ABSTRACT

Updates for XML are a quite new research area. Many applications need not only to transform or search XML documents but also to alter them. Nowadays, there are some suggestions for update languages, and some XML database systems support updates operations via special APIs. In all update languages the content and the structure of documents can be changed. That means, documents may invalidate the XML schema after or during accomplishing an update.

In this article, we suggest different architectures for processing such updates. Some of them reject updates that violate the schema, others entail a schema evolution that generalizes the schema. We point out necessary evolution steps and show implications of the schema evolution. With the discussion of architectures for executing updates we will focus on problems that can occur by updating XML documents. These questions have to be considered in all implementations that update XML documents.

Keywords

1. INTRODUCTION

In databases, there is a strict differentiation between changing values of tuples and changes of structure. Changes of values are called update operations. They are expressed with UPDATE clauses in SQL. Every update operation that does not comply to the integrity constraints of the associated schema is rejected. Updates of the database schema (also called schema evolutions) are part of the SQL standard, and also are expressed for instance with an ALTER TABLE statement. Whereas updates of tuples occur very often, a schema evolution should appear very seldom.

For XML documents, we cannot clearly distinguish between updates of values and altering the schemas since updates of XML documents comprise updates of values and changes of structure. Both types of update operations always have to ensure the well-formedness, otherwise the update operation, e.g., inserting an attribute which is already present in an element, has to be rejected. In fact, an XML update language should not allow for formulating updates that violate well-formedness.

In the presence of schema information for a given XML document, some update operations cannot be executed because they would violate the validity of the XML document. We use the following schema fragment (the popular and frequently used XML document book) as a running example:

```
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="fm">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="author" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="year" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="chapter" minOccurs="0" maxOccurs="unbounded">
        ...
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The complete XML Schema, the equivalent DTD (for most people easier to read), and a sample document instance can be found in Appendix A–C.

We want to rename the element author with value C.J. van Rijsbergen by editor because some books are collections of articles. Those books have an editor instead of an author. The operation not conforms to the given schema part and therefore it should be rejected. But, sometimes it is our intention really to change the XML document and evolve the schema. This is reasonable if the application has no exact knowledge about the structure, or the structure of the objects frequently changes. One characteristic of semistructured data is that the schema is not as static as for instance in databases. The structure can be irregular or rapidly evolving [1]. Such applications want to update XML documents and evolve the XML schema. As a result, we get this updated content definition (it contains changes in line 5–8):

```
<xs:element name="fm">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="author" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="editor" type="xs:string" minOccurs="0"/>
      <xs:element name="publisher" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```
There is more than one schema implied by this update operation. An equivalent schema using an alternative would be:

```
<xs:element name="fm">
    <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="2">
            <xs:element name="editor" type="xs:string"/>
            <xs:element name="author" type="xs:string"/>
        </xs:choice>
    </xs:complexType>
</xs:element>
```

In this example, an additional element `choice` is added (line 4-7). Throughout this article, we stay with schema update steps which change quantifiers in the first instance. There are further techniques to normalize or minimize such a schema with respect to certain criteria (see Section 5).

In this article, we demonstrate most of the problems arising from updates of XML documents and the implied schema changes. To overcome these problems, we recommend using evolution strategies for XML documents. Evolution in this context means the schema of an XML document collection is changed and the documents have to be updated and aligned to the modified schema.

In this article, our contribution is to:

- illustrate each basic update operation and show their effects on the XML documents and on the schema,
- suggest several architectures for processing updates,
- show that the schema evolution steps initiated by updates do only extend the information capacity of a schema, and
- enumerate open problems related with this quite new research field.

The article is organized as follows: In the next section, we present basic update operations. Section 3 shows which consequences each update operation has on the XML documents. We enumerate which update operations cause which kind of schema changes. In Section 4, we outline different architectures dealing with those update operations. Thereby, some architectures reject update operations that violate schema constraints. Other architectures do adapt the schema, too. We discuss the relationships between schema evolution and document updates. In Section 5, we focus on some aspects not discussed in detail and open issues in that field. In Section 6, we enumerate the related work in the field of updating XML and schema evolution. In the last section, we sum up and give some outlook.

