Language-Integrated Querying of XML Data in SQL Server

James F. Terwilliger[*]
Portland State University
jterwill@cecs.pdx.edu

Sergey Melnik and Philip A. Bernstein
Microsoft Research, USA
{Sergey.Melnik, Phil.Bernstein}@microsoft.com

ABSTRACT

Developers need to access persistent XML data programmatically. 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. Our prototype addresses this problem by transforming object-based queries and updates into queries and updates on XML using declarative mappings between classes and XML schema types. Our prototype extends the ADO.NET Entity Framework and leverages its object-relational mapping capabilities.

We demonstrate how a developer can interact with stored relational and XML data using the Language Integrated Query (LINQ) feature of .NET. We show how LINQ queries are translated into a combination of SQL and XQuery. Finally, we illustrate how explicit mappings facilitate data independence upon database refactoring.

1. INTRODUCTION

XML data has become ubiquitous in data-centric applications. Many commercial database management systems support XML storage. Yet, the problem of translating between XML and objects automatically is largely unsolved, due to differences in the expressive power of their type systems [5] and the difficulty of translating object queries into an XML query language such as XQuery. In hybrid relational/XML databases, this problem is compounded by the object-relational impedance mismatch, since XML data can be partitioned across multiple relational tables.

Several object-relational mapping (ORM) frameworks [2] 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. However, typically ORMs do not handle the mismatch between objects and XML. Recently, substantial research was done on the XML-relational mismatch, including shredding XML into relations and rewriting XQuery as SQL. However, as far as we know this line of work did not consider programming language integration.

We developed a prototype, called *LINQ-to-Stored XML*, that enables programmatic access to persistent XML and relational data from .NET applications by using explicit mappings between object classes, XML schema types, and relations. The mappings drive query and update processing. They can be generated automatically or provided by the developer. Using the language-integrated query (LINQ) feature of .NET [7], the developer can write object queries that resemble SQL but are statically compiled and type-checked in an object-oriented programming language. Our prototype translates LINQ queries into a mixture of SQL and XQuery to execute in the database, using the native SQL dialect and XML features of that database. Our implementation extends the ADO.NET Entity Framework [1] and leverages its object-relational mapping capabilities. The demonstration runs on Microsoft SQL Server 2005.

Our research goes beyond the capabilities of existing XML-to-object mapping tools in two important ways. First, existing tools work only on XML documents while they are in memory; LINQ-to-Stored XML supports in-memory translation of XML documents into objects, but can also push queries to the database. Second, existing tools we are aware of can map XML to objects only in a pre-determined, canonical fashion. LINQ-to-Stored XML can begin with a canonical mapping, but then allows the programmer to adjust or rewrite the mappings to suit the needs of the application, e.g., to support schema evolution.

Section 2 details the capabilities of our prototype and what we demonstrate. We then describe the run-time and design-time components of our prototype's implementation in Section 3. We conclude in Section 4 by comparing our contributions against related work.

2. WHAT IS DEMONSTRATED

Our running example is based on AdventureWorks, a sample database distributed with SQL Server 2005. This database contains several tables whose columns store XML data. For example, the table `JobCandidate` shown in Figure 1(a) has a column `Resume` whose contents are XML documents. Figure 2(a) shows part of the XML schema for those documents.

2.1 Querying XML as Strongly-Typed Objects

We start by demonstrating how XML data that has an associated XML schema can be queried using classes mapped to XML schema types. The following C# program uses LINQ to list email addresses and schools attended by job candi-

