An XQuery-Based Language for Processing Updates in XML

Gargi M. Sur, Joachim Hammer, and Jérôme Siméon‡

CISE Department, University of Florida, {gsur, jhammer}@cise.ufl.edu
‡Bell Labs, Lucent Technologies, simeon@research.bell-labs.com

Abstract

Extensive usage of XML for information exchange and data processing has led to the development of standard languages for querying and publishing of XML data. However, a key problem in XML data management is the absence of a standard language to update XML. As a result, updates are carried out in a mostly ad-hoc fashion at the application level by writing code to modify tree-based structures representing XML documents in memory. So far very little research has been conducted on language-based updates and their impact on XML processing. In this paper we present UpdateX, our implementation of a declarative update language that can be used to directly update XML data using queries. Our update language is based on XQuery 1.0, W3C’s XML query language, and seamlessly integrates with all of its constructs and capabilities. We describe the user-level syntax of our update language and present a framework for its implementation. The UpdateX prototype, which has been implemented on top of the Galax XQuery 1.0 processor, is fully functional and its source code distribution is publicly available on the Web.

1. Introduction

In the recent years, many applications have been migrated to XML to satisfy flexible data processing needs. As larger amounts of XML data must be processed and maintained efficiently, an expressive language to update XML becomes of primary importance. Update capabilities are necessary not only to modify existing XML documents but also to manage native XML repositories or XML data published from relational sources. However, there is no agreed upon XML update language yet and there is very little experience in building XML update processors. As a result, existing XML storage engines have little support for XML updates, and often rely on ad hoc solutions. Most XML applications use the XML Document Object Model\(^1\) (DOM) programming interface to update XML. The DOM interface contains primitives which operate on a tree-based representation of the XML document or fragment in memory. Hence, programmers are subjected to writing code tailored to every XML document that must be processed. In the absence of a high-level, declarative update language, maintaining that code can also be tedious and error-prone. Finally, the lack of an easily accessible XML update infrastructure makes research about the impact of updates on XML processing difficult to carry out.

In this paper, we describe an XML update language that is tightly integrated with XQuery 1.0 [BCF+], the W3C XML query language, and present an implementation of that language. The language itself is based on an internal W3C working draft [CFL+] that extends the powerful, declarative syntax of XQuery to provide update semantics. Thanks to its tight integration with XQuery, this language is powerful yet intuitive and easy to

\(^1\) http://www.w3c.org/DOM/
learn. Moreover, an implementation can be obtained fairly rapidly by modifying an existing XQuery processor. We have developed a complete implementation of that language as part of the UpdateX project between University of Florida and Bell Laboratories. Our implementation has been carried out on top of the Galax XQuery 1.0 engine. Galax and its source code, including the update implementation, are publicly available from the Galax Web site².

In this paper, we describe the update language itself using examples based on the XML Benchmark Project [AWK+], also known as XMark. The XMark benchmark proposes a set of queries operating on a document containing information about auctions, and has been widely accepted in the XML research community. The XMark queries have been carefully picked up in order to exercise various aspects of XML querying as is meant to evaluate the performances of XML query implementations for different application contexts. It is interesting to note here that the authors of XMark have cited the lack of support for updates as one of the prominent shortcomings of the current W3C XQuery specification. The demand for concurrent XML queries and updates is in fact not specific to the XMark scenario, but prevails in a number of other application domains, such as in Web services (e.g., an on-line travel agent must support both access to flight information and perform changes to reservations), in the context of the Semantic Web (to query and maintain an ontology written in RDF), in XML messaging (e.g., to route existing customer transactions or add annotations to them), or in XML publishing (e.g., to update an XML view over an underlying relational store).

Our work is based on an XML update language originally proposed in [CFL+], which addresses the issues discussed in this paper by providing the required update extensions on top of XQuery itself. Although a number of previous XML updates proposals have been made [TIH+,MR02,PL01], this language is the first that provides XML update support tightly coupled with XQuery. Our UpdateX prototype, which is based on Galax, validates the claim that this language can be easily incorporated into existing XML query engines, and is targeted to provide the desired DML functionalities in the above-mentioned and many other domains.

The rest of the paper is organized as follows. Section 2 gives an overview of the XML update language itself by means of examples. Section 3 presents the low-level XML update operations at the data model level, which are used to implement the language. Section 4 presents the rest of the architecture for UpdateX and details about the compilation process from the declarative XML updates to the low-level operations. Section 5 reviews related work and Section 6 concludes the paper.

