Comput Sci Res Dev (2014) 29:1–19
DOI 10.1007/s00450-010-0132-2

REGULAR PAPER

XTrigger: XML database trigger

Anders H. Landberg · J. Wenny Rahayu · Eric Pardede

Received: 2 September 2009 / Accepted: 5 August 2010 / Published online: 26 August 2010

Abstract An ever increasing amount of data is being exchanged and stored in the XML data format brings with it a number of advanced requirements to XML database functionality, such as trigger mechanism. Triggers are commonly used to uphold data integrity constraints and are yet to be rigorously defined for XML. To make trigger functionality for XML databases as practical and applicable as it is in the relational context, the hierarchical nature of the XML data model must be considered. However, current research on XML triggers shows clear limitations with respect to multi-document applicability and ancestor-descendant relationships.

We introduce path-level granularity in combination with the novel concept of trigger scope, which plays a critical role in the validation of complex constraints. In order to effectively and efficiently support both traditional data integrity constraints and advanced semantically aware constraints, a trigger methodology that embeds several points of innovation in the context of XML triggers is hence proposed and deeply investigated in this paper.

Keywords XML · Trigger · Path granularity · Trigger scope

A.H. Landberg · J.W. Rahayu · E. Pardede (✉)
Department of Computer Science and Computer Engineering,
La Trobe University, Bundoora, VIC 3083, Australia
e-mail: E.Pardede@latrobe.edu.au

A.H. Landberg
e-mail: A.Landberg@latrobe.edu.au

J.W. Rahayu
e-mail: W.Rahayu@latrobe.edu.au

1 Introduction

The concept of the database trigger was first introduced in 1976 [1]. Prior to this, the need for a mechanism to automatically react to constraint violations had been identified as necessary [2–4]. Some of the most common areas where triggers can be used are: monitoring constraints on the data [5–7], view maintenance [7], temporal changes [18], versioning [19] and logging manipulation of data [5]. However, several authors show how entire business logic can be incorporated into triggers [8].

The W3C standard does not yet include an XML trigger support method [9]. Therefore, this paper investigates and discusses what is lacking in previous approaches, the reasons for these absences, and it considers solutions where the XML trigger best fulfils its duty within the XML database.

As a fundamental functionality of XML databases, the development of XML trigger methodologies poses a number of requirements.

First, trigger granularity is a fundamental and low-level underlying concept that discusses the level of detail in which modified data is to be treated by a trigger. This involves determining the partitions of data that make up a particular grain. As triggers provide reactive functionality in response to XML data updates, it is crucial to identify which parts within an XML document are impacted during the update [10, 20, 21]. This is so important because the structure of XML is hierarchical instead of flat. Therefore, in the XML context, one type of granularity may be the set including a particular target node together with its parent and child nodes [11].

When examining data with this granularity, each target node is not being considered by itself alone, but as part of this hierarchical relationship. Similar to how sibling nodes can be put into a horizontal relationship, namely to be on the same hierarchical level, nodes can also be put into a vertical

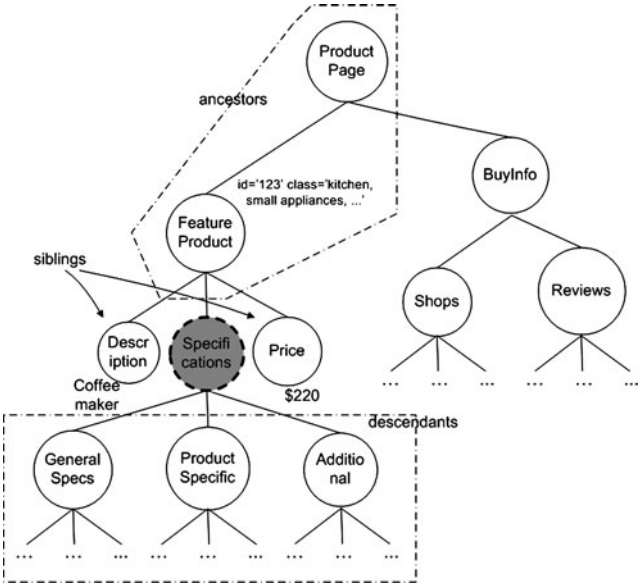


Fig. 1 Ancestor and descendant nodes

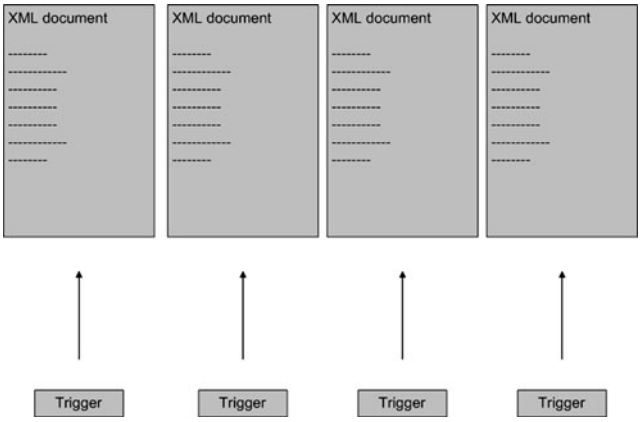


Fig. 2 Multiple identical triggers

relationship which considers neighbouring nodes on different hierarchical levels. The consideration of such a vertical, hierarchical node relationship in regards to trigger granularity is vital because it strongly draws focus to the concept of data hierarchy that is one of the major discrepancies between relational and XML data models [12].

Figure 1 illustrates a situation where the (highlighted) node `Specifications` has been modified by a database update. When making changes to `Specifications`, subsequent changes in nodes `FeatureProduct`, `ProductPageGeneralSpecs`, `ProductSpecific`, etc. may be necessary to uphold constraints along the ancestor-descendant axis.

A second requirement is the support for multiple documents. This is so because apart from a document-internal structuring, which is hierarchical, there also exists a further structuring within the database, namely the structuring of documents. As such, several documents in an XML database may contain similar types of information, i.e. have a similar or even the same structure. The reasons for this can be that slightly different content has been grouped into different documents to maintain a better overview, or, that some operations can be performed quicker on these separate documents.

The issue arises when circumstances force a certain operation to be monitored for several XML documents within a database. These documents could either have a similar (or even the same) structure, hence possibly having a common schema definition, or belonging together in some other way of relationship such as linking. As an example, the ACM SIGMOD record XML edition[1] contains collections of XML documents which apply to a common DTD. If all these documents should be monitored by a trigger to ensure that after a deletion important parts of the documents are backed up, it would be desirable to have a trigger concept that is capable of applying to a collection of documents. Or, even more convenient, to the respective schema definition, which indirectly would link the trigger to the collection of documents.

A possible solution to the above described problem is to apply one trigger to each of the documents. Such a scenario is illustrated in Fig. 2. In current approaches [5–7] this is the only possibility to solve such a scenario. However, this will inevitably result in redundant triggers, as it takes one trigger with the same functionality for each document. So in the case of a change in the trigger code, each of them will have to be rewritten, which again will lead to errors and inconsistency.

Another way around this problem would be to merge XML documents so that all data is in one document. Then it is sufficient to apply a single trigger to this document that monitors the desired operations and takes action where necessary. Although this approach does not bear the risk of having multiple identical triggers, it forces data to be joined that now must be separated by queries.

In a previous work on XML triggers, we proposed the fundamental concept of path-granularity and how it solves a number of current approaches' limitations with respect to vertical node relationships [17]. In this publication we reintroduce this concept in combination with trigger scope.

Proposal This paper proposes two interconnected concepts in XML database triggers, namely (i) *path-level granularity* and (ii) *trigger scope*. The path-level granularity is a novel concept that considers vertical node relationships during updates on XML data. XML triggers with this new functionality will overcome current approaches' limitation in respect to node-relationships and granularity and make the trigger more applicable and useful for more complex

[1] Available for download at http://www.sigmod.org/record/xml.

constraint checking as we proposed in a recent publication [22].

The concept of trigger scope is an extension to the context within which the trigger can be applied to by defining the entities in the database that are affected by the trigger. As current approaches only limit their trigger to be applicable to one single document instance (single document context), it must be considered what would be the impact and benefits of extending this context. These extensions can be made in order to apply the trigger to (i) a schema, and (ii) all documents in the database.

Organisation Section 2 gives background information on context paths and nodes that will be used to determine the trigger granularity and scope. Section 3 proposes definitions and methodologies to specify the new level of granularity and apply it to the XML context as well as the new levels of trigger scope. Section 4 classifies XML trigger types and offers a comprehensive trigger categorisation. Section 5 discusses execution issues that result from the trigger categories. Sections 6 and 7 describe the implementation that was done in order to prove and evaluate the proposed concept. Related work is summarised in Sect. 8. Finally, Sect. 9 concludes the paper and offers ideas for future work.

2 Preliminaries

Following definitions are necessary to describe a context path. First, it is important that the nodes in a document tree are be labelled in a way that uniquely identifies them. Also, the labelling must allow further definitions being dependent on it. In our approach, we use the pre-order and post-order scheme [13, 14].

Notation A '/' is a single separator between two nodes A and B, where B is a child node of A.

Definition 1 A *root* node root is defined as a tuple (pre, post), where preorder value $pre = 1$ and post-order value $post = 2n$, and where n is the number of nodes in the tree.

Definition 2 An intermediate node *interm* is defined as a tuple (pre, post), where $pre > 1$ and $post < 2n$ and $post\text{-}pre > 1$.

Definition 3 A *leaf* node leaf is defined as a tuple (pre, post), where $post\text{-}pre = 1$.

Definition 4 A context node *con* is defined as $con \in \{\text{root, interm, leaf}\}$. The context node is the node that is being addressed to by a trigger's event path.

Definition 4a A context node *con* has pre-order and post-order values $(\text{pre}_{con}, \text{post}_{con})$.

Definition 5 An ancestor node *anc* of a context node is defined as $anc \in \{\text{root, interm}\}$.

