Updating XML

Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, Daniel S. Weld
{igor, zives, alon, weld}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington, Seattle, WA 98195

ABSTRACT

As XML has developed over the past few years, its role has expanded beyond its original domain as a semantics-preserving markup language for online documents, and it is now also the *de facto* format for interchanging data between heterogeneous systems. Data sources export XML "views" over their data, and other systems can directly import or query these views. As a result, there has been great interest in languages and systems for expressing queries over XML data, whether the XML is stored in a repository or generated as a view over some other data storage format.

Clearly, in order to fully evolve XML into a universal data representation and sharing format, we must allow users to specify updates to XML documents and must develop techniques to process them efficiently. Update capabilities are important not only for modifying XML documents, but also for propagating changes through XML views and for expressing and transmitting changes to documents. This paper begins by proposing a set of basic update operations for both ordered and unordered XML data. We next describe extensions to the proposed standard XML query language, XQuery, to incorporate the update operations. We then consider alternative methods for implementing update operations when the XML data is mapped into a relational database. Finally, we describe an experimental evaluation of the alternative techniques for implementing our extensions.

1. INTRODUCTION

Over the past several years, there has been a tremendous surge of interest in XML as a universal, queryable representation for data. This has in part been stimulated by the growth of the Web and e-commerce, where XML has almost instantly emerged as the *de facto* standard for information interchange. Nearly every vendor of data management tools has added support for exporting, viewing, and in some cases even importing, XML-formatted data. Tools for querying and integrating XML are still largely in their infancy, but are beginning to emerge. XML document repositories like ObjectDesign's eXcelon [18] and Software AG's Tamino [16] are now available, and XML publishing capabilities have been added to the latest relational database systems from Oracle, IBM, and Microsoft.

Ultimately, it is expected that these relational database engines will provide standardized, integrated support for querying and publishing XML views of databases. This will allow the sharing of data from both XML repositories and traditional relational databases using a single, unified, queryable model — namely, XML views with an XML query language. The World Wide Web Consortium is in the process of developing a standard for this XML query language, called XQuery [3]. Meanwhile, the database research community has been hard at work in addressing the challenges of providing XML views of relational databases, with systems such as SilkRoute [9] and XPERANTO [2].

The next step in making XML into a full-featured data exchange format is to support not only queries, but updates, over XML content. It should be possible to modify content within XML documents and to express updates to XML views, which are percolated back to the original data. The ability to encapsulate an update operation is also necessary for expressing incremental changes ("deltas") over content, which is important for Continuous Queries [4], XML document mirroring, caching, and replication.

In this paper we propose a set of constructs for expressing such updates in both an ordered and unordered XML data model, we map these constructs into the syntax of the XQuery language, and we describe implementation techniques over a relational database system. In particular, we make the following contributions:

- We propose a set of primitive operations for modifying the structure and content of an XML document.
- We present update extensions to the World Wide Web Consortium's proposed XQuery standard query language.
- We provide algorithms for implementing XML update capabilities within an XML repository based on a relational database system, and describe how these techniques can also be applied to the problem of updating relational databases through XML views. Most frequently-updated data tends to reside in relational systems, and we feel it is important to have well-studied techniques for updating complex XML structure mapped to a relational system.
- We provide an analysis of the performance of our different update strategies using a number of workloads and document structures.

This paper is structured as follows. We begin with a description of related work in Section 2. In Section 3, we provide an overview of the XML data model and describe a set of primitive operations for updating both ordered and unordered XML data. Section 4 describes how these operations can be added to the XQuery language. Section 5 reviews key concepts in mapping between XML and relational databases, and Section 6 presents our implementation strategies. We evaluate these techniques in Section 7 and present conclusions and future directions in Section 8.

ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA

2. RELATED WORK

The topic of updating XML data has received little attention thus far, as the XML community has primarily focused on development of query languages such as XML-QL [6] and XQuery [3] and their semantics. The eXcelon XML repository [17] is one of the few XML systems that support updates, and it expresses simple insertions and deletions using an extension to the XPath [5] language. The Lorel [1] query language from Stanford's Lore semistructured database system supports simple insertions and deletions of nodes into the Lore data graph. Recent extensions to Lorel [11] have migrated the query language to the XML data model, but the update features were not ported in the process. Object-oriented systems also support updates, but they tend to use a more programming-language-like set of operations: assignments from one object to another, or insertions or deletions from collection types. Our language extensions allow insertions, deletions, and assignments at multiple levels within a hierarchy, in order to support updates to complex structure — including derived XML Schema [19] types.

The problem of storing XML in relational database systems has been extensively studied [10, 7, 14]; the problem of extracting an XML view of relational tables has been addressed by [9, 15, 2]. Our techniques leverage these works, supplementing them with Access Support Relations [12], and our implementation focuses on the problem of supporting updates. While the problem of updating hierarchical data stored across multiple tables is not a new one — having arisen in the context of object-relational systems [13] — we believe we are the first to address it at the SQL level, and for storage of XML data.

3. XML UPDATE FUNDAMENTALS

Numerous XML query languages have been proposed by both the document and database communities. We have chosen to focus our paper on the emerging XQuery [3] standard, which attempts to combine the best features of the leading XML query languages and is expected to become the "SQL of XML."

In this section, we present our basic data model and update operations in a language-independent way in order to carefully define our semantics. The next section maps our basic set of operations into the XQuery language.

3.1 Data Model

XQuery uses the World Wide Web Consortium's XML Query Data Model [8], which views an XML document as a node-labeled tree with references. We use a simplified version of the model for purposes of illustration here.

Figure 1 shows an example XML document and Figure 2 illustrates its tree-structured representation. We draw the tree such that a left-to-right, depth-first traversal describes the order of the XML content within our document. Note that in this representation, we model all attributes uniformly, including those with special meaning, such as `IDREF`s. Attributes are unordered with respect to one another; however, it is important to note that a given `IDREFS` attribute is itself an ordered list of references. In XQuery, all `IDREF` attributes are explicitly dereferenced in path expressions by using the `->` operator.

In our discussion we need to refer to different kinds of XML content. We use the term *object* to refer to any component of XML, which can be any of the following:

- An attribute is a pair, with an name and a string value.
- An `IDREFS` is a named ordered list of `ID`s. For simplicity of presentation, we assume that `IDREF` and `IDREFS` types are equivalent, *i.e.*, an `IDREF` is a singleton list.
- An element is a tuple with a name, set of attributes, set of references, and list of child elements or `PCDATA`.
- `PCDATA` (scalar) content is a `string` value that exists within an element.

3.2 Update Operations

We now describe a set of update operations on XML documents. Our goal is to provide a uniform and clean semantics for updating not simply scalar or leaf-node values (as in Lorel), but also complex, structured, and irregular types (in particular, complex XML Schema [19] types *and* their derived subtypes). Here we distinguish between `IDREF` types and attribute content, as `IDREF`s encode structural information as opposed to data values, and they may appear within ordered `IDREFS` lists.

