Preserving XML Queries during Schema Evolution

Mirella M. Moro
University of California Riverside
mirella@cs.ucr.edu

Susan Malaika
IBM Silicon Valley Lab
malaika@us.ibm.com

Lipyeow Lim
IBM T J Watson Research Center
liplim@us.ibm.com

ABSTRACT

In XML databases, new schema versions may be released as frequently as once every two weeks. This poster describes a taxonomy of changes for XML schema evolution. It examines the impact of those changes on schema validation and query evaluation. Based on that study, it proposes guidelines for XML schema evolution and for writing queries in such a way that they continue to operate as expected across evolving schemas.

Categories and Subject Descriptors: H.2.1 [Logical Design]: Schema. H.2.3 [Languages]: Query Languages.

General Terms: Design, Languages.

Keywords: XML Query, XML Schema Evolution.

1. INTRODUCTION

As XML gains widespread use as the data exchange standard, the ability to persist, validate and query XML documents becomes increasingly important. Most commercial database systems support XML persistence in some form. For example, IBM's DB2 pureXML [3] provides support for storing XML documents natively in XML typed columns, validating documents against schemas, and querying documents. Documents can be in the same column even if they conform to different schemas.

With the proliferation of web services and mash-ups, there is also a need for web application developers to query and transform XML messages. Such messages may come directly from a web service or indirectly from a database in which they are persisted. Moreover, the evolving context of the web imposes challenges on accommodating new functionalities (e.g. an expansion of a corporation's business) and new data types (e.g. RSS feed messages) in the database. New schema versions may be released as frequently as twice a month. Managing applications that operate on XML documents from different schema versions can be confusing. However, applications and their users still need to interact with XML data regardless of their schema evolution.

For all these reasons, *schema evolution* is a very important topic that has been researched in the context of relational, object-oriented, and XML databases [4, 2]. Specifically for XML databases, schema evolution still is one of the killer use cases for XML [5]. Whereas most previous work considers the technical aspects of the schema evolution (i.e. how to efficiently store different schema preserving *data consistency* over the changes), the goal of this poster is to provide guidelines for preserving *query integrity* on evolving schemas. Following the guidelines, databases are able to keep up with the evolving scenarios of the web applications, and the web applications are able to keep their interaction with databases (through queries) working as well.

In this context, the contributions of this poster are as follows. (1) We propose a taxonomy for the changes during XML schema evolution. (2) Based on that taxonomy, we overview their impact on schema validation. (3) We present and discuss the impact of schema evolution on query formulation. (4) We introduce guidelines for managing XML schema evolution by controlling schema changes and writing queries across schema versions.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

2. XML SCHEMA CHANGES TAXONOMY

During schema evolution, any component of a schema may change from one version to the next one. This section introduces the types of schema changes with respect to validation and summarizes how each change affects the *forward compatibility* (document instances from the original schema are valid on the new schema) and *backward compatibility* (document instances from the new schema are valid on the original schema).

Basic Changes. A taxonomy of five types of schema changes is proposed in [1]. In summary, Refinement adds optional or required elements to the schema, while Removal deletes elements from the schema. These changes are compatible between schemas only if the affected elements are optional (not instantiated by the documents). Then Extension adds new constructs to the schema (e.g. complex types), Reinterpretation changes the semantics of an element without changing its structure, and Redefinition updates the schema without changing the document instances format (e.g. to factor out common constructions). Those changes are also syntactically compatible as long as names and types are preserved.

Complex Changes. This poster extends the previous taxonomy by including 9 important types. Element Composition: groups related elements under a new element (e.g. grouping *lastName* and *firstName* on a new element *name*). Element Decomposition: ungroups subelements into individual elements. Renaming: updates the name of an element or attribute. Optionality: changes the participation semantics of an element from optional to required, or vice-versa. Renumbering: changes the element cardinality (e.g. from singleton to multiple elements). Retyping: modifies the data type of an element, which includes changing its facets. Namespaces: changes the namespace. Default values: changes the element (or attribute) default value. Reordering: changes the order of the elements inside a complex type. Most of the complex changes are not compatible unless the application on top of the database specifies some type of translation rules.

3. EVOLUTION IMPACT ON QUERIES

Besides the impact on validation, each change may impact the query formulation: queries evaluated against the documents from the original schema may not work on the new schema (and vice-versa). This section summarizes the impact of schema evolution on queries and their results (the poster presentation will detail better the impact through a series of illustrative examples).

General Queries. A rule of thumb is that if the queries do not evaluate or retrieve any element influenced by the changes, then the queries will keep working in any of the schema versions.

Basic Changes. From the five basic changes, only Refinement and Removal impact on query formulation and results (Extension causes no problem unless new elements are defined using the new construct – equivalent to Refinement; Reinterpretation does not

affect the structure of an element and nor does Redefinition).

Refinement. If a query has a new element in a predicate clause with the *exist* constraint, then the query processing is limited to the document instances from the new schema, because all instances from previous schema evaluate the predicate as *false*.

Removal. The opposite situation from Refinement, where only documents from original schemas will be evaluated.