Definition 5a An ancestor node *anc* of a context node is defined as a tuple $(\text{pre}_{anc}, \text{post}_{anc})$, where $\text{pre}_{anc} < \text{pre}_{con}$ and $\text{post}_{anc} > \text{post}_{con}$.

Definition 6 A descendant node *desc* of a context node is defined as $desc \in \{\text{interm, leaf}\}$.

Definition 6a A descendant node *desc* of a context node is defined as a tuple $(\text{pre}_{desc}, \text{post}_{desc})$, where $\text{pre}_{desc} > \text{pre}_{con}$ and $\text{post}_{desc} < \text{post}_{con}$.

Definition 7 A *Valid Path* $VP = N_1/N_2/N_3/\ldots/N_m/$, with length m, where N_1 is a root node, and node $N_i + 1$ is a descendant node of N_i, and N_m is a leaf node $(1 \leq i \leq m - 1)$.

Definition 8 A *Context Path CP* is a valid path that contains a context node.

Definition 8a A *Context Path* $CP = \{A, c, D\}$, where the *set of ancestor nodes* $A = \{a_1, a_2, a_3, \ldots\}$ and the context node c and the *set of descendant nodes* $D = \{d_1, d_2, d_3, \ldots\}$.

Definition 9 The number of context paths of a context node is equal to the number of leaf nodes that are descendants of that context node.

Definition 9a A root context node has k context paths, where $k = all\ valid\ paths\ in\ the\ tree$. Hence, context paths = valid paths.

Definition 9b A leaf context node has 1 context path.

To illustrate the above defined node-labelling scheme and context node and context path definitions, consider Fig. 3. The context node *con* is marked with a bold circle, and all nodes that stand in a vertical node relationship to it are shaded in grey colour. All the nodes that are coloured grey will be considered by the path-level granularity. The remaining nodes are of no importance. The context node with labelling (13, 22) has the following context paths related to it:

$CP_1 = /(1, 26)/(10, 25)/(13, 22)/(14, 17)/(15, 16)$

$CP_2 = /(1, 26)/(10, 25)/(13, 22)/(18, 19)$

$CP_3 = /(1, 26)/(10, 25)/(13, 22)/(20, 21)$

Note that the total number of context paths within a document is equal to the sum of context path of all context

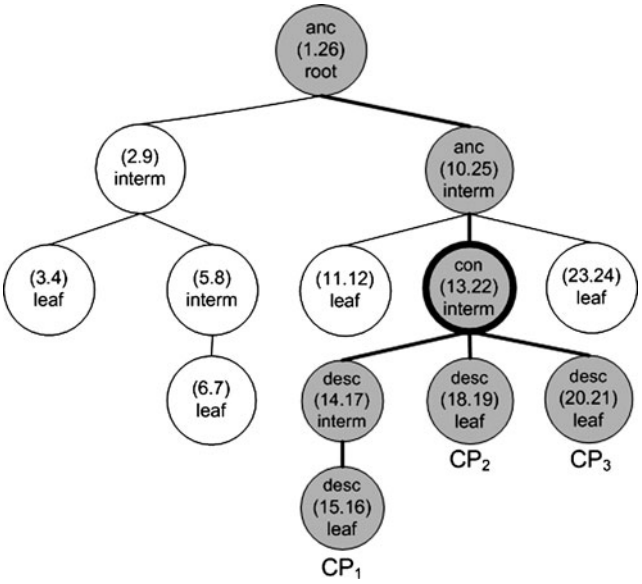


Fig. 3 Labelled document tree

nodes. So given that there are n context nodes with p_n context paths each, then the total number of context paths is $p_1 + p_2 + \cdots + p_{n-1} + p_n$.

3 Incorporating granularity and scope into triggers

Based on the preliminary definitions described in the previous section, the notions of XML trigger granularity and scope will be formalised and outlined in this section.

3.1 Granularity

This section defines three levels of granularity that can apply for an XML trigger: node, path, and document granularity.

3.1.1 Definitions for trigger granularity

Definition 10 *Trigger granularity* G_X of a trigger T is defined as a tuple $T(X, P, N)$, where granularity level $X \in \{Node, Path, Document\}$, and P is the set of valid paths, and N is the number of trigger executions.

Proposition 1a *Node granularity* G_N *for a trigger* T *is defined as a tuple* $T(X, P, N)$, *where* $X =$ Node, *and* $P =$ *a valid path that specifies the set of context nodes, and* $N =$ *size of* (*context nodes*).

Proposition 1b *Path granularity* G_P *for a trigger* T *is defined as a tuple* $T(X, P, N)$, *where* $X =$ Path, *and* $P =$ *a valid path that specifies the set of context nodes, and* $N =$ *size of* (*context paths*).

Proposition 1c *Document granularity* G_D *for a trigger* T *is defined as a tuple* $T(X, P, N)$, *where* $X =$ Document, *and* $P =$ *a valid path that specifies the set of context nodes, and* $N = 1$.

Example 1 Given a trigger T, and given $X =$ Path, and given $P =$ '$/A/B/C$', and given $N = 5$, then this means that there exist 5 context paths that traverse context nodes specified by '$/A/B/C$', and that T will be executed 5 times.

3.1.2 Vertical and horizontal granularity

This section defines the methods and techniques that are used to incorporate path-granularity to the XML trigger mechanism. The notions of *vertical node granularity* and *horizontal node granularity* are introduced.

Vertical node granularity Vertical node granularity regards each node as belonging to a vertical relationship that includes ancestor and descendant nodes. Similarly, horizontal node granularity addresses a particular node and its siblings.

A path is a set of related nodes that belong to vertical relationship as explained above. The set of nodes in such a relationship is a subset of the document, and as such it must be considered as an additional level of trigger granularity. As previous research has neglected the path notion in respect to granularity [5, 7], paths and nodes that are interconnected with the modified node, are neglected, too. This means that neither ancestor nodes (`FeatureProduct`, `ProductPage`), nor descendant nodes (`GeneralSpecs`, `ProductSpecific`, `Additional`, ...) are being considered when making updates (see Fig. 1).

Horizontal node granularity Horizontal node granularity considers nodes which stand in a horizontal relationship towards one another. As for the granularity, current approaches introduce two different levels, namely node-level and document-level granularities [6]. They represent the XML equivalent of row and statement granularities in the relational context. Node-level granularity considers each trigger event separately, and the XPath expression in the trigger's event part is associated with a single event. Document-level granularity assumes that each trigger considers at once all relevant event instances.

Hence, a document-level trigger is fired once for all modified nodes. Although this mapping from the relational to the XML context works for some cases, it neglects the basic discrepancies between the different data models.

In Fig. 4, nodes 10 and 11 have been modified by an update statement and have been matched by a trigger event path. In the left example, the (node-level) trigger will be fired twice for node 10 and 11. In the right example in

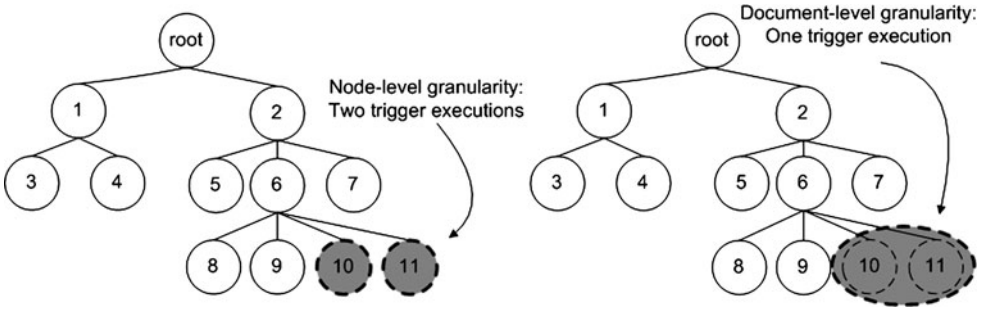


Fig. 4 Node and document level granularity

Fig. 4, the (document-level) trigger will be fired once for both nodes.

As can clearly be seen, neither of these existing approaches to granularity solve the above described problem, as they only consider horizontal node relationships and do not focus on vertical node relationships

3.1.3 Trigger path methodology

Based on the previously proposed concepts, this section outlines how we form a path-level trigger methodology. Our main contributions in this section aim to extend XML trigger path methodologies, and are as follows:

1. Applying the concept of a context path into trigger execution based on the context path definition
2. Expressing the new path-granularity using a proposed syntax

First, we will discuss the concept of context paths. To do this, it is necessary to analyse which nodes in a document tree will possibly be affected by an update. We will refer to these nodes as context nodes. In existing research papers, context nodes are often referred to as affected nodes [7, 8]. Similarly, a context document is a document that is modified, i.e. that contains context nodes.

Figure 5 illustrates the steps that are necessary to perform path-level granularity. The first step is to identify the context node. This is the node, or set of nodes, that is affected by an update statement, such as an insertion, deletion, or modification of nodes. All ancestor and descendant nodes of the context node are extracted and represent a sub-tree of the entire document tree. The second step traverses this sub-tree and assembles the context paths. These context paths are then available to the trigger in the form of transition variables during execution.

The advantage of this level of granularity is that content of various node types in an entire document sub-tree can be considered by a single trigger and that all relevant paths that traverse the context node are considered. A path-level trigger will therefore create one event instance for every path that traverses the context node.

Next, we will focus on the syntax that is necessary to describe a trigger that incorporates path-level granularity. The expression `DO FOR EACH [doc|path|node]` on line 5 in the listing specifies the level of granularity. *doc* and *node* denote document-level and node-level granularities respectively, and *path* denotes path-level granularity.

When a trigger is fired and after its condition (refer to line 4 in the below listing) has evaluated to true, it will be executed according to its level of granularity (see line 5). In accordance with our definitions, a trigger with path-level granularity will therefore be fired once for each context path that traverses the context node. If the number of context nodes in a document is n, then the trigger will be fired for each context path of each of the n context nodes. When a trigger is fired, its action body, denoted by `<XQuery-expr>` in the listing, is executed according to the level of granularity.

