Sedna. Nevertheless, to make our methods useful for broader class of XML database systems, we elaborate our solution to make them depended only of those Sednas features which are essential to these methods. Moreover we consider these features in the most common way to make them free of Sedna-specific particularities. Below is an overview of such essential features.

We assume that a system supports the XQuery language and an update language based on XQuery and its data model [1]. There is no standard update language proposed by W3C. Examples of this paper are given in the Sedna update language [1] that is very closed to that proposed in [18]. In addition to operations comparing nodes according to their document order, we assume support for operations that allows comparing nodes in ancestor-descendant order. The both types of comparison operations are implemented in Sedna using advanced numbering scheme as described in [3]. The universal method specified in Section 3.2 strongly relies on the latter comparison operations.

Optimized method described in Section 3.3 exploits the following two advanced features.

The first one is support for descriptive schema (also referred to as DataGuide [19]). In contrast to prescriptive schema that is known in advance and is usually specified in DTD or XML Schema, descriptive schema is dynamically generated (and increasingly maintained) from the data and presents a concise and accurate structure summary of these data. Formally speaking, every path of the document has exactly one path in the descriptive schema, and every path of the descriptive schema is a path of the document. As it follows from the definition, descriptive schema for XML data represented in the XQuery data model is always a tree. In the optimized algorithm, descriptive schema is used for compile-time analysis and simplification of XPath expressions. Using descriptive schema instead of prescriptive one gives the following advantages: (1) descriptive schema is more accurate then prescriptive one; (2) it allows us to perform compile-time simplification of XPath expressions when prescriptive schema is not available.

The second advanced feature is an extension to the standard XQuery execution model. In the extended execution model items obtained as an intermediate result of execution may be annotated with some metadata. Annotations are used to mark nodes for which triggers might be triggered later during the update execution process. There is also a set of predefined operations for dealing with annotations. The specification of the annotations and operations will be introduced in Section 3.3. Here we only notice that this extension is orthogonal to the XQuery execution model in a sense that marked items are just common items for standard XQuery operations.

Using extensions to the standard XQuery execution model requires us to turn to implementation-level details to describe our methods. We need to define a query logical plan that supports XQuery and the extension as well. The logical plan used in this paper is very close to that used in Sedna and defined formally in [20]. The exhaustive definition of the plan is not the goal of the paper we will describe it to the extent that should be sufficient to understand methods proposed in this paper. The logical plan includes operations to represent all kinds of XQuery expressions such as XPath expressions, FLWRs, etc. For the purposes of this paper we need only to describe logical plan for XPath expressions. XPath allows traversing an XML tree in the directions determined by axes. The logical plan for an XPath expression is a composition of operations each of which implements traverse by an axis. For example, the logical plan for the following XPath expression

```
doc("foo.xml")/library//book/@id
```

is as follows

```
attribute(descendent(child(doc("foo.xml"),
  elem(library)),elem(book)),id)
```

where `child` and `descendent` are operations implementing the child and descendent-or-self axes respectively; `elem(library)`, `elem(book)`, `id` are representations of the corresponding node tests.

XPath predicates are represented in logical plan using the `select` operation that takes a sequence to be filtered and the predicate represented as a function of one parameter. `select` applies the function to each element of the sequence and returns a new sequence containing items for which the function application results to true. For example, the following XPath expression

```
doc("foo.xml")/library/shelf[@nr=45]
```

is represented as follows

```
select(child(child(doc("foo.xml"),elem(library)),
  elem(shelf)),f(x|child(x,elem(@nr)=45))))
```

The features essential to our methods such as support for descriptive schema and advanced numbering scheme are not Sedna-specific and supported by a number of systems. The mechanism for marking nodes during query execution, though not known to be used in other systems, can be easily implemented due to its property of orthogonality to the standard XQuery execution model.

3.2 Universal Method

The method provided below is a straighforward one. It consists in preliminary evaluation of the update path and trigger paths and identification of the nodes associated with triggers among those affected by given update statement.

The basic steps of this method are the following:

- Evaluate update-path expression. Thus, we have *affected nodes*- a set of nodes that are to be updated.
- For `INSERT` and `REPLACE` statements, construct a new fragment of data.
- *Expanded* path is a path-expression that excludes ambiguities: '//', '*'. Using descriptive scheme and the result of evaluation from the previous step build all possible expanded paths without predicates.
- For a set of expanded update-paths and trigger-paths pick out the triggers that are probably triggered by the update statement (this procedure is described in detail in the 3.3 step 1: searching for probable triggers).
- For each trigger picked out on the previous step evaluate its trigger path expressions. From the result set of nodes, applying ancestor-descendant comparison operations introduced in Section 3.1, pick out those that are:
  - coinciding with some of affected nodes or
  - the descendants of some of the affected nodes or
  - the ancestors of some of the affected nodes
- For each node and its associated trigger pick out the nodes and their associated trigger according to Figure 2. Thus, we have got a set of nodes affected by the update statement on which some triggers are triggered.
- Evaluate the update statement and consider triggers on the nodes determined on previous step.

