Full-Fidelity Flexible Object-Oriented XML Access

James F. Terwilliger
Microsoft Research, USA
James.Terwilliger@microsoft.com

Philip A. Bernstein
Microsoft Research, USA
Phil.Bernstein@microsoft.com

Sergey Melnik
Google Inc.
melnik@google.com

ABSTRACT

Developers need to programmatically access persistent XML data. Object-oriented access is often the preferred method. Translating XML data into objects or vice-versa is a hard problem due to the data model mismatch and the difficulty of query translation. We propose a framework that addresses this problem by transforming object-based queries and updates into queries and updates on XML using flexible, declarative mappings between classes and XML schema types. The same mappings are used to shred XML fragments from query results into client-side objects. Information in the XML store that is not mapped using the mapping language, such as comments and processing instructions, are also made available in the object representation.

1. INTRODUCTION

XML has become ubiquitous in data-centric applications, especially for ones that involve data exchange. Most of these applications need to store the XML that they receive and send. To support these applications, many commercial database systems offer XML storage, which enables an XML document to be stored as a single column value. By storing the original XML document in the database, an application retains a perfectly accurate copy of the original XML and enables users to query parts of the document that are not shredded into relational columns.

The importance of offering an object-oriented data access layer to databases is well known — to assemble objects that are spread across many tables and data models and to augment the data with business logic. All large-scale database applications that we know of include such a layer. Several object-relational mapping (ORM) frameworks [3] have emerged to help application developers bridge objects and relations. They leverage the performance and scalability of databases by translating queries on objects into equivalent queries in SQL. ORM tools initially used fixed mappings, which were insufficiently flexible for most applications and hence not widely used. Recent tools that are driven by flexible mappings have become more popular [1, 6].

Mapping flexibility is important because application interfaces and database schemas can differ substantially and for good reasons: applications exploit inheritance to improve encapsulation and code reuse, whereas database schema design emphasizes performance, normalization, replication, access control, etc.

Typically, ORMs do not handle the mismatch between objects and XML. Translating between XML and objects automatically is a largely unsolved problem, due to differences in the expressive power of their type systems [9] and the difficulty of translating object queries into an XML query language such as XQuery.

Currently, developers have two options for accessing XML stored in a relational database. One option is to use XQuery to access XML from an imperative language (object-oriented or otherwise) by sending a query as a quoted string to a function, such as Execute("for $a in doc('b.xml') return ..."). This approach is brittle and complex since it offers no static type-checking and requires diving into XQuery, an unfamiliar territory for many developers. The alternative is to bring the entire XML document to the client and shred or wrap it into objects client-side [5, 8]. For large documents or large document collections, this approach can be prohibitively expensive.

To address this problem, we developed an object-oriented data access layer that exposes both XML and relational data as objects of user-defined (custom) classes. Initial feedback we got from application developers suggests that such a layer can provide up to an order-of-magnitude increase in developer productivity. Using an object-oriented programming language to access classes that encapsulate XML and relational data brings several essential advantages. First, developers get the type-checking benefits of a custom-class interface to their XML data. Second, they get an object-oriented query language called LINQ to access XML and relational data in a uniform fashion. LINQ is the new language-integrated query mechanism in .NET [12]. LINQ queries resemble SQL, but are statically compiled and type-checked in the developer's object-oriented programming language, such as C# or Visual Basic. Finally, the integrated development environment can use type information to offer auto-completion functionality for class and member names in both imperative code and queries.

We propose an architecture called LRX (LINQ over Relations and XML) for implementing an object-oriented data access layer to SQL plus XML. It has several new features. The first is a flexible mapping language to the underlying SQL and XML data.

To offer object-oriented access to XML data via custom classes, we need a schema that describes the XML parts of the database. We use the XML Schema (XSD) standard for this purpose. XSD enables the specification of structural features of XML data and constraints that XML data is required to satisfy. It is currently supported by Oracle, IBM DB2, and Microsoft SQL Server. We use the SQL and XSD schemas as the target of the object-to-database mappings. In addition, we generate customizable classes from the SQL and XSD schemas and customizable mappings between the classes and the schemas.

With these classes and mappings in place, LRX translates queries and updates expressed in LINQ on the client into SQL and XQuery queries and updates on the server. Using these mappings, LRX can also retrieve XML fragments as part of a query result and materialize client objects in a way that respects the mappings.

VLDB '09, August 24-28, 2009, Lyon, France.

One problem that is intrinsic in any schema-based approach to XML mapping is potential data loss. Data loss arises because XML documents may contain information that is not described in the document's schema, such as comments or extra attributes. Because this information is invisible to the document's schema, it is invisible to any schema-based mapping approach.

We say that an object-oriented representation of XML data has *full fidelity* if it contains all of the information necessary to reconstruct the original document. To address the problem of lost data, LRX includes a *delta representation* that describes all information that cannot be exposed through the mapping or is not captured in a given XML schema. Having a full-fidelity object representation means that developers have client-side access to all data in the original XML, and that the original document can be reconstructed in a lossless fashion.

The primary contributions of this paper are as follows:

1. A mechanism to translate object queries expressed in LINQ against custom-typed objects (and XPath expressions against generically typed, XML-like objects) into queries in SQL and XQuery to push query execution to the server.
2. A flexible mapping language that allows choosing a variety of object representations for XML.
3. A full-fidelity object-oriented representation that presents to the developer information that is in the XML document but is not exposed as instances of the custom-typed objects.
4. A performance analysis demonstrating the performance gain achieved by pushing query evaluation to the server.

The rest of this paper proceeds as follows. Section 2 lays out motivating examples to illustrate the process of pushing queries expressed in LINQ to be executed on the database. We briefly describe the components of LRX in Section 3. In Section 4, we formalize our mapping language and discuss query translation. Section 5 describes how LRX allows a developer to interact with information that does not participate in the XML-to-object mappings. We discuss update translation in Section 6. Our performance results are introduced in Section 7. Section 8 provides an analysis of related work, and Section 9 concludes the paper with some final analysis.

2. MOTIVATING EXAMPLES

Our running example is based on the XMark benchmark [16]. We modified the schema and queries of the auction site example by partially shredding it, thereby creating a hybrid relational and XML data source. Figure 1 shows the database schema used in our motivating examples and subsequent performance analysis. This database contains several tables whose columns store XML data. For example, the table `Items` has a column Item whose content in each row is an XML document.

We call a part of an XSD schema *custom-mapped* if it has a direct correspondence as a custom class in the object-oriented type definitions. An XML fragment is custom-mapped if it is an instance of a custom-mapped part of the document's XSD schema.

2.1 XML as Custom-Typed Objects

Consider the following code fragment written in C#, which uses LINQ to list the name of the item with ID "item20748" registered in North America:

```
using (AuctionDB db = new AuctionDB()) {
    var q = from i in db.Items
            where i.ID == "item20748" &&
                  i.Region == "North America"
            select i.item.Name;
    foreach (var o in q)
        Console.WriteLine(o); }
```

This query is an object version of Q1 from the XMark benchmark. The "var q" declaration indicates that the return type of the query is inferred by the compiler (as a collection of strings, in this case). LRX translates the query into the following SQL and XQuery:

```
WITH XMLNAMESPACES('http://.../Auction' AS a)
SELECT VALUE(I.item, '*[1]/a:Name', varchar)
FROM Items I
WHERE I.region = 'North America'
  AND I.id = 'item20478'
```

The function "VALUE" is a placeholder for a DBMS-specific XML feature. The VALUE function executes a query expressed in XQuery and casts the resulting atomic value into a relational data type. This function exists as the XMLCAST function in Oracle, and the `.value` method in SQL Server.

Although in this example the query could be expressed almost entirely in XQuery, our translation algorithm uses the relational operators whenever possible to leverage the relational capabilities of the query processor and to support queries that span both relational and XML data.