[*]Research conducted at Microsoft during an internship

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-306-8/08/08

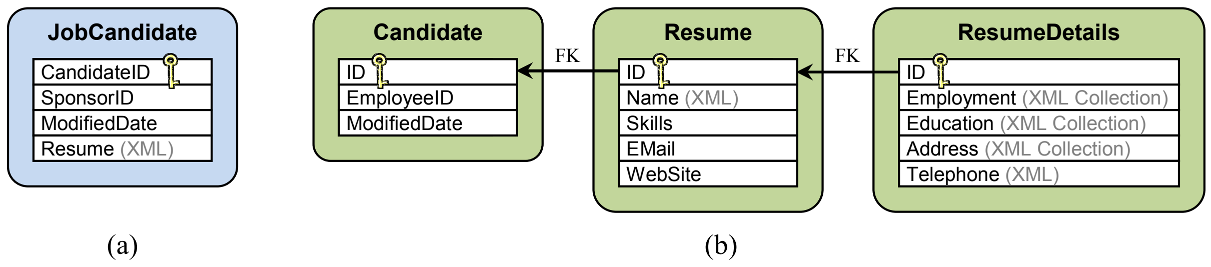
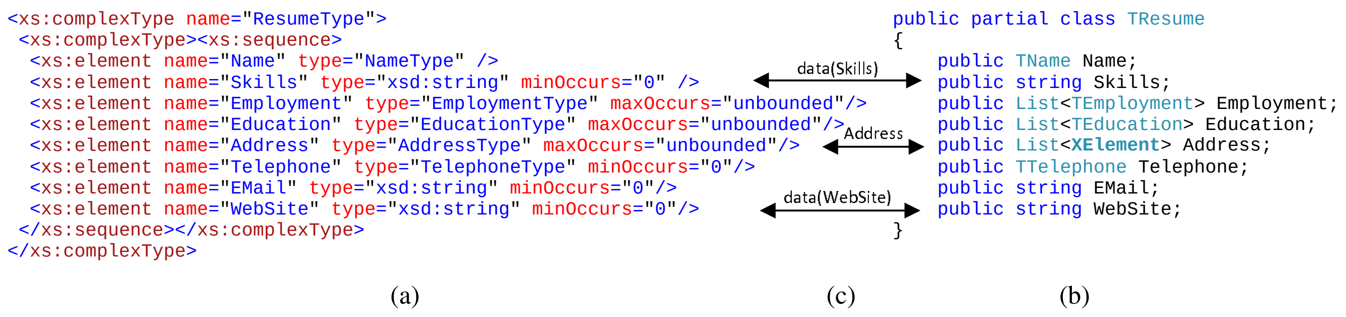


Figure 1: (a) The relational portion of the job candidate table in the AdventureWorks database and (b) an alternative representation where the XML data has been refactored into multiple tables



```
<xs:complexType name="ResumeType">
 <xs:complexType><xs:sequence>
  <xs:element name="Name" type="NameType" />
  <xs:element name="Skills" type="xsd:string" minOccurs="0" />
  <xs:element name="Employment" type="EmploymentType" maxOccurs="unbounded"/>
  <xs:element name="Education" type="EducationType" maxOccurs="unbounded"/>
  <xs:element name="Address" type="AddressType" maxOccurs="unbounded"/>
  <xs:element name="Telephone" type="TelephoneType" minOccurs="0"/>
  <xs:element name="EMail" type="xsd:string" minOccurs="0"/>
  <xs:element name="WebSite" type="xsd:string" minOccurs="0"/>
 </xs:sequence></xs:complexType>
</xs:complexType>
```

```
public partial class TResume
{
    public TName Name;
    public string Skills;
    public List<TEmployment> Employment;
    public List<TEducation> Education;
    public List<XElement> Address;
    public TTelephone Telephone;
    public string EMail;
    public string WebSite;
}
```

Figure 2: Part of mapping (c) between an XML schema type (a) and a class (b)

dates who have an email address and a Bachelor's degree:

```
using(AWdb db = new AWdb()) {
  var q = from c in db.JobCandidates
          from e in c.Resume.Education
          where c.Resume.EMail != null &&
                e.Degree.Contains("Bachelor")
          select new { c.Resume.EMail, e.School };
  foreach (var i in q)
    Console.WriteLine(i);
}
```

The "var q" declaration indicates that the return type of the query is inferred by the compiler (as a collection of string pairs). We illustrate how the query can be written easily using the IntelliSense feature of Microsoft Visual Studio 2008, which automatically suggests member names such as Degree and School (they belong to the TEducation class whose definition is omitted for brevity, as are subelements of EducationType). Our prototype translates the query into the following SQL and XQuery:

```
WITH XMLNAMESPACES('http://.../Resume' AS r)
SELECT
  C.Resume.value('*[1]/r:EMail', 'varchar(max)'),
  E.value('./r:Edu.School', 'varchar(max)')
FROM HumanResources.JobCandidate C
CROSS APPLY C.Resume.nodes('*[1]/r:Education') T(E)
WHERE C.Resume.exist('*[1]/r:EMail')=1 AND
 E.exist('./r:Edu.Degree[contains(.,"Bachelor")]')=1
```

The functions "value", "nodes", and "exist" are SQL Server-specific directives that apply XQuery expressions to XML fragments. We generate the XPath fragments in the query (e.g., *[1]/r:EMail) based on mappings that describe relationships between XML and objects. We describe these mappings in more detail in the implementation section.