This method does not impose any restrictions on update path, i.e. it can be an arbitrary XQuery expression. But, obviously it has significant disadvantages: the preliminary identification of the nodes that match both update statement and some trigger may lead to working with a great number of triggers that appear to be not considered while given update statement.

Hence, in practice the method that excludes such an inefficiency is strongly needed. Below we provide triggers support method, simplifying the problem by introducing the restriction: in update statements XPath expressions are used (instead of arbitrary XQuery expressions) for determining the data to be updated.

3.3 Optimized Method for XPath-based Update Statements

Below we provide a three-step algorithm to support triggers efficiently. The effectiveness is attained thanks to the building of the specific execution plan that allows identifying the eligible triggers and marks their associated data at the run-time of an update statement execution.

Each step is a sufficiently complicated procedure that can be considered as a separate algorithm, thus, we try to describe it in full detail and carry out the step operations for our example given in Section 1.2. So, we have an update operation and a list of triggers.

3.3.1 Step 1: Searching for Probable Triggers

Probable triggers are triggers that can be probably triggered while the given update statement. The operations of this step are carried out on triggering operations, trigger path expressions, update operation and update path expression.

1. *Parent-expanded path* is an XPath expression rewritten in a way that it excludes '..' (parent axes). Build parent-expanded paths from update-path and each trigger-path by means of rewriting these path expressions to avoid '..' in the path. A set of rewriting rules for XPath expressions aimed at avoiding '..' is proposed in [11].

   In our example we do not have '..' in update path and trigger paths. So, we do not have to rewrite update path and trigger paths. Parent-expanded paths look as follows:

```
parent-expanded-upd-pth =
 document("lib.xml")/library/shelf[@nr=45]
   /book[@id="AO97"]
parent-expanded-tr1-pth =
 document("lib.xml")//book[year<1980]
parent-expanded-tr2-pth =
 document("lib.xml")//book[year<1980]/author
```

2. *Expanded* path is a path expression that excludes ambiguities: '//', '*'. Each of these ambiguities can be expanded using descriptive scheme. For example take '//': by means of descriptive scheme tree traversal we can obtain a set of paths that match '//'. Build expanded paths from update path and each trigger path by getting rid of the ambiguities using descriptive scheme.

```
expanded-upd-pth =
 document("lib.xml")/library/shelf[@nr=45]
 /book[@id="AO97"]
expanded-tr1-pth =
 {document("lib.xml")/library/shelf
   /book[year<1980],
  document("lib.xml")/library/box
   /book[year<1980]}
expanded-tr2-pth =
 {document("lib.xml")/library/shelf
   /book[year<1980]/author,
  document("lib.xml")/library/box
   /book[year<1980]/author}
```

   Thus, now for each trigger we have a set of expanded trigger paths associated with it.

3. Now pick out the probable triggers by means of following two procedures. Only triggers that are picked out in the first procedure below will be considered in the second procedure.

- For each trigger and for each expanded trigger path compare the names of the elements (and document function parameter) on each step of path with the corresponding names (names on the same steps of path) in expanded update path until one of the paths is ended. If on some step names are not equal this trigger is not triggered by the considered update operation. If on all steps of these two expanded paths corresponding names are equal (no matter if one of the paths is shorter) pick out this expanded trigger path and its associated trigger (consider it in the next procedure).

  In our example we carry out this procedure for two sets of expanded trigger paths that both consist of two expanded trigger paths. For tr1 we pick out

```
expanded-tr1-pth =
 document("lib.xml")/library/shelf
  /book[year<1980]
```

  For tr2 we pick out

```
expanded-tr2-pth =
 document("lib.xml")/library/shelf
  /book[year<1980]/author
```

  The other two expanded trigger pathes

```
expanded-tr1-pth =
 document("lib.xml")/library/box
  /book[year<1980]
expanded-tr2-pth =
 document("lib.xml")/library/box
  /book[year<1980]/author
```

  were not picked out because they have names `box` at the third step of path, but expanded update path has `shelf`.

- The table on Figure 2 provides the triggering operations. For each cell of table only triggers with triggering operations from this cell can be triggered by the specified update operation with given trigger path vs. update path length. For each expanded trigger path according to the table pick out only those expanded trigger paths and its trigger that have associated triggering operations provided in the table.

  In our example for tr1 we have expanded update path and expanded trigger path of tr1 of equal length and triggering operation is `REPLACE`. So, according to the table we pick out this expanded trigger path and its associated trigger. For tr2 we have expanded trigger path longer than expanded update path and triggering operation is `DELETE`. According to the table this expanded trigger path and its trigger is also suitable to pick out.