Following the style of XQuery, we assume the presence of a path-expression-matching operation that binds variables to objects within the input XML document and returns tuples of references to the selected objects. One of these bindings will be the *target* of the sequence of operations, and is assumed implicit in the specification below. The update operations also take a set of parameters (`content` and `name` below). An update is a sequence of primitive operations of the following types:

- **Delete**(*child*): if the *child* is a member of the *target* object, it is removed. Valid types for *child* include `PCDATA`, attribute, `IDREF` within an `IDREFS` list, and element. If the *child* is a reference within an `IDREFS`, only the single entry is removed — the remainder of the `IDREFS` is preserved.
- **Rename**(*child*, `name`): if the *child* is a non-`PCDATA` member of the *target* object, it is given a new name. Note that we cannot rename an individual `IDREF` within an `IDREFS`; such a rename operation will rename the entire `IDREFS`.
- **Insert**(`content`): inserts new `content` (which can be `PCDATA`, element, attribute, or reference) into *target*. An attempt to insert an attribute with the same name as an existing attribute fails. An attempt to insert a reference with the same name as an existing `IDREFS` adds an extra entry into the `IDREFS`.

  In an ordered execution model, all non-attribute insertions are defined to occur at the *end*, *i.e.*, the new content is appended.
- **InsertBefore**(*ref*, `content`): (defined only for ordered execution). If *ref* is a child element of *target* or `PCDATA`, then `content` must be an element or `PCDATA`, and it will be inserted directly before *ref* in *target*'s list of children. If *ref* is an entry in an `IDREFS`, then `content` must be an `ID` and it is inserted directly ahead of *ref* in the `IDREFS`. **InsertAfter**(*ref*, `content`) is defined analogously.
- **Replace**(*child*, `content`): atomic replace operation, equivalent to InsertBefore(*child*,`content`) followed by Delete(*child*) in the ordered model, or (Insert(`content`), Delete(*child*)) under unordered execution.

```
<db lab="lalab">
 <university ID="ucla">
  <lab ID="lalab" managers="smith1 jones1">
   <name>UCLA Bio Lab</name>
   <city>Los Angeles</city>
  </lab>
 </university>
 <lab ID="baselab" managers="smith1">
  <name>Seattle Bio Lab</name>
  <location>
    <city>Seattle</city>
    <country>USA</country>
  </location>
 </lab>
 <lab ID="lab2">
  <name>PMBL</name>
  <city>Philadelphia</city>
  <country>USA</country>
 </lab>
 <paper ID="Smith991231" source="lab2"
        category="spectral" biologist="smith1">
  <title>Autocatalysis of Spectral...</title>
 </paper>
 <biologist ID="smith1">
   <lastname>Smith</lastname>
 </biologist>
 <biologist ID="jones1" age="32">
   <lastname>Jones</lastname>
 </biologist>
</db>
```

Figure 1: Sample XML document representing biology labs and publications

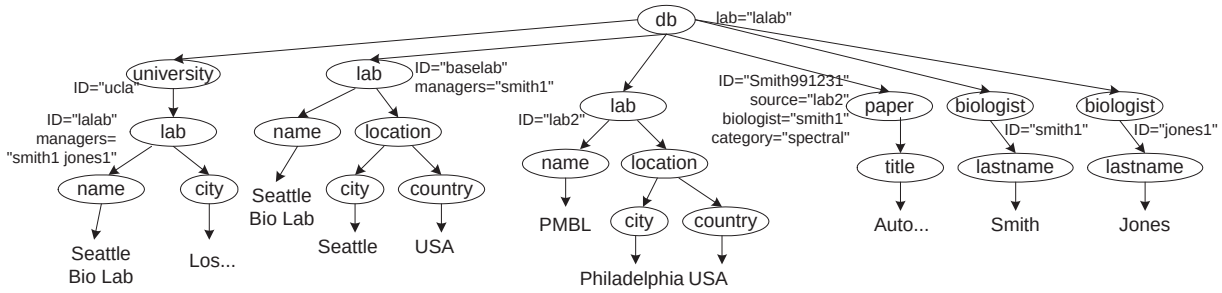

Figure 2: Data model representation for Figure 1.

- **Sub-Update**(*patternMatch*, *predicates*, *updateOp*): starting at the *target* element, invokes a new pattern-matching operation over the input, returning bindings that are filtered by the *predicates*. For each valid combination of bindings, recursively invokes the update operation. This allows us to express updates at multiple levels within a complex XML structure.

A full update-operation may consist of several of these sub-operations that execute in sequence. Therefore, we add several additional restrictions to the semantics of our operations in order to prevent ill-defined semantics. All bindings within **Sub-Update** operations are made over the input *before* any updates take place. Likewise, *content* is evaluated for each *target* before the sequence of updates is executed. Finally, a binding that has been deleted cannot be used by any operations later in the sequence (except as *content*).

4. XQUERY EXTENSIONS FOR UPDATES

The basic set of operations presented in Section 3 express XML updates logically; the next step is to take these operations and map them into the XQuery language syntax. We begin by describing the general form of our update extensions, and then elaborate upon specific operations.

4.1 Basic Form of Updates

We extend XQuery with a `FOR...LET... WHERE...UPDATE` structure for updates. Within the `UPDATE` clause, a sequence of sub-operations (following the same semantics as our logical **update-ops**) are specified:

`FOR $`*binding1* `IN` *XPath-expr*, ...
`LET $`*binding* `:=` *XPath-expr*, ...
`WHERE` *predicate1*, ...
updateOp, ...

where **updateOp** is defined in EBNF as:

`UPDATE $`*binding* { **subOp** {, **subOp**}* }

and **subOp** is:

`DELETE $`*child* |
`RENAME $`*child* `TO` *name* |
`INSERT` *content* [`BEFORE` | `AFTER $`*child*] |
`REPLACE $`*child* `WITH $`*content* |
`FOR $`*binding'* `IN` *XPath-subexpr*, ...
`WHERE` *predicate1*, ... **updateOp**

The nested `FOR...WHERE` clause allows one to specify an XPath expression to be matched beginning at $*binding*, as well as a set of predicates that may restrict its bindings. A nested update operation may be performed over any of the bindings from the outer scope or the `FOR` clause. If multiple **updateOp**s are specified, they are performed consecutively for each iteration of variable bindings, following the semantics described in the previous section.

4.2 Language Extensions in Detail

The previous section makes the assumption that the XPath expressions are sufficiently expressive that one can select the XML objects one wishes to manipulate. While this is certainly true of elements, XQuery does not actually define how to select an attribute as a whole (only its value), nor an individual `IDREF` out of an `IDREFS` list. Thus we introduce conventions for binding to attributes and `IDREF`s.

Specifically, we assume that a variable bound to an attribute (*e.g.*, `FOR $x IN tag/@attr`) represents a reference to that attribute (and not simply the value) within the document. Thus one can modify the attribute by updating the object `$x`. However, an expression over the *value* of `$x` will get its string *content*.

To bind a variable to a specific reference within an `IDREF` or `IDREFS`, we introduce a function `ref(label, target)` that can be bound to a variable. An example that binds `$l` to the reference to `lalab` in the document of Figure 1

follows:

```
FOR $l IN document("bio.xml")/ref(lab, "lalab")
```

Now we examine each of the basic operations individually, providing an XQuery example in each case.