XML Trigger Syntax

```
1. CREATE OR REPLACE XMLTRIGGER
   <trigger-name>
2. ON [insert|modify|delete]
3. OF [doc(<document-name>)
   /<update-path>|
   schema(<schema-name>
   /<update-path>)|doc(ANY)]
4. IF <XPath-qualifier>
5. DO FOR EACH [doc|path|node]
   [<XQuery-expr>]
6. END XMLTRIGGER;
```

3.2 Scope

Scope is defined as an enclosing area in which something carries out actions on some content that is within this area. The challenge is now to map this definition to the trigger context, and in particular, to the XML database trigger context.

3.2.1 Definitions for trigger scope

Definition 11 *Trigger scope* S_X of a trigger T is defined as a tuple $T(S, M)$, where scope level $S \in \{\text{Schema, Schema-less}\}$, and execution multiplicity $M \in \{1, \text{Multiple}\}$.

Definition 11a *Instance scope* S_I for trigger T is defined as a *tuple* $T(S, M)$, where $S =$ Schema-less and $M = 1$.

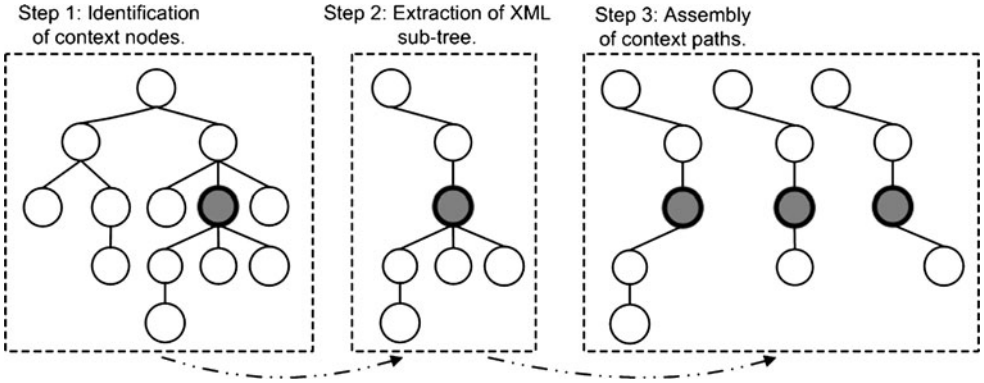


Fig. 5 Identification and assembly of context paths

Definition 11b *Schema-less scope* S_{SL} for trigger T is defined as a tuple $T(S, M)$, where $S =$ Schema-less and $M =$ Multiple.

Definition 11c *Schema scope* S_S for trigger T is defined as a tuple $T(S, M)$, where $S =$ Schema and $M =$ Multiple.

3.2.2 Trigger scope methodology

Based on above proposed definitions, this section explains our proposed methodology to extend the trigger scope.

Our main contributions for this section aim to extend XML trigger methodologies, and are as follows:

1. Extending XML triggers' applicability to multiple documents
2. Identification and definition of scope levels

This paper introduces two additional types of scope. Schema scope assumes that there exists a schema definition that describes documents. A schema scope trigger can be applied to a schema and hence applies to all documents corresponding to it. Schema-less scope assumes that there are documents that are not corresponding to any schema definition. Hence, a trigger with schema-less scope will only consider documents that have no associated schema associated.

The new levels of trigger scope are instance, schema, and schema-less scope. They are ordered in ascending granularity, regarding a document instance as the smallest grain of scope and the set of all documents (addressed by the schema-less scope) as the largest grain.

3.2.3 Instance scope

A trigger which is applied to a single document instance in an XML database is defined as being an instance scope trigger. Hereby it is irrelevant whether this particular document has got a schema definition associated to it or not. Current approaches apply their trigger to a particular document instance (see Fig. 6) and the notion of a trigger scope has not been considered yet. However, the existing papers' approaches can be classified as instance scope triggers.

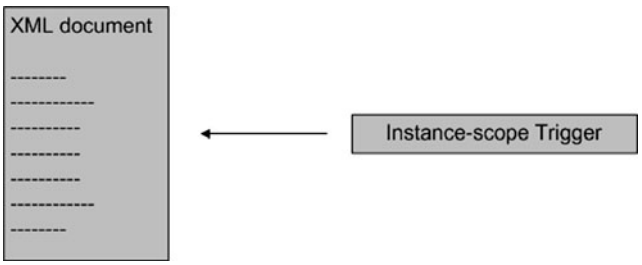


Fig. 6 Instance scope

3.2.4 Schema scope

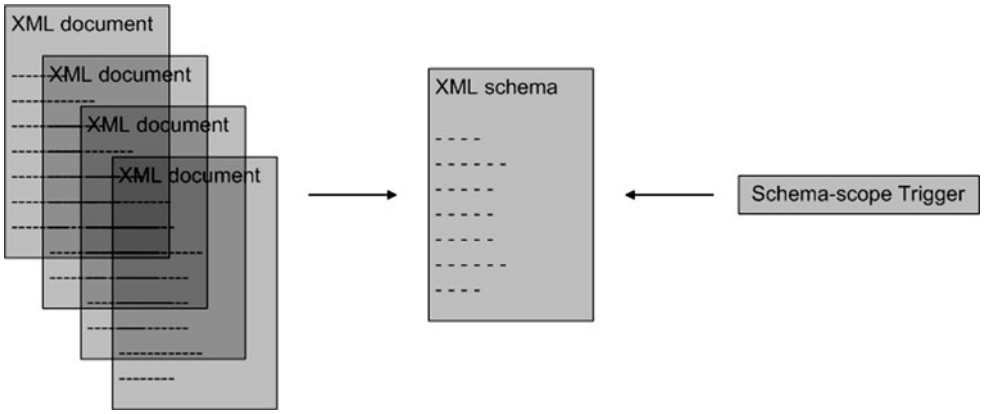
When defining the trigger, an associated schema definition (XML Schema, DTD, Relax NG) must be provided that correctly describes a collection of documents. It is important to note that when providing a schema as trigger scope, all documents that conform to it will possibly be affected by the trigger, dependent on the trigger's event and condition paths. As such, the schema serves as identification of the set S_{schema} of documents which must be considered. So, whenever a document which conforms to a certain schema is modified, it must be checked if there are any triggers associated to that schema. If the trigger's condition evaluates to TRUE and is fired, it will be executed according to its level of granularity within the context document. The granularity of the schema-trigger applies to each document that conforms to the specified schema as illustrated in Fig. 7.

3.2.5 Schema-less scope

This scope level is similar to the schema scope, as it includes a collection of documents rather than a single document instance by itself. It includes documents that do not have an associated schema (see Fig. 8). Hence, it includes the set $S_{schema\text{-}less}$ of all documents that are not considered by the schema scope trigger. The schema-less scope provides the possibility to make a trigger available throughout the entire set of documents in the database.

Referring back to the XML trigger syntax listing, line 3 illustrates how the scope level can be defined. To define a

Fig. 7 Schema scope



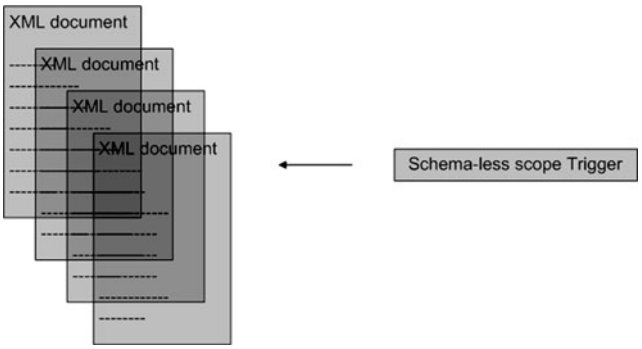


Fig. 8 Schema-less scope

schema-scope trigger, the keyword "schema" is used, and is followed by the schema's file name in brackets. For the conventional instance-scope trigger, the XML document can be referred to by its file name enclosed by the keyword "doc". To apply a trigger to all documents in the database, we use "doc(ANY)" to denote this.

4 XML trigger categorisation

Table 1 shows how the new trigger types are classified according to their scope and granularity levels. The labels are to be read as follows: S and G denote scope and granularity respectively. The subscripted abbreviations N, P, and D for the scope stand for node, path, and document level. Subscripts I, S, and SL for the granularity level stand for instance, schema, and schema-less level.

In the examples that will follow, the nodes in the document tree illustrations are labelled with incrementing numbers instead of the pre-order and post-order labelling techniques for simplicity reason.

Note S_IG_* refers to all triggers with instance-scope, independent of their granularity. Therefore, $S_IG_* = \{S_IG_N, S_IG_P, S_IG_D\}$. The same yields for the notation S_*G_D, where $S_*G_D = \{S_IG_D, S_SG_D, S_{SL}G_D\}$.

Table 1 Trigger types, where: *Node* ⊂ *Path* ⊂ *Document*

Granularity	Trigger scope		
	Instance	Schema	Schema-less
Node	S_IG_N	S_SG_N	$S_{SL}G_N$
Path	S_IG_P	S_SG_P	$S_{SL}G_P$
Document	S_IG_D	S_SG_D	$S_{SL}G_D$

4.1 Trigger S_IG_N

Scope The scope of the trigger is an XML document instance. To declare a trigger of type S_IG_*, an XML document instance ("filename.xml") must be explicitly specified. The trigger will then listen to modifications that are performed to this document instance only. It is not important whether this document has a schema associated with it, as the document instance is explicitly declared within the trigger.

Granularity Within the instance scope, a node-level trigger will execute n times, where $n = number\ of\ context\ nodes$. For each node in the document which (i) matches the modification operation's update path, and (ii) matches a node-level trigger's event path, an event instance will be generated and the trigger will then be executed for each of these instances.

Transition variables As explained above.

Trigger execution multiplicity This category specifies how many times this type of trigger will possibly be executed during an arbitrary modification operation on an XML document. Hereby it is being distinguished between single execution and multiple executions as well as the number of context nodes, context paths, and context documents.