## 2. UPDATING XML

Up to now, there exists no "official" update language for XML documents, but there are some suggestions for describing updates [18, 29, 20, 36]. All these update languages contain similar operations. Tatarino et al [29] developed an update language as an extension to the XML query language (XQuery [33]). XQuery is the actual working draft of the W3C for an XML query language. In this article, we use a syntax similar to Tatarino's for simplicity reasons.

In that proposal, update operations are always evaluated in the context of node bindings. The binding of a variable to a node is usually done in the `FOR` or `LET` clause in XQuery. Subsequent expressions, e.g., updates, are always related to a so called `context node`, which is defined by the current node binding. These are the basic operations performed on the context node:

- **DELETE child** removes the child from the sub-node list of the context node.
- **INSERT content [ (BEFORE | AFTER) ref ]** allows adding new content beneath the context node. The `BEFORE/AFTER` operations are used to put the new content at a specific position, i.e., before or after the node `ref` in an ordered model, otherwise `INSERT` appends after the last child node.
- **RENAME child TO name** assigns a new name to the child node.
- **REPLACE child WITH content** substitutes the child node with the specified content. This operation can also be accomplish by a back-to-back `INSERT BEFORE` and `DELETE` operation.

To illustrate the basic structure of such XQuery update expressions, we give an example that updates the `year` content for a certain book.

```
FOR $f IN document("books.xml")/book/fm,
    $y IN $f/year
WHERE $f/title = "XML and Databases"
UPDATE $f {
    REPLACE $y WITH ELEMENT year { "2003" } 
}
```

While the above example only modifies content, other updates may also change the structure. The following example demonstrates that by renaming the last author element to `editor`:

```
FOR $f IN document("books.xml")/book/fm,
    $a IN $f/author[last()]
WHERE $f/title = "XML and Databases"
UPDATE $f {
    RENAME $a TO "editor"
}
```

At first glance, this update operation doesn't look much more complicated than the one before. But we will see in the next section that updates that change the structure raise several questions.

## 3. EFFECTS ON THE XML SCHEMA

In this section, we want to enumerate which effect each of the basic update operations has on XML documents. If an external schema is associated with the XML documents we also have to check if the updated XML documents are still valid. If the update operation violates the schema constraints then we have several opportunities as there are:

- rejecting the update operation or
- evolving the XML schema so that the schema constraints are relaxed. The update operation can be repeated afterwards. The schema evolution can be processed
We will discuss these alternatives in detail in section 4.

In this section, we examine which are the consequences of updates for an schema. We describe which changes of the schema are required. Thereby, we estimate the worst case, the most extensive alteration that can be necessary.

We focus on operations which only change the structure of XML documents since updates of attribute values and textual element content don’t influence the validity (beside type mismatch in XML documents). We use XML Schema for explanation and comment only some special cases for DTDs.

### 3.1 DELETE

This operation can be used for deleting elements or attributes of XML documents.

If we delete an attribute a that is defined as required or fixed then we have to change the attribute declaration so that the attribute is set to optional now. If we delete an element e then we have a similar situation. So, it can be necessary to decrease the facet minOccurs for instance: old value 1, new value 0. In DTDs the same update operation drags changes of quantifiers of the element declaration as follows: (e → e?, e+ → e*).

For example, we want to delete the element publisher by the following update operation:

```xml
LET $f := document("book.xml")/book/fm,
$p := $f/publisher
UPDATE $f {
DELETE $p
}
```

As a result, we get the updated XML document. The schema also has to be changed as shown below. The element declaration of publisher has now an additional facet minOccurs="0" in line 8. The schema evolution can be carried out automatically or manually.

```xml
1 <xs:element name="fm"/>
2 <xs:complexType>
3  <xs:sequence>
4   <xs:element name="title" type="xs:string"/>
5   <xs:element name="author" type="xs:string" minOccurs="0"/>
6   <xs:maxOccurs="unbounded"/>
7   <xs:element name="publisher" type="xs:string" minOccurs="0"/>
8   <xs:element name="year" type="xs:string"/>
9 </xs:sequence>
10 </xs:complexType>
11 </xs:element>
```

The deletion of an ID-Attribute can result in dangling references. To solve this problem, one of the following three alternative strategies can be applied:

1. The deletion is rejected if referencing IDREF/IDREFS attributes still exist.
2. The ID is deleted because there could be another update operation that insert an ID containing the same ID value.
3. Retrace all incoming reference edges and do a cascading delete of all associated IDREF/IDREFS.