2.2 XML as Generically Typed Objects

Not all XML schema types can be mapped to custom types. For instance, the type of an XML element may be declared as "xsd:anyType", which cannot have a statically typed object counterpart any more descriptive than "any XML data". Also, mixed-content elements are hard to map to custom-typed objects due to text nodes that may be sprinkled between child elements. Finally, the developer may prefer to query persistent XML directly using XPath.

LRX supports these scenarios by mapping XML schema elements to the .NET type `XElement`. For example, an XML schema element "Text" that has mixed content is mapped to a class member of type `XElement`. Each XPath axis has a counterpart in the object layer as a method of the class `XElement`.

The following LINQ query illustrates how one can use custom and generically typed portions of the query in a single expression:

```
var q = from c in db.ClosedAuctions
        where c.closed_auction.price > 20
        from t in c.closed_auction
                .annotation.description.text
                .Descendants("emph")
        select t;
```

The query above selects all of the emphasis nodes in the descriptions of auctions that closed for more than 20 dollars. This query is translated into the following SQL/XQuery expression:

```
WITH XMLNAMESPACES('http://.../Auction' AS a)
SELECT T
FROM (SELECT QUERY(I, '*/a:Description//a:bold')
FROM ClosedAuctions C,
    SEQUENCE(C.closed_auction,
    '*[1]/annotation/descriptiontext//emph') AS T)
WHERE VALUE(C.closed_auction, '*[1]/price',
      int) > 20
```

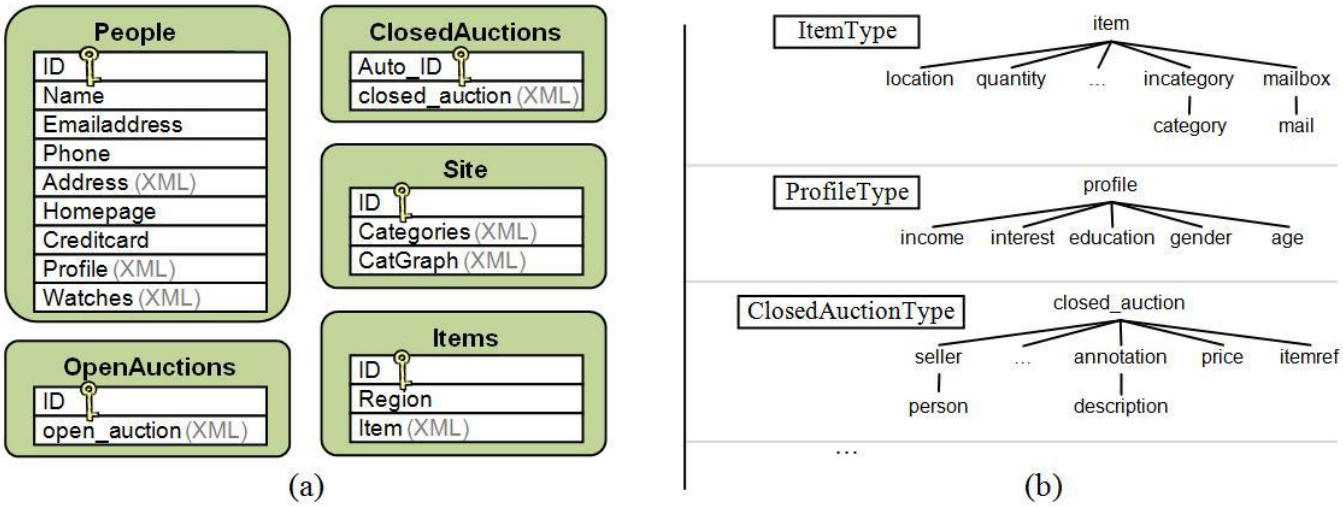

Figure 1. The schema of our test data, both relational (a) and XML (b)

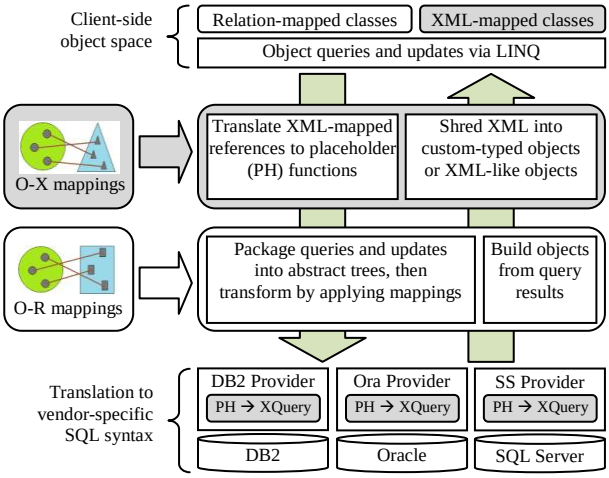

Figure 2. An overview of the LRX architecture

The XPath accessor Descendants() is passed through and becomes the abbreviated descendant axis "//" in the final query. The SEQUENCE function (corresponding to the XMLSEQUENCE function in Oracle and the `.nodes` method in SQL Server) takes a list of XML elements and returns one row per element.

3. LRX Architecture

XML stored in a relational database typically resides in relational columns. Therefore, we propose an architecture that plugs into existing relational data access frameworks and can leverage its object-relational mapping capabilities and database connection APIs. We add new components to the data access framework in several ways to support handling of XML data, outlined in Figure 2. In the figure, all of the components in white come from the existing data access framework, and objects in gray are components that are new to the architecture.

To validate our proposed architecture, we implemented it as extension to the Microsoft Entity Framework (EF) [1]. EF already supports mappings between object classes and relations; LRX adds support for mappings between object classes and types defined in XML Schema. These two sets of classes are disjoint, so a class can be mapped to either a relation or an XML type but not both. Though the developer can write a mapping manually, we have a tool that generates a default class for each XML type and a default object-XML mapping (which can be altered or rewritten by the developer).

The mapping engine is responsible for several functions. The first function of the engine is to handle references to XML-mapped objects in queries or updates. All references to XML-mapped objects are packaged into function calls that reference the relational objects in the query; therefore, the resulting query can be operated upon by the relational engine in the data access framework without further alteration to the engine. We call these function calls *placeholder functions* because they take the place of the object references that the relational engine cannot handle.

The placeholder functions that LRX supports are:

- VALUE(*c, q, t*), which executes the XQuery expression *q* against the XML-type relational column *c* and, expecting the query to return an atomic value, casts the answer into an object of type *t*.
- QUERY(*c, q*), which executes the XQuery expression *q* against the XML-type relational column *c* and returns the result as a single XML fragment.
- SEQUENCE(*c, q*), which executes the XQuery expression *q* against the XML-type relational column *c* and returns one row for each XML element returned by the query.
- TEST(*c, q*), which executes the XQuery expression q against the XML-type relational column c and returns a Boolean value, true if the query returns any data and false if result is empty.
- XINSERT(*c, q, m, n*), which inserts the nodes *n* into the documents in column *c* at the position indicated by XPath expression *q*. The mode parameter *m* is one of "before", "after", "as first", or "as last", indicating the position of the new nodes relative to the result of the XPath expression.

