Incorporating Active Rules Processing into Update Execution in XML Database Systems

Maria Rekouts
Institute for System Programming of Russian Academy of Sciences
B. Kommunisticheskaya, 25, Moscow, Russia
rekouts@ispras.ru

Abstract

We address the problem of efficient support for active rules in XML database systems. We describe a general form of active rules for XML based on XQuery and a previously defined update language and present our method to support these rules. The method consists in active rules processing along with the execution of an update statement. The method efficiency is attained due to the identification of rules triggered by the update statement during the execution of this statement that allows avoiding the redundant processing of not triggered rules.

Keywords: Active rules, XML, Database systems

1 Introduction

The concept of rules itself originated from the early times of relational database systems. Not long after the idea of integrity constraints in relational databases appeared, researches recognized the importance of automatic "reactions" to constraint violations, the idea expanded to the more general concept of triggers, also now known as event-condition-action (ECA) rules, or active rules [4].

In database system, active rules are used for incremental maintenance of materialized views, support of integrity constraints. Nowadays active rules are often considered as a mean to support business logic of e-commerce applications [3]. 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 active rules for XML we confronted with the problems induced by the irregular and hierarchical nature of XML data:

- 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 [10], concerned in this paper, and in another well-known proposal of update extension of XQuery [12]. Such a "bulk" nature of update primitives causes the question: does a new XML fragment that is inserted contains a data that matches any rules available in the system by the moment?

- Practically, XPath [14] is used in XML update languages to specify the data that are to be updated, as well it is used for rules, to determine the data associated with given rule. The problem is that analyzing update and rule XPath expressions at compile time it is impossible to know if the rule needs to be triggered under the given update statement, because the result of an XPath expression usually depends on data, but data are not available at compile-time. Thus, rules 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 a arbitrarily large number of rules, such evaluation must be carried out in an efficient way.

To illustrate these problems let us consider an example.

1.1 A First Glance at Active Rules for XML

This example is borrowed from [2], except some modifications that were brought in to illustrate our methods better. Assume a scenario based on the following lib.xml document, that belongs to an XML database of a library:

```xml
<lib>
  ...
  <shelf_nr="45">
    <book_count>2</book_count>
    <book_id="AO97">
      <author>J. Acute</author>
      <title>Triangle Inequalities</title>
      <year>1973</year>
  </book_id>
</shelf_nr>
</lib>
```
The library automatically maintains an index of all authors of the book from the shelf that was published before 1980.

Suppose rules are responsible to guarantee the correctness of the index. We do not provide full set of rules needed to guarantee the correctness as it can involve a reader into an unnecessary complexity. We consider only those that better illustrate the work of our methods in Section 3:

Rule r1 is triggered when an author of a book that was published before 1980 is deleted. If there are no other books that was published before 1980 by this author then his entry is deleted from the index. Rule r2 has UPDATE-CONTENT event 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.

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 under 1980. Here is corresponding update statement s0:

```
UPDATE
replace document("lib.xml")/lib/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 triggering of all of the two rules: r1 is associated with data that are descendants of the replaced node and its triggering operation is DELETE; r2 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 rules are triggered by the given update statement and how are the triggered rules executed along with the update statement.

The main contribution of this paper is our method to support active rules in XML database systems efficiently by means of incorporating active rules processing into the update statement execution.

The rest of the paper is organized as follows. In Section 2 we briefly present the general form of active rules for XML. In Section 3 we provide a method to support active rules in XML DBMS. Section 4 gives a survey of the related work. Section 5 outlines our future work and concludes the paper.

2 Active Rules for XML

An active rule consists of four components: the triggering operation, the path to the data that is in the scope of this rule, the condition and the action. A rule is triggered when one of its triggering operations occurs on the data addressed by the rule path; once a rule is triggered, its condition is considered; if the evaluation of condition is successful, the rule action is performed. Thus, active rule takes the following form:

```
RULE NAME r1
ON triggering operation (, triggering operation)*
OF path (,path)*
IF condition
DO action
```

- **RULE_NAME** specifies the name of the rule that is used to identify the rule in the system.
- Each rule is associated with a set of triggering operations (INSERT, DELETE, REPLACE), adopted from the update language [10], and it can be also associated with UPDATE-CONTENT that is not a member of [10]. UPDATE-CONTENT is introduced by us. Rule with this triggering operation associated with some node is triggered, when any descendant of this node is updated.
- The rule is relative to elements that match paths specified after the OF keyword. Each path is an XPath expression.
- The condition is specified after the IF keyword and can be an arbitrarily XPath expression.
- The rule condition is specified after the IF keyword and can be an arbitrarily XPath expression.
- The rule action is specified after the DO keyword and can be an arbitrary complex update statement.
The transition variables OLD and NEW denote the affected XML element before and after the execution of update statement. These variables are available for use within the condition and action part of the rule.\footnote{Thus, we concern the rules of a node-level granularity, i.e. rule is executed once for each node affected by the update statement. For not to overload this paper, we do not concern any different kinds of granularity (e.g. statement-level), but the methods provided below can be propagated on such rules by means of minor revisions.}

For a complete syntax of XQuery refer to [15]. For the syntax of the update language, refer to [14].

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

Figure 1 provides the triggering operations. For each cell of the table only rules with triggering operations from this cell can be triggered by the specified update statement with given rule path vs. update path length.

<table>
<thead>
<tr>
<th>update operation</th>
<th>Delete</th>
<th>Insert</th>
<th>Replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\texttt{x+y}$</td>
<td>Delete</td>
<td>Delete</td>
<td>Delete, Insert</td>
</tr>
<tr>
<td>$\texttt{x+y}$</td>
<td>Delete</td>
<td>Insert</td>
<td>Delete, Replace</td>
</tr>
<tr>
<td>$\texttt{x+y}$</td>
<td>Update-content</td>
<td>Update-content</td>
<td>Update-content, Insert</td>
</tr>
<tr>
<td>$\texttt{x+y}$</td>
<td>Update-content</td>
<td>Update-content</td>
<td>Update-Content</td>
</tr>
</tbody>
</table>

Figure 1. Rule/update statement correspondence

- The transition variables OLD and NEW denote the affected XML element before and after the execution of update statement. These variables are available for use within the condition and action part of the rule.

For a complete syntax of XQuery refer to [15]. For the syntax of the update language, refer to [14].

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

Figure 1 provides the triggering operations. For each cell of the table only rules with triggering operations from this cell can be triggered by the specified update statement with given rule path vs. update path length.

3 Active Rules Support Method

In this section we present our method to support active rules in XML database efficiently. Traditionally, to achieve maximal efficiency, a rule support method is tightly bound with the architecture of a particular database system and cannot be discussed in isolation from the architecture. Our method uses the following features of the database system: support for descriptive schema and an extension to the standard XQuery execution model, so, we first give a brief overview of these features.

Descriptive schema (also referred to as DataGuide [6]) 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 our method, descriptive schema is used for compile-time analysis and simplification of XPath expressions.

The second 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 rules might be triggered later during the update execution process. Predefined operations mark1 and mark2 (they are presented formally later) are introduced for dealing with annotations. This extension is orthogonal to the XQuery execution model in a sense that marked items are just common items for standard XQuery operations.

Method uses the concept of a query logical plan. The logical plan used in this paper is very close to that used in Sedna and its exhaustive definition is provided in [7]. Here, we give only a brief overview of the logical plan for an XPath expression. The logical plan for an XPath expression is a composition of operations each of which implements traverse by an axis. XPath predicates are represented in logical plan using the \texttt{select} operation that takes a sequence to be filtered and the predicate represented as a function of one parameter. For example, the logical plan for the following XPath expression

\[
\text{doc("foo.xml")/lib//book[@nr=45]/@id}
\]

is as follows

\[
\text{attr(select(descendant(child(doc("foo.xml")), elem(lib)), elem(book)), f(x|attr(x,nr)=45)), id)}
\]

where \texttt{child} and \texttt{descendant} are operations implementing the child and descendant-or-self axes respectively; \texttt{elem(library)}, \texttt{elem(book)}, \texttt{id} are representations of the corresponding node tests.

The support for descriptive schema is provided 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.

Now we pass on to the description of our method. It consists of three steps. We carry out each step operations for our example from Section 1.1.

Step 1: Searching for Probable Rules

We have an update statement and a list of rules. \textit{Probable rules} are rules that can be probably triggered during the execution of the given update statement. The operations of this step are carried out on triggering operations, rules path expressions, update statement and update path expression.

1. Expanded path is a path expression that excludes ambiguities: `//`, `*`, `.`. `//` and `*` ambiguities can be expanded using descriptive schema tree traversal. `.`...
(parent axes) can be excluded using a set of rewriting rules forXPath expressions proposed in [9]. Build expanded paths from update path and each rules path by getting rid of the ambiguities using descriptive schema and rewriting rules. For our example this looks as follows:

\[
\begin{align*}
\text{exp-upd-pth} &= \text{document("lib.xml")/lib/shelf[@nr=45枝]} \quad \text{/book[@id="AO97"]} \\
\text{exp-r1-pth} &= \{\text{document("lib.xml")/lib/shelf /book[year<1980]/author}, \\
&\quad \text{document("lib.xml")/lib/box /book[year<1980]/author}\} \\
\text{exp-r2-pth} &= \text{document("lib.xml")/lib/shelf[@nr=45枝]} \\
\end{align*}
\]

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

2. For each rule and for each expanded rule path compare the names of the elements (and the document function parameter) on each step of the expanded rule path with the corresponding names (names on the same steps of path) in the expanded update path until one of the paths is ended. If on some step names are not equal this rule is not triggered by the considered update statement (do not consider it further). 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 rule path and its associated rule (consider it on the further steps).

In our example for r1 and r2 we pick out

\[
\begin{align*}
\text{exp-r1-pth} &= \text{document("lib.xml")/lib/shelf} \\
&\quad /\text{book[year<1980]/author} \\
\text{exp-r2-pth} &= \text{document("lib.xml")/lib/shelf[@nr=45枝]} \\
\end{align*}
\]

The other expanded rule path

\[
\begin{align*}
\text{exp-r1-pth} &= \text{document("lib.xml")/lib/box} \\
&\quad /\text{book[year<1980]/author} \\
\end{align*}
\]

was not picked out because it has names box at the third step of the path, but expanded update path has shelf.

3. The table on Figure 1 provides the triggering operations. For each expanded rule path according to the table pick out only those expanded rule paths and its rule that have associated triggering operations provided in the table.

In our example, for r1 we have the expanded rule path longer than the expanded update path. r1 triggering operation is DELETE, so according to the table we pick out this expanded rule path and its associated rule. For r2 we have the expanded rule path shorter than the expanded update path and triggering operation is UPDATE-CONTENT. According to the table this expanded rule path and its rule is also suitable to pick out.

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

Step 2: Building Merged Execution Plan

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

On this step for the expanded update path and for each rule 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 rule path expression. Merged execution plan is constructed in a way that as the result of its execution we retrieve a sequence of nodes that are to be updated and identify those nodes in the sequence on which rule is triggered by the given update statement. Such a sequence with identified nodes we call marked sequence. To mark nodes we use the facilities of extended XQuery execution model described above in this section.

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

To build merged execution plan follow the next instructions:

1. Compare the lengths of expanded rule 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 not divided.

For our example we get the following.

\[
\begin{align*}
\text{common-exp-upd-pth} &= \text{document("lib.xml")/lib/shelf[@nr=45枝]} \\
\text{tail-exp-upd-pth} &= \text{/book[@id="AO97"]} \\
\text{common-exp-r1-pth} &= \text{document("lib.xml")/lib/shelf} \\
&\quad /\text{book[year<1980]/author} \\
\text{common-exp-r2-pth} &= \text{document("lib.xml")/lib/shelf[@nr=45枝]} \\
\text{tail-exp-r2-pth} &= \'' \\
\end{align*}
\]

2. For the expanded update path build logical plan.
3. Each predicate from expanded rule 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 rule path mark1 is inserted, for the rest of the elements mark2 is inserted. Thus, for our example we build the following:

```
mark1(mark1(select(child(child(doc("lib.xml"), elem(lib)), elem(shelf)), f(s|attr(s,nr)=45)),
f(r|attr(r,nr)=45),"r2"),f(r|true()),"r1")
```

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

**Step 3: Combined Execution of Update and Rules**

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

```
begin
for each $i in eval(merged_exec_plan)
{
    let $what := eval_what($i+update_tail);
    let $where := eval_where($i+update_tail);
    begin
    update_op ($what, $where);
    if (marked($i)) then
        for each $rule in {rules with which $i is marked}
        {
            let $old := <determine by the table>
            let $new := <determine by the table>
            if <rule_condition($old, $new)>
            then <rule_action($old, $new)>
        }
    end;
}
end;
```

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

**function eval_what( update path )** - If the update statement is an insert, function constructs data that is inserted. If the update statement is a replace, function constructs data to replace with. This function 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. This function returns nothing.

Our method has the following advantages:

1. During update execution the rules that are not triggered by the update are excluded, thus, redundant processing of not triggered rules is avoided. The rules 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 rules are processed together along with the update execution, thus, there is no need to consider each rule separately.

However, we are aware of a disadvantage: the method strongly relies on the fact that update and rule 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 statements and to automatically maintain views of data has been extensively studied in database systems [4]. Research projects [13],[5] provided substantial contributions to the field of active databases.

[11] 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 recent works related to ours, we point out [2], [1]. In [2] Active XQuery – an active language for XML repositories is presented as an extension of XQuery. The authors propose the syntax and semantics of Active XQuery, provide an algorithm to 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, at compile-time. The algorithm presents a good solution of the problem of "bulk" nature of update statements stated in the introduction of this paper, but it does not consider all of the triggers triggered by an update statement (the triggers associated with ancestor elements of updated elements are not considered). Secondly, we argue that trigger processing cannot be done completely at compile-time as update statement expansion requires accessing the data affected by the update statement.
This leads to a tight integration of a trigger subsystem with a query engine that is exactly what we provide in the method proposed in this paper.

[1] 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

The paper proposes a method of efficient active rules support in XML database systems by means of incorporating active rules processing into update execution. The method is being prototyped in Sedna XML database system [8]. Our future work consists in conducting experiments and thorough analysis of their results.

References