Each of these approaches has advantages and disadvantages but one of it has to be chosen.

The table in figure 1 summarizes schema changes caused by a delete operation:

<table>
<thead>
<tr>
<th>Schema</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>e - minOccurs=&quot;x&quot;</td>
<td>e - minOccurs=&quot;x-1&quot;</td>
</tr>
<tr>
<td>Attribute</td>
<td>a - required</td>
<td>a - optional</td>
</tr>
</tbody>
</table>

#### Figure 1: Effects of delete operations on an XML schema

In a DTD, it can be necessary to change the quantifiers (e → e?, e+ → e*) and an attribute declaration from required to fixed or implied.

### 3.2 INSERT [BEFORE | AFTER]

Insert operations add elements or attributes to XML documents, or even whole sub-trees. Since all operations have the same effect on a corresponding schema we discuss it together.

If we insert an attribute a which is not defined in the schema so far, we insert the attribute declaration with the facet use="optional".

If an additional reference is inserted into an IDREF attribute the schema declaration has to be changed to IDREFS.

While inserting an element e it must be added as an optional element with minOccurs="0" to the schema if not already present. It is also possible that we have to extend an element’s declaration in a way that more than one element can occur. In that case, we have to increase the value of the facet maxOccurs.

If we insert a new element editor before the publisher then we have to adapt the schema with the following update operation:

```xml
LET $f := document("book.xml")/book/fm,
$p := $f/publisher
UPDATE $f {
INSERT ELEMENT editor ("Norbert Fuhr") BEFORE $p
}
```

As a result, we get an XML schema containing a new element declaration for editor in line 7-8.

```xml
1 <xs:element name="fm"/>
2 <xs:complexType>
3  <xs:sequence>
4   <xs:element name="title" type="xs:string"/>
5   <xs:element name="author" type="xs:string"/>
6   <xs:maxOccurs="unbounded"/>
7   <xs:element name="publisher" type="xs:string" minOccurs="0"/>
8   <xs:element name="year" type="xs:string"/>
9   <xs:element name="editor" type="xs:string"/>
10 </xs:sequence>
11 </xs:complexType>
12 </xs:element>
```

Beside insert operations with simple content, we can insert complex sub-trees. Nevertheless, inserting whole sub-trees can be viewed as stepwise executing simple inserts in a top-down manner and changing the schema accordingly.

The schema evolution cause by the insert operations is summarized in the table in figure 1: the schema.

The definition of an attribute of type enumeration has to be extended accordingly.

In a DTD, we can add additional optional elements, change quantifiers (e → e?, e+ → e*) or add attributes which are defined as #IMPLIED.
### 3.3 RENAME and REPLACE

Because **RENAME** and **REPLACE** can be considered as combined **DELETE** and **INSERT** operations they have to be treated in the same manner.

These operations can (in worst case) be seen as a combination of **DELETE** \(a_1\) and **INSERT** \(a_2\) for an attribute or **INSERT** \(e_2\) **BEFORE** \(e_1\), **DELETE** \(e_1\) for an element. If so, both steps must be performed as one atomic operation in the sense of database transactions and regarding the schema change. The table in figure 3 lists the results of these operations.

#### Extension of Data Types

If we use XML Schemas then there are further problems that occur during updating documents. Updates can violate the defined data types of elements and attributes. The evolution of data types is difficult to treat. Problems that occur are strongly related to reverse engineering tasks. Updates cause an extension of the data types. In [27], a method for that is developed that originally was applied for a schema extraction approach that derives an XML schema from a set of given XML documents. The same algorithm is applicable for updates, too.

#### 3.4 Conclusion

All update operations that change the XML document structure can also require an evolution of the explicit schema independently, if a user or the system process these schema evolution. If we want to execute such schema updates then we can summarize that no other XML documents that are associated with the schema have to be modified. As a result, we got the following *evolution steps* of a schema:

- adding optional attributes or elements,
- changing attribute declaration from *required/fixed* to *optional*.
- decreasing *minOccurs* and increasing *maxOccurs* values.