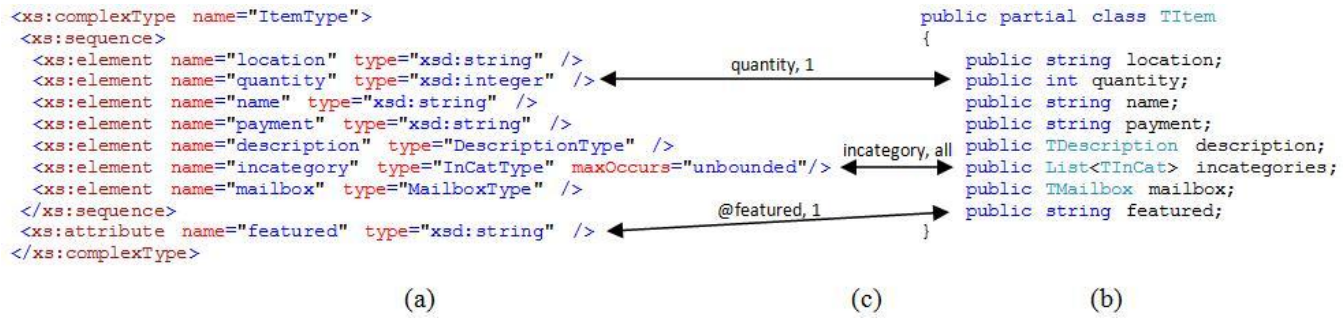

Figure 3. A relational to XML mapping using LRX. A schema element in an XML type (a) is associated with a member in a client class (b) using a Component Designator expression and a position reference (c)

- XUPDATE(*c*, *q*, *d*), which replaces the data in the nodes returned by the XPath expression *q* against the documents in column *c* with the data *d*.
- XDELETE(*c*, *q*), which deletes the nodes returned by the XPath expression *q* from the documents in column *c*.

Once packaged into placeholder functions, the remaining query or update can be processed as usual by the relational mapping engine of the data access framework.

The second function handled by the mapping engine is object materialization. Queries may return XML fragments that must be deserialized into objects in client space. The mapping engine generates an automaton based on the XML Schema and the type mappings that handles the shredding of XML fragments into objects in a type-and-mapping-specific, efficient way.

When the query or update has successfully been passed through the relation components of the data access framework, it is processed by a database vendor-specific provider that translates the statement or statements into vendor-specific SQL syntax. Each database provider can be extended to look for our placeholder functions and translate them into the appropriate dialect of XML functions for that system; we extended the SQL Server provider to translate placeholder functions into XML functions supported by SQL Server 2008.

4. QUERY TRANSLATION

Before we can have object-oriented query and update access to stored XML data, we need to map the XML types in schematized XML database columns to classes. Figure 3 shows the mapping that associates type ItemType from the XMark schema with a C# class TItem, part of the mapping required for the sample queries in Section 2. Each mapping is specified using Component Designator (CD) expressions, one for each class member.

EF uses a mapping compilation approach; a developer specifies an object-relational mapping, which EF then compiles into views that it uses to process queries and updates [13]. LRX uses a similar approach. For each member of an XML-mapped class, LRX compiles a mapping fragment associated with the member into an XQuery expression that represents the query for all elements that match the CD expression.

4.1 XSD Component Designators

XSD Component Designators (CD) [20] is an XPath-like language that is used to refer to individual elements within an XML Schema. Component Designator expressions can be either absolute paths, which describe the location of a schema element relative to the schema document root, or relative paths, that describe the location of a schema element relative to some current location. In LRX, we use CD absolute paths to describe the location of types, and within each type, we use relative paths to describe the location of schema elements that map to members.

For example, consider the following Component Designator relative path expression:

```
model::choice[3]/schemaElement::ns:Y[2]
```

The expression above references the third choice particle in the current type definition, and within that particle, references the second element declaration with the name "ns:Y".

Just like XPath, Component Designators have an abbreviated syntax. For expressions that only reference `schemaElement` axes, one can use a very XPath-like expression like the following:

```
ns:X[2]/ns:Y
```

This expression references the second "ns:X" element reference in a type, then references the first "ns:Y" element of the type of element associated with "ns:X". If there is only one matching element in a type, then one can omit the position reference (as in the example above, where "ns:Y" refers to the first and only "ns:Y" element in a type).

Because of the Unique Particle Attribution requirement of XML Schema, each element of an XML document validates against exactly one particle in an XML schema. Since each CD path represents a particle in an XML schema, a CD expression has a clear semantics as a collection of XML nodes. We can think of a CD path and the set of elements that validates against the referenced particle interchangeably.

4.2 Mapping Formalism

CD expressions only constitute part of a mapping expression. For instance, mappings may optionally specify conditions that must be met on either the XML or object side or both, allowing a single XML schema type to be conditionally mapped to different classes, and vice versa.

A *mapping fragment* is a triple (*m*, *d*, *i*), where *m* is a class member reference (e.g., a field or a property), *d* is a CD path expression, and *i* is a position designator. The position designator *i* can be either a positive integer (designating a position reference within a collection of elements, where "1" (one) is the first element in the collection) or the special value "all", indicating that the mapping refers to all of the elements in the collection.

A *type mapping* is a 5-tuple (*c, cp, x, xp, f*), where *c* is an object class, *cp* is a (possibly empty) list of predicates on the class *c*, *x* is an XML Schema type, *xp* is a list of predicates on *x*, and *f* is a list of mapping fragments. Predicates in *cp* (on objects that belong to class c) may be equality conditions on the values of members of the object or conjunctions of such expressions. Predicates in *xp* (on elements that satisfy XML type *x*) may be equality conditions on the child element values or the name of the element, conditions on the existence of child elements, or conjunctions of the above.

A type mapping (*c, cp, x, xp, f*) is valid if it meets a non-redundancy condition: For any two mapping fragments *f1* and *f2* in *f*, *f1.d* (the component descriptor for fragment *f1*) is not a prefix of *f2.d*. This condition eliminates the possibility that a single XML element can map to multiple locations in object space.

A *schema mapping* is a triple (*cs, xs, ts*), where *cs* is a client-side object schema, *xs* is an XML Schema instance, and *ts* is a collection of *valid* type mappings. A schema mapping (*cs, xs, ts*) is *valid* if it meets all of the following conditions:

- For each type mapping (*c, cp, x, xp, f*) in *ts*, class *c* is a valid class in *cs*, *x* is a valid type in *xs*. Also, for each fragment (*m, d, i*) in *f*, *m* is a valid member of *c* relative to schema *cs*, *d* is a valid CD expression for *x* relative to schema *xs*, and *m* does not participate in any of the equality predicates in *cp*.
- For each mapping fragment (*m, d, i*) in each type mapping, if the type of member *m* is scalar, then position value *i* is integral. If *m* has a collection type, then *i* is the special value "all".
- For each mapping fragment (*m, d, i*) in each type mapping, if the type of member *m* is atomic (i.e., a numeric or string type) or is a collection of atomic items, then the element type referred to by CD expression *d* is also atomic. If *m* is not atomic, then the element type referred to by expression d is also not atomic.
- Given any two type mappings *t1* and *t2* in *ts*, if *t1.x* is a subtype of *t2.x*, then either *t1.c* = *t2.c* or *t1.c* is a subclass of *t2.c*. This condition prevents type conflicts in object space. If an XML element has an associated `xsi:type` attribute saying it has type *t1.x*, it will become an object of type *t1.c*, which can still be assigned to variables of type *t2.c*. Similarly, if *t1.c* is a subclass of *t2.c*, then either *t1.x* = *t2.x* or *t1.x* is a subtype of *t2.x*.
- Given any two type mappings *t1* and *t2* in *ts*, if *t1.x* = *t2.x* (both mappings refer to the same XML type) and *t1.xp*∧*t2.xp* ≡ *t1.xp* (*t1.xp* is a stronger condition than *t2.xp*), then either *t1.c* = *t2.c* or *t1.c* is a subclass of *t2.c* (for the same reason as the previous condition). Similarly, if *t1.c* = *t2.c* and *t1.cp*∧*t2.cp* ≡ *t1.cp*, then either *t1.x* = *t2.x* or *t1.x* is a subtype of *t2.x*.
- Given any two type mappings *t1* and *t2* in *ts*, if *t1.x*=*t2.x*, then exactly one of the following conditions must be true:
  - *t1.xp* ∧ *t2.xp* = *t1.xp* (*t1.xp* is strictly more selective)
  - *t1.xp* ∧ *t2.xp* = *t2.xp* (*t2.xp* is strictly more selective)
  - *t1.xp* ∧ *t2.xp* = *false* (*t1.xp* and *t2.xp* have no overlap)

  Similarly, if *t1.c*=*t2.c* for two type mappings *t1* and *t2*, then *t1.cp* must either be strictly more selective than *t2.cp*, strictly less selective than *t2.cp*, or have no overlap with *t2.cp*.
- Finally, for every class *c* that participates in a type mapping, there must exist some type mapping *t* such that *t.cp* = *true*. This type mapping is called the *base mapping* for *c*.