- Single execution on single document if context nodes = 1
- Multiple executions on single document if context nodes > 1

Listing 1 Insert statement
```
INSERT BELOW node ( 2 )
node(5)
node(6) (node(8) node(9) node(10)
node(11) node(12))
node(7)
```

Listing 2 Trigger SIGN example code
```
ON INSERT BELOW fn:
document(<XML doc 1>)/root/2
IF <trigger condition>
FOR EACH NODE DO <trigger action>
```

As this trigger type can only be applied to a single document instance, trigger execution multiplicities are limited to a single document. In case there is only one context node, the execution multiplicity is 1. For n_{CN} context nodes, there will be n_{CN} trigger executions.

Example Figure 9 shows a scenario where three nodes are being inserted into an XML document tree. One of the newly inserted nodes, node 6, contains child nodes 8 to 12. An example insert statement could look like in Listing 2. The nested child nodes of node 6 are enclosed in round brackets in the insert statement following their parent node.

4.2 Trigger $S_I G_P$

Scope See scope of $S_I G_N$ trigger.

Granularity Within the instance scope, a path-level trigger will execute n_{CP} times, where $n_{CP} =$ *number of context paths*. For each path in the document which (i) contains the context nodes addressed by the modification operation's update path, and (ii) has the property "path-level trigger's event path IS SUBSET OF document path", an event instance will be generated and the trigger will then be executed for each of these instances.

Transition variables The $S_I G_P$ trigger will execute for each identified context path. As such, the transition variables $old and $new are bound to the respective context path for each execution.

Note that different context paths can belong to different context nodes, i.e. context node A has CP_{a1} and CP_{a2} and context node B has CP_{b1}. For this reason it is necessary to introduce an additional transition variable $con, which denotes the context node and is available only for path-level triggers.

Trigger execution multiplicity

- Single execution on single document if context nodes = 1 v context paths = 1
- Multiple executions on single document if context nodes ≥ 1 v context paths > 1.

A context node can have many context paths. Hence, a single execution can only occur if there is one context node which has one context path.

Example The scenario in Fig. 10 demonstrates how a delete operation (see Listing 3) is monitored by a $S_I G_P$ trigger. In this example, the trigger (see Listing 4) is fired in reaction to the deletion of (context) nodes 5 and 6. The deletion of node 6 also leads to a cascading deletion of all descendant nodes. For these two nodes, the following context paths are now considered by the trigger.

$CP_1 =$ /root/2/5

$CP_2 =$ /root/2/6/8

$CP_3 =$ /root/2/6/9

$CP_4 =$ /root/2/6/10

$CP_5 =$ /root/2/6/11

$CP_6 =$ /root/2/6/12

The above described trigger executes six times, once for each context path. For the first execution, $con will contain node 5, as this is the context node which has context path CP_1. For executions 2 to 6, $con will contain node 6, and the transition variable $old will contain CP_2 to CP_6 respectively.

4.3 Trigger $S_I G_D$

Scope See scope of $S_I G_N$ trigger.

Granularity Within the instance scope, a document-level trigger will execute once for all context nodes in that document. As opposed to the node-level trigger $S_I G_N$ where an event instance is created for each context node, this trigger creates only one event instance.

Transition variables During trigger execution, variables $old and $new are bound to the set of all context nodes, or,

Listing 3 Delete statement
```
DELETE node(5)
node(6) (node(8) node(9) node(10)
node(11) node(12))
```

Listing 4 Trigger $S_I G_P$ example code
```
ON DELETE BELOW fn:
document(<XML doc 1>)/root/2
IF node(5) AND node(6)
FOR EACH PATH DO <trigger action>
```

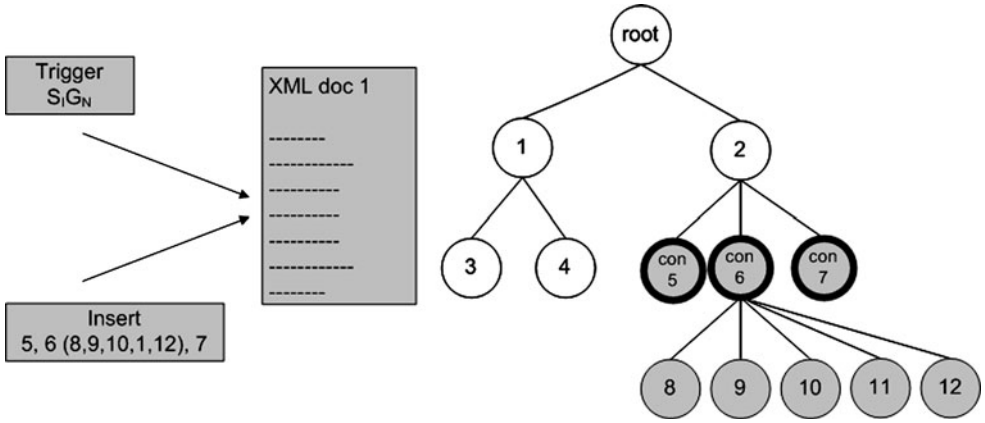


Fig. 9 Trigger $S_I G_N$ example scenario

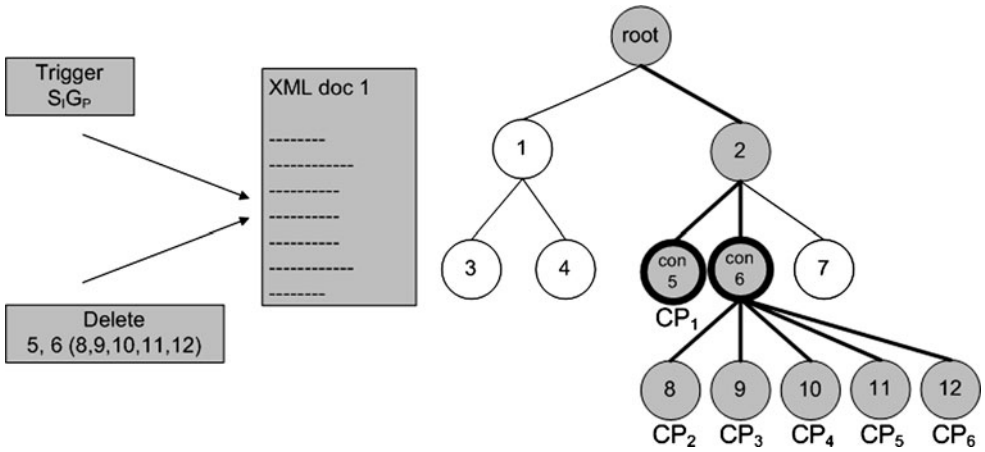


Fig. 10 Trigger $S_I G_P$ example scenario

in case of a single context node, to that node only. So, given the case that there are n context nodes when the trigger fires, $old and $new will contain a set of n nodes. The availability of these transition variables is identical to the $S_I G_N$ trigger.

Trigger execution multiplicity

– Single execution on single document if context nodes ≥ 1

This is independent of how many context nodes were identified. So even if there are many context nodes of the same type, the trigger will still only be fired a single time.

Example The example used to demonstrate the $S_I G_D$ trigger in action is the same as for the example used for the $S_I G_N$ trigger. However, there is a significant difference. First, consider the scenario in Fig. 11, the insertion operation in Listing 1, and the $S_I G_D$ trigger example in Listing 5. As pointed out in the figure by the perforated ellipse, the trigger will be executed once for all the context nodes and its transition variable $new will contain the set of these context nodes. The contents of the $new variable are shown in Listing 6.

Listing 5 Trigger $S_I G_D$ example code

```
ON INSERT BELOW fn :
document(<XML doc 1>)/ root /2
IF <trigger condition>
FOR EACH DOC DO <trigger action>
```

Listing 6 Content of $new transition variable

```
<(node 5)>
. . .
</(node 5)>
<(node 6)>
<(node 8)> . . . </(node 8)>
<(node 9)> . . . </(node 9)>
<(node 10)> . . . </(node 10)>
<(node 11)> . . . </(node 11)>
<(node 12)> . . . </(node 12)>
</(node 6)>
<(node 7)>
. . .
</(node 7)>
```

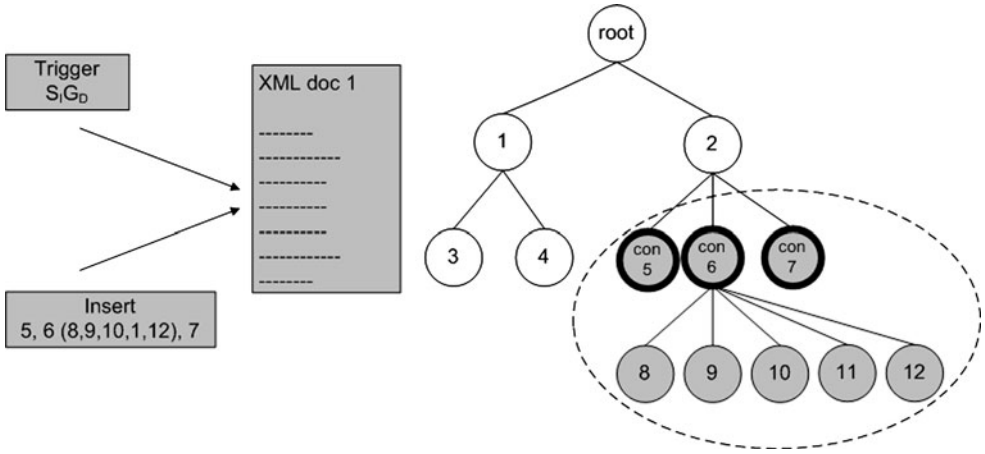


Fig. 11 Trigger $S_I G_D$ example scenario

4.4 Trigger $S_I G_D$

The trigger class $S_S G_*$ includes all schema-scope triggers $S_S G_N$, $S_S G_P$, and $S_S G_D$. The differences between the triggers in this class are the same as the instance-scope triggers', and therefore it is not necessary to discuss each individual type by itself within this section.

Scope $S_S G_*$ triggers can be applied to a collection of XML documents, rather than to a single document instance ($S_I G_*$ triggers). To specify the scope of this trigger type, the collection of documents must be specified. This is done by supplying the trigger with an XML schema definition.