All evolution steps caused by update operations extend the information capacity [11] of the schema and allow more alternatives in the XML documents. The result is a more generalized schema.

---

**Figure 2: Effects of insert operations on an XML schema**

<table>
<thead>
<tr>
<th>Schema</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>e - minOccurs=&quot;0&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e - maxOccurs=&quot;x&quot;</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>a - IDREF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a - NMTOKENS</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3: Effects of rename/replace operations on an XML schema**

<table>
<thead>
<tr>
<th>Schema</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>e1 - minOccurs=&quot;x&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e2 - maxOccurs=&quot;y&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a1 - required</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>a2 - IDREF</td>
<td></td>
</tr>
</tbody>
</table>

---

4. UPDATE ARCHITECTURES

The articles suggesting update languages concentrate on the syntax. Implementing such updates the question arises how to handle updates that change the structure of XML documents so that they are don’t conform to a given schema anymore. There exist some methods for testing the conformity of XML documents efficiently.

3. Based on the incremental validation of schemas, we can find out which updates violate schema constraints. We can determine that there are different strategies for handling update operations. We want to systematize these approaches in the following. For that, we will identify four possible architectures (IGNORE, REJECT, REDO, EVOLVE) for handling such update operations. We discuss the advantages and disadvantages of each method.

#### 4.1 IGNORE — Accepting all update operations

The first architecture executes all updates and doesn’t check the validity after updates (see Figure 4). This approach is used for example in eXcelon [5].

**Figure 4: Architecture IGNORE: Accepting all update operations**

If we accept all update operations without checking validity then the disadvantage is obvious: It is possible that XML documents are invalid and therefore cannot be used in other applications that rely on validity. The advantage of this architecture is that all specified update operations can be executed. No validity tests are necessary during updating XML documents.

#### 4.2 REJECT — Refuse updates that invalidate the schema

The second alternative rejects update operations if the XML documents would not be valid after the update. Figure 5 demonstrates this approach which is used for example in Tamino [35].

At least, this approach demonstrates the need for integrating a transaction concept for executing sequences of update operations as an atomic operation. Validity should be checked only at transaction boundaries else it is impossible to make changes that only temporarily violate the given schema, e.g. the implementation of **REPLACE** as a sequence of **INSERT** and **DELETE**. This scenario has on the one hand the disadvantage that some updates cannot be processed because they violate the schema. On the other hand it is an advantage that all updated XML documents are valid.

---

3. Methods for testing conformity are enumerated in section 6.
4.3 REDO — Schema evolution by hand

Some applications intend to alter the schema by issuing updates on documents. That’s why, we suggest a third method for executing updates. This method requires that the user extends the schema manually. Figure 6 demonstrates this method. First a user tries to update a document. The update operation is rejected because it violates the schema. Then, the user has to adapt the schema definition as demonstrated in section 3. Further, all associated XML documents are adapted so, that the modified documents are aligned to the new schema definition. Next, he can redo the update operation. If the evolution process was correct then the update operation is possible now.

This method requires that the user evaluates the schema by hand. An advantage is that all updates can be processed. The user is responsible for changing the schema. For most users this method seems to be the best alternative because the schema design process is controlled by the designer.

A disadvantage is that this process is very time-consuming and error-prone. Two kind of errors can occur:

- the evolution step defined by a user are not sufficient or wrong so that the original update operation cannot be executed after the schema evolution.
- The evolution steps are not only a generalization. They can change a schema so that other XML documents associated to the schema have to be adapted, too.

Although this architecture is for many application the best choice, we also see the disadvantages mentioned before. That’s why, we want to suggest a fourth approach for executing updates and altering the schema so that the XML documents are valid again.

4.4 EVOLVE — Updates forcing schema evolution

The fourth method allows for executing all updates. If the updates change the structure and violate the schema constraints then the schema is adapted, too. Figure 7 demonstrates this case.