2. The XML Update Language

The proposed XML update language borrows design principles from both XQuery and SQL. The main features of the language include statement-based update execution, use of XQuery expressions to compute target nodes and update content, constraint checking, complex updates, and snapshot semantics to enforce consistency of complex updates.

The primary building block in XQuery is an expression, which always evaluates to a value. XQuery uses various types of expressions such as path, sequence, arithmetic, logical, comparison, conditional, and FLWOR expressions. The XML update language,
on the other hand, introduces the notion of statement, which is semantically different from the expression used in XQuery: an update statement modifies an existing value rather than simply returning one.

Updates are classified into simple and complex updates. Simple updates represent the basic data modification operations such as add, remove, or update. Complex updates can be either conditional or iterative, allowing complex operations based on simple updates. In the following subsections we highlight some of the most important features of our update language.

3.1. Simple Updates

Simple updates support either insertion of new XML fragments, deletions of existing XML fragments, or ‘replacement’ of an existing XML fragment by a new one. In each case, XQuery is used to compute the location where the update occurs and the content of the update. The syntax of the simple update statements is as follows:

- **INSERT** UpdateContent InsertLocation TargetNode
  
  where InsertLocation = INTO / AS FIRST INTO / AS LAST INTO / BEFORE / AFTER

- **DELETE** TargetNode

- **REPLACE** <value of> TargetNode WITH UpdateContent

For example, the simple insert statement inserts a copy of the item sequence returned by the ‘UpdateContent’ expression into the location determined by the ‘TargetNode’. The order of the inserted nodes is specified using the ‘InsertLocation’ clause. The language provides several means to indicate the insert location, either as the children of a given node, or before/after a given node as a sibling. For example, a new item can be inserted into the XMark database using following insert statement:

```
insert
  <item id="{id}">
    <location>Brazil</location>
    <quantity>200</quantity>
    <name>XML in a Nutshell</name>
    <payment>Credit Card, Personal check</payment>
    <shipping>Will ship internationally</shipping>
    <incategory category="category1"/>
  </item>
as last into document("xmark.xml")/site/regions/samerica;
```

**Update 1.** Insert Statement.

Note that in this case the item is inserted as the last element. Also note that XQuery element node constructor is used to construct a new item element, and the parameter ‘id’ can be computed using XPath, e.g. $auction/regions/item[last()] + 1, where the $auction variable binds to the document node ‘site’. Similarly the following delete statement can be used to delete the last bid of the second open auction:

```
delete
  $auction/open_auctions/open_auction[@id = "open_auction2"]/bidder[last()];
```

**Update 2.** Delete Statement.
Finally, the following replace statement updates the credit card information of person with reference identifier 10:

\[
\text{replace} \quad \$\text{auction/site/people/person[@id="person10"]}/\text{creditcard} \\
\text{with} \quad <\text{creditcard}>2334 \ 3423 \ 3484 \ 2743</\text{creditcard}>;
\]

\text{Update 3. Replace Statement.}

### 3.2. Complex Updates

Complex updates can be built from simple updates using either conditionals or FLWOR expressions. The syntax for conditional update and FLWUpdate is as follows:

- **Conditional Update**:  
  \[
  \text{UPDATE} \quad \text{IF} \ (\text{ConditionExpr}) \ \text{THEN} \ \text{SimpleUpdate1} \ \text{ELSE} \ \text{SimpleUpdate2}
  \]

- **FLWUpdate**:  
  \[
  \text{UPDATE} \quad (\text{FORClause}|\text{LETClause})^+ \ \text{WHEREClause}? \ \text{SimpleUpdate}^+
  \]

A complex conditional update first computes the value of the ‘ConditionExpr’. If the Effective Boolean Value\(^3\) of ConditionExpr is true, then the SimpleUpdate in the `then` clause is evaluated; otherwise, the SimpleUpdate in the `else` clause is evaluated. The ‘then’ and ‘else’ branches can also contain an empty update statement, which is written as “( )” and used to indicate that no data modification needs to be performed. The following conditional update statement replaces the category name of ‘category4’ if it exists or inserts a new category into the database as follows:

\[
\text{update} \quad \text{if} \ (\$\text{auction/site/categories/category[@id="category4"]}) \\
\text{then} \quad \text{replace} \ \$\text{auction/site/categories/category[@id="category4"]}/\text{name} \\
\text{with} \quad <\text{name}>2003 \text{ Car Sales}</\text{name}> \\
\text{else} \\
\text{insert} \quad <\text{category id="category4"}>
\text{name}>2003 \text{ Car Sales}</\text{name}> \\
<\text{description}>
<\text{parlist}>
<\text{listitem}>
<\text{New} \ <\text{bold}> \text{Toyota Camry} \</\text{bold}> \ 2003</\text{text}>
</\text{listitem}>
</\text{parlist}>
</\text{description}>
</\text{category}>
\text{into} \ \$\text{auction/site/categories};
\]

