XQuery Update for the impatient

A quick introduction to the XQuery Update Facility

Xavier Franc

May 25, 2025

Table of Contents

The XQuery Update Facility is a relatively small extension of the XQuery language which provides convenient means of modifying XML documents or data. As of March 14, 2008, the XQuery Update Facility specification has become a "Candidate Recommendation", which means it is now pretty stable.

Why an update facility in XML Query? The answer seems obvious, yet after all the XQuery language itself - or its cousin XSLT2 - is powerful enough to write any transformation of an XML tree. Therefore a simple "store" or "put" function, applied to the result of such transformation, could seem sufficient to achieve any kind of database update operation. Well, perhaps. In practice this would be neither very natural, convenient, nor very efficient (such an approach requires storing back the whole document and makes optimizing very difficult). So as we will see the little complexity added by XQuery Update seems quite worth the effort.

This tutorial attempts to give a quick yet comprehensive practical introduction to the XQuery Update extension, while highlighting some of its peculiarities.

Prerequisites: the reader is presumed to have some acquaintance with XML Query and its *Data Model* (the abstract representation of XML data, involving *nodes* of six types: document, element, attribute, text, comment, processing-instruction).

1. Basics

1.1. The update primitives

The XQuery Update Facility (abbreviated as **XQUF** hereafter) provides five basic operations acting upon XML *nodes*:

- **insert** one or several nodes inside/after/before a specified node
- **delete** one or several nodes
- **replace** a node (and all its descendants if it is an element) by a sequence of nodes.

- **replace** the **contents** (children) of a node with a sequence of nodes, or the **value** of a node with a string value.
- **rename** a node (applicable to elements, attributes and processing instructions) without affecting its contents or attributes.

These primitives are detailed hereafter.

1.2. How do the update primitives combine with the base language?

Typically, we use some plain query to select the node(s) we want to update, then we apply update operations on those nodes. This is similar to the SQL **`UPDATE ... WHERE ...`** instruction.

Example 1: in a document `doc.xml`, rename as `SECTION_TITLE` all `TITLE` elements children of a `SECTION`:

```
for $title in doc("doc.xml")//SECTION/TITLE   (: selection :)
return rename node $title as SECTION_TITLE    (: update :)
```

Example 2: for all `ITEM` elements which have an attribute Id, replace that attribute with a child `NID` in first position:

```
for $idattr in doc("data.xml")//ITEM/@Id     (: selection :)
return (
   delete node $idattr,                      (: update 1 :)
   insert node <NID>{string($idattr)}</NID>  (: update 2 :)
      as first into $idattr/..
)
```

With the latter script the following fragment

```
<ITEM Id="id123">some content</ITEM>
```

would be modified into:

```
<ITEM><NID>id123</NID>some content</ITEM>
```

In the latter example, it is completely irrelevant whether the **delete** is written after or before the **insert node**. This surprising property of XQUF is explained below.

There are some restrictions in the way the 5 updating operations can mix with the base XQuery language. XQUF makes a distinction between *Updating Expressions* (which encompass update primitives) and non-updating expressions. Updating Expressions cannot appear anywhere. This topic will be explained in more detail.

1.3. Processing models

There are two main ways of using the update primitives:

Direct update of an XML database:

In the examples above, nodes belonging to a database are selected then updated.

The XQUF notion of a database is very general: it means any collection of XML documents or well-formed fragments (trees).

XQuery Update does not define precisely the protocol by which updating operations are applied to a database. This is left to implementations. For example transaction and isolation issues are not addressed by the specifications.

It is simply assumed that updates are applied to the database when the execution of a script completes. The language is designed in such a way that semantics of the "apply-updates" operation are precisely defined, yet as much space as possible is left for optimization by database implementations.

Points to be noticed:

- *Updates are not applied immediately as the updating expression executes*. Instead they are accumulated into a "Pending Update List". At some point at the end of the execution, Pending Updates are applied all at once, and the database is updated atomically.

  A noticeable consequence is that *updates are not visible during the script execution*, but only after. This can be fairly off-putting for a developer. It also has a definite influence on programming style. We will see later examples of this effect and how to cope with it.

- The same expression can update several documents at once. The examples above could be applied to any collection of documents instead of the single document doc.xml. Example:

  ```
  for $title in collection("/allbooks")//SECTION/TITLE
  return rename node $title as SECTION_TITLE
  ```

Transforms without side effects:

The XQUF has a supplementary operation called *transform* which updates a *copy* of an existing node, without modifying the original, and returns the transformed tree.