The question arises if the evolution of the schema requires further update operations on the other XML documents referencing the same schema. As demonstrated in Section 3, these documents need no modification since they have been already valid to a specialization of the new (generalized) schema. If we execute the updates automatically then we can ensure that only a subset of all possible evolution steps is applied, namely only those evolution steps that generalize a schema.

The advantage of this method is that all updates can be executed and all updated XML documents are valid. It is not necessary that the schema evolution is defined by the user. Update operations specify the schema evolution. Thereby, only some schema evolution operations are applied. They ensure that all other XML documents associated to the same schema are still valid after the schema evolution.

The disadvantage of this method is that it is not obvious for a user that a simple update operation causes far-reaching changes of the schema. Therefore, we suggest four update modes in the context of update operations.

1. REJECT: This mode will discard changes that invalidate the schema.
2. EVOLVE: Update operations are allowed to cause a schema evolution.
3. USER-DEFINED: The user decides whether to REJECT or to EVOLVE by specifying a user-defined function.
4. SEMI-AUTOMATIC: The consequences for the schema are discussed with the user. If he confirms the operations then the schema is evolved and the update operation can be performed. Otherwise the schema is not changed and the update operation is rejected.

A combination of the REJECT and EVOLVE mode is reasonable if some update operations shall not influence the schema but other shall extend it. That’s why in USER-DEFINED mode it is possible to do a REJECT or go into EVOLVE mode depending on conditions checked by the user. We suggest that the condition checking is implemented by XQuery expressions, e.g. FLWR-expressions, or conditional expressions embedded in a user-defined function.

The USER-DEFINED mode can be used for schemas that have static portions that cannot be changed and additionally special extensions that can underlie changes. Such scenarios often occur if the core of a defined schema is a standard and individual extensions are allowed. The USER-DEFINED update mode can protect this condition.

It can be argued that one really needs an IGNORE mode as found in some implementations of XML update languages. It would result in a mismatch of the schema implied by the documents and the schema definition given. At least the schema has to be derived from the document collection at certain points. Actually, when applications require schema validity for document processing then the update mode should be set by the user, e.g. as a special command in the prolog of an XQuery update statement sequence.

Furthermore, a SEMI-AUTOMATIC approach is possible. This method generates suggestions for schema evolution steps. Then, these evolution steps are discussed with a user before the schema is changed.

![Figure 5: Architecture REJECT: Rejecting updates that invalidate the schema.](image)

![Figure 7: EVOLVE: Updates and automatic schema evolution](image)
5. FURTHER INVESTIGATIONS

Some aspects left out in the previous sections. There are solutions for some of them while others are still open issues.

Updates on stored and indexed XML documents

If we want to process an update on XML documents, we have to consider which storage method is used on the physical layer. All updates on the logical layer have to be mapped to updates on the physical storage structure. Some storage methods, for instance generic storage methods [37] enable schema evolution. In other cases, the physical storage schema may prohibit schema changes, e.g. if a mapping of XML document type definitions to relational or object-relational schemas is used for storing XML documents as described in [26] and [13]. In these cases, an XML schema evolution would also require a database evolution of the underlying database. The effort of the schema evolution is very high that's why we can assume that for some storage methods the architectures redo (see 4.3) and evolve (see 4.4) are not suited.

Further restrictions on update operations appear if XML views are established on database structures. Even if the update does not invalidate the XML schema problems may arise that are similar to updates on SQL views.

Transactions

Transactions provide a context for executing sequences of update and query operations and have ACID properties (atomicity, consistency, isolation, and durability). A "real" update language must provide certain constructs for defining transaction boundaries and rolling back work.

Consistency (e.g. validity) may be violated during the transaction but should be preserved at the end of a transaction. It seems to be obvious that evaluating schema changes at the end of a transaction can reduce processing costs since some steps may overlap or annihilate each other. Even the consequences from single step update operations have to be reconsidered for sequences of update operations.

Schema generalization and normalization

There are some constraints and properties of XML schemas to be fulfilled in general. Especially if the content model of an element is altered, the resulting content model must conform to an one-unambiguous regular languages (see Brüggemann-Klein and Wood [4]). For example the model \( (a, b^+, b) \) is ambiguous, but it can be transformed into an equivalent model \( (a, b, b^+) \) which is unambiguous. So, several normalization steps and schema rewriting techniques are needed.