4.2.1 Deletion

Deletion is in many ways the simplest operation, as it simply requires a binding to a parent and a child. The following query illustrates an operation in which we select the `paper` element to update, and remove its `category` attribute, its `biologist` reference to `smith1`, and its `title` subelement.

```
FOR $p IN document("bio.xml")/paper,
    $cat IN $p/@category,
    $bio IN $p/ref(biologist,"smith1"),
    $ti IN $p/title
UPDATE $p {
  DELETE $cat,
  DELETE $bio,
  DELETE $ti
}
```

Example 1: Deleting an attribute, `IDREF`, and subelement

Deleting a subelement node will typically remove all of its content. However, there may be *references* to deleted subelements. XQuery allows "dangling" references to data that is not present in the results, so we support these same semantics in a delete operation — a reference is allowed to dangle.

4.2.2 Insertion

With insertion, we must introduce a constructor for new attributes or references to be inserted. We illustrate this with the example below, which inserts an `age` attribute, two `worksAt` references, and a `firstname` subelement into biologist `smith`'s entry:

```
FOR $bio in document("bio.xml")/db/
                biologist[@ID="smith1"]
UPDATE $bio {
  INSERT new_attribute(age,"29"),
  INSERT new_ref(worksAt,"ucla"),
  INSERT new_ref(worksAt,"baselab"),
  INSERT <firstname>Jeff</firstname>
}
```

Example 2: Inserting an attribute, two references, and a subelement

In an unordered execution model, the child elements within the biologist entry would appear in any arbitrary order as a result of this update. For the ordered model, each successive reference would be inserted at the end of the `worksAt` list, and the `firstname` subelement would appear after any existing subelements. We can also specify positional insertion — in this case, adding a `street` after the `name` element in a `lab` and adding `"jones1"` as a first `managers` reference:

```
FOR $lab in document("bio.xml")/db/
                lab[@ID="baselab"],
    $n IN $lab/name,
    $sref IN ref(managers,"smith1")
UPDATE $lab {
  INSERT "jones1" BEFORE $sref,
  INSERT <street>Oak</street> AFTER $n
}
```

Example 3: Inserting a subelement and a reference relative to existing content

Note that the `new_ref()` constructor is not necessary here, because the insertion is relative to an `IDREFS` binding.

4.2.3 Replacement

Replacing an item has the same effect as inserting a new item before it and deleting it, but it is often convenient to be able to express it as a single, atomic operation. The simple example below replaces `lab baselab`'s manager and name:

```
FOR $lab in document("bio.xml")/db/lab,
    $name IN $lab/name,
    $mgr IN $lab/ref(managers, *)
UPDATE $lab {
  REPLACE $name WITH <appellation>Fancy Lab</>,
  REPLACE $mgr WITH new_attribute(managers,"jones1")
}
```

Example 4: Replacing elements, references, and attributes

Note the use of the wildcard character,"*", within the `ref()` function to match against any references. A reference binding can *only* be replaced with another reference of the same label, *i.e.*, a manager can only be replaced with other managers.

4.2.4 Combining Operations with Nested Updates

The nested `FOR` clause enables us to express updates at multiple levels in an XML document, selecting the desired nodes at each level. We illustrate these with an example that updates the `university ucla` by adding a `labs` attribute on the number of labs, creating a new `lab`, and modifying the existing one:

```
FOR $u in document("bio.xml")/db/university
            [@ID="ucla"],
    $lab IN $u/name
WHERE $lab.index() = 0
UPDATE $u {
  INSERT new_attribute(labs,"2"),
  INSERT <lab ID="newlab">
           <name>UCLA Secondary Lab</name>
         </lab> BEFORE $lab,
  FOR $l1 IN $u/lab,
      $labname IN $l1/name,
      $ci IN $l1/city
  UPDATE $l1 {
    REPLACE $labname WITH <name>UCLA Primary Lab</>,
    DELETE $ci
  }
}
```

Example 5: Multi-level nested update

Our outermost `UPDATE` operation works on the `university` tag, and a nested pattern match is performed inside the operation so the `lab` can also be modified. See Figure 3 for the output document.

5. STORING XML IN RELATIONS

Although XML repository systems have been built over object-oriented [18] and hierarchical [16] databases, they are most commonly constructed over a relational database system, using a middleware layer that translates queries from an XML query language into SQL queries over the database [10, 14, 7]. Another similar source of queryable XML data is an XML mediator placed over an existing relational database [2, 9]. The mediator takes a user-provided mapping and generates a hierarchical XML view of the existing relational tables; XML-based queries made over this view are translated to SQL. There is a key difference between an XML repository with a relational core and an XML view manager over a relational database: in the former case, the relational schema and the relationships between tables are

```
<db lab="lalab">
 <university ID="ucla" labs="2">
  <lab ID="newlab">
    <name>UCLA Secondary Lab</name>
  </lab>
  <lab ID="lalab" managers="smith1 jones1">
   <name>UCLA Primary Lab</name>
  </lab>
 </university>
 <lab ID="baselab" managers="smith1">
  ...
 </lab>
 <lab ID="lab2">...</lab>
 <paper ID="Smith991231" source="lab2"
        category="spectral" biologist="smith1">
  ...
 </paper>
 <biologist ID="smith1">
   ...
 </biologist>
 <biologist ID="jones1" age="32">
   ...
 </biologist>
</db>
```

Figure 3: Results of multi-level update to university and its labs, Example 5

automatically generated from the XML input; in the latter, the relations are provided along with a view definition for the XML mapping. As a result, the query translation algorithms are somewhat different. However, both types of systems can make use of the same techniques for translation from hierarchical XML queries to generated SQL queries, and in transforming from relational output to XML.

Likewise, the same basic XML update techniques can be applied to both repository and mediator contexts. Since XML-relational techniques cover such a wide portion of the XML query spectrum, we have selected this domain for implementing our XQuery extensions. In this section, we review and summarize the basic techniques used for representing XML in relational tables and for converting relational query results back into XML. Then we present a number of alternatives for executing our XQuery update extensions over relational tables.

The first issue in mapping between XML and a set of relations is the correspondence between a relation and some portion of the XML document's hierarchy. With an XML mediator, this is provided by the XML view definition; in an XML repository, the system must determine the mapping. Once a mapping to tables has been established, a means of converting from tabular output to XML must be determined. Finally, queries across multiple tables can be sped up with the use of Access Support Relations, an indexing technique from object-oriented databases. We now examine each of these topics in turn.

5.1 Mapping XML into Relations

Several methods have been proposed for mapping XML documents into relations [10, 14, 7]. In [10] the authors describe the *Edge* and *Attribute* approaches. In the former, each element or attribute is stored as a tuple in a single "edge" relation, while in the latter we create a similar binary table for *each* tag or attribute name in the document. The main drawback of these two approaches is that they cause excessive fragmentation of XML elements across multiple tuples and relations, and therefore traversing XML structure or outputting XML content requires many joins, which can be very expensive. The *Edge* approach does have one advantage over most other techniques, which is that it works with input documents that do not have DTDs.

The *Shared Inlining* method [14] exploits a DTD to better cluster parent and child elements. Informally, the method tries to store child elements and attributes in the same tuple as their parents, thus reducing the number of joins required. The DTD provides the information necessary to determine where inlining is possible. For an example of the inlining method, we begin with the DTD in Figure 4, which is a simplified version of the TPC/W web benchmark's schema.

```
<!ELEMENT CustDB (Customer*)>
<!ELEMENT Customer (Name, Address, Order*)>
<!ELEMENT Address (City, State)>
<!ELEMENT Order (Date, OrderLine*)>
<!ELEMENT OrderLine (ItemName, Qty)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Address (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT State (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT ItemName (#PCDATA)>
<!ELEMENT Qty (#PCDATA)>
```

Figure 4: DTD of the example customer database.