\text{Update 4. Conditional Update Statement.}

Finally, for more complicated updates, the FLWUpdate statement can be used to apply simple updates through iterations. The `for` (and `let`) clause in the FLWUpdate allows the binding of a variable to the results of a query expression. The FLWUpdate then iterates

---

\(^3\) The effective Boolean value is defined as the result of invoking the `fn:boolean` function on a XQuery sequence.
through this sequence of variable bindings to execute the list of simple updates sequentially. For instance, the following FLWUpdate can be used to insert a count of total number of bids for every open auction into the database:

```xml
update
  for $a in $auction/site/open_auctions/open_auction
    where fn:count($a/bidder) > 1
  insert <bid_count>{ fn:count($a/bidder) }</bid_count> as last into $a;
```

**Update 5.** FLWUpdate Statement.

### 3.3. Joins

The FLWUpdate can also compute joins over various parts of a document or across documents. For example, the following FLWUpdate performs a join between ‘person’ and ‘closed auction’ elements to compute the total purchase amount for each person and inserts it into his profile:

```xml
update
  for $p in $auction/site/people/person
  let $s := fn:sum(for $o in $auction/site/closed_auctions/closed_auction
    where $o/buyer/@person=$p/@id
    return $o/price)
  insert <purchase_history>{ $s }</purchase_history> into $p
```

**Update 6.** FLWUpdate Statement.

### 3.4. Constraints and Semantics

A set of basic semantic constraints must be preserved while executing updates. These constraints aim to preserve the logical structure of the data model instance. The most important constraints are as follows:

- The target node of a simple insert must be a single node. If the insert location evaluates to an empty value, or contains more than one node, then an error must be raised and the insertion is not performed.
- When the `into` clause is used in a simple insert, the target node must evaluate to document or element node; when the `after` or `before` clause is used, the target node must be an element, comment, or processing instruction node.
- During insertion or execution of a replace statement, for each adjacent sequence of one or more atomic values, a new text node is constructed. This new text node contains the result of casting each atomic value to a string, with a single blank character inserted between adjacent values.
- After insertion or deletion, adjacent text nodes must be coalesced into a single text node by concatenating their contents, with no intervening blanks.
- While replacing the value of an element or document node, the update content must be a **content sequence**. A content sequence is any sequence of zero or more element nodes, atomic values, processing instruction, and comment nodes.

In addition to these basic constraints, the data model must satisfy additional constraints in the case the input XML document has a DTD or XML Schema associated with it. Using XML schema one can describe key and referential integrity constraints, and restrictions on the values of attributes and elements. Such value-based constraints must
be handled before the updates are processed. Most of the value-based constraints can be checked explicitly using the conditional update statement. Note that a promising direction would be to hook up the update processor with an incremental constraint checker, such as 

One of the main difficulties in designing and implementing such an XML update language is to ensure its semantic integrity. Most notably, one must carefully deal with cases where consecutive updates in a single FLWUpdate statement impact the same XML nodes. To avoid inconsistent results, the language imposes a so-called “snapshot semantics”. According to snapshot semantics, all the variables in the ‘for’ and ‘let’ clauses of FLWUpdate must be bound with respect to the initial snapshot before the simple updates in the body of the FLWUpdate are executed. The simple update statements are then executed sequentially based on the initial snapshot and evaluated independently of each other. In the future, a complete specification for XML updates using XQuery will be published by the W3C XML Query Group, highlighting all the necessary constraints and semantics.

3. The UpdateX Data Model API

We now describe the underlying data model and its API which is based on the XQuery data model. Again, all examples are drawn from the XMark benchmark application.

3.1 The XQuery Data Model

In XQuery and XPath, every XML document instance is represented using a set of nodes. The XQuery 1.0 and XPath 2.0 data model specification [FMM+] defines seven different kinds of nodes: document, element, attribute, namespace, comment, processing instruction, and text. These nodes are used to capture the abstract, logical structure of a document, which is known as its data model. Every node in the data model is assigned a unique identifier by the query processor. These identifiers are used to maintain the original document order amongst the data model nodes. The query specification also defines various terms to describe content type such as atomic value, item, sequence, etc. Atomic value defines a value corresponding to the XML Schema atomic type (a simple type). An item is either a node or atomic value. A sequence is an ordered collection of zero or more items.