An XML element *e matches* a type mapping *t* if *e* has type *t.x* and *e* satisfies the predicate *t.xp*. The element *e maximally matches t* if no other type mapping matches *e*. Because of the properties of a valid schema mapping, any time an element matches a type mapping, there must exist a type mapping that maximally matches the element. The definition for an object *o* matching or maximally matching a type mapping *t* follows from the same logic.

A schema mapping *s* is *complete* if, for any XML element *e* that conforms to a type definition in *s.xs*, there is a type mapping that matches *e*, and for any object that conforms to a class in *s.cs*, there is a type mapping that matches *o*. LRX checks mapping validity, but not completeness. Practically-speaking, the developer may not write type mappings for every contingency if they know *a priori* the characteristics of the objects or XML elements.

4.3 Mapping Compilation

There are two cases to consider when compiling a CD path expression into XQuery, depending on whether the XSD type in a mapping is *simple*. We call an XSD type *simple* if its type definition has no repeated element definitions and no sequence, all, choice, or group particles with a `maxOccurs` value of greater than one. The regular expression for the validation of elements of a simple type has no repeated symbols unless they are adjacent to each other (to allow constructs like `maxOccurs=3` for an element) and no Kleene star symbols except on a single element. One special case that we allow in the simple case is a disjunction of permutations (e.g., (abc|acb|cab |cba|bac|bca)), which accommodates an "all" particle. So, a type with regular expression "ab*ccc?c?(d|e)" is simple, but "aba" or "a(bc)*" are not.

Simple types have the property that queries that retrieve all elements corresponding to a CD expression can be done in XPath. For example, consider the CD expression "a/b", which refers to the "a" schema element in a type definition, and the "b" schema element of the type associated with "a". To retrieve all matching elements from an instance of the parent type, one executes the XPath expression "/a/b". A simple type has the property that component designators and XPath expressions are interchangeable. The only exception is CD expressions that reference structural particles, such as "model::choice[2]/schemaElement::foo"; for a simple type, we can drop the axis reference for the particle, leaving the XPath expression "/foo".

For types that are not simple, retrieving all of the elements that match a particular CD expression is more complex, as it is not interchangeable with XPath. Such compiled expressions generally require finding elements by relative location to other elements (e.g., /A[. >> ../B[3]], or "select all A elements after the third B"). In more complex cases, the compiled expressions are too expressive for XPath and require full XQuery. This problem is significant and difficult on its own. We have developed an algorithm that can compile CD expressions for non-simple types in most cases, and are working on incorporating it into our prototype.

4.4 Member Translation Algorithm

As mentioned in Section 3, LRX packages references to XML-mapped objects into placeholder functions that are then tunneled through the existing object-relational mapping functionality. For queries, LRX walks the tree that represents the LINQ query and translates any references one at a time. Therefore, when processing the statement `foo.bar.att`, the algorithm translates the member reference `foo.bar`, then the member reference `.att`.

LRX also performs some processing on conditions in the WHERE clause of a query. Because LRX type mappings can have client-side predicates, if those predicates appear in a query, LRX translates them into conditions on XML elements and types according to the XML side of the associated mapping.

The ExpressionTranslate algorithm in Figure 4 outlines LRX query translation. For brevity, we assume in our exposition that the conditions in the WHERE clause of the query are in disjunctive normal form. The algorithm uses the following additional terminology:

Q1∘Q2: The result of composing query Q1 with query Q2.

PH(E): Represents the placeholder function that results from running ExpressionTranslate on E. If P is the result, P.function refers to the function name, P.column refers to the function's column argument, and P.query refers to the query argument.

COND(P, M, V): A temporary placeholder function, representing that the condition "P.M = V" was found in the query, and that M participates in a client-side condition of a type mapping. The first phase of ExpressionTranslate (member translation) creates instances of COND. The second phase (condition translation) packages them up to identify a non-base type mapping for the type of P, which translates to server-side XQuery conditions.

There are two helper algorithms provided in Figure 4. The first, PropertyTranslate, finds the appropriate compiled mapping fragment for a dereferenced member and composes it with the query in the current placeholder function (and creates instances of the COND temporary function). The second helper function, MethodTranslate, translates some method calls into XQuery equivalents. LRX recognizes three kinds of methods:

- Methods on atomic types that have clear XQuery analogs. For instance, the expression s.Contains("foo") maps to the XQuery expression "contains(q, 'foo')" if s translates to query q. Computed properties like string length belong in this category. LRX includes a catalog of such method mappings.
- The OfType<T> method, which filters a collection of objects, returning only those of type T.
- XPath-like methods for the class XElement, described in Section 2.2. For instance, the methods .Attribute(X), .Elements(Y), and .Descendants(Z) map to the XPath expressions "/@X", "/Y", and "//Z" respectively.

LRX only supports literal arguments to translated methods. For simplicity of exposition, we treat the variable $method in the algorithm description as the method call with its arguments.

4.5 Object Materialization

Consider the following query:

```
var q = from r in db.Items select r.Item;
```

This query returns a list of objects of type "TItem". On the database side, the query returns a list of XML fragments of type "ItemType". The object materializer component shreds each XML fragment in the query result into an object of type TItem according to the XML fragment's maximally matched type mapping. If the type mapping includes equality predicates on the client-side class, the generated object fills in the appropriate values in the object to satisfy those predicates. Section 5.2 describes in more detail how object materialization occurs.

```
ExpressionTranslate($query):
  For each member reference $parent.$member in $query:
    If $parent is not XML-mapped:
      If $member is not XML-mapped or is a method call:
        Move to next member
      ▷ Need to create placeholder function for XML
      Else if called from the "from" clause of the query:
        ▷ Need to enumerate through results, so return a list
        Replace with SEQUENCE ($parent.$member, /*)
      Else:
        Replace with QUERY ($parent.$member, /*)
    $P ← PH($parent)

    If $member is a property:
      Replace with PropertyTranslate($parent, $member, $P)
    If $member is a method call:
      Replace with MethodTranslate($parent, $member, $P)
  For each conjunctive clause $c in the where clause:
    For each object $parent referenced by a COND function
                                    appearing in $c
      $cond ← conjunction of all COND($parent, _, _)
      $M ← mapping for type of $parent where $M.cp = $cond
      If $M does not exist, throw error
      $M_old ← base mapping for type of $parent
      $P ← PH($parent)
      $func ← $P.query ∘ .[. instance of $M.x][$M.xp]
      Replace $cond with TEST ($P.column, $func)

PropertyTranslate($parent, $member, $P)
  $M ← base mapping for the type of $parent
  If $member has a mapping fragment in $M:
    $comp ← compiled XQuery corresponding to the mapping
                          fragment for $M.$member
    If $member is of atomic type:
      $type ← type of $member
      Return VALUE ($P.column, $P.query ∘ $comp, $type)
    Else, return QUERY ($P.column, $P.query ∘ $comp)
  Else if $member participates in a predicate in $M, and also
                    participates in a predicate "= $V" in the query
    Return COND ($parent, $member, $V)
  Else, throw error   ▷ Member is not mapped to anything

MethodTranslate($parent, $method, $P)
  If $parent is atomic type:
      && $method has a known XQuery equivalent
    $T ← return type of $method
    $func ← translation of $method into XQuery equivalent
    If $T = Boolean:
      Return TEST ($P.column, $P.query ∘ $func)
    Else:
      Return VALUE ($P.column, $P.query ∘ $func, $T)
  Else if $parent is type XElement:
      && $method is a method representing an XPath axis
    $func ← XPath expression corresponding to $method
    Return QUERY ($P.column, $P.query ∘ $func)
  Else if $method = OfType<T>():
    $M ← base mapping for type T
    ▷ Create query that filters on the type and predicates of $M
    $func ← $P.query ∘ .[. instance of $M.x][$M.xp]
    Return $P.function ($P.column, $func)
  Else, throw error      ▷ Unsupported method
```

Figure 4. The LRX query translation algorithm

5. FULL-FIDELITY REPRESENTATION

When an XML document is exposed through a schema mapping, information that is not mapped may be lost. In this section, we describe how we present information that would normally be lost in translation to the user, which may happen for several reasons:

Information exists in the document that is not addressed in the schema. A schematized XML document can include three kinds of information that cannot be described by its XSD: processing instructions, comments, and whitespace. This kind of information can appear anywhere in an XML document, with our without a schema. Since it is not described in an XML schema, it ordinarily would not be part of the custom-typed object-oriented type definitions that are created to correspond to the XSD schema.