The following XML schema fragment and update illustrates the problem (one-unambiguity) with respect to our running example.

```xml
<xs:element name="fm">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="author" type="xs:string" minOccurs="0"/>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="author" type="xs:string"/>
        <xs:element name="editor" type="xs:string"/>
      </xs:choice>
      <xs:element name="publisher" type="xs:string"/>
      <xs:element name="year" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Deleting the first author of a book and applying our schema changing rules would result in following new schema fragment (changes in line 6):

```xml
<xs:element name="fm">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="author" type="xs:string"/>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="author" type="xs:string"/>
        <xs:element name="editor" type="xs:string"/>
      </xs:choice>
      <xs:element name="publisher" type="xs:string"/>
      <xs:element name="year" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Since the result is one-ambiguous the update has to be rejected or we have to find different rules which honour the required schema property (one-unambiguity). Our current work is to find more general rules with respect to one-unambiguity.
A similar natured problem arises if we observe the life-cycle of an XML schema. As a consequence of permanent update operations, the schema is growing and its complexity increases. So, some normalization steps should be applied and result in a generalization of the XML schema. All XML documents referencing the schema have to be still valid for the generalized schema.

Normalizing, minimizing, and optimizing schemas should be based on equivalence rules for schema transformations. Certain criteria like minimal description length (MDL) [24] or one-ambiguity [4] should be applied during the transformation process. These criteria are partially studied in recent work on schema extraction from schema-less XML document collections [2, 7, 24], in [27] additionally the determining of the XML schema data types is considered.

**Update language for schema**

For this approach (and lots of other task), a language for updating a schema is necessary. There exist a published proposal in [14] and [28], but there is till no W3C recommendation for a schema evolution language. In our group, we also developed an approach for DTD evolution ([6],[38]). Currently, we are developing a complete XML schema evolution approach ([30]), starting with a language for evolving the schema (an XML schema). For that, an XML update language can be applied because an XML schema has XML syntax. After it, XML documents associated to the schema are adapted, too. For adaptation of the XML schema and the XML documents the update facilities of a DOM processor are used because there exist many available tools.

**6. RELATED WORK**

**Update languages.** Updates in database systems are well studied. In the field of XML there are only a few publications. In most XML applications updates are often performed using the DOM-API [31] or implemented as transformations using XSLT [32]. Both have drawbacks. While DOM uses a navigation approach XSLT is a more descriptive languages but can only transform documents into new ones which can be used to replace the (whole) old document.

XUpdate [36] and SixXML [20] are two similar approaches. These languages are XSLT-like and use an XML syntax. MMDOQL [17] is a logic-based approach, which builds upon the path-predicate-calculus. XML Lore! [3] uses the data manipulation language of OQL to evaluate update operations on semi-structured data.

The forthcoming XQuery 1.0 [33] will not contain any manipulation facilities. But there are some language extensions suggested by Tatarino et al [29]. W3C XML Update group members [23], and there is a similar language called SUNFLWR suggested by H"ansel [9].

**Updates in XML systems.** Today, some systems managing XML data offer update facilities. eXcelon [5] uses XUL (XML Update Language) as a manipulation language. XUL is an XSLT extension. XUpdate [36] is a very similar language supported in several XML servers. Microsoft SQL Server uses a middle-ware approach. Updates on XML views can be performed by so-called Updaterns [25]. Software AG’s Tamino offers a manipulation language very similar to the proposed XQuery update extensions [35].

**XQuery update implementations.** Lethi [16] discusses and implements XQuery update extension on top of a persistent DOM storage while H"ansel [9] discusses an implementation using a relational mapping of the XML tree structure by Shimura et al [37]. A similar approach for XML documents stored in Oracle databases was developed by Below in [3]. That approach translates XML Update operations into SQL statements.

Kane, Su, and Runnesteiner suggest a method that only supports updates that don’t violated the schema [12]. Update operations (with the syntax suggested from Tatarino et al [29]) are extended with XQuery fragments that check whether the operation violates schema constraints. The approach only executes updates that are safe. So, this method fulfills our architecture REJECT because all update operations violating the schema are rejected.