2.2 Querying XML using Loosely-Typed Objects and Embedded XPath

Not all XML schema types can be mapped to strongly-typed classes. For instance, the declared type of an XML element may be "xsd:anyType", which does not have a statically-typed object counterpart any more descriptive than "any XML data". Also, mixed-content elements are hard to map to strongly-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.

Our prototype supports these scenarios by mapping XML schema elements to a .NET type called "XElement" in the LINQ-to-XML API that represents an XML element [8]. For example, in Figure 2 the XML schema element Address, which has an unbounded number of occurrences, is mapped to a list of XElements. Each XPath axis has a counterpart in the object layer as a method of the class XElement.

The following LINQ query illustrates how strongly-typed and loosely-typed portions of the query can be used in a single expression. The nested subquery that ends with "Any()" restricts the result to those job candidates who have at least one address with the postal code 98052. ModifiedDate has a strongly-typed .NET type DateTime, which has a member Year and is mapped to the relational type "datetime". Address is loosely-typed XML; to access the PostalCode member of an Address in variable "a", we must use the Element method (corresponding to the XPath child axis):

```
from c in db.JobCandidates
where c.ModifiedDate.Year <= 2007 &&
 (from a in c.Resume.Address
  where (int)
    a.Element("{http://...}Addr.PostalCode") == 98052
  select a).Any()
select c.Resume.Name;
```

The above query is translated into the following SQL/XQuery expression:

```
SELECT C.Resume.query('*[1]/r:Name')
FROM HumanResources.JobCandidate AS C
WHERE DATEPART(year, C.ModifiedDate) <= 2007 AND
 EXISTS (SELECT 1
  FROM C.Resume.nodes('*[1]/r:Address') AS T(A)
  WHERE A.value('./r:Addr.PostalCode[1]', 'int')
       = 98052)
```

Our prototype rewrites LINQ's Element() function (and the cast to "int") as value() in T-SQL and applies it to the field variable A iterating over Address elements. The nested subquery becomes a SQL EXISTS clause. The condition on the date is expressed using a built-in SQL function DATEPART. Each XML document returned by the query is materialized on the client as an object of type TName, a strongly-typed class mapped to NameType in the XML schema.

2.3 Refactoring XML into Relations

Finally, we demonstrate how the prototype introduces a level of data independence that isolates queries from a changing persistence model. We allow for both the relational and XML schemas of the database to evolve; the application code remains intact as long as the mappings are updated with the schema and can compensate for the changes. We show how we can alter the schema of Figure 2(a) (and the data to match) so that all of the Education nodes are moved beneath a new EducationListing parent node. We compensate for this change by appending the "EducationListing/" prefix to all XPath expressions in the mapping that reference the Education nodes, without modifying the application.

Moreover, the prototype allows data to change from relational to XML or vice versa. For example, the schema in Figure 1(b) is a refactoring of the schema in Figure 1(a), where the original XML data has been partitioned across three tables containing relational and XML columns. Such selective shredding of XML data may be motivated by performance concerns; retrieval of frequently-accessed elements of XML may be significantly faster if those elements are moved into their own tables and columns. Once we adjust the mappings to conform to the new schema, all of the demonstrated queries still work without alteration.

3. IMPLEMENTATION OVERVIEW

Our implementation builds on the ADO.NET Entity Framework [1], which enables applications to interact with the database via a conceptual entity model and an object surface that encapsulates conceptual types (Figure 3). We extended the framework by adding an XML mapping layer. Specifically, we modified the LINQ query translator and the Data Provider[1] for SQL Server to recognize XML mappings and embed XQuery statements into the generated SQL. All of the XML-related features shown in Figure 3 are unique to our prototype.