Complex Changes. Element Composition. There are two problems with this kind of change. First, the new composed element may have a different name from the original structured element (with similar consequences to Renaming). Second, the query may refer to a specific substructure of the original schema. The impact on the query on this last case is extremely important because, even though no data is lost (from one document to the other), the qualification of the data does not exist anymore.

Element Decomposition. The same problems from Composition apply inversely to Decomposition.

Renaming. The impact on the queries depends on whether these element names are directly referred to or not. One way of solving this problem is to specify a table of synonyms in the application on top of the database. A second way is to specify all names (from previous and current schema versions) on the query. This solution is less elegant and requires specific knowledge of the schema.

Optionality. This change is related to Refinement and Removal. Changing from optional to required is related to Refinement for those optional elements that are not instantiated on the original document. Likewise, changing from required to optional is related to Removal, for its impacts on the results of the queries that reference the ex-required elements. In this case, it may happen that the result is empty. The same concerns from Refinement and Removal are therefore applied here as well.

Renumbering. Changing from singleton to multiple elements impacts on the query results that will have more instances of the same element. Changing from multiple elements to singleton is easier because it just reduces the cardinality of the results.

Retyping. This is probably the hardest change from the compatibility point of view and query impact because it requires specific procedures in order to keep the queries working. For example, consider a query that specifies a predicate with an integer comparison for a value *v*, and then *v* has its type changed to string. In this case, *casting* functions are needed in order to keep the queries working, for instance *$elem cast as xs:integer*.

Default values. Changing the default value does not affect the query formulation if the data type is the same, but it affects the results for queries on those values. Also, special attention is needed on queries that retrieve all values except the default ones by using comparison constraints. In this case, there should be different queries to access documents from different schemas.

Namespaces. In XML schema evolution, namespaces may be used to denote different schema versions. Queries that are sensitive to namespace will silently return no results on documents whose namespace are different from the namespace specified in the query. If the desired behavior is to return the same results, the query should be written with a wildcard for the namespace.

Re-ordering. Most XML queries are not order sensitive. However, if positional predicates are used in the XML queries (e.g. *//book/author[2]*), the query may return the wrong result, which could have serious consequences for the application functionality.

4. XML EVOLUTION GUIDELINES

Controlling schema changes. A related issue is how the evolving schemas are managed. It is important to decide when the application may break, for example: (i) it works across all versions of the XML schema; (ii) it breaks for major version changes; and (iii) it breaks even for minor version changes. Likewise, evaluating queries on different schemas may result in the following outcomes: (i) the query returns the correct results (correctness regarding the application semantics); (ii) it returns no results; and (iii) it returns incorrect results. The application architect needs to decide which of the three outcomes the application requires and how the outcomes should be processed.

Writing queries across schema versions. This poster assumes that web applications or services must work across all versions of the XML schema and that the query must return the correct results across all versions. Therefore, based on the previous discussions, an initial set of guidelines for keeping queries working across schema versions is provided as follows. (1) Do not add required elements (or attributes) in the middle of the hierarchy. If necessary, the queries must have *ancestor*//*descendant* axis to work on new schemas. (2) Likewise, do not delete required elements (or attributes) from the middle of the hierarchy. (3) Do not change the order of the elements on the schema when the queries consider ordered predicates. (4) Do not change atomic types if queries are strongly typed or have value comparisons. (5) Do not change element names when they are referred in a query. If necessary, then consider including a synonym control in the application such that the old and the new names refer to the same element. (6) Likewise, do not change the type or facet of elements that are referred in any query. If necessary, then update the queries as well, by adding cast functions. (7) If using namespaces to distinguish schema versions, then do consider adding wildcards (*) to the namespace specification within the queries. (8) Certain functions, such as *exist*, need special attention because they may constrain the query evaluation to a set of versions that have a specific element. (9) Querying composed/decomposed elements return results with different structures. On the other hand, if those elements are evaluated in comparisons (within a predicate) or path expression, then most probably the query will not consider document instances from all schema versions.

5. CONCLUSION

This poster presented a taxonomy for changes during XML schema evolution, and the impact of those changes on validation, query formulation and results. It proposed an initial set of guidelines for preserving queries across different schema versions. This work was not intended to exhaust the subject but rather to cover the most common situations an XML designer faces on schema evolution. Also, there are cases where queries should not work across major versioning changes. Those cases are not covered in this study and will be covered in a future article.

6. REFERENCES

[1] Technical Note: Versioning. FpML Public Documents, February 2003. http://www.fpml.org/documents/technical.html.
[2] K. S. Beyer et al. DB2/XML: Designing for Evolution. In Proceedings of SIGMOD, 2005.
[3] M. Nicola and B. V. der Linden. Native XML Support in DB2 Universal Database. In Proceedings of VLDB, 2005.
[4] J.F. Roddick. A Survey of Schema Versioning Issues for Database Systems. Info. and Software Technology, 37(7), 1995.
[5] E. Sedlar. Managing Structure in Bits & Pieces: The Killer Use Cases for XML. In Proceedings of SIGMOD, 2005.