Granularity Because $S_S G_*$ triggers can be seen as a collection of $S_I G_*$ triggers, the concept of granularity is the same. The only difference is that the $S_S G_N$ trigger can possibly be fired after modifications on various XML documents as opposed to a single document.

Transition variables With equivalent granularity behaviour as $S_I G_*$ triggers, the concept of the transition variables is mostly equivalent, too. Given the case that there is a collection of 100 documents $C = \{d_1, d_2, \ldots, d_{99}, d_{100}\}$ that have a common schema, and there is an $S_S G_*$ trigger that has been applied to this schema. Now given the case that some modification is being made to d_{86}, the user that implements the trigger needs a way to identify the context instance. To do this, an additional transition variable $doc is introduced that holds the instance name of the respective context instance.

Trigger execution multiplicity Refer to the $S_I G_*$ trigger class' execution multiplicity section.

Example Figure 12 illustrates the example that was already explained in the "Transition variables" part of this section. To point out the usefulness of this trigger class, the scenario

Listing 7 Trigger $S_S G_D$

```
ON INSERT fn: schema(<XML schema 1>)
IF \$doc != null
FOR EACH DOC DO <increment
the document counter>
```

Listing 8 Trigger $S_S G_P$

```
ON INSERT BELOW fn: schema(<XML
schema 1>)/root/2
IF \$doc > ' doc 99 '
FOR EACH PATH DO <record all paths
into a path index table>
```

Listing 9 Insertion of new XML document

```
INSERT DOCUMENT 'doc101. xml '
```

Listing 10 Insertion of new nodes

```
INSERT BELOW fn:
document('doc101.xml')/root/2
'<(node 5)> . . . </(node 5)>
<(node 6)> . . . </(node 6)>
<(node 7)> . . . </(node 7)>'
```

considers two consecutive operations O_1 and O_2 on the collection C of documents described by schema s_1. Two $S_S G_*$ triggers $S_S G_D$ and $S_S G_P$ are being applied to s_1 and will monitor document—and node insertions.

First, a new document d_{101}: 'doc101.xml' is to be inserted into the database. This document has s_1 associated to it. Then, three new nodes will be inserted into that document. $S_S G_D$ and $S_S G_P$ monitor insertion of new documents and insertion of new nodes for the above mentioned collection specified by s_1. Listing 7 and Listing 8 show the triggers. Listing 9 and Listing 10 show the operations performed on the database.

O_1 will trigger $S_S G_D$. The trigger's event states insertion of documents that conform to s_1. As d_{101} has this schema

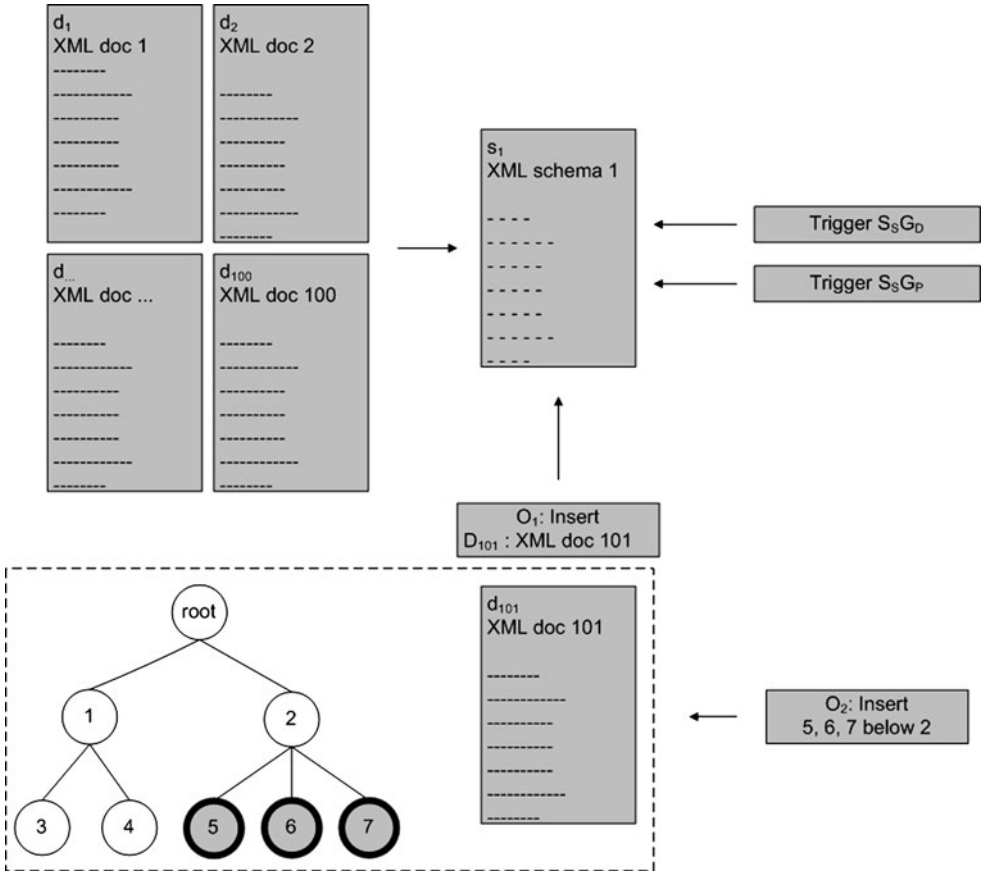


Fig. 12 Trigger class $S_S G_*$ example scenario

associated to it and trigger $S_S G_D$'s condition holds true (the document name is not null), $S_S G_D$ is executed.

For the second operation, O_2, three nodes are inserted below node 2. This insertion operation triggers $S_S G_P$, whose event path matches the insertion path. Hence, the trigger will be executed for each context path of the newly inserted nodes. As all inserted nodes are leafs, there will be three context paths considered by the trigger.

4.5 Trigger $S_{SL} G_*$

The trigger class $S_{SL} G_*$ includes all schema-less scope triggers $S_{SL} G_N$, $S_{SL} G_P$, and $S_{SL} G_D$. The differences between the triggers in this class are the same as the instance-scope triggers', and therefore it is not necessary to discuss each individual type by itself within this section.

Scope The class of $S_{SL} G_*$ triggers can be regarded as an extension to the $S_S G_*$ trigger class. It applies to documents that have no schema associated to them as well as to documents that have an associated schema, i.e. to all documents.

Granularity The concept of granularity is identical to the $S_S G_*$ trigger class'.

Transition variables Identical to the $S_S G_*$ trigger class'.

Trigger execution multiplicity Refer to the $S_I G_*$ trigger class' execution multiplicity section.

5 XML trigger execution

After this classification, it can be observed that there are certain similarities and hierarchical ordering among the trigger types.

For each level of granularity, the trigger scopes are hierarchical (see Table 2). Therefore, for each granularity level that is chosen, the scope can be selected from the above explained. The hierarchical ordering of scope levels is from left to right, where instance scope is below schema and schema-less scopes.

For each level of scope, the trigger granularity is hierarchical (see Table 3). Node granularity is hierarchically below path granularity, which again is hierarchically below document granularity. Therefore, for each scope level that is chosen, the granularity can be selected from the above explained.

5.1 Execution order

The hierarchical ordering of triggers can now be illustrated as follows (see Table 4). This is also the recommended exe-

Fig. 13 Trigger execution order

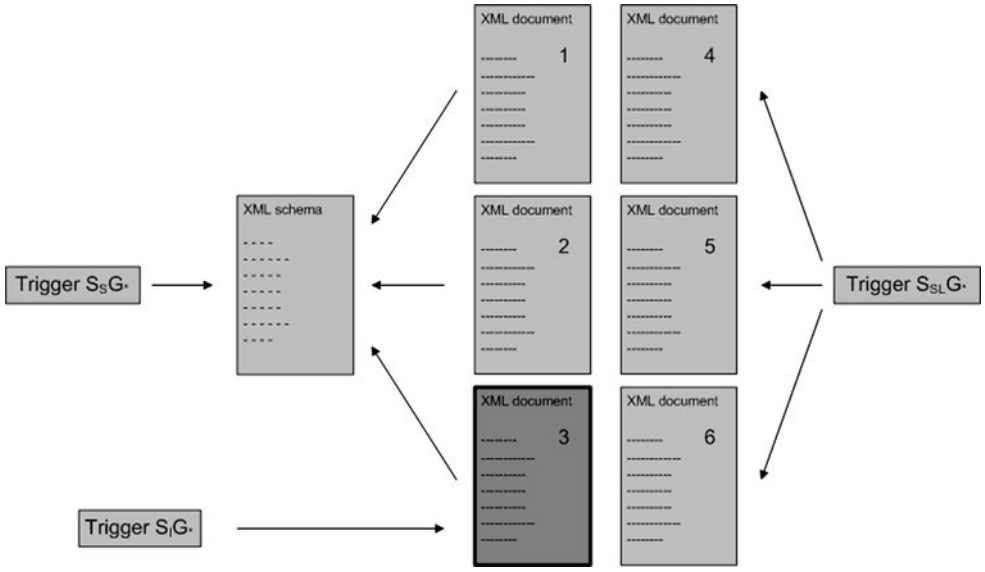


Table 2 Scope hierarchy

Granularity	Trigger scope		
	Instance	Schema	Schema-less
Node Path Document	→ Scope hierarchy →		

Table 3 Granularity hierarchy

Granularity	Trigger scope		
	Instance	Schema	Schema-less
Node Path Document	↓ Granularity hierarchy ↓		

Table 4 Execution hierarchy

Granularity	Trigger scope		
	Instance	Schema	Schema-less
Node Path Document	↘ Execution hierarchy ↘		

cution hierarchy for triggers that are fired simultaneously in response to the same update operation. It is desirable to execute the trigger types according to their hierarchical orderings from low to high, such that the updates are performed on the smallest grain of data first (node) and then on paths (sets of nodes) and documents (sets of paths).