3.1 Flexible X-O Mappings

Before we can execute LINQ queries on objects, we need to represent the XML data in the object layer. In our running example, we need to map the XML types used in the Resume column to classes. Figure 2 shows a portion of the mapping that associates ResumeType from the XML schema with a C# class TResume. The mapping is specified using XPath expressions, one for each class member, shown as labels on double-headed arrows. For example, the XPath "Address" retrieves all of the Address children of a Resume element, while the XPath "data(WebSite)" identifies the scalar value of the WebSite child element. 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. For example, an XML schema type AddressType may be mapped to classes TAddress or TUSAddress depending on whether a state element appears in the XML instance.

While XPath expressions describe how to transform XML into objects, they do not specify the reverse transformation explicitly. For a mapping to be reversible, we require that the collective set of XPath expressions for a single class decompose the XML structure into non-overlapping, contiguous segments along the descendant axis. We restrict the supported subset of the XPath language to enable validation of these roundtripping conditions. The XPath subset we currently support includes:

- Child element axis (e.g., `Q/R/S` for qualified names `Q`, `R`, and `S`)
- Attribute axis (`@Q` for some qualified name `Q`)
- Self axis (`.`)
- Filter by absolute position (`[i]` for some positive integer `i` or the function `last()`)
- Filter by relative position (e.g., `Q[.<<../R[1]]` finds all instances of `Q` before the first instance of an `R` sibling)
- The `data()` function, to retrieve scalar values

3.2 Design-time Mapping Generation

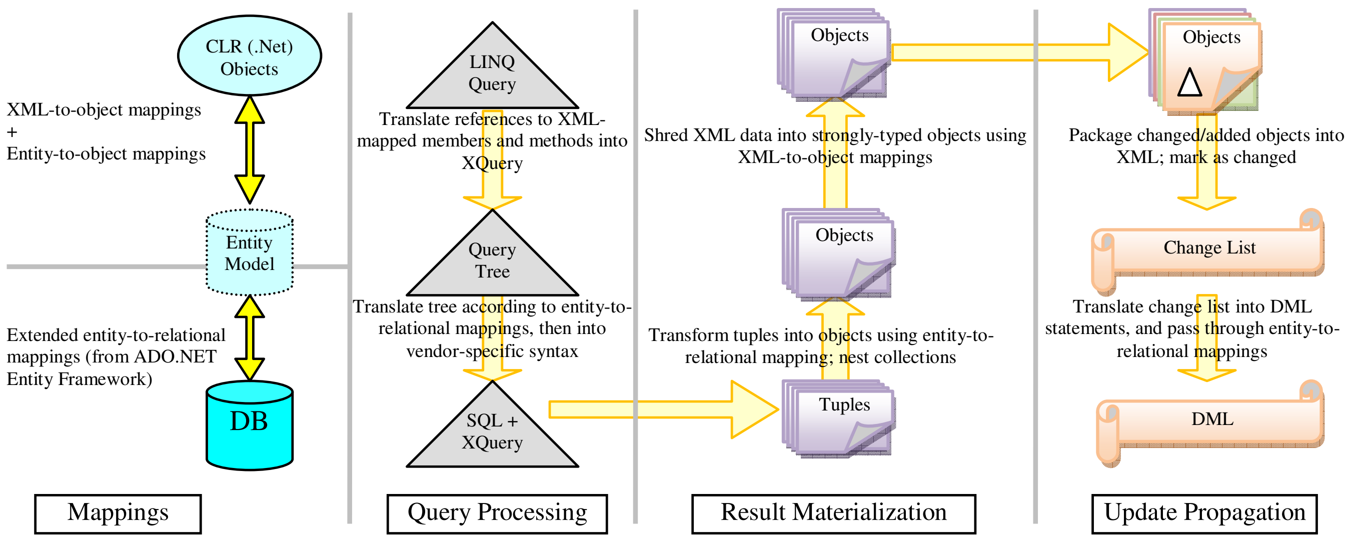
We provide several ways to automatically generate a default mapping. First, we have written a tool that reads the XML schemas that are stored in the database or in the local file system and generates a set of .NET classes and mappings to those classes. Generation of classes and mappings are driven by a set of canonical rules. For example, if an XML element can be repeated it is mapped to a .NET collection type, etc.