**Schema validation.** There are several publications dealing with incremental schema validation after updates [19, 22] and schema inference in general [21]. On one side, these papers tackle the same problems – managing changes — from the more restrictive validation point of view. On the other hand side, schema inference and validation is a prerequisite for the update and schema evolution management.

**Schema evolution.** As stated before, updates can also cause a schema evolution. We only know a few publications on XML schema evolution. In [14] and [28] an XML Evolution Management is suggested. The articles describe a language for schema evolution and open on a DTD and XML documents.

The DOM Level 3 [34] allows for validating against an XML schema during tree manipulations.

There is some work by our own group on DTD and XML schema evolution. In [38] all DTD changes are enumerated and discussed. Each operation is classified as increasing, decreasing, changing, or keeping information capacity. The implementation of the evolution steps for DTDs is based on DOM update methods. A similar method starting from XML schema is currently under development [30].

Another approach [6], [15] suggests a language for schema evolution based on attributed grammars and has implemented it in Prolog.

**7. SUMMARY AND FUTURE WORK**

Updates on XML documents can change the structure. That’s why, processing updates for XML documents is quite complicated because update operations can violate schema constraints. If such updates are executed then the XML documents are not conform with the schema given.

While in databases all updates violating schema constraints are rejected, we stated that the schema of XML documents has other characteristics – it is not as static and often underlies changes. That is the reason why we suggest several alternatives to handle such update operation. Two of these change the schema, too. That means, an update operation can cause a schema evolution process.

The difference between schema evolution and XML update is that schema evolution first changes the schema. The schema evolution operation can extend or reduce the information capacity of a schema. Then, all XML documents assigned to a schema have to be adapted, too. Updates first modify an XML document or a set of XML documents. Sometimes, the associated schema is evolved, too. The schema evolution contains only a few simple operations and can only extend the information capacity of a schema.

Summarizing, the execution of update operations is very com-
plicated. In this article, we enumerated some of the associated problems and showed some solutions and architectures for updating XML documents.

The problems described in this article will occur in every implementation of updates. One of the strategies demonstrated in this paper has to be used. As shown in Section 5, managing XML updates rises plenty of challenges not only for researchers but system implementers and will still require a lot of research in the future.

8. REFERENCES


APPENDIX

A. BOOK DTD

```xml
<ENTITY % fulltext "#PCDATA|section|par|ref">
<ELEMENT book (fm, chapter*)>
<ELEMENT fm (title, author+, publisher, year)>
<ELEMENT title (#PCDATA)>
<ELEMENT author (#PCDATA)>
<ELEMENT publisher (#PCDATA)>
<ELEMENT year (#PCDATA)>
<ELEMENT chapter (%fulltext;)*>
<ATTLIST chapter title CDATA #REQUIRED
label ID #IMPLIED>
<ATTLIST section (%fulltext;)*>
<ATTLIST section level CDATA #REQUIRED
label ID #IMPLIED>
<ATTLIST chapter title CDATA #REQUIRED
label ID #IMPLIED>
<ELEMENT par (#PCDATA|ref)*>
```

B. XML SCHEMA FOR BOOK

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="fm">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string" maxOccurs="unbounded"/>
              <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
              <xs:element name="year" type="xs:string" maxOccurs="unbounded"/>
              <xs:element ref="section"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="chapter" type="xs:IDREFS"/>
        <xs:element name="section" type="xs:IDREFS"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

C. XML DOCUMENT

```xml
<book>
  <fm>
    <title>The Geometry of Information Retrieval</title>
    <author>Keith van Rijsbergen</author>
    <publisher>Cambridge University Press</publisher>
    <year>2004</year>
  </fm>
  <chapter title="Preface" label="ir_preface">
    <section level="1" title="Scene">
      <par>This book begins and ends in information retrieval, but ...
      </par>
    </section>
  </chapter>
  <chapter title="Prologue" label="ir_prologue">
    <par>Where did that come from? Strictly Ballroom, film, directed ...
    </par>
  </chapter>
  <chapter title="Conditional logic in IR" label="ir_logic">
    <ref label="ir_logic"/>
    <Paragraph>
      All this is spelt out in some detail in Chapter ...
    </Paragraph>
  </chapter>
</book>
```