Attributes have been added to elements that are not in the schema. The set of attributes in a schema is the minimum set; new attributes can also be specified that are not anywhere in the associated schema (e.g., RDF tags). Since these attributes are not part of the schema, they cannot participate in a schema mapping.

Parts of the schema have not been mapped. We do not require that a schema mapping contain mapping fragments that refer to every element in a schema. A schema mapping can leave out some information in the XSD schema from the custom-typed object-oriented type definitions. That is, the information is schematized and could be custom-typed, but the designer leaves it out, presumably because it is not of interest to the application.

Order and element representations are not maintained. Once a custom-typed object has been generated for an XML fragment, some aspects of the original fragment are lost. The element order is lost, since object members have no explicit order (save for elements in a list). If elements came from an XML "all" group, the fragment's schema cannot be used to recover order. Also, the element content may not directly translate into an OO counterpart. For instance, the value "+06.43000" in an XML element would translate to the value "6.43" if assigned to a numeric variable, even if the trailing and leading zeroes were significant.

5.1 Delta Representation

Information loss when accessing an XML document via object-oriented type definitions presents the problem of how to obtain high fidelity when it is required. LRX addresses this problem by maintaining a *delta representation* that includes the information in an XML document that is not custom-typed, including comments, processing instructions, whitespace, and extra-schematic elements and attributes. The delta representation also includes the relative order of information in the document. If the delta representation is complete, one can construct an exact copy of an XML document from the parts of the document that are exposed as instances of its object-oriented type definitions and its delta representation.

We define an *anchor* to be a 2-tuple (c, i), where c is a Component Designator expression and i is a position reference. We know from Section 4 that, when considered together, these two pieces of information represent a unique element in an XML document. Unlike mapping fragments, the position reference in an anchor must be an integer (i.e., cannot be the value "all").

A delta representation is a function Δ: (c, i) → (R_{SP}, R_{SC}, R_C, R_{EP}, R_{EC}, R_{SC}) that associates each anchor with a collection of six *regions*. Each region represents a location of delta information in an XML document relative to an anchor's corresponding element:

- StartPrefix (R_{SP}): The region before element's opening tag.
- StartContent (R_{SC}): The region inside the element's opening tag, after the tag name. Elements in this region are indexed by their position relative to attributes of the element that may be mapped. Therefore, delta information immediately before attribute X will be indexed by the string "X", or indexed by "null" if the information follows all attributes.
- ElementContent (R_C): The region inside the element, only if the anchor corresponds to an atomic type (e.g., cannot contain other elements). Elements in R_C are indexed by character position.
- EndPrefix (R_{EP}): The region before the element's closing tag.
- EndContent (R_{EC}): The region after the closing tag's name inside the tag.
- SelfClosing (R_{SC}): A Boolean value indicating whether the tag was self-closing.

Each region can only contain certain kinds of information due to syntactic restrictions of XML. For instance, one cannot place comments within an element declaration, so comments cannot appear in the StartContent or EndContent regions. Table 1 shows the kinds of delta information that each region supports.

Not all of these regions are applicable to anchors that represent XML attributes. Attributes may contain whitespace before and after the equals sign. In other words, the following two attribute declarations are syntactically valid and have equivalent data:

```
Attribute="value"
Attribute     =         "value"
```

For the anchor associated with this attribute, the whitespace before and after the equals sign is captured in the StartContent and EndContent regions respectively.

LRX presents an interface that allows a user to access or update the delta representation for an object. The following example retrieves all attributes present in the delta information for member "foo" of object "bar", where "foo" is a scalar value:

```
bar.delta["foo", null].StartContent
                                  .OfType(XAttribute);
```

The following statement creates a new comment for the third member in list "foo2" to be placed before the opening element:

```
bar.delta["foo2",3].StartPrefix.Add(
                            new XComment("Testing"));
```

The first argument to the "delta" member can be either an object member name or a component designator. Each object member corresponds with a component designator (with or without a position reference), but there may be anchors that do not match with an object member. For instance, the type mapping may flatten several levels of hierarchy in an XML element, but there may be delta information associated with a level of that hierarchy.

Two special anchors exist to handle document-level delta information, and are only present in the delta representation of the object representing the root element. The first, ".", corresponds to the document's root element and covers all regions relative to the root element in the normal way. The second, "null", corresponds to data after the root element's closing tag. By convention, this delta information is stored in the null anchor's EndPrefix region.

Region	Supported by Attributes	Processing Instructions	Comments	Whitespace	Extra-Schematic Attributes	Unmapped Content
StartPrefix	No	X	X	X		X
StartContent	Yes			X	X	
ElementContent	No	X	X	X		
EndPrefix	No	X	X	X		X
EndContent	Yes			X		
SelfClosing	No					

Table 1. Describing what kinds of data are compatible with each anchor region, and which regions are supported by attributes.

Anchors are stored in the order that they are discovered in the document. Therefore, ordering of document elements and attributes is preserved. For elements that have an atomic value, the element value is stored as a string value as the item at index "null" in the ElementContent region for that element's anchor.

5.2 Parsing Automata

Current commercial implementations of XML offer separate tools to validate an XML document with respect to an XSD schema and then to parse the XML so that it can be accessed via an object-oriented API. This is wasteful, since much of the computation time in validating the document involves parsing. It is therefore more efficient to integrate these two tasks so they are performed together in a single pass over the document [2, 7].

LRX compiles an XSD schema into an automaton that accepts documents that conform to the schema. When the automaton recognizes a part of the document that corresponds to a custom-typed object-oriented type definition, it creates an instance of that type, extracts the recognized part of the document, and uses the extracted part to populate the instance in the same pass. Since the automaton is parsing the document relative to the XSD schema, it is in effect validating the document too. For example, the automaton might parse a DateTime value to recognize its type, which tells it that this part of the document is valid and tells it how to populate an object that is supposed contain that value.

One way in which our compiled automaton differs from previous projects that compile XSD-specific parsers in that we build the delta representation in the same single pass. When the automaton identifies a part of the document that is not covered by the schema (such as a comment), it retains this information in the delta representation, in the region in which it is found relative to the anchor point at the current position of the cursor (or next anchor point, if the delta information is found between elements). Thus, the automaton effectively splits an input document into the parts extracted to populate custom-typed objects and the remainder that is retained in the delta. We call this an *input automaton*.

A type mapping may associate a class member *m* with an XML element *x* that is optional. When parsing an XML fragment, if the input automaton discovers that element *x* is not present, it will leave member *m* in the generated object as null. However, the automaton will create a delta representation anchor with all empty regions. If member *m* is given a value on the client, its anchor is already in the appropriate order placement in the delta.