Conversely, the developer can also start with a set of .NET classes. In this case, we use a technology from Windows Communication Foundation (WCF [9]) called *Data Contract*. WCF includes a schema generator that creates an XML schema from a set of classes based on canonical choices. The default choices can be overridden using class-level attributes provided by the developer. We built a tool that creates a mapping that respects the XML schema generated by WCF.

3.3 Runtime Query and Update Processing

We leverage multiple existing features and extensibility mechanisms of the Entity Framework (EF) to combine XML and relational mappings. For example, we exploit user-defined functions to tunnel XML queries through the query pipeline. We also use the EF's query rewriting mechanism to return nested collections in query results, which enables

[1]An ADO.NET Data Provider translates queries represented in an abstract syntax into the native SQL dialect (and XML features) supported by the database system.



CLR (.Net) Objects
XML-to-object mappings + Entity-to-object mappings
Entity Model
Extended entity-to-relational mappings (from ADO.NET Entity Framework)
DB
Mappings
LINQ Query
Translate references to XML-mapped members and methods into XQuery
Query Tree
Translate tree according to entity-to-relational mappings, then into vendor-specific syntax
SQL + XQuery
Query Processing
Objects
Shred XML data into strongly-typed objects using XML-to-object mappings
Objects
Transform tuples into objects using entity-to-relational mapping; nest collections
Tuples
Result Materialization
Objects
Δ
Package changed/added objects into XML; mark as changed
Change List
Translate change list into DML statements, and pass through entity-to-relational mappings
DML
Update Propagation

Figure 3: A high-level view of our implementation

convenient query formulation over XML content. For instance, the query

```
from c in db.JobCandidates select c.Resume.Address
```

returns a collection of collections of XElements. The EF automatically flattens such queries prior to execution in the database and nests their results on the client. Not least, we directly leverage EF's built-in support for .NET functions and data types, such as DateTime.

The query and update translation performed at runtime is decoupled from the schema translation algorithm used by the tool, which is essential to support schema evolution and advanced mapping scenarios that require data reshaping. Once we have defined or generated a set of classes, we can construct LINQ expressions that reference both classes that map to relational types and classes that map to XML types.

We also perform some optimization during query processing. For instance, we collapse multiple XQuery fragments from nested SQL queries into a single XQuery expression if possible. Also, note that the example query from Section 2.1 could be expressed almost entirely in XQuery; our translation algorithm often uses relational operators to leverage the relational capabilities of the query processor and to support queries that span both relational and XML data.

4. RELATED WORK

Several tools are currently available that provide access to XML documents as strongly-typed objects. XML Beans [10] can expose XML documents as typed Java objects, while Liquid XML [6] can expose XML documents as typed objects in a variety of languages. Finally, a strongly-typed LINQ interface to XML was proposed in the initial LINQ-to-XSD work of Lämmel et al [4] 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 each use a fixed mapping that cannot be controlled by the user. Our work addresses both of these limitations. To the best of our knowledge, LINQ-to-Stored XML is the first system that supports accessing typed XML stored in a database through LINQ. Our XML-to-object mappings can be generated, but are not hardwired and can be edited at design-time. We believe these features allow persistent XML programming to be accessible to a larger developer audience.

Tools such as XJ [3] and LINQ-to-XML offer an XPath-like interface to loosely-typed XML objects in a programming environment. These tools are also limited to manipulating XML in memory.

Our work was introduced at the *XML'07* conference [8] as a future direction for LINQ-to-XSD but has never been demonstrated previously. Our proposed demonstration shows an internal Microsoft prototype and does not imply any product commitments.

5. REFERENCES

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

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

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

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

[5] R. Lämmel, E. Meijer. Revealing the X/O Impedance Mismatch (Changing Lead into Gold). In *Datatype-Generic Programming*, Lecture Notes in Computer Science. Springer-Verlag, June 2007.

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

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

[8] S. Pather. LINQ to XML: Visual Studio 2008, Silverlight, and Beyond. XML '07, http://2007.xmlconference.org/public/schedule/detail/369.

[9] S. Resnick, R. Crane, C. Bowen. *Essential Windows Communication Foundation (WCF): For .NET Framework 3.5*. Addison Wesley, 2008.

[10] XML Beans. http://xmlbeans.apache.org/.