Thus, we have a list of triggers that can be probably triggered by the given update operation. For each probable trigger we have pick out a set of expanded trigger paths that address data on which this trigger can be probably triggered. Pass to the Step 2.

3.3.2 Step 2: Building Merged Execution Plan

The operation of this step are carried out on expanded update path and expanded trigger paths that were pick out on the previous step.

On this step for expanded update path and for each trigger path we build *merged execution plan*. *Merged execution plan* is a logical plan for execution expanded update path expression taking into account predicates from expanded trigger path expression. Merged execution plan is constructed in a way that as the result of its execution we retrieve sequence of nodes that are to be updated and identify those nodes in the sequence on which trigger is triggered by the given update operation. Such a sequence with identified nodes we call *marked sequence*. To mark nodes we use the facilities of extended XQuery execution model described in Section 3.1.

For marking nodes we introduce two logical operations: `mark1` and `mark2`. `mark1(unmarked_seq, predicate, trigger_name)` — takes unmarked sequence of nodes and marks (with `trigger_name`) those nodes that satisty the predicate. `mark2(marked_seq, predicate, trigger_name)` — takes marked sequence of nodes and for each marked node (that was marked with `trigger_name`) checks the predicate. If marked node satisfies the predicate it remains to be marked, if it does not satifies the predicate it is unmarked.

Thus, to build merged execution plan follow the next instructions:

1. Compare the lengths of expanded trigger paths and update path. If the lengths are not equal divide all of the expanded paths except the shortest one into two parts: *common expanded path* and a *tail expanded path*. For each expanded path, except the shortest one, common expanded path is a first part of expanded path of the same length as the shortest expanded path, tail expanded path is the rest of the expanded path. The shortest expanded path is divided into common expanded path that is equal to the shortest expanded path and tail expanded path that is empty.

   In our example expanded-tr2-pth is longer than expanded-update-pth, so we get

```
common-expanded-upd-pth =
 document("lib.xml")/library/shelf[@nr=45]
  /book[@id="A097"]
tail-expanded-upd-pth = ''
common-expanded-tr1-pth =
 document("lib.xml")/library/shelf
  /book[year<1980]
tail-expanded-tr1-pth = ''
common-expanded-tr2-pth =
 document("lib.xml")/library/box
  /book[year<1980]
tail-expanded-tr2-pth = /author
```

2. For expanded update path build logical plan by means of operations defined in [].

```
select(child(select(child(child(
 doc("lib.xml"),elem(library)),
  elem(shelf)),f(s | attribute(s,nr)=45)),
   elem(book)),f(b|attribute(b,id)="A097"))
```

Triggering operation	$OLD	$NEW	e		
			length(tr)<length(up)	length(tr)=length(up)	length(tr)>length(up)
INSERT	----	e	----	$what	tr_tail_pth($what)/ node()
DELETE	e	----	----	$i	tr_tail_pth($i)
REPLACE	e	$what	----	$i	----
UPDATE-CONTENT	e	evaluate in temporary space	tr_tail_pth($i)	----	----

Figure 3: Table for evaluating OLD and NEW variables

3. Each predicate from expanded trigger path must be inserted into the logical plan for their corresponding elements by means of logical operations `mark1` or `mark2`. For the first predicate starting from the end of the expanded trigger path `mark1` must be inserted, for the rest of the elements `mark2` must be inserted.

   Thus, for our example we build the following:

```
mark1(mark1(select(child(select(child(child(
 doc("lib.xml"),elem(library)),elem(shelf)),
  f(s|attribute(s,nr)=45)),elem(book)),
   f(b|attribute(b,id)="A097")),"tr1"),"tr2")
```

Thus, having passed through this step we obtain merged execution plan. Pass to the final step 3.

3.3.3 Step 3: Combined Execution of Update and Triggers

On this step we provide the procedure for combined execution of update statement and one or more triggers that is triggered by this update statement in an intuitive language.

```
begin
 for each $i in eval(merged_exec_plan)
 {
  let $what := eval_what(update_tail($i));
  let $where := eval_where(update_tail($i));
  begin
   trigger_func ($i, "before");
   update_op ($what, $where);
   trigger_function ($i, "after");
  end;
 }
end;
```

Function `trigger_func` executes triggers on marked nodes according to the `time` variable: `BEFORE` or `AFTER`. Function returns nothing.

```
function trigger_func( node $i, string $time )
begin
   if (marked($i)) then
   for each $trigger in
        {triggers with which $i is marked}
   {
    if(trigger_time($trigger) == $time)
    then
    {
     let $old := <determine by the table>
     let $new := <determine by the table>
     if <trigger_when_clase($old, $new)>
     then <trigger_action($old, $new)>
    }
   }
end;
```

`function eval( plan )` - evaluates merged plan, builded on the step 2, returns marked sequence of nodes.