The following example produces a transformed version of doc.xml without actually touching the original document:

```
copy $d := doc("doc.xml")
modify (
  for $t in $d//SECTION/TITLE
  return rename node $t as SECTION_TITLE
 )
return $d
```

Notice that *within the modify clause*, XQUF forbids modifying the *original version* of copied trees (here the document doc.xml itself); only the copied trees can be modified. The following expression would cause an error:

```
copy $d := doc("doc.xml")
modify (
  for $t in doc("doc.xml")//SECTION/TITLE (: *** wrong *** :)
  return rename node $t as SECTION_TITLE
 )
```

```
return $d
```

2. Going deeper

2.1. Primitive operations revisited

delete nodes

Syntax:

```
delete node location
```

```
delete nodes location
```

The expression `location` represents a sequence of nodes which are marked for deletion (the actual number of nodes does not need to match the keyword **node** or **nodes**).

insert nodes

Syntax:

```
insert (node|nodes) items into location
```

```
insert (node|nodes) items as first into location
```

```
insert (node|nodes) items as last into location
```

```
insert (node|nodes) items before location
```

```
insert (node|nodes) items after location
```

The expression `location` must point to a single target node.

The expression `items` must yield a sequence of items to insert relatively to the target node.

Notice that even though the keyword **node** or **nodes** is used, the inserted items can be non-node items. What happens actually is that the *string values* of non-node items are concatenated to form text nodes.

- If either form of **into** is used, then the target node must be an element or a document. The items to insert are treated exactly as the contents of an element constructor.

  For example if `$target` points to an empty element `<CONT/>`,

  ```
  insert nodes (attribute A { 2.1 }, <child1/>, "text", 1 to 3)
  into $target
  ```

  yields:

  ```
  <CONT A="2.1"><child1/>text 1 2 3</CONT>
  ```

Therefore the same rules as in constructors apply: item order is preserved, a space is inserted between consecutive non-node items, inserted nodes are copied first, attribute nodes are not allowed after other item types, etc.

- When the keywords **as first** (resp. **as last**) are used, the items are inserted before (resp. after) any existing children of the element.

  For example if `$target` points to an element `<parent><kid></parent>`

  ```
  insert node <elder/> as first into $target
  ```

  yields:

  ```
  <parent><elder/><kid></parent>
  ```

  When the only keyword **into** is used, the resulting position is implementation dependent. It is only guaranteed that **as first into** and **as last into** have priority over **into**.

- If **before** or **after** are used, any node type is allowed for the target node.
- Attributes are a special case: regardless of the **before** or **after** keyword used, attributes are always inserted **into** the *parent element* of the target. The order of inserted attributes is unspecified. Name conflicts can generate errors.

replace node

Syntax:

```
replace node location with items
```

The expression *`location`* must point to a single target node.

The expression *`items`* must yield a sequence of items that will replace the target node.

- Except for `document` and `attribute` node types, the target node can be replaced by any sequence of items. The replacing items are treated exactly as the contents of an element/document constructor.

  For example if `$target` points to an element `<P><kid/>some text</P>`,

  ```
  replace node $target/kid with "here is"
  ```

  yields:

  ```
  <P>here is some text</P>
  ```

- Attributes are a special case: they can only be replaced by an attribute node. Name conflicts can generate errors.

replace value of node

Syntax:

```
replace value of node location with items
```

Here the identity of the target node is preserved. Only its value or contents (for an element or a document) is replaced.

- If the target is an element or a document node, then all its former children are removed and replaced. The replacing items are treated exactly as the contents of a text constructor (so all node items are replaced by their string-value).

  For example if `$target` points to an element `<P><kid/>some text</P>`,

  ```
  replace value of node $target with (<text>let's count: </text>, 1 to 3, "...")
  ```

  yields:

  ```
  <P>let's count: 1 2 3 ...</P>
  ```

  So the element contents are replaced by a *text node* whose value is the concatenation of the string values of replacing items.

- If the target node is a leaf node (attribute, text, comment, processing-instruction) then its string value is replaced by the concatenation of the string values of replacing items.

  For example if `$target` points to an element `<P order="old">some text</P>`,

  ```
  replace value of node $target/@order with (1 to 3, <ell>...</ell>)
  ```

  yields:

  ```
  <P order="1 2 3...">some text</P>
  ```

rename node

Syntax:

```
rename node location as name-expression
```

The expression *`location`* must point to a single target element, attribute or processing-instruction.

The expression *`name-expression`* must yield a single QName or string item.

For example if `$target` points to an element `<CONT A="a">some text</CONT>`

```
rename node $target as qName("some.namespace", "CONTAINER"),
rename node $target/A as "NEWA"
```

yields:

```
<ns1:CONTAINER NEWA="a" xmlns:ns1="some.namespace">some text</ns1:CONTAINER>
```

transform

Syntax:

```
copy $var := node [, $var2 := node2 ...]
modify updating-expression
```

```
return expression
```

Each `node` expression is copied (at least virtually) and bound to a variable.

The `updating-expression` contains or invokes one or several update primitives. These primitives *are allowed to act only upon the copied XML trees*, pointed by the bound variables. Therefore the transform expression has no side effect.

Before the `return` expression is evaluated, all updates are applied to the copied trees. Typically the `return` expression would be a bound variable, or a node constructor involving the bound variables, so it will yield the updated tree(s).