According to the DTD, some of the child elements appear exactly once for each occurrence of their parents, *e.g.*, every customer has a name. These elements will be "inlined" as attributes within their parent elements' tuples (the `Customer` table will contain an attribute for `Name`). Conversely, all subelements with $1:n$ relationships to their parents cannot be inlined and must be stored in separate tables. Shared Inlining will create 4 relations for our example: `CustDB`, `Customer`, `Order`, and `OrderLine`. In addition to the corresponding PCDATA attributes, each relation will also have an `id` and a `parentId` attribute. These attributes will be used to link tuples corresponding to parent/child element pairs in the stored XML document.

In our work we experimented with several storage schemes. In the rest of the paper we focus on the inlining approach both because of its flexibility and because the other storage schemes did not yield any different results or insights. We do not consider the issue of document order in our discussion.

5.2 XML Results as Outer Unions

When an XML structure is stored across multiple tables, there are a number of possible ways of returning the results. One approach might be to open a series of nested cursors across each table, but this would require modification to the underlying RDBMS. Another approach could be to create a "wide" tuple containing all possible attributes within the XML subtree, then joining the constituent relational tables and returning the results, but this produces large amounts of redundant data. An alternate approach essentially simulates the nested cursor approach: different levels in the XML hierarchy (*i.e.*, different relational tables being returned in the results) are returned in separate tuples in the same stream. Since each of these tuples comes from a different table and schema, we must create a "wide" tuple with all possible attributes, and pad any given table's tuples with NULLs so it

```
WITH Q1(C1, C2, C3, C4, C5, C6, C7, C8, C9) AS (
                                -- Customer subquery
  SELECT id, Name, Address_City, Address_State,
         NULL, NULL, NULL, NULL, NULL
  FROM Customer
  WHERE Name = 'John'

), Q2(C1, C2, C3, C4, C5, C6, C7, C8, C9) AS (
                                -- Order subquery
  SELECT C1, NULL, NULL, NULL,
         id, Status, NULL, NULL, NULL
  FROM Q1, Order O
  WHERE O.parentId = Q.C1

), Q3(C1, C2, C3, C4, C5, C6, C7, C8, C9) AS (
                                -- OrderLine subquery
  SELECT C1, NULL, NULL, NULL,
         C5, NULL, id, ItemName, Qty
  FROM Q2, OrderLine OL
  WHERE OL.parentId = Q.C5
) (
  SELECT *
  FROM Q1
) UNION ALL (
  SELECT *
  FROM Q2
) UNION ALL (
  SELECT *
  FROM Q3
)
ORDER BY C1, C5, C7
```

Figure 5: Outer Union query for Example 6

fits into the wide tuple. This technique was first proposed in [15] and is called the "outer union" method. To simplify the job of reconstructing the XML document at the client, output tuples are sorted so that child element data comes after parent data and child elements of different parents are not intermixed.

Consider the following simple XQuery expression that retrieves data for customers named John:

```
FOR $c IN document("custdb.xml")/
          CustDb/Customer[Name="John"]
RETURN $c
```

Example 6: Returning customer "John" in XQuery

Instead of separately joining the relations `Customer`, `Order` and `OrderLine`, the outer-union approach would produce the query of Figure 5[1], which employs the IBM DB2 `WITH` clause for reusable subexpressions. Note that child tuples include the key attributes, but not the content, of their parents, so the appropriate association between parent and child can be maintained. This also facilitates the use of `ORDER BY` to sort the tuple stream such that child elements follow their parents. If the WHERE clause of an XQuery expression contains conditions on elements' values, all such conditions have to be tested in the first, base subquery of the WITH clause, since the other "branches" of the Outer Union cannot remove tuples. A fix-point query can be used to evaluate regular path expressions as described in [14].

[1]For readability, this Figure is a simplification, omitting the required typecasting of NULLs to match attribute types.

5.3 Access Support Relations for Path Expression Evaluation

Access support relations (ASRs) are an effective method of speeding up the evaluation of path expressions in object-oriented and semi-structured databases [12]. An ASR indexes a specific path, and it contains an attribute for each object along the path. This principle extends naturally to the XML mapping described in this paper: each object along the path is a tuple within a table at a particular depth in the element hierarchy. Each tuple has an XML element ID, corresponding to an Object ID in a traditional ASR. Hence, an ASR tuple encodes a path from an XML element to one of its descendants. XML queries are generally expressed "downwards" from a specified node in the XML tree, and certain children may or may not exist within the semistructured data, so we use a left-complete extension (*i.e.*, NULLs are allowed only at the bottom of the tree).

To see how ASRs speed path expression evaluation, suppose the customer XML document has been extended with part information stored in a separate Parts table, and consider the following query that returns the names of customers who have ordered an item built with part 123:

```
FOR $c IN document("custdb.xml")/CustDb.Customer
          [Order.OrderLine.Item.Part.Number=123]
    $n IN $c/Name
RETURN $n
```

Example 7: Customers Ordering Items with Part 123

This query can be evaluated with three joins: select `Part` tuples with number 123, then join with the `Item`, `Order` and `Customer` relations to retrieve corresponding customer names. However, if we have an ASR that contains paths from the root element to leaf (`Part`) elements, then the above query can be evaluated using only two joins: first, `Part` and ASR relations are joined, and then the result is joined with `Customer` to obtain customer names. (These queries can be further sped up by also indexing the various tables and the ASR by the required IDs.)

Clearly, ASRs are especially effective for long paths. However, even for short paths (*e.g.*, `Customer.Order.OrderLine`), joining with the ASR can be faster than joining with the intermediate relation, because the ASR can be more compact since it does not contain data values.

6. UPDATING STORED XML

The challenges of translating XML updates into SQL go beyond those of translating queries. Clearly we would like to generate efficient SQL update statements; ideally, as with the case of XML queries, we would translate each XML update into a single SQL command, because issuing multiple separate SQL statements incurs overhead and prevents the RDBMS from performing large-scale optimizations. Unfortunately, a SQL update statement can only modify a single relation, so we *must* issue multiple queries to update an XML document at several levels of hierarchy. For example, an XML INSERT of a new item of customer data into our sample XML database may easily require three SQL INSERT statements to modify the `Customer, Order`, and `OrderLine` relations. However, certain approaches to insertion may require more statements than others; we examine a number of alternatives in detail.

Another difficulty is the creation and maintenance of the

element ID-parent ID foreign-key relationship that encodes XML structure. When an XML subtree is being copied from one location to another, all tuples must be replicated and given new IDs that have the same connectivity.

There are also subtle issues to be considered to ensure correctness of the translation. Consider the following example query, which selects customer orders that are ready and contain an order line for tires. The status of the selected orders is set to "suspended" and a comment is added to each tire order line.

```
FOR $o IN document("custdb.xml")//Order
            [status="ready" and
             OrderLine/ItemName="tire"]
UPDATE $o {
  INSERT <Status>suspended</Status>
  FOR $i IN $o/OrderLine[ItemName="tire"]
  UPDATE $i {
    INSERT <comment>recalled</>} }
```

Example 8: Suspending Orders of Tires in XQuery

Suppose this XML update statement is translated into two SQL updates to update the `Order` and `OrderLine` base relations. Clearly, if the first update is executed before the second, the second update will not apply to any tuples because the status of all relevant orders has been changed to "suspended." For this pair of updates, the second statement must be executed first. The example also illustrates a potential performance problem: the second update performs the same joins and selection operations as the first. Later in this section, we present methods for avoiding this redundant work.