In the following scenario there are 6 documents in the database. Documents 1–3 do conform to a schema definition, where documents 4–6 have no schema associated to them. Trigger S_SG_* is a schema-scope trigger, which means it applied to documents 1–3. Trigger $S_{SL}G_*$ applies to all documents that have no schema associated to them. Trigger S_IG_* is an instance-scope trigger which has been applied to document 3.

It is assumed that S_SG_*, S_IG_*, and $S_{SL}G_*$ have identical events and conditions. This means, that for a given update operation on document 3, they will fire simultaneously. In such a case, it is necessary to identify the order in which the different types of trigger will be executed. If the conflicting triggers are of the same type, it does not matter in which order they are executed. The below scenario only specifies the scope level of the triggers. For simplicity, the granularity it omitted.

In Fig. 13, document 3 is affected by three triggers during an update operation. All triggers are fired in response to the update. According to the execution hierarchy, trigger S_IG_* will be executed first, then trigger S_SG_*, and last trigger $S_{SL}G_*$.

6 Implementation and case study

Figure 14 gives a rough overview over the components that were necessary to realise the XML trigger support prototype. The user interface component was added in to make interaction with the database and the trigger functionalities easier and better visible to the user. The BDB XML component represents the Oracle Berkeley XML database [15] which can be accessed by the XML trigger support system using the BDB XML Connection component. It was necessary to develop two parsers in order to support (i) the XML

Fig. 14 Trigger support system architecture

trigger syntax (to create a new XML trigger) and (ii) the update syntax which had to be adjusted to be more suitable for the system.

Although the main functionalities all are carried out by the XML trigger engine component, several additional classes with helper- and utility methods were developed to support the XML trigger functionality. Figure 15 shows a screen shot of the interaction with the command line prompt when creating a new XML trigger.

The communication between the components works as follows. After the user enters a command which is recognised by the system, it is sent to the parser component for further processing. If the user has entered a trigger create statement or a database insert- or delete statement, the parser will return the respective trigger- or query object. This parsed input is then passed to the XML trigger engine which determines the type of database operation and initiates associated trigger processing. When data needs to be analysed or evaluated which resides in the database, then the BDB XML Connection component is invoked. It has methods that enable access to the XML database (represented by the BDB XML component) in various ways.

The XML trigger engine is the core component of the XML trigger support system. Its main functionalities include the detection of triggers that can be fired in response to an update statement, the evaluation of these possible triggers, and the execution of triggers whose evaluation was successful.

Implementation tools and system The prototype is implemented in JAVA SE 5.0 and utilises an existing API to interact with the Berkeley XML database. The Eclipse IDE 3.2.2 was used to develop the system.

6.1 Case study

Product information for a large online-shopping website is stored in an XML database. For each product, there is one XML document. Particular information, such as the product name and price, is often retrieved separately so that subscribers to this service can make price comparisons with similar products. One document in the catalogue is used to promote lowest-price products. Hence, it is necessary to keep this document up-to-date, whenever new products are added or old products are taken out of the visible catalogue.

Another important task of the database system is to maintain a collection of meta-data keywords for each product. Preferably, the extraction of keywords is done automatically

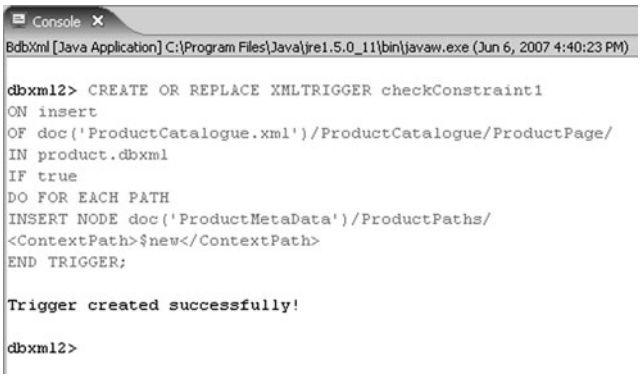


Fig. 15 Create a path-level XML trigger

Listing 11 Validate product ID $S_S G_N$ trigger

```
1 INSERT XMLTRIGGER
  validateProductIdTrigger
2 ON insert OF schema
  ('ProductCatalogue.xsd')
  /ProductPage/FeatureProduct
3 IN product.dbxml
4 IF true
5
6 DO FOR EACH NODE
7
8 IF ( substring( \$new [@id], 1, 3 )
  != 'PRD' )
9 THEN RAISE\_ERROR()
10
11 END TRIGGER;
```

upon new product insertion. Figure 16 illustrates the case study. Further, it must be ensured that the format of the product ID is correct. To maintain and check the complex constraints on this database, we create five XML triggers that monitor insertions, updates, and delete statements to the database.

Triggers The first trigger *validateProductIdTrigger* (Listing 11) ensures that the format of the product ID is correct and starts with the three capital letters 'PRD'. When new products are inserted into the database, this trigger is executed for each *FeatureProduct* node that is inserted, and its ID is validated. If the validation fails, an error is raised.

The second trigger *alertBargainTrigger* (Listing 12) checks if a new product was inserted into the lowest prices document. For each newly inserted product, the trigger

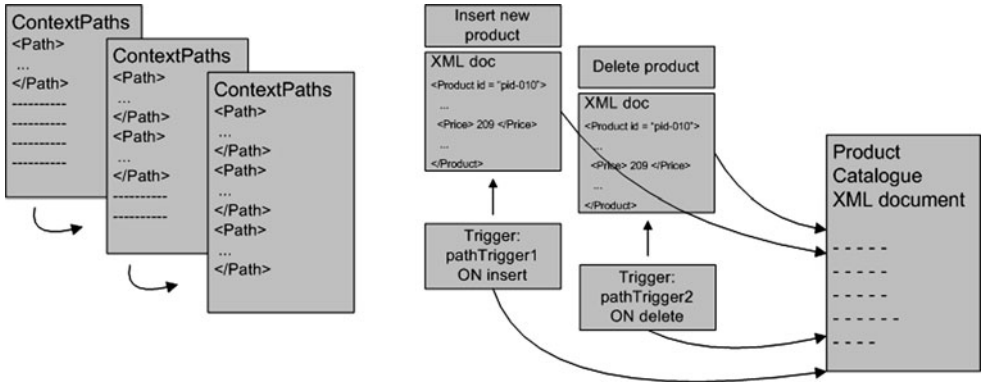


Fig. 16 Product catalogue scenario

Listing 12 Alert when a new lowest price product is inserted $S_I G_N$ trigger

```
1 INSERT XMLTRIGGER alertBargainTrigger
2 ON insert OF doc('LowestPrices.xml')/
3 IN product.dbxml
4 IF true
5
6 DO FOR EACH NODE
7
8 IF ( \$new/ProductPage/FeatureProduct
  /PriceRange[@min]
9 < doc('LowestPrices.xml')
  /PriceInfo/MinPrice )
10 THEN SEND\_BARGAIN\_ALERT ()
11
12 END TRIGGER;
```

sends a notification alert if the price of the inserted product has the currently lowest price.

The third trigger *updateMetadataTrigger* (Listing 13) is a path-level trigger that monitors insertions of new product records and then extracts the content of each context path's text node. This extracted value is then inserted into a document that records products' meta-data.

The fourth trigger *updateLowestPricesTrigger* (Listing 14) is an document-level trigger ($S_S G_D$) which retrieves the minimum price attribute from the product's price range and makes a check against a document which stores lowest and highest prices. If the minimum price of the newly inserted product is the lowest, then it will be replaced and a new version of the lowest-prices document is created.

The fifth trigger *returnToPrevVersionTrigger* (Listing 15) monitors documents deletions in the database. If the product is deleted whose price is listed as the lowest one in the price-info document, this deletion must either (i) be prevented and the next higher layer (e.g. application layer) must be informed, or (ii) the lowest-price document is set back to the previous version. In this scenario the trigger will raise an application error. The granularity of this trigger is document-level ($S_S G_D$).

Listing 13 Update meta-data $S_S G_P$ trigger

```
1 INSERT XMLTRIGGER
  updateMetadataTrigger
2 ON insert OF doc('ProductPage.xml ')
  / IN product.dbxml
3 IF true
4
5 DO FOR EACH PATH
6
7 \$id = \$new/ProductPage
  /FeatureProduct[@id]
8
9 INSERT NODE doc('ProductMetaData')
  /Products/
10 Product[ @id = \$id ]/Keyword
11 <Keyword>
12 \$new/ProductPage/descendant::text()
13 </Keyword>
14
15 END TRIGGER;
```

Evaluation The insertion of a new product document caused the path-level trigger *updateMetadataTrigger* to execute n number of times, where n is the number of context paths that were identified for the context node. For each of the paths, the text node was extracted and its content recorded.

The *updateLowestPricesTrigger* trigger that was fired in response to the insertion caused a new version of the lowest-prices document to be created and at the same time the *alertBargainTrigger* trigger sent a notification because the product inserted had the lowest price that had been recorded to-date. The XML data from the old version was written into a new file, and the file was then modified, i.e. the price was updated.

The delete operation caused the *returnToPrevVersionTrigger* trigger to be fired. The version of the current lowest-price document was detected and then deleted. Then, the previous version of that document was copied to become the current version.