`function eval_what( update path )` - If update statement is an insert, function constructs data that is inserted. If update statement is a replace, function constructs data to replace with. Functions returns variable `$what` bound to the constructed data. In case of delete `$what` is unspecified.

`function eval_where( update_path )` - returns a sequence of nodes that are to be updated by means of evaluation of `update_path`.

`function update_op($what, $where)` - executes update statement. Function returns void. `function trigger_time( trigger )` - returns trigger time: `before` or `after`.

Our method has the following advantages:

1. During update execution the triggeres that are not triggered by the update are excluded, thus, redundant processing of not triggered triggeres is avoided. The triggeres exclusion is carried out by means of update path and rule path comparison using descriptive schema and, after that, by means of incorporating the evaluation of rule path predicates into the update path execution.

2. All triggered triggeres are processed together along with the update execution, thus, there is no need to consider each triggere separately.

However, we are aware of a disadvantage: the method strongly relies on the fact that update and trigger paths are an XPath expressions. The method needs modifications to be suitable for XQuery expressions that address data in update statements.

4 Related Work

Active rules to enforce the correctness of update operations and to automatically maintain views of data has been extensively studied in database systems [5]. Many research projects provided substantial contributions to the field of active databases (among others, Starburst [9], Hipac [16], Sentinel [17], and IDEA [8]).

[22] presents implementation techniques of rules in Version 2 of POSTGRES (in particular, tuple level processing of rules deep in the executor) that we partially adopted while elaborating on method proposed in this paper.

As for the works specifically focused on triggers for XML, we point out [14], [21]. [14] proposes Active XQuery - an active language for XML repositories presented as an extension of XQuery. The authors propose the syntax of Active XQuery and its semantics by describing an algorithm for support triggers and a sketchy system architecture. The algorithm consists in expanding bulk update statements into a collection of equivalent statements, each one relative to a smaller fragment. This, as authors expect, simplifies trigger consideration, but in our opinion, the problem what triggers are triggered by the given update statement remains unsolved.

Secondly, authors claim their approach leads to a separation between the two system components: trigger subsystem and query engine. But at the same time they note that update statement expansion requires accessing the

data affected by the update statement. That confirms our assertion that complete trigger processing is impossible at compile-time, that is the trigger processing requires tight integration of trigger subsystem with query engine.

[21] investigates ECA rules on XML of the form similar to ours. The paper provides techniques for statical analysis of the triggering and activation dependencies between rules - the issue that we do not address in this paper. The proposed techniques can be a good supplement for our method.

5 Conclusion and Future Work

In this paper we proposed the definition of XML triggers and methods to support XML triggers in XML database systems. Methods described in this paper are being prototyped in the Sedna XML database system.

In many implementations of relational database systems triggers have been successfully used to support integrity constraints. In future work we will analyse the trigger capabilities in order to support different kinds of integrity constraints for XML databases.

References

[1] Sedna Programmer's Guide. MODIS ISP RAS, http://www.modis.ispras.ru/Development/sedna.htm

[2] Extesible Markup Language (XML) 1.0, W3C Recomendation. 2nd edition (2000), http://www.w3.org/TR/2000/REC-xml-20001006

[3] M. Grinev, A. Fomichev, S. Kuznetsov, K. Antipin, A. Boldakov, D. Lizorkin, L. Novak, M. Rekouts, P. Pleshachkov, "Sedna: A Native XML DBMS", Submitted to International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P), 2004.

[4] XQuery 1.0: An XML Query Language. W3C Working Draft. 29 October 2004.

[5] 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.

[6] 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.

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

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

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

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

[11] D. Olteanu, H. Meuss, T. Furche, F. Bry. XPath: Looking Forward. EDBT 2002 Workshops, LNCS 2490, pp. 109-127.

[12] K. P. Eswaran. Aspects of a Trigger Subsystem in an Integrated Database System. IBM Research Laboratory. San Jose.

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

[14] A. Bonifati, D. Braga, A. Campi, S. Ceri. Active XQuery. In Proc. of the 18th International Conference on Data Engineering (ICDE'02).

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

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

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

[18] P. Lehti. Design and Implementation of a Data Manipulation Language. Technische Universitt Darmstadt Technical Report No. KOM-D-149. August, 2001.

[19] R. Goldman, J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB 1997: 436-445.

[20] M. Grinev, P. Pleshachkov. Rewriting-based Optimization for XQuery Transformational Queries. In Proc. of IDEAS, Montreal, Canada. 2005.

[21] J. Bailey et al. An Event-Condition-Action Language for XML. In *Proc. of the WWW2002*, 2002.

[22] M. Stonebraker, A. Jhingran et al. On Rules, Procedures, Caching and Views in Data Base Systems. In *Proc, of SIGMOD Conference*, 1990.