Yet another issue, one which we do not address in this paper, is validation of operations against a DTD. In this section, we present a number of algorithms that all assume a valid operation is being performed.

6.1 XML Deletion Techniques

Certain "simple" deletion operations may involve the removal of an item from a single table — generally this occurs during deletion of an XML element that has been inlined with its parent (or ancestor). The major expense of such an operation lies in the query that finds the specific element; the actual operation can be performed by a simple SQL UPDATE that sets the element's attribute(s) to NULL. A possible caveat arises when a non-leaf element and its children have all been inlined together with a parent, and are deleted: the non-leaf element does not have a data value of its own, so if we set its children to NULL, this results in ambiguity as to whether the non-leaf element has been deleted or it exists with empty content. In this case, we add a flag specifying whether the non-leaf element is present or absent.

The operation that is of greater interest to us is a deletion of a subtree that is stored across multiple relations — in this case, tuples must be removed from subsidiary tables as well as from the target of the deletion. In the remainder of this section we describe several possible translations of such a "complex" XML delete to SQL. As our example, we will be using the following delete statement, which deletes customer data for all customers whose name is "John".

```
FOR $d IN document("custdb.xml"),
    $c IN $d/Customer[Name="John"]
UPDATE $d {
  DELETE $c }
```

Example 9: Deleting customers named "John" in XQuery

6.1.1 Trigger-Based Delete

SQL triggers are perhaps the most natural mechanism for initiating successive deletes of subsidiary tuples based on a single delete. Triggers, which are supported by most commercial RDBMSes today, are event-initiated rules that can be attached to specific relations. Whenever a relation is updated, the trigger is fired and its SQL code is executed. Triggers can be fired either individually for each affected tuple, or once for an entire delete statement.

Triggers greatly simplify the implementation of the multi-level delete operation that we need. For each relation that has child relations, we define a post-delete trigger that is fired whenever a parent tuple is deleted. The body of the trigger contains SQL code to delete relevant tuples from the child relations. To delete an element that is mapped into a relation, one only needs to delete the element's root tuple. For example, our "delete Johns" query can be translated into the following simple SQL statement:

```
DELETE FROM Customer WHERE Name='John'
```

The relevant `Order` and `OrderLine` tuples will be automatically deleted by triggers set up during the creation of the relational schema. Since only a single SQL statement needs to be issued and the entire delete operation is handled inside the RDBMS, we expect trigger-based deletes to be very efficient.

6.1.1.1 Per-Tuple vs. Per-Statement Triggers.

As we mentioned above, triggers can be fired on either a per-tuple or per-statement basis. Accordingly, there are two ways to propagate deletes through an XML tree. In a per-tuple trigger, the deleted tuple is still accessible, as is its `id`. Therefore, the trigger should delete all tuples from child relations that link to that tuple through their `parentId` attribute. That, in turn, will cause the triggers on the child relations to be fired, and so on. In contrast, a per-statement trigger is fired after all relevant tuples have been deleted from the relation. To propagate the delete, it is necessary to delete orphaned tuples from the child relations.

6.1.2 Cascading Delete

When a DBMS does not support triggers, it is possible to simulate the effect of per-statement triggers by first deleting relevant tuples from parent relations and then deleting orphaned tuples from their children. We stop as soon as a delete operation does not remove any tuples. Note that we can apply this method even if the schema is recursive.

To delete a `Customer` element for John, the following three statements can be executed in sequence:

```
DELETE FROM Customer WHERE Name = 'John'
DELETE FROM Order
  WHERE parentId NOT IN (SELECT id FROM Customer)
DELETE FROM OrderLine
  WHERE parentId NOT IN (SELECT id FROM Order)
```

Intuitively, the Cascading Delete method should have similar performance characteristics to the per-statement trigger method, except that it has slightly more overhead since it

requires more SQL statements. As discussed in Section 7, this is indeed the case.

6.1.3 ASR-Based Delete

Access support relations can be used not only to improve query response time (as discussed in Section 5), but also to speed up deletions. Specifically, once the internal tuple `id` of an element to be deleted is known, the `id`s of its descendants can be obtained from the ASR. A single join of the deleted tuple with the ASR yields the `id`s of child tuples below the delete point. To perform the delete, we mark each ASR tuple containing the deleted object. Then, for each child table, we delete all tuples whose IDs match a deleted row in the ASR. Finally, we must update the ASR to reflect the current state of the data.

We expect that the ASR method will not perform as well as the trigger based delete methods because of the larger number of SQL statements that needs to be executed and the extra overhead of updating the ASR.

6.2 XML Insertion Techniques

As with deletions, we can divide insertions into "simple" (flat) and "complex" (hierarchical) operations. Simple insertions, in which the element to be inserted is completely inlined, can be performed using a single SQL UPDATE operation. However, if we want to generate a warning on any attempt to insert "over" an item that may only occur once in the DTD, we must initially query the table to ensure that every tuple we wish to update has NULL values in the appropriate attributes.

A complex insertion contains an XML subtree and requires updates to tuples across multiple tables. This operation may insert literal content and/or copy data from within an existing XML document, and it may even impose a correlation between the destination and source data. Insertion of data from within an existing document is more challenging, so we provide an example. Suppose we want to copy `Customer` elements describing Californians into a different document (with an appropriate DTD):

```
FOR $source IN document("custDB.xml")/CustDB/
                  Customer[Address/State="CA"],
    $target IN document("CA-customers.xml")/CustDB
UPDATE $target {
  INSERT $source }
```

Example 10: Inserting Californian Customers in XQuery

There are two particularly challenging aspects to executing this insertion. The first is in minimizing the number of SQL insertions or updates required. The second and more difficult problem lies in generating new `id`s for the duplicate data while maintaining its structure. The insertion operation has copy semantics, so we cannot simply link to the existing tuples; moreover, `id`s must be unique within a document, so we cannot simply copy the tuples.

One solution to the `id` assignment problem would be to provide an `id` mapping function similar to a Skolem function in XML-QL: given a particular source and destination `id`, it would return a unique new `id` as a "perfect hash." Unfortunately, no RDBMS provides this capability, so we would have to perform the operation ourselves (and maintain some sort of mapping state). We prefer to avoid the overhead of this method, so in the following two sections, we discuss two approaches of the complex insert operation that do not use an `id` mapping function.

6.2.1 Tuple-Based Insert

The tuple-based insertion method begins by querying for the source subtree using the Sorted Outer Union method and reading a tuple at a time. Then every source element within the tuple is given a new and unique `id` (parent elements shared across tuples will, of course, share the same new `id`, so a mapping structure must be maintained during execution). Finally, the Sorted Outer Union tuple is partitioned back into components matching the underlying tables, and these are inserted.

The memory requirements of this method are very low, since they are bounded by the size of the Outer Union tuple. However, there is a large penalty in terms of the number of SQL INSERT statements required – one must be generated for each table within each tuple. One advantage of the tuple method is that it allocates tuple `id`s without gaps, unlike the table and ASR methods described below.