The XMark database, which we have used in our examples, is modeled after an Internet auction site. The main entities are: item, open auction, closed auction, person, and category. An item is an object on sale. Every item is assigned a unique identifier. An open auction has properties like privacy status, bid history, along with references to the bidders and sellers, the current bid and default increase, the type of auction, status of transaction, and a reference to the item being sold. Closed auctions describe completed auctions and include seller and buyer references, sold item reference, date when the transaction was closed, its type, and the annotations after the bidding process. The person elements are characterized by personal details, credit card number, profile of their interests, and a set of open auctions they watch. Categories feature name and descriptions used to classify items while a category-graph links categories into a network.
Figure 1 shows the data model for the complete XMark document. As one can see, the root entity site is represented by a top-level document node while the remaining entities and their properties are represented using nested element and associated attribute nodes.

3.2 Data Model Update Primitives

We require two data interfaces for the UpdateX language. Firstly, basic primitives to create and access the data model elements, and secondly, update primitives to execute basic data-modification operations such as insert, delete, and replace. Specifically, the update API has the following interfaces:

- **Insert(Sequence update-content, Node parent-node, Node sibling-node).** The insert primitive adds new nodes into the data model which are inserted as children of the parent node and can be ordered with respect to a sibling node.
- **Delete(Node target-node).** The delete primitive removes given target node from the data model.
- **ReplaceNode(Sequence update-content, Node target-node, Node parent-node).** This particular replace primitive replaces a given target node with one or more new nodes and updates the parent node pointers.
- **ReplaceValue(Sequence update-content, Node target-node).** This replace primitive replaces the value of given target node with one or more nodes given in the update content.

Other useful update operations are move and rename [TIH+, MR02], but we do not include them in our API as they can be simulated using basic operations. For example, the move operation which moves an existing sub-tree from one location to a new one can be simulated by an insertion of a copy of the sub-tree at the new location followed by a
deletion of the original sub-tree. Similarly, the rename operation is a specific case of the replace, i.e., replacement of qualified name of a data model node. We have implemented the update primitives using tree-based, recursive algorithms. These primitives are available to the query engine in the form of library functions in Galax.

4. Implementation of the UpdateX Language

We have implemented a fully functional prototype to serve as a reference implementation of the UpdateX language. The prototype was implemented on top the Galax query engine [FSB+] developed at Bell Laboratories. Galax is an open source implementation of XQuery 1.0 and closely conforms to the XQuery specification suite. It implements most of the features of XQuery like path and FLWOR expressions, arithmetic and comparison operators, node constructors, document order, etc. It also supports several advanced features namely XML Schema import and validation, static type checking, and type/value based optimization. Galax is written in OCaml and is portable to various platforms (Linux, Solaris, Macintosh, and Windows). OCaml [XL02] is an object-oriented variant of ML with sophisticated features, like typed compilation, automatic memory management, polymorphism and abstraction. It is an ideal language to implement XQuery as well as update extensions, as its algebraic types and higher-order functions help to simplify the symbolic manipulation that is central to query transformation, analysis, and optimization.

The update processing model is based on the XQuery Formal Semantics [DFF+]. The architecture of the UpdateX processor is shown in Figure 2. The inputs to the processor consist of update statements and one (or more) XML document(s) that are subject to modification. The system consists of three main layers: parsing, normalization and execution. We explain the main features in more detail below.

![Figure 2. Architecture of the Update Processor.](image-url)
4.1. Parsing Layer

The parsing layer implements the parsing phase of the XQuery processing model. It takes the inputs, parses them, and builds abstract syntax trees (AST) corresponding to the inputs. The parsing layer consists of two modules: update parser and XML parser. The update parser parses the input update statement to build an update AST. Updates are parsed according to the grammar rules specified in the Appendix. The input XML document is processed by a SAX-based XML parser. The advantage of using a SAX parser is that the AST corresponding to the XML document is never materialized. The SAX parser optimizes memory usage by generating stream of SAX events which are directly consumed by the data model loader to create an instance of the XQuery data model in memory.

4.2. Normalization Layer

The update normalizer implements the normalization phase of the XQuery processing model. It maps the update AST into an internal representation that can be directly executed by the evaluation engine. XQuery normalization judgments are applied to transform the update AST to equivalent XQuery Core expressions. Normalization judgments are transformations or rewriting rules that expand the abbreviated, hidden, or implied syntax of the top-level language and make them explicit.