When the automaton identifies a part of the document that does not conform to the schema (such as an element that is out of place according to the schema), it retains the violating part of the document in the delta. An exception is only thrown if the document is missing required data. This relaxed schema validation can be used to cope with schema evolution. Documents that conform to the evolved schema may still be accepted by the automaton for the original schema. In this case, the data that conforms to the new parts of the schema but not the original schema are not lost, because they are retained in the delta.

In most cases, given an XML type that participates in a type mapping, LRX can uniquely determine the corresponding type from the schema mapping and thus can generate a type-specific parsing automaton. However, there are two exceptions:

- There is an xsi:type attribute in an element. Consider the case that element *e* would normally map to class C1, but an xsi:type attribute indicates that *e* should translate into an instance of C2. We try to parse *e* into an instance of C1, but upon seeing the xsi:type attribute, we restart parsing in a C2-specific automaton. Because xsi:type attributes appear early, little time is wasted.
- There are XML conditions in the type mapping. In this case, we cannot uniquely determine the output object type because it is determined by the content of each parsed element, and that information is not available until after the parse. In this case, we revert to parsing the element into an XML fragment in client space, read the data to determine the correct type, then run the type-specific parser to build the object and delta representation. This process is an open area of optimization.

LRX also precompiles an *output automaton* from the schema, which joins together data saved as custom-typed objects with the deltas to generate the original XML document with full fidelity. Some verification occurs in the output automaton to ensure the XSD semantics are preserved. For instance, if the XML schema has a choice restriction and there is not exactly one qualifying data member, the output automaton will throw an error.

6. UPDATE TRANSLATION

A developer updates a database through LRX by updating the results of a query. To be updatable, a query result must contain enough information to identify the row that an XML fragment belongs to (e.g., key values) and also must refer to the root node of the XML column. This updatability restriction is compatible with the restrictions currently in place for updates through EF. So, the following query would return a result that is updatable:

```
var q = from i in db.Items
        select new {i.ID, i.Item};
```

LRX translates updates against client-side objects into update statements on the database. For non-simple types, LRX handles updates to the members of objects of that type or their descendants by using the output automaton for that type to reconstitute the XML document associated with the updated object and updating the entire document on the client side.

For simple types, LRX translates object modifications into XINSERT, XUPDATE, or XDELETE placeholder functions, which represent in-place modifications of stored XML documents. All three major DBMS systems support the ability to insert, update, or delete nodes relative to the result of an XPath expression. For instance, for deleting nodes, the DELETEXML function in Oracle can delete all nodes in a document that match an XPath expression. The .modify method in SQL Server and the xmlquery function in DB2 are both patterned after the XQuery Update Facility [19] and can delete in-place all nodes in a document that are returned by an XQuery expression. Since mapping fragments on simple types compile to XPath, they can be used to identify update locations across all three implementations.

For example, consider the following code expression:

```
foreach (var i in q) if (i.ID == "item20748")
  i.Item.Name = "NewName";
db.SaveChanges();
```

This code immediately changes the name of a single item in client space, but also generates an update statement that is sent to the database once the SaveChanges method is called. On SQL Server, the update statement takes the following form:

```
UPDATE Regions SET Item.modify('
  declare namespace a = "http://.../Auction";
  replace value of (/*/Name) with "NewName"')
WHERE ID = 'item20748'
```

Every time a member *m* (or collection element) is modified on a simple-type, XML-mapped object *p*, LRX runs algorithm UpdateTranslation (Figure 5) to generate placeholders. LRX then uses existing EF features to identify the relational tables, columns, and conditions of the update. If *m* is updated or deleted, UpdateTranslation uses compiled mapping fragments to generate a query that identifies the XML document location of *m*. If *m* is inserted, UpdateTranslation uses *p*'s delta representation ordering to identify the members that occur just before and after *m* to place *m*'s data in the correct location in the document.

UpdateTranslation has two helper functions. The first, BuildPath, compiles into a single XQuery the list of CD expressions that are necessary to get from the root of the XML document to the element for the updated object's parent. The second, PadValue, nests a node that is to be inserted ($element) within new elements so that the inserted node is at the necessary level of the document hierarchy (represented by the CD expression in parameter $cd). Finally, ⇔ (query1, query2) is shorthand that represents a query for all of the sibling nodes before query2, and also after query1 if it is not empty. We use this shorthand to identify, for a node *n* to be deleted, the nodes corresponding to the delta information for *n*.

The association between an anchor and delta information in its regions implies a positional relationship, not necessarily a semantic one. UpdateTranslation presumes that the positional relationship is also a semantic relationship and maintains the relative positions of delta information with respect to its anchor. For instance, consider the region R_{SP} for a deleted object (or a member whose object reference is set to null) with comments or processing instructions preceding the original element's start tag. UpdateTranslation deletes all delta information for that object, so it also deletes the nodes in that region in the original document. UpdateTranslation only creates XINSERT statements with "as first" or "after" modes, so that new nodes do not come between an existing node and the delta information immediately before it.

```
UpdateTranslation ($parent, $member):
  ($column, $query) ← BuildPath($parent)
  If $member has a mapping fragment defined for the base
                                    mapping for the type of $parent:
    ($cd, $index) ← mapping fragment for $member
    $qmap ← compiled query for mapping fragment
    $v ← new value for $member, serialized to XML using
                                    output automaton if not atomic-typed
    ($cb, $ib) ← first anchor with a non-null member value
                    before ($cd, $index) in the delta for $parent
    ($ca, $ia) ← first anchor with a non-null member value
                    after ($cd, $index) in the delta for $parent
    $pfxb ← common prefix between $cb and $cd
                                    or "." if $cb does not exist
    $pfxa ← common prefix between $ca and $cd
                                    or "." if $ca does not exist
    If $v = null (or $member is item deleted from a collection):
      Issue XDELETE ($column, ⇔($cb[$ib], $query ∘ $qmap))
      Issue XDELETE ($column, $query ∘ $qmap)
      Delete the contents of Δ($cd, $index)
    Else if old value of $member is null
                    (or $member is item inserted into a collection):
      If length of $pfxb < length of $pfxa:
        $mode ← "as first"
        ▷ Case: new element is the first entry in a list
        If $cd = $pfxa, ($cn, $in) ← ($cd.parent, 1)
        ▷ Case: new element is the first entry in a sequence
        Else, ($cn, $in) ← ($pfxa, 1)
      Else:
        $mode ← "after"
        ▷ Case: new element is an added entry in a list
        If $cd = $pfxb, ($cn, $in) ← ($cb, $ib)
        ▷ Case: new element is an added entry in a sequence
        Else, ($cn, $in) ← ($pfxb', $ia')
            where $pfxb' is $pfxb plus one more step from $cb
            and $ib' = $ib if $pfxb' = $cb, 1 otherwise
      $v ← PadValue ($cd, $v, $cn)
      $qprev ← compiled fragment for anchor ($cn, $in)
      Issue XINSERT ($column, $query ∘ $qprev, $v, $mode)
    Else:
      Issue XUPDATE ($column, $query ∘ $qmap, $v)
      Delete the contents of Δ($cd, $index).R_C
  If $member participates in a type mapping condition:
    ▷ Object now may maximally match a different mapping
    $o ← the result of repackaging object $parent into XML
    Issue XUPDATE ($column, $query, $o)

BuildPath ($parent):
  $root ← ancestor object of $parent corresponding to the root
                                    of the XML document
  $query ← "/*"
  For each member reference $p.$m in the path of
                    member references from $root to $parent
    $map ← maximally matched type mapping for $p
    $query ← $query ∘ (XQuery for $m's fragment in $map)
  Return ($root, $query)

▷ Nest an XML element to the needed hierarchy level
PadValue ($cd, $element, $cdcompare):
  $common ← common prefix between $cd and $cdcompare
  $diff ← part of expression $cd after $common
  ▷ With simple types, $diff is interchangeable with XPath
  For each step $step in $diff, read backwards
    $element ← <$step>$element</$step>
  Return $element
```