6.2.2 Table-Based Insert

A logical way to reduce the number of statements required by the tuple-based insert is to remap all source tuples in a single operation, and then insert the remapped tuples *en masse* within each underlying table. Often, this operation will simply require buffering of the source data within memory, but in the worst case, there might be too much data and a temporary table will be required. In the table-based insert method, we always make use of the temporary table as our work area, and assume that the database's buffer manager will keep the results in memory when possible.

In order to quickly remap `id`s across a large number of tuples, we employ the following heuristic. If we find the minimum and maximum tuple `id` values ($minId$ and $maxId$, respectively) in the source tree (the temporary table), we know the upper bound on the number of used `id`s is given by $maxId - minId + 1$. Given a systemwide "next available `id`" counter $nextId$, we can add an offset of $nextId - minId$ to each `id` in the temporary table, and can advance $nextId$ by $maxId - minId + 1$.

For even moderately sized subtrees, the table-based insert method should perform better than the tuple-based approach because it executes a single SQL insertion per data relation.

6.2.3 ASR-Based Insert

The main reason for the temporary table in the insert method described above was to find the source subtree data and its `id`s. This is the primary role of Access Support Relations, so it is not surprising that ASRs can be used to enhance insert performance. Instead of creating a temporary table, an ASR-based insertion operation can scan the ASR for all child tuple `id`s and compute the remapping offset (as described above for table-based inserts). Then the ASR can be used to retrieve and replicate the source tuple data. Our ASR-based insert uses a marking scheme (similar to the ASR delete approach described previously) to identify ASR paths through the source; thus it requires SQL update statements to mark and unmark the ASR, to add new paths to the ASR for the inserted data, and to update each source table. This is actually a higher number of operations than the table-based insert, but does not require excess duplication of data, nor does it require the join-intensive generation of Outer Union results.

6.3 Other Update Operations

We can leverage the basic algorithms for querying, inserting, and deleting to form the remaining operations. Thus, because of space constraints, in this section we only briefly highlight the algorithms and their potential optimizations.

The replace operation removes an existing subtree and adds a new one in its place. Clearly, this can be implemented as a deletion followed by an insertion. Note, however, that it is possible to replace a tree with the value of one of its subtrees. In such cases, a special-case operation can be performed: the new subtree is linked to its new parent, and the remainder of the "old" subtree is deleted.

The rename operation logically changes the element or attribute name around content; in our relational mapping, this involves moving the data from one attribute or table to another. This can be simulated by an insertion to copy a subtree to its new place, followed by a deletion of the original. However, the process can be greatly optimized by two observations: a rename only affects the outermost level of a subtree, so only the top-level table needs updating; and since renaming involves movement but not creation of data, no new `id`s need to be generated.

One unique aspect of our update extensions is their use of nested update expressions. In order to prevent interaction between different levels, language semantics define that all bindings are made prior to updates. This actually simplifies the mapping to SQL.

Our approach to implementing a multilevel update is as follows. First, we perform a Sorted Outer Union query (potentially making use of an ASR) to compute *all* of the various source and target bindings within the update statement, including all sub-operations. Then we sequentially execute the sub-operations over the appropriate bindings.

7. EXPERIMENTAL RESULTS

In order to determine which update methods work best and under what conditions, we ran a comprehensive series of experiments. Since space limits preclude presentation of all of our results, this section reports the highlights.

Our code was written in Java; it translated update queries into SQL, and used JDBC to communicate with IBM DB2 UDB 7.1. To avoid network delays, all processing was done on the same computer, an 866 MHz Pentium III with 1GB of main memory, running Windows 2000. Since we are interested primarily in optimal performance of a relational database for XML updates, our experiments focused on the case when all data could fit within memory.

In order to ensure consistency, each experiment consisted of set of 5 runs with the results of the first run discarded. Thus, each graph point represents the average time for five runs; since variance was small (10-15%), we do not show confidence intervals.

To avoid reloading the database after each run, we did not commit our transactions. We have run several experiments with commits and did not notice any significant increase in running time. This is because DB2, like most other commercial databases, commits transactions very efficiently – a log flush is the only I/O operation that is performed at commit time.

7.1 Workloads and Test Data

We compared the update methods on both *bulk* and *random* workloads. With a bulk workload, an update operation was applied to every subtree element of the synthetic input document. Thus, a bulk delete would leave only the root element deleting everything else.

To simulate a random workload, we applied an update operation to 10 randomly chosen subtrees. Clearly, the performance of an efficient update operation on a random workload should not depend significantly on the size of the input document.

As described below, we used three sets of test data for our experiments. In the first set, the data is synthesized and the document structure is fixed. In the second set, the data is still synthetic but the document structure is randomized. For the third set of experiments, we used real-life data from the DBLP bibliography database.

7.1.1 Fixed Synthetic Data

As a first step, we compared the performance of the update methods on a set of relatively simple synthetic documents. Each test document was generated based on the following three parameters:

- *Scaling factor* specifies the number of subelements (subtrees) at the root level. Intuitively, scaling factor denotes the length of an XML document.
- *Depth* determines the number of levels in each subtree, and affects the complexity of a synthetic XML document.
- *Fanout* specifies the number of child nodes (subelements) in the internal nodes of a subtree, also a measure of document complexity.

Since the space of possible parameter combinations is huge, we explored the space by alternately holding one parameter constant while varying the other two, using the values summarized in Table 1. To simulate content, each element in the synthetic XML documents contains two data subelements: a 50 character string and an integer.

7.1.2 Randomized Synthetic Data

In this set of experiments, the input document's subtrees do not have a fixed structure, and this necessitates changes to the document parameters. As before, scaling factor measures the number of subtrees at the root level. Depth now specifies the maximum depth of a subtree, so the actual depth of each subtree is a random number between the minimum depth, two, and this maximum depth. Similarly, the fanout at each node is a random number between one and the specified maximum fanout.

7.1.3 Real-Life Data

In order to verify our results, we augmented our synthetic data with the conference publications portion of the DBLP bibliography. The data was organized as an XML document in which upper-most elements correspond to conferences. Each conference element has publication subelements which contain author and citation subelements, among other data. The size of the resulting document is 40MB, which produced more than 400,000 database tuples.

7.2 Effect of Access Support Relations on Path-Expression Evaluation

Although pure query evaluation is not the focus of this study, ASRs affect both query and update performance so

Experiment	fixed param	variable params	max data size	data size growth
fixed fanout (f)	fanout=1	d=2,4,8 sf=100,200,400,800	6400 tuples (0.8MB)	linear in depth and sf
fixed depth (d)	depth=2	f=1,2,4,8 sf=100,200,400,800	7200 tuples (0.7MB)	linear in fanout and sf
fixed scaling factor (sf)	sf=100	d=2,3,4,5 f=2,4,8	58500 tuples (7MB)	exponential in depth

Table 1: Parameter values which were evaluated using synthetic data

we started by testing the effectiveness of ASRs for path-expression evaluation. Surprisingly, we found that ASRs improve performance only on documents with small fanout. For example, on a synthetic document with a fanout of 4, a query containing a path expression of length 3 ran two times slower when an ASR was used. If the path length is increased to 4, the conventional method (4 data relation joins) has the same performance as the ASR method, which does only two joins. If the path expressions are even longer, then ASRs do help.