For example if `$target` points to an element

```
copy $target := <CONT id="s1">some text</CONT>
modify (
   rename node $target as "SECTION",
   insert node <TITLE>The title</TITLE> as first into $target
)
return element DOC { $target }
```

returns:

```
<DOC><SECTION id="s1"><TITLE>The title</TITLE>some text</SECTION></DOC>
```

2.2. The Problem of Invisible Updates

The fact that updates are applied only at the end of a script execution has two consequences on programming, one disturbing, one pleasant:

- The disturbing consequence is that you don't see your updates until the end, therefore you cannot build on your changes to make other changes.

  An example: suppose you have elements named PERSON. Inside a PERSON there can be a list of BID elements (representing bids made by this person), and you want the BID elements to be wrapped in a BIDS element. But initially the PERSON has no BIDS child.

  Initially:

  ```
  <PERSON id="p0234">
    <NAME>Joe</NAME>
  </PERSON>
  ```

  We want to insert `<BID id="b0012">data</BID>` to obtain:

  ```
  <PERSON id="p0234">
    <NAME>Joe</NAME>
    <BIDS>
       <BID id="b0012">data</BID>
    <BIDS>
  </PERSON>
  ```

  Classically, for example using the DOM, we would proceed in two steps:

1. If there is no BIDS element inside PERSON, then create one
2. *then* insert the BID element inside the BIDS element

In XQuery Update this would (incorrectly) be written like this:

```
declare updating function insert-bid($person, $bid)
{
  if(empty($person/BIDS))
    then insert node <BIDS/> into $person
    else (),
  insert node $bid as last into $person/BIDS
}
```

Don't try that: it won't work! Why? Because the BIDS element will be created only at the very end, therefore the instruction `insert ... as last into $person/BIDS` will not find any node matching `$person/BIDS`, hence an execution error.

So what is a correct way of doing ? We need a self-sufficient solution for each of the two cases:

```
declare updating function insert-bid($person, $bid)
{
 if(empty($person/BIDS))
   then insert node <BIDS>{$bids}</BIDS> into $person
   else insert node $bid as last into $person/BIDS
}
```

- The pleasant consequence is that the document(s) on which you are working are stable during execution of your script. You can rest assured that you are not sawing the branch you are sitting on. For example you can quietly write:

  ```
  for $x in collection(...)//X
  return delete node $x
  ```

  This is perfectly predictable and won't stop prematurely. Or you can replicate an element after itself without risking looping forever:

  ```
  for $x in collection(...)//X
  return insert node $x after $x
  ```

2.3. Mixing Updating and Non-updating Expressions

Updating Expressions are XQuery expressions that encompass the 5 updating primitives.

There are rules about mixing Updating and Non-updating Expressions:

- First of all, let us remember that Updating Expressions do not return any value. They simply add an update request to a list. Eventually the updates in the list are applied at the end of a script execution (or at the end of the **modify** clause in the case of the **transform** expression).
- Updating Expressions are therefore not allowed in places where a meaningful value is expected. For example the condition of a **if**, the right hand-side of a **let :=**, the **in** part of a **for** and so on.

- Mixing Updating and Non-updating Expressions is not allowed in a sequence (the comma operator). Though technically feasible, it would not make much sense to mix expressions that return a value and expressions that don't (remember that the sequence operator returns the concatenation of the sequences returned by its components).

  The `fn:error()` function and the empty sequence `()` are special as they can appear both in Updating and in non-updating expressions.

- In the same way, the branches of a **if** or a **typeswitch** must be consistent: both Updating or both Non-updating. If both branches are Updating then the **if** itself is considered Updating, and conversely.

- If the body of a function is an Updating Expression, then the function must be declared with the **updating** keyword. Example:

  ```
  declare updating function insert-id($elem, $id-value) {
     insert node attribute id { $id-value } into $elem
  }
  ```

  A call to such a function is itself considered an Updating Expression. Logically enough, an updating function returns no value and therefore is not allowed to declare a return type.

2.4. Order and Conflicts

Another consequence of the "Pending Updates" mechanism is that the order in which updates are specified is not important. In the following example you can without any issue delete the attribute `Id` (pointed by `$idattr`), and *after* use `$idattr/..` (the parent ITEM element) for inserting! Or you could insert first and delete after.

```
for $idattr in doc("data.xml")//ITEM/@Id   (: selection :)
return (               (: updates :)
   delete node $idattr,
   insert node <NID>{string($idattr)}</NID> as first into $idattr/..
)
```

But because of that, some conflicting changes can produce unpredictable results. For example two **rename** of the same node are conflicting, because we do not know in which order they would be applied. Other ambiguous operations: two **replace** of the same node, two **replace value** (or contents) of the same node.

The XQUF specifications take care of forbidding such ambiguous updates. An error is generated (during the apply-updates stage) when such a conflict is detected.

A bit ironically, no error is generated for meaningless but non ambiguous conflicts, for example both renaming and deleting the same node (**delete node** has priority over other operations).

3. Conclusion

The XQuery Update Facility is a powerful, convenient and elegant extension of the XQuery language, in spite of a few peculiarities that can be slightly off-putting for programmers.

We are looking forward to its wide adoption as the language of choice for updating XML databases. At the time this tutorial was written, there were already a few implementations: Monet DB (CWI), Qizx (XMLmind), Oracle Berkeley DB XML (Oracle), XQilla.