Figure 5. The LRX update translation algorithm

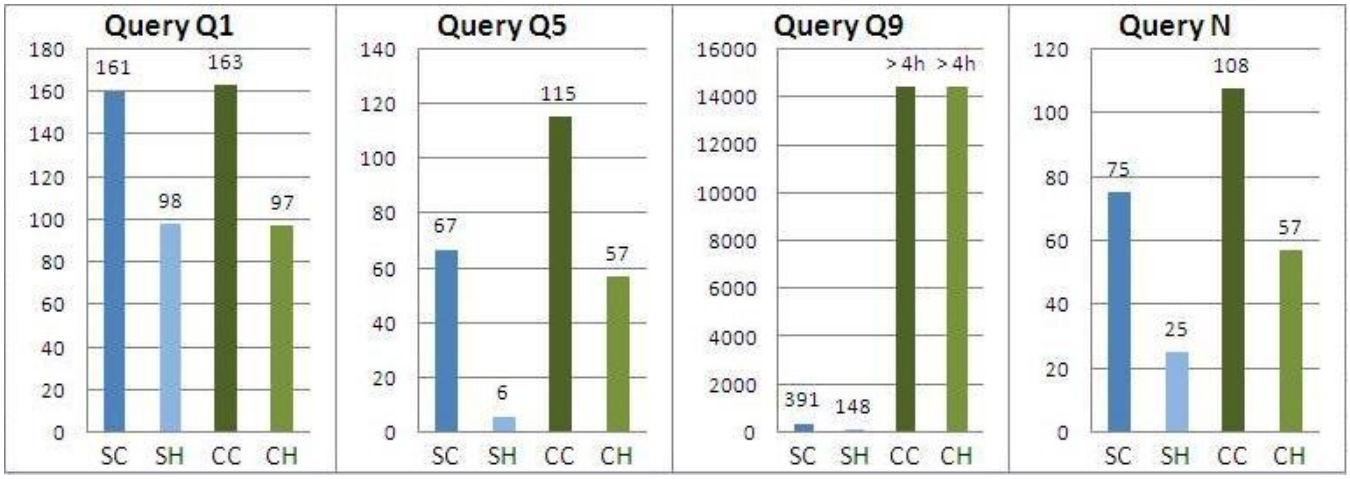

Figure 6. Timing results from running four queries, pushing the entire query to the server using LRX (SC and SH) versus evaluating the XML portion client-side (CC and CH). Results are in seconds

7. EVALUATION

We believe that a primary contribution of LRX is developer productivity. Anecdotally, we have seen examples of SQL and XQuery queries that take hours to construct and debug. Using LRX, there is no need to worry about SQL dialects, XQuery constructs, namespaces, correct embedding of XQuery into SQL, etc. Without long-term developer adoption, this hypothesis is impossible to test. However, if LRX is at least as fast as existing approaches, then we believe that LRX can offer an overall advantage. Our performance results demonstrate LRX is just as fast as client-side execution in the worst case, and often offers dramatic performance gains.

To evaluate LRX performance, we built a prototype that extends EF in the fashion described in Section 3. In this section, we present timing analysis of different scenarios that demonstrate the performance benefit of pushing object queries to the server as SQL and XQuery when compared to performing the XML portion of the same queries on the client.

We tested our implementation by measuring the execution time of four different queries. Included in this set of queries are equivalents of queries Q1, Q5, and Q9 in the XMark benchmark. The fourth query that we ran is the query from Section 2.2 above, referred to as Query N. Our data set was generated using the XMark data generator with a scaling factor of 40, generating a 4GB XML file that we then shredded into our database.

Execution time was measured from when the query is sent to LRX to when the query has finished computation and has been processed by the client, iterating through all of the rows of the result. This ensures that client-side and server-side processing are compared on an even playing field, as server execution time alone does not account for the time cost of moving results to the client.

The performance tests were run on a machine running Windows Vista with a 3GHz dual-core processor and 4GB of RAM, with a commercially-available database management system. Client and server were run on the same machine to eliminate network overhead, and to ensure isolation of the database. Figure 6 presents the results of our timing analysis. We ran each query 400 times, 100 times each for the following cases:

- Using LRX, clearing buffers and plan cache between executions (SC – 'C' for "cold buffers" case)
- Using LRX, leaving buffers and plan cache alone between executions (SH – 'H' for "hot buffers" case)
- Pushing the relational part of the query to the server (clearing buffers and plan cache), but evaluating XML client-side (CC)
- Evaluating XML client-side, leaving the buffers and plan cache alone between executions (CH)

In the text of the queries, there are member references that include the suffixes "_Typed" and "_XML". The "_Typed" member represents the custom-typed version of whatever the member is, with an associated XML mapping so that queries referencing the member are pushed to the server. The "_XML" member represents the generically typed version of the member, where the raw XML stored in the corresponding relational column is pulled to the client and validated and processed there as XML.

These performance tests are not intended to evaluate specific XML parsing or query engines, either on the client or the server. Our contention is that, by pushing queries to the server, performance improves because indexes can be exploited or built, queries can be optimized at the server, and the high likelihood that far fewer data rows are returned by the query over the network.

7.1 Simple XPath Processing (Q1)

Query Q1 from the XMark benchmark returns a single, scalar value representing the name of a particular item. This query is only different than the one listed in the XMark documentation in that we have changed the item ID to one that corresponds to an item in our data set. Here is Q1, expressed in LINQ:

```
var q = from o in db.Items
        where o.region == "namerica"
        where o.id == "item522031"
        select o.item_Typed.name;
foreach (var o in q) var s = o;
```

Here is the same query, expressed as a strictly relational query with client-side XML processing:

```
var q = from o in db.Items
        where o.region == "namerica"
        where o.id == "item522031"
```

```
        select o.item_XML;
foreach (var o in q)
  string s = (string) o.Element("name");
```

Figure 6 shows that pushing query Q1 entirely to the server takes approximately the same time as running the XML portion on the client. Comparing case SC versus CC and SH versus CH shows a negligible performance difference between client and server execution. This result is unsurprising, since query Q1 retrieves a single row in a table, then parses and validates the XML value for that row to run the XPath query. Whether the parsing, validating, and querying of that one small document occurs on the server or the client does not have a large effect on performance.

7.2 Aggregation and Filtering (Q5)

Query Q5 from the XMark benchmark is an aggregation query, counting the number of auctions that closed for at least a price of 40. Here is the query, expressed entirely as an object query:

```
var c = (from o in db.ClosedAuctions
         where o.closed_auction_Typed.price >= 40
         select o.auto_id).Count();
```

Here is the same query, expressed as a strictly relational query with client-side XML processing:

```
var q = from o in db.ClosedAuctions
        select o.closed_auction_XML;
int i = 0;
foreach (var o in q)
  if ((decimal)o.Element("price") >= 40) i++;
```

From Figure 6, by comparing case SC versus case CC for query Q5 we see that pushing the query entirely to the server has a 42% performance gain over client-side execution on cold buffers. On hot buffers, the performance gain is substantially greater.

7.3 Joining on Values (Q9)

Query Q9 from the XMark benchmark is a join query, which joins on the values within the XML. In XMark, this query joins along ID references. In our partially-shredded data set, two of the join parameters have become relational columns, so the join is between XML values and relational values. Here is the query expressed as an object query:

```
var q = from o in db.People
        from c in db.ClosedAuctions
        from i in db.Items
        where c.closed_auction_Typed
               .itemref.item == i.id
           && c.closed_auction_Typed
               .buyer.person == o.id
        select new { name = o.name,
                     item = i.item_Typed.name };
int j = 0;
foreach (var o in q)
{ var s = o.name; j++; }
```

Here is the same query, expressed as a strictly relational query with client-side XML processing:

```
var q = from o in db.People
        from c in db.ClosedAuctions
        from i in db.Items
        select new { iid  = i.id,
                     cl   = c.closed_auction_XML,
                     pid  = o.id,
                     name = o.name };
int j = 0;
foreach (var o in q)
  if (o.iid == (string) o.cl.Element("itemref")
                           .Attribute("item")
   && (string) o.cl.Element("buyer")
                   .Attribute("person") == o.pid)
   { j++; }
```

The performance improvement by pushing this query to the server was dramatic. The client-side version of Q9 ran reliably for longer than four hours without completing, versus about 6.5 minutes for the LRX version with cold cache and buffers. Because the three-way join is based entirely on values within the closed auction XML trees, the relational-only query sent to the server is just a massive cross product.