We believe that our ASR implementation failed to improve path-expression performance for the following reasons. First, the DBMS which we used does not expose internal row identifiers that could serve as physical addresses. With no physical row ids, we had to rely on generated tuple ids to link ASRs and data relations and resort to index joins instead of physical references. Second, with larger fanouts, the ASR relation quickly becomes very large, since it contains a tuple for each full path in the XML tree. As a result, it is often more efficient to perform a multi-way join on the original data relations than to join the path's start and end relations with the ASR. Third, in many cases it appears that the DBMS optimizer made poor choices, which exacerbated the first two issues.

7.3 Performance of Delete Methods

Our next experiment compares four methods of performing deletes: per-tuple trigger, per-statement trigger, cascading delete, and ASR-based delete. In general, the trigger-based delete methods performed best. As we expected, the cascading delete method performed much like the per-statement trigger-based delete; the former simulates the latter, but does it at the application level rather than inside the DBMS. Since the difference in performance of these methods was almost negligible, less than 5%, we omit cascading delete from our graphs.

Figures 6 and 7 compare the performance of the delete methods on a bulk and random workloads. We show data for the case where fanout was fixed at one and depth set to eight. Note that running time on the bulk workload is frequently less than that for the random workload, because only one SQL statement is issued for a bulk delete, while 10 SQL statements are necessary for the random workload, one per deleted subtree.

On the bulk workload, per-statement triggers perform noticeably better than per-tuple triggers. Since the entire contents of relations are deleted, executing deletes on a per-statement (or equivalently per-relation) basis is more efficient than doing that on a per-tuple basis.

On the random workload however, per-tuple triggers perform much better than per-statement triggers. More importantly, with per-tuple triggers, performance does not degrade as the data set increases — the per-tuple line is virtually flat, which is appropriate since we are not varying the amount of deleted content. Per-statement triggers, on the other hand, slow significantly as the document size (scaling factor) is increased.

A quick review of the trigger mechanism (Section 6) yields an explanation for this phenomenon. When a tuple from a parent relation is deleted, a per-tuple trigger uses the id of the deleted tuple to look up newly orphaned tuples in child relations. The number of such lookups equals the number of deleted tuples. The size of the document does not directly impact per-tuple triggers. A per-statement trigger, on the other hand, gets invoked when all relevant tuples from a parent relation have been deleted. The trigger then deletes all orphan tuples from child relations, *i.e.*, those tuples for which a parent tuple cannot be found. Since this delete operation involves a scan of entire child relations (or their indexes on the `id` attribute), the time required increases with the overall size of the data.

Figures 8 and 9 show the running times of the delete methods on documents with scaling factor fixed at 100 and fanout equal to four. Since linear growth in the independent variable, depth, results in exponentially larger documents, we use a logarithmic y axis. On the bulk workload, the trigger-based methods clearly outperform the ASR-based approach. As before, the per-tuple trigger method performs best on the random workload. Per-statement trigger delete is slow because it involves a full index scan for each relation involved.

The results on randomized synthetic data are similar to those shown above, and are omitted. The per-tuple trigger-based delete was again a clear winner on random workloads, and it performed slightly below per-statement trigger delete on bulk workloads.

DBLP results are shown in Table 7.2. For our experiments, we used a query that deletes publications that appeared in the year 2000. Once again, the per-tuple trigger delete method performed better than the other methods. The per-statement trigger and cascading delete methods did not perform well because DBLP data is very "bushy" and only a small portion of the document was deleted; as discussed above, these methods do not perform well on random workloads.

7.4 Inserts

To measure insert performance, we sought to simulate a query that copies complex XML elements from within a single document.[2] To this effect, we used a simple query that replicates ten (random workload) or all (bulk workload) subtrees of the root element. The results of our experiments are shown in Figures 10 and 11.

Clearly, the table method outperforms the other methods for bulk inserts. With random inserts, if the amount of of copied data is small, the tuple method is preferable since it does not have the overhead of the other methods. If many tuples need to be copied, i.e. the source elements are deeper, the tuple method does not perform as well and the table method performs better. This can be explained by the large number of SQL operations (one per source tuple), that the tuple method has to execute.

[2]Copying data into a different document with the same DTD is similar.

operation / method:	per-tuple trigger	per-stm trigger	cascade	ASR	table	tuple
delete running time	1.6	4.6	4.8	2.2	-	-
insert running time	-	-	-	4.2	1.7	15.4

Table 2: Experimental results on DBLP data.

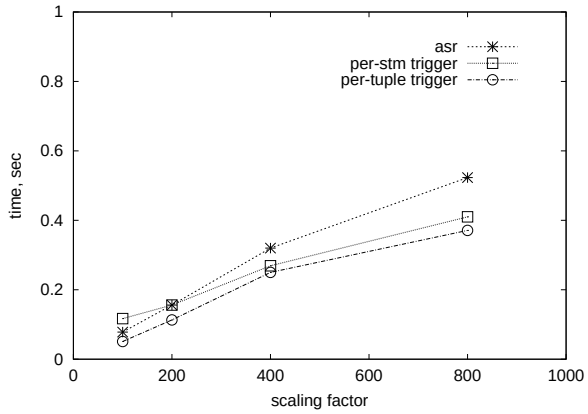

Figure 6: Delete performance on *bulk* workload, fixed fanout=1, depth=8.

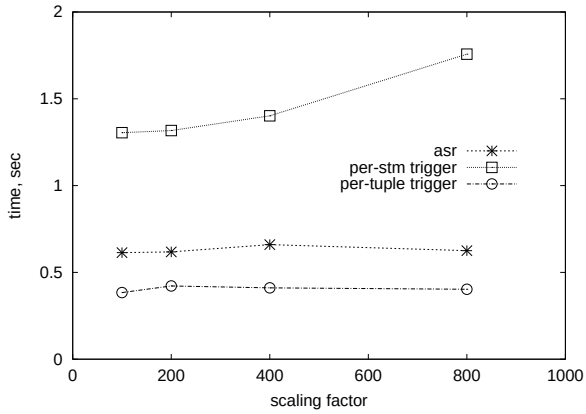

Figure 7: Delete performance on *random* workload, fixed fanout=1, depth=8.

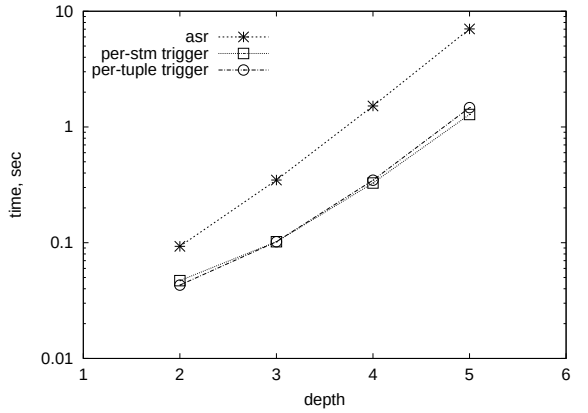

Figure 8: Delete performance on *bulk* workload, fixed scaling factor=100, fanout=4.