Listing 14 Update low prices $S_S G_D$ trigger

```
1 INSERT XMLTRIGGER
  updateLowestPricesTrigger
2 ON insert OF schema
  (‘ProductCatalogue.xsd’)
  / IN product.dbxml
3 IF doc(’ LowestPrices .xml’)
  / PriceInfo / MinPrice
4 [. <\$ new/ProductPage/FeatureProduct
  /PriceRange[@min]]
5 DO FOR EACH DOC
6 \$ version = doc (’LowestPrices’)
  /PriceInfo[@version]
7 COPY ( doc(’LowestPrices’)) doc
  (’LowestPrices\_v’ + \$version )
8 \$version += 1
9 UPDATE ATTRIBUTE
10 doc (’LowestPrices.xml’)
  /PriceInfo[@version] = \$version
11 DELETE doc(’ LowestPrices ’)
  /PriceInfo/MinPrice
12 INSERT doc(’ LowestPrices ’)
  /PriceInfo/MinPrice
13 <MinPrice prodIdRef =
  ’\$new/ProductPage
  /FeatureProduct[@id]’>
14 doc (’LowestPrices.xml’)
  /PriceInfo/MinPrice/text()
  </MinPrice>
15 END TRIGGER;
```

Listing 15 Return to previous version $S_S G_D$ trigger

```
1 INSERT XMLTRIGGER
  returnToPrevVersionTrigger
2 ON delete OF schema
  (’ProductCatalogue.xsd’)
  /IN product.dbxml
3 IF doc(’LowestPrices’)/ PriceInfo
  / MinPrice
4 [.= \$new/ProductPage/FeatureProduct
  /PriceRange[@min]]
5 DO FOR EACH DOC
6 \$ version = doc (’LowestPrices’)
  /PriceInfo[@version]
7 DELETE DOCUMENT doc(’LowestPrices’)
8 \$version -= 1
9 COPY ( doc(’LowestPrices’
  + \$version )) doc (’LowestPrices’)
10 END TRIGGER;
```

7 Experimental Performance Evaluation

In this section, we evaluate the effectiveness of the proposed trigger methodology. We compare it to a traditional approach, whereby the trigger is replaced by a sequence of manual XQuery updates. The following figures show how the comparison is made against manual XQuery updates which perform similar or the same functionality. As can be seen, the traditional approach requires context path traversals of all possible context nodes. In the new method, the context node is known, and using our proposed trigger methodology, the trigger will only be executed on the relevant context paths.

We compare the two approaches by running queries against individual XML documents. The reason why we tested against individual documents is so that we could have a fairer comparison with a manually implemented XQuery approach. Further, we decided to perform our comparisons on the lowest scope level (instance scope) because a test on multiple document instances would result in an equal increase in cost for both approaches. Thus, a comparison on a document-collection can be deduced by multiplying the cost of processing a single document by the number of documents in the collection, given that the cost of processing each document is the same.

The tests described in this section were conducted using the above described implementation of a trigger engine on a machine equipped with a 2.0 GHz Intel Pentium M 760 processor and 1 GB DDR2 memory, running Windows XP professional as operating system.

As there is no trigger functionality available in existing XML databases, we used our developed trigger engine that incorporates path-level granularity as well as the above suggested XML trigger syntax.

We used a set of eight equally structured XML documents, all of which had been generated from the SIGMOD Record that is available online. The largest set contained $n = 23047$ nodes, the next largest $\frac{7}{8}n$ nodes, until finally the smallest set contained $\frac{n}{8}$ nodes.

The queries and triggers were divided into three test groups.

In the first group, a trigger was applied to the context node, and all queries were executed on the root node, such that the number of context nodes was equal to 1, and the number of context paths was equal to n, where n is equal to the number of leaf nodes in the document. In the second group, the trigger was applied to an intermediate node, and test queries were executed such that we generated various ratios of actual context nodes versus possible context nodes. In the third group, triggers were applied to leaf nodes, and therefore, executing queries resulted in a number of context paths equal to 1.

This test configuration allows us to thoroughly test the path-trigger against a traditional alternative method in all

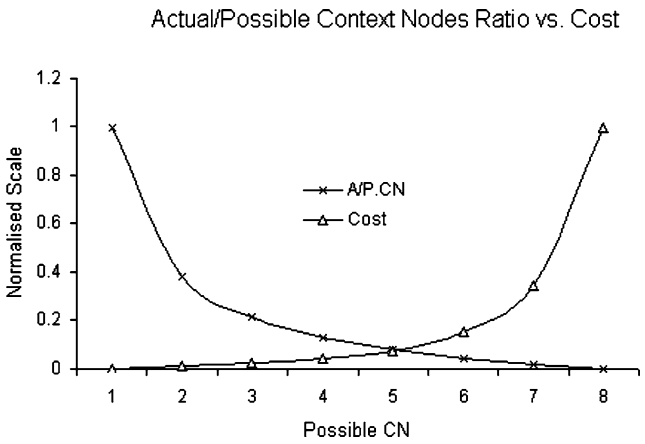


Fig. 17 Actual/possible context nodes ratio vs. cost

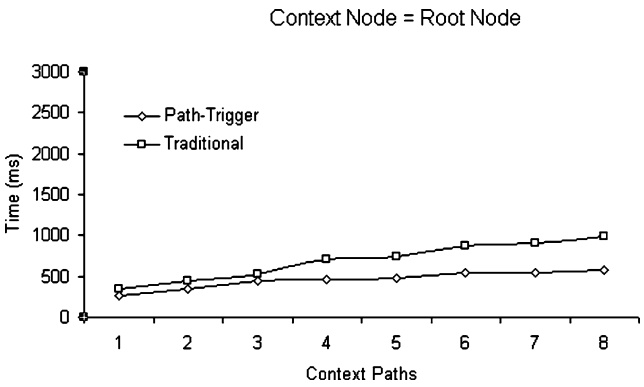


Fig. 18 Context node is root node

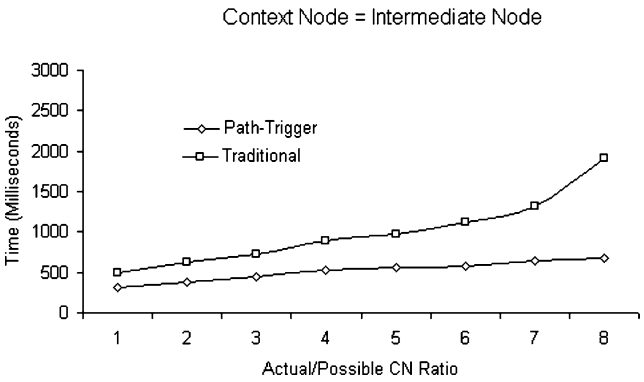


Fig. 19 Context node is intermediate node

possible scenarios, and to analyse how both methods perform.

The major advantage of the XML trigger approach is that it identifies actual context nodes on compile-time, as opposed to the traditional approach, where context nodes are identified on query-time. In order to make the difference between the two approaches more clear, refer to the two graphs in Fig. 17. They represent normalised ratios of actual/possible context nodes (A/P.CN), and the related computational cost (Cost). The cost curve in this figure represents the additional computational cost that the traditional approach would require as opposed to the new path-trigger.

Graph A/P.CN is created by choosing an arbitrary number of (intermediate) context nodes from the smallest document by querying for example /SigmodRecord/issue/articles/article [authors/author='Michael Stonebraker'], and then querying all other documents in such a way that the number of actual (identified) context nodes stays constant. This means that with growing document sizes and constant number of actual context nodes, the ratio actual/possible context nodes will decline.

The graph that represents the cost is opposite proportional of the former graph, and represents the expected performance differences between the approaches. With declining ratios A/P.CN, the computational cost will exponentially increase for the traditional method, as no context nodes are known in advance, and an increasing number of redundant node checks need to be performed.

Figure 18 shows the result of the first set of experiments, as previously described for scenario 1. Here, we choose the root node to be the context node. This scenario is an exceptional case, as none of the approaches clearly outperform each other. The reason for this is that the ratio A/P.CN is 1, and therefore we don't expect any large differences in performance. Nevertheless, we observe that our approach performs slightly better. This is due to the more effective way of identifying and assembling context paths at trigger execution. The XML trigger can do context path identification on compile-time, so this is the cost difference that is visible in the graphs.

The second scenario is tested, and its results are displayed in Fig. 19. The x-axis describes a decreasing A/P.CN ratio for test documents 1 to 8, and we can now start to see the resemblance with our expected outcomes from Fig. 17. An intermediate node has been chosen to be the context node, and with an increasing number of sibling nodes that are possible (before trigger evaluation), but not actual (after successful trigger evaluation) context nodes, the performance gap grows larger.

Our test results for the last scenario are given in Fig. 20, where the context node is a leaf node, and therefore, the number of context paths and decreasing A/P.CN ratios, the XML trigger clearly outperforms the traditional approach, and our test results show the closest resemblance with our expected graph in Fig. 17. The reason for this is that the traditional approach requires traversing all possible context nodes, all which are leaf nodes, and leads to a very clear exponential growth in cost. On the other side, the XML trigger stays almost constant in cost as opposed to the traditional method, and even performs better than in scenario 2.

Figure 21 summarises the resulting graphs of the test scenarios, and shows the performance patterns of the traditional and new approaches. It can be observed that for XML trig-

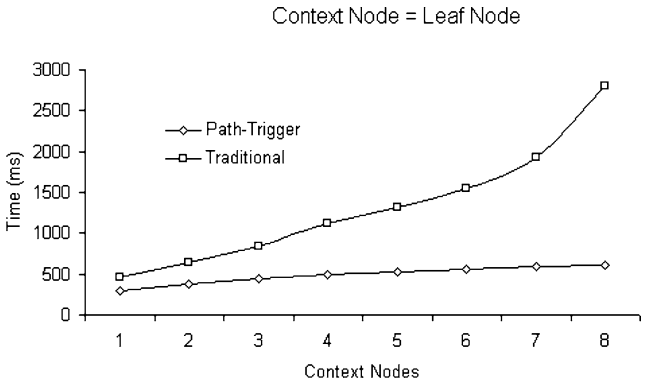


Fig. 20 Context node is leaf node

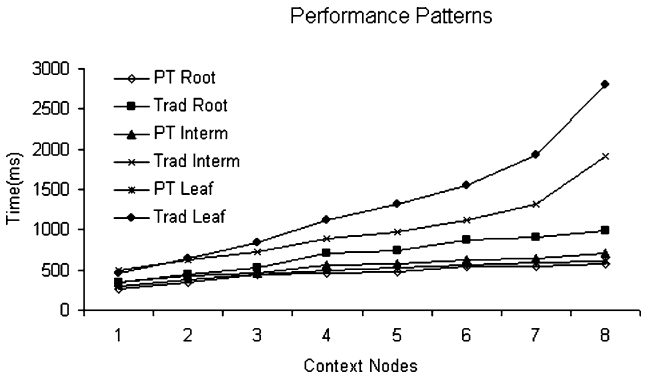


Fig. 21 Performance patterns

gers that operate on leaf and intermediate nodes give the largest difference in performance when being executed. Leaf nodes are usually the nodes where the actual data is stored, and therefore are queried most often, followed by intermediate nodes, which represent aggregations or summaries of the underlying data. For this reason, the XML path-trigger is much preferred, as its performance addresses especially these node types.