7.4 Descendant Query with Sequence (N)

This query is roughly related to query Q6 from the XMark benchmark in that it evaluates a descendant's XPath expression. Query Q6 did not have a direct analog in our test data, since the interesting part of Query Q6 referenced XML that was shredded into relations. This query returns a list of all of the emphasized text elements in the description of any closed auction that closed for more than 20. The server-side version of the query includes a call of the descendants (//) axis of XPath and the use of the SEQUENCE placeholder function. Its object query is:

```
var q = from c in db.ClosedAuctions
        where c.closed_auction_Typed.price > 20
        from e in c.closed_auction_Typed
                   .annotation.description
                   .text.Descendants("emph")
        select e;
int j = 0;
foreach (var o in q) j++;
```

Here is the same query, expressed as a strictly relational query with client-side XML processing:

```
var q = from c in db.ClosedAuctions
        select c.closed_auction_XML;
int j = 0;
foreach (var o in q)
  if ((decimal) o.Element("price") > 20)
  { var f = o.Element("annotation")
             .Element("description")
             .Element("text");
    if (f != null)
      foreach (var d in f.Descendants("emph"))
        j++; }
```

In Figure 6, we see that, for hot cache buffers, there is a 56% performance gain using LRX for this query.

8. RELATED WORK

There is a great deal of research that has been done in the areas of data and schema mapping, query and update translation, and data exchange. For brevity, we narrow our focus here to topics specific to XML, and direct the reader to previous publication on the Entity Framework for additional discussion [1,13]. We have previously demonstrated an earlier prototype of LRX [17], and we have additional work describing compilation of mapping fragments for non-simple types [12].

Several tools are available that provide access to XML documents as custom-typed objects. XML Beans [18] exposes XML documents as typed Java objects, while Liquid XML [10] exposes XML documents as typed objects in a variety of languages. Finally, a custom-typed LINQ interface to XML was proposed in the initial LINQ-to-XSD work of Lämmel et al. [8] at Microsoft.

Each of these tools is limited to XML in main memory. They do not push any operations to a database. In addition, they use fixed mappings that cannot be controlled by the user. Our work

addresses both of these limitations. To the best of our knowledge, LRX is the first system that supports accessing typed XML stored in a database through any object-based query language.

Both XML Beans and LINQ-to-XSD support full fidelity representations of XML by maintaining a complete copy of the original document. In these tools, a custom-typed object is just an interface to the document, and updates occur in place. The advantage of the full-document approach is that it is never necessary to reconstruct the original document, since it is always present. The disadvantage is that access to the full fidelity of the information is not object-oriented. To access delta information, for instance, one must traverse the XML using a cursor. The advantage of the delta representation is that one can ask the question, "What is the delta information for object member X". In LRX, if XML-like, generically typed access is preferred, it is always possible to return query results as XML documents and still be able to push queries to the server.

Tools such as XJ [5] and LINQ-to-XML offer an XPath-like interface to generically typed XML objects. These tools are also limited to manipulating XML in memory. Lenses [4] and bidirectional XQuery [11] are additional approaches for updatable access to XML. Rather than translating queries and updates, these approaches are state-based, taking an updated materialized query result and reconstructing an updated base document.

Other research projects have considered the benefits of compiling an XML schema into a parser-validator, including XML Screamer [7] and work by Chiu and Liu [2]. These approaches demonstrate the performance advantage of schema-specific parser-validators over standard parsers and validators. Our approach differs from those research projects in that the compiled automata are specific to both the XML schema and the schema mapping, and that data that either is not schematized or does not match the schema is saved and made available in the delta representation.

Clio [15] and Oracle TopLink [14] are two tools that map to XML schemas and have flexible mapping schemes that allow a developer to customize mappings. Oracle TopLink uses XPath expressions as its mapping language, while Clio uses a graphical mapping tool that is roughly equivalent to using Component Designator expressions. Both tools are designed to marshal and unmarshal data from XML and thus do not handle query or update translation or full-fidelity representation.

9. CONCLUSION

In this paper, we introduced an architecture that allows queries and updates expressed in an object-oriented language to be pushed to the server for processing. Object types and XML types are associated using a flexible mapping scheme. Query results are made available through custom types or generic types based on the developer's needs. Query results also include a delta representation that allows object-oriented access to the parts of the XML document not addressed by mappings. Component designator expressions are the key to both the mapping language and the delta representation. Performance numbers strongly indicate the benefit of pushing XML queries to the server.

One area of future work that remains to be considered is query optimization between relational and XML components. For instance, depending on the circumstance, it may be more efficient to evaluate conditions on XML nodes using SQL functions or comparators, or push as much as possible into XQuery processing. Other future work includes incremental updates to the delta representation (currently, updating the delta triggers full re-serialization of the enclosing element) and translating keyref or idref nodes in XML into client-side object pointers.

10. REFERENCES

[1] A. Adya, J. A. Blakeley, S. Melnik, S. Muralidhar, The ADO.NET Team. Anatomy of the ADO.NET Entity Framework. *SIGMOD*, 2007.

[2] K. Chiu, W. Lu. A compiler-based approach to schema-specific XML parsing. *First International Workshop on High Performance XML Processing* (May 2004).

[3] W. R. Cook, A. H. Ibrahim. Integrating Programming Languages and Databases: What is the Problem? ODBMS.ORG, Expert Article, Sept. 2006.

[4] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, May 2007, 29(3).

[5] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, V. Sarkar. XJ: facilitating XML processing in Java. *WWW*, 2005.

[6] Hibernate. http://www.hibernate.org/.

[7] M. G. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, M. Mercaldi. XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization. *WWW 2006*, 93–102.

[8] R. Lämmel. LINQ-to-XSD. *PLAN-X*, 2007.

[9] R. Lämmel, E. Meijer. Revealing the X/O Impedance Mismatch (Changing Lead into Gold). In *Datatype-Generic Programming*, LNCS 4719. Springer-Verlag, June 2007.

[10] Liquid XML. http://www.liquid-technologies.com/.

[11] D. Liu, Z. Hu, M. Takeichi. Bidirectional interpretation of XQuery. *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, 2007.

[12] E. Meijer, B. Beckman, G. M. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. *SIGMOD*, 2006.

[13] S. Melnik, A. Adya, P. A. Bernstein. Compiling Mappings to Bridge Applications and Databases. *SIGMOD 2007*.

[14] Oracle TopLink. http://www.oracle.com/technology/products/ias/toplink/index.html.

[15] L. Popa, M. A. Hernández, Y. Velegrakis, R. J. Miller, F. Naumann, H. Ho. Mapping XML and Relational Schemas with Clio. *ICDE*, 2002.

[16] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, R. Busse. XMark: a benchmark for XML data management. *VLDB*, 2002, 974–985.

[17] J. F. Terwilliger, S. Melnik, P. A. Bernstein. Language-integrated querying of XML data in SQL Server. *PVLDB*, 1(2).

[18] XML Beans. http://xmlbeans.apache.org/. [17]

[19] W3C. XQuery Update Facility. Candidate Recommendation. http://www.w3.org/TR/xqupdate/.

[20] W3C. XSD Component Designators. Working Draft. http://www.w3.org/TR/xmlschema-ref/.