For instance, the FLWUpdate syntax allows one or more For- (and Let) clauses in the same statement. During normalization, every FLWUpdate statement containing nested For- (and Let) clauses is normalized separately. Also, a For-clause in a FLWUpdate can contain more than one variable, whereas a core expression can bind and iterate over only one variable. In such a case the FLWUpdate is normalized to sequence of nested For-expressions with a single variable binding.

![Normalization process for FLWUpdate](image)

Figure 3. Normalization process for FLWUpdate

In addition to the above judgments, a special tuple construction rule is applied that helps to enforce snapshot semantics. Consider the FLWUpdate described in Update 5. When the FLWUpdate executes, the For-expression is evaluated to a sequence of tuples, which is used to evaluate expressions in the simple update list. This implies that a tuple (structure) must be present in the update AST to hold the tuple sequence values. When
the FLWUpdate is normalized, we insert a “for-tuple” construct into the update AST, which is later materialized during execution phase. The normalization process is shown in Figure 3. The “Build” and “Get_X” clauses represent internal tuple construction and extraction operations respectively.

4.3. Execution Layer

The execution layer implements the dynamic evaluation phases of the processing model. After an input update statement is parsed and normalized, it is executed by the evaluation engine. The update execution consists of three logical phases: pre-update processing, semantic checking, and update evaluation, which are shown in Figure 4 below.

4.3.1. Pre-update Processing

In the pre-update processing phase, all the XQuery expressions in the normalized AST are evaluated. A pre-update list is used in the algorithm, which holds the values of the query expressions based on the initial snapshot. For example, when FLWUpdate is pre-processed, the following steps are performed:

- The variable bindings in the ‘for’ and ‘let’ clauses of the update statement are computed by evaluating the query expressions and stored in a tuple sequence. These bindings actually define the scope of the update.
- The evaluation engine iterates through the list of simple updates and evaluates the sub-expressions in each simple update statement, with respect to the values in the tuple sequence.
- The resulting values are stored in a pre-update list according to the order of update statements in the simple update list.

For instance, the statement in Update 5 is normalized to the following core expression:

```
update
  for-tuple $dot in
    (for $a in $auction/site/open_auctions/open_auction
      return [ a: $a ])
  insert <bid_count>{fn:count($dot#a/bidder)}</bid_count>
  as last into $a;
```

**Normalized Update 5. Core expression.**

When this expression is executed, the ‘for’ expression is evaluated first. The intermediate results are stored in the for-tuple data structure that is inserted in the AST during the normalization phase. Next, the values of the ‘bid-count’ element for each iteration of ‘$a’ variable in the simple insert is computed and stored in a pre-update list.
4.3.2. Semantic Checking

In this step, semantic checks are applied to ascertain whether the target nodes and update content values are valid and whether basic constraints are satisfied. The evaluation engine performs necessary checks on the values in the pre-update list. During this validation step, if any constraint is not met, the execution is aborted and an update error is raised.

4.3.3. Update Evaluation

In the final step, updates are executed by invoking the data model primitives. Actual modifications to the underlying data model are performed by these primitives. Based on the type of update, the evaluation engine invokes the required primitives and passes the values stored in the validated pre-update list as arguments. The invoked primitive modifies the data model and returns control to the evaluation engine for further processing. After the update operations are completed, the update processor may perform additional book-keeping tasks such as updating the node identities and indexes if required.

The data model primitives are implemented as atomic functions. Here we discuss the implementation of the insert primitive. Its signature is given below:

\[ \text{Insert}(\text{Sequence update-content, Node parent-node, Node sibling-node}) \]

The \textit{update-content} consists of a sequence of items, which are inserted as children of the parent node. The insert operation is deterministic. It preserves the order of children nodes in the parent node. A pseudo-code version of the insert algorithm is as follows:

- Create a deep copy of the sequence of items to be inserted, viz., copy of update-content.
- Check the node kind of the parent; the parent node should be a document or an element node; otherwise raise an error.
- Compute the child axis of the parent node to retrieve its children.
• If a sibling node is specified, determine its position in the children node sequence.

• Compute the insert location as follows: if the sibling node parameter is null, then the insert location is after the last child of the parent node; otherwise the insert location is before the current position of the sibling node in document order.

• Insert the items in the update-content recursively at the insert location.

• Update the parent pointers of the newly inserted items.