Summarising the above graphs, it can be said that if the number of context nodes for which the trigger executes is equal to the number of possible context nodes, the traditional method performs nearly at the same efficiency, as nearly no redundant traversals and node checks are necessary. However, with increasing difference between possible context nodes and identified context nodes, the new method clearly outperforms the traditional approach.

8 Related work

This chapter provides a summary of the literature review that was conducted in the area of XML triggers. Besides a very limited amount of previous research in this new area, the review confirmed that several aspects of XML triggers had not been covered in enough detail and some aspects had not been considered yet. In the following sections the literature review's findings are summarised and discussed according to a set of predefined criteria.

Trigger granularity Despite the big differences between relational and XML data, the levels of granularity were almost directly adopted from the relational model by all existing literature. What is in the relational context row-level and statement-level has now become node- level and document-level granularity [5]. Some approaches refer to document-granularity as set-granularity [8] or statement-granularity [6]. Another group only addresses the node-level granularity for the reason of simplicity in presenting the approach [7, 16]. Although different terms are used, they describe one and the same thing. So is the node level trigger equivalent to the row-level trigger and the document- or set-level trigger equivalent to the statement-level trigger, for XML and relational contexts respectively. Further granularity levels, such as the path-granularity, are not discussed by any previous research works. In a previous paper, we proposed path-granularity, however, not in conjunction with the notion of trigger scope [17]. Also, in this paper we have further elaborated the implications on extending XML triggers with the combination of path-granularity and scope, which includes the execution hierarchies and classifications of XML triggers.

Transition variables The largest group of works refers to the trigger's transition variables as `old` and `new` (or `old_node` and `new_node`) [6–8, 16]. In each case, these variables represent the affected nodes before and after the update operation has taken place. One approach remains where a single transition variable `$delta` is used [5].

A rather unusual way of dealing with transition variables is introduced in [5], which also has a remarkable way of dealing with the distinction between different levels of granularity. First, the paper introduces the transition variable `$delta`, which represents the set of nodes that was inserted/deleted by the simple XPath expression e in the event part. `$delta` is available in both the condition and action parts. Depending on whether the variable is being used or not, the trigger is said to be of node-level or document-level granularity, respectively.

Other approaches [6, 7] have the same set of transition variables for both node and document level granularities.

The types of transition variables that are available in a trigger, depends on the type of the trigger itself. Conceptually, this is similar to the relational database triggers, where for example an INSERT trigger only has the *:new* transition variable, which holds the value of the newly inserted data. With XML triggers, this concept has been adopted by the majority of approaches. However, some strategies in the choice of transition variables, such as for example Bailey et

Table 5 Comparison of existing XML trigger approaches

	[8]	[16]	[5]	[6]	[7]
Trigger structure					
Event-Condition-Action	√	√	√	√	√
ECA plus additional components	X	X	X	√	√
Granularity					
Node-level	√	√	√	√	√
Document-level	√	X	√	√	X
Considers node relationships	X	X	X	X	X
Trigger execution					
Update decomposition	X	X	X	√	X
Compile-time analysis	X	X	X	X	√
Trigger scope					
Instance scope	√	√	√	√	√
Schema scope	X	X	X	X	X
Other levels of scope	X	X	X	X	X

al. [5], are not very efficient, because they only provide a single type of variable. The drawback of this is that it cannot be distinguished whether this transition variable refers to new or old data. Another shortcoming is that it is unusable for the distinction between node and document granularity, because for node granularity the transition variable must be a set of document nodes, whereas for document granularity the transition variable is an XML subset (subtree).

Scope One limitation that has been overseen by all analysed literature is found in the context of the trigger's scope. The scope levels that have been proposed so far are incomplete as they do not address the fact that a trigger may not only be applied to document instances, but may well be applied to a schema (if existent) as well. A "multi-document" support is not possible with existing trigger techniques, which means that a trigger does not yet have the functionality to be applied to a collection of XML documents.

Triggers in general represent an important factor to add a dynamic and reactive feature to database systems. It has been found that not only traditional tasks of the XML trigger are in demand, such as enforcement of constraint violations and view updates. New application domains such as the support of business rules in applications are increasingly popular for XML triggers. The analysis of recent research in the area of triggers for XML databases shows that there are some unresolved problems which may be the reason that there is still no standardized XML trigger. A summarising comparison is shown in Table 5.

This research paper has proposed methods to fill in these gaps and made a contribution to the field of XML triggers.

9 Summary and future works

This paper has proposed two new concepts for XML triggers, namely path-granularity and scope. Resulting from these novel XML trigger extensions, we have constructed a categorisation of trigger types, and have introduced a formal model to express them.

Path-granularity adds a further level of fine-grained document segmentation during trigger execution, by which horizontally related ancestor and descendant nodes are bound to context-paths and can be accessed and manipulated when the trigger is fired.

Trigger scope is an extension that enables XML triggers to be applied to more than one document. A trigger can be applied to document instances and schemas. In the latter case, the trigger applies to all documents that have the schema registered.

We have analysed and evaluated our approach in respect to the execution strategies that will govern the different trigger types, and implemented a prototype trigger execution engine. Finally, a performance evaluation discusses test results of the new trigger types against traditional methods.

Future work involves optimising the context-path detection and further exploring application areas where the path-level granularity and trigger scope can be effectively used.

The conclusion of this paper is, that the proposed concepts of XML trigger granularity and scope have proven to be a beneficial to XML trigger approaches, and it can improve the robustness and integrity of XML databases.

References

1. Eswaran KP (1976) Specifications, implementations and interactions of a trigger subsystem in an integrated database system. Tech rep, IBM Research Report RJ 1820, IBM San Jose Research Laboratory, San Jose, California
2. Ceri S, Cochrane R, WidoM J (2000) Practical applications of triggers and constraints: Success and lingering issues (10-year award). In: VLDB. Morgan Kauffman, San Mateo, pp 254–262
3. Fan W, Siméon J (2000) Integrity constraints for XML. In: PODS. ACM, New York, pp 23–34
4. Fan W (2007) XML publishing: Bridging theory and practice. In: DBPL. Springer, Berlin, pp 1–16
5. Bailey J, Poulovassilis A, Wood PT (2002) An event-condition-action language for XML. In: WWW. ACM, New York, pp 486–495
6. Bonifati A, Braga D, Campi A, Ceri S (2002) Active XQuery. In: ICDE. IEEE, New York, pp 403–412
7. Rekouts M (2005) Incorporating active rules processing into update execution in XML database systems. In: DEXA workshops. IEEE, New York, pp 831–836
8. Bonifati A, Ceri S, Paraboschi S (2001) Active rules for XML: A new paradigm for e-services. VLDB J 10(1):39–47
9. W3C. W3c recommendations. Available at: http://www.w3.org/TR/#Recommendations
10. Tatarinov I, Ives ZG, Halevy AY, Weld DS (2001) Updating XML. In: SIGMOD. ACM, New York, pp 413–424

11. Barbosa D, Mendelzon AO, Libkin L, Mignet L, Arenas M (2004) Efficient incremental validation of XML documents. In: ICDE. IEEE, New York, pp 671–682
12. Bailey J, Poulovassilis A, Wood PT (2002) Analysis and optimisation of event- condition-action rules on XML. Comput Netw 39(3):239–259
13. Grust T (2002) Accelerating XPath location steps. In: SIGMOD. ACM, New York, pp 109–120
14. Jagadish HV, Al-Khalifa S, Chapman A, Lakshmanan LVS, Nierman A, Paparizos S, Patel JM, Srivastava D, Wiwatwattana N, Wu Y, Yu C (2002) Timber: A native XML database. VLDB J 11(4):274–291
15. Brian D (2006) The definitive guide to Berkeley DB XML (Definitive guide). Apress, Berkely
16. Bonifati A, Ceri S, Paraboschi S (2001) Pushing reactive services to XML repositories using active rules. In: WWW. ACM, New York, pp 633–641
17. Landberg AH, Rahayu JW, Pardede E (2007) Extending XML triggers with path-granularity. In: WISE. Springer, Berlin, pp 410–422
18. Wang F, Zaniolo C, Zhou X (2008) ArchIS: an XML-based approach to transaction-time temporal database systems. VLDB J 17(6):1445–1463
19. Marian A, Abiteboul S, Cobena G, Migner L (2001) Change-centric management of versions in an XML warehouse. In: VLDB. Morgan Kaufmann, San Mateo, pp 581–590
20. Ghelli G, Onose N, Rose K, Simeon J (2008) XML query optimization in the presence of side effects. In: Proceedings of the 2008 ACM SIGMOD
21. Benedikt M, Bonifati A, Flesca S, Vyas A (2005) Verification of tree updates for optimization. In: Lecture notes in computer science, vol 3576. Springer, Berlin, pp 379–393
22. Landberg AH, Rahayu JW, Pardede E (2010) Privacy-aware access control in XML databases. In: ADC. CRPIT, ACS, Washington, pp 85–92

Anders H. Landberg is a PhD candidate at the Department of Computer Science and Computer Engineering La Trobe University and is supervised by Assoc. Prof. Wenny Rahayu and Dr. Eric Pardede. His research areas include Data Privacy, Data Security, XML Databases, Health Information Systems.

J. Wenny Rahayu is an Associate Professor at the Department of Computer Science and Computer Engineering La Trobe University. Her research areas cover a wide range of advanced databases topics including XML Databases and Data Warehousing, Object-Relational Databases, and Semantic Web and Ontology. She is currently the Head of Data Engineering and Knowledge Management Laboratory at La Trobe University.

Eric Pardede is a Lecturer at the Department of Computer Science and Computer Engineering La Trobe University. His current research areas include Data Modeling, XML Databases, Data Quality, Information Systems and Software Engineering.