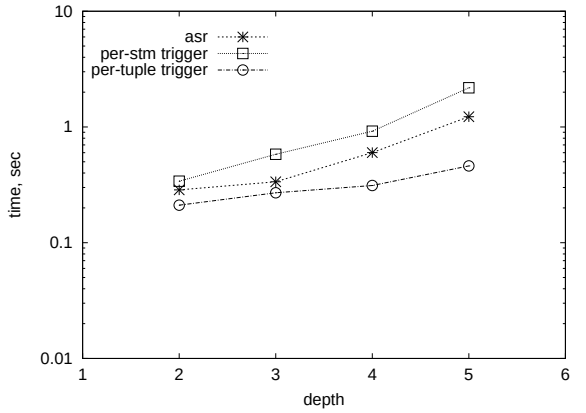

Figure 9: Delete performance on *random* workload, fixed scaling factor=100, fanout=4.

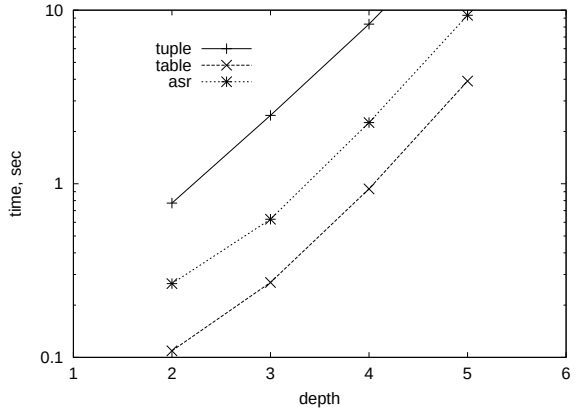

Figure 10: Insert performance, *bulk* workload, fixed scaling factor=100, fanout=4.

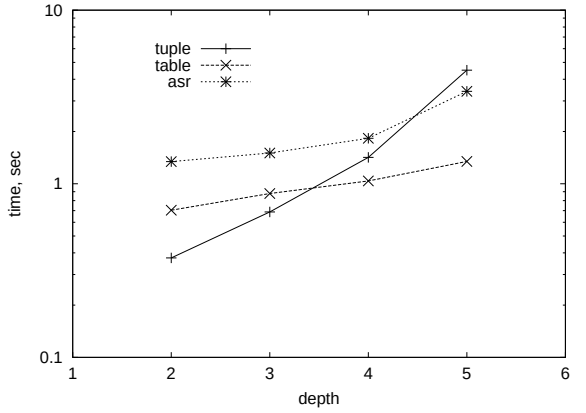

Figure 11: Insert performance, *random* workload, fixed scaling factor=100, fanout=4.

7.5 Other Update Operations

As explained in Section 6.3, our other update operations can be generally implemented using sequences of inserts, deletes, and queries. Thus we do not separately show the performance of different strategies for these operations; their best implementations make use of the fastest delete method (a per-tuple trigger-based delete) and the fastest insert method (a table-based insert).

8. CONCLUSIONS AND FUTURE WORK

Though the XML data management community has recently been focused on issues related to querying XML, it is clear that the problem of expressing updates over XML data will become prominent in the near future. An XML update language provides a general-purpose way to express changes to any data that can be represented as XML — whether the data is actually stored within an XML repository or a relational database with an XML view. We have proposed a set of primitive XML update operations that can be composed to express modifications at multiple levels within an XML document — reflecting the hierarchical structure of XML and its query languages — and have incorporated our operations as a set of language extensions to XQuery.

Furthermore, we have taken our language extensions and implemented them in the most widely researched means of storing data presented as XML — the relational database. We have compared numerous approaches for implementing the core operations (insert and delete), and have determined which work best. For the most part, per-tuple trigger-based deletes dominated other methods and that table-based insert method was the fastest.

While we feel we have made a fairly comprehensive study of XML updates, there are several potential areas for future work. The topic of typechecking updates is an important one, and we plan to investigate whether it is possible to directly use the techniques developed for queries. For our update language and its implementation, we would like to be able to efficiently provide a deterministic result if a particular XML update target is reachable more than once in a path expression (*e.g.*, it is the target of more than one reference). In terms of our XML repository implementation, we would like to extend our relational techniques to support the preservation of order within the XML document. In a query-only repository, this is traditionally done by storing each element along with its child index or position; the results can then be sorted into their original order. Since updates can insert new content between existing data, we encounter a problem of "pushing" the position of the old data forward to accommodate the insertion.

Our final assessment is that updates to an XML document can be expressed in a concise and natural way, even with support for ordering. We have shown that our set of basic constructs can be efficiently implemented over a relational database, with a number of challenges not encountered by a query-only XML storage manager.

ACKNOWLEDGEMENTS

We would like to thank Dan Suciu for his helpful suggestions regarding our update extensions, and Jayant Madhavan and the anonymous reviewers for their insightful comments on the paper. Halevy was supported by a Sloan Fellowship, NSF Grants IIS-9978567 and IIS-9985114, and gifts from Microsoft Research, NEC and NTT. Weld was supported by NSF Grant DL-9874759 and Office of Naval Research grant N00014-98-10147. Ives was supported by an IBM Research Fellowship.

REFERENCES

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Winer. The Lorel query language for semistructured data. In *Proceedings of International Journal on Digital Libraries*, volume 1(1), pages 68–88, April 1997.

[2] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing object-relational data as XML. In *ACM SIGMOD WebDB Workshop '00*, 2000.

[3] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanascu. XQuery: A query language for XML. Technical report, World Wide Web Consortium, February 2001. Available from http://wwww.w3.org/TR/xquery/.

[4] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD '00*, 2000.

[5] J. Clark and S. DeRose. XML path language (XPath) recommendation. http://www.w3.org/TR/1999/REC-xpath-19991116, November 1999.

[6] A. Deutsch, M. F. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Eighth International World Wide Web Conference*, 1999.

[7] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD '99*, pages 431–442, 1999.

[8] M. Fernandez and J. Robie. XML query data model, W3C working draft 11 may 2000. Technical report, World Wide Web Consortium, May 2000. Available at www.w3.org/TR/2000/WD-query-datamodel-20000511.

[9] M. Fernandez, W.-C. Tan, and D. Suciu. SilkRoute: Trading between relations and XML. In *Ninth International World Wide Web Conference*, November 1999.

[10] D. Florescu and D. Kossman. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report 3684, INRIA, March 1999.

[11] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *ACM SIGMOD WebDB Workshop '99*, pages 25–30, 1999.

[12] A. Kemper and G. Moerkotte. Access support in object bases. In *SIGMOD '90*, pages 364–374, 1990.

[13] B. Mitschang, H. Pirahesh, P. Pistor, B. G. Lindsay, and N. Südkamp. Sql/xnf - processing composite objects as abstractions over relational data. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 272–282, 1993.

[14] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB '99*, pages 302–304, 1999.

[15] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. R. B. Lindsay, and H. Pirahesh. Efficiently publishing relational data as XML documents. In *VLDB '00*, 2000.

[16] Tamino: The information server for electronic business. http://www.softwareag.com/tamino.

[17] Updating XML data. eXcelon 1.1 User Guide, Chapter 7, 1998.

[18] eXcelon: The XML application development environment. http://www.odi.com/excelon/main.htm.

[19] XML Schema part 1: Structures. http://www.w3.org/TR/1999/WD-xmlschema-1-19991217/, 17 December 1999. W3C Working Draft.