It is important to note that the syntax for simple insert allows five InsertLocation clauses: insertion into, as last, as first, before, or after the target node. Although this may indicate the need for multiple insert primitives, we have implemented a single primitive that simulates all the required variations. The evaluation engine computes the values of the parent node and target node based on the InsertLocation clause, and then determines if the sibling node value is to be computed. These values are passed to the insert primitive when it is invoked during update execution.

The distinction made between low-level data model operations and a higher-level update evaluation provides a degree of independence with respect to the physical implementation of the update processor. The data model primitives can be implemented using in-memory, tree-based algorithms (this prototype), on top of indexed XML structures in a native XML repository, or by converting them into SQL DML queries which can be issued to a relational database system. For example, the delete primitive can be implemented as a pre-delete SQL trigger that can be fired when parent-child nodes of an XML document are stored in the form of tuples in a relational database [SGT+] and a parent tuple is deleted. The body of the trigger contains SQL code to delete the relevant tuples from the child relations. Moreover, since data model operations execute atomically, it becomes possible to define transactions over the XML data model by enhancements to the update processing model.

5. Related Work

Until recently, there has been relatively little work on XML updates. One of the first languages for XML updates was proposed by Tatarinov et al in [TIH+]. They proposed a fairly simple update language not directly integrated with XQuery, and whose expressiveness is limited to the basic updates supported by UpdateX at the data model level. In [LM00], Laux and Martin proposed an ad hoc language using XPath 1.0 to support XML updates in the XML:DB native XML database. Many of the updates presented in Section 3 cannot be expressed with those languages. Our work relies on a more recent proposal by the XML query working group [CFL+], which itself was inspired by proprietary proposals in [MR02] and [PL01]. An important limitation of the language is the lack of compositional interplay between the update expressions and the rest of the language. Even though powerful, the update primitives can only occur as top-level statements, in a way similar to SQL. This has to be distinguished from ‘updates’, or in-place modifications as present in most modern functional programming languages, such as ML [LW91]. Trying to support this kind of feature raises difficult questions in terms semantics and optimizations. We believe this is an important requirement for complex
XML applications, notably for Web services, and we hope to work on XML updates as first-class expressions in the language in the future.

6. Conclusion

We have described a new approach to XML update processing that uses a declarative, functional language based on XQuery 1.0. The language is easy to learn and powerful. It consists of various constructs, such as simple insert, delete, replace, conditional, and FLWUpdate statements, which can be used to efficiently modify XML data model instances. These constructs have been explained with examples from the XMark auction database. We also describe a complete framework for processing and implementation.

One of the most important contributions of the processing framework is that it eliminates tedious and ad hoc programming techniques (e.g., updating XML using DOM primitives) that are currently used to update XML documents. In addition, application developers benefit from use of an integrated query and update language to perform many different types of retrievals and modifications on XML documents (e.g. embedded XQuery queries or statements) in their software applications. Moreover, in this research, we have developed a vendor-independent, standard API to update native and relational XML repositories, which is beneficial to the database community at large. Lastly, for researchers, we provide a flexible architecture to study the impact of updates on XML processing and scope for refinement or improvement of update processing algorithms.

References


Andreas Laux and Lars Matin. XUpdate working draft. October 2000.
http://www.xmldb.org/xupdate


**Appendix** UpdateX Grammar in EBNF

1. Update ::= SimpleUpdate | ComplexUpdate
2. SimpleUpdate ::= Insert | Delete | Replace | EmptyUpdate
3. ComplexUpdate ::= FLWUpdate | ConditionalUpdate
4. FLWUpdate ::= "update" (ForClause | LetClause)+ WhereClause? SimpleUpdate+
5. ConditionalUpdate ::= "update" "if" "(" ConditionExpr ")" "then" SimpleUpdate "else" SimpleUpdate
6. ConditionExpr ::= XQueryExpr
7. Insert ::= "insert" UpdateContent InsertLocation
8. InsertLocation ::= (("as" "last""> | "as" "first"">)? "into" TargetNode_ ("after" TargetNode) ("before" TargetNode)
9. UpdateContent ::= XQueryExpr
10. TargetNode ::= XQueryExpr
11. Replace ::= "replace" "value" "of">? TargetNode "with" UpdateContent
12. EmptyUpdate ::= "(" ")"
13. Delete ::= "delete" TargetNodes
14. TargetNodes ::= XQueryExpr

Note: XQueryExpr refers to the Expr [40] grammar production in the XQuery specification.