ADO.NET Entity Framework: Raising the Level of Abstraction in Data Programming

Pablo Castro
Microsoft Corporation
One Microsoft Way
Redmond, WA, 98052, USA
pablo.castro@microsoft.com

Sergey Melnik
Microsoft Research
One Microsoft Way
Redmond, WA, 98052, USA
sergey.melnik@microsoft.com

Atul Adya
Microsoft Corporation
One Microsoft Way
Redmond, WA, 98052, USA
adya@microsoft.com

ABSTRACT

The ADO.NET Entity Framework provides a persistence layer for .NET applications that allows developers to work at a higher level of abstraction when interacting with data and data-access interfaces. Developers can model and access their data using a conceptual schema that is mapped to a relational database via a flexible mapping. Interaction with the data can take place using a SQL-based data manipulation language and iterator APIs, or through an object-based domain model in the spirit of object-to-relational mappers.

We demonstrate how the Entity Framework simplifies application development using sample scenarios. We illustrate how the data is modeled, queried and presented to the developer. We also show how the provided data programming infrastructure can result in easier-to-understand code by making its intent more explicit, as well as how it can help with maintenance by adding a level of indirection between the logical database schema and the conceptual model that applications operate on.

Categories and Subject Descriptors:
H.2 [Database Management], D.3 [Programming Languages]

General Terms: Algorithms, Management, Design, Languages

Keywords: Data Programming, Conceptual Modeling, ADO.NET

1. INTRODUCTION

A large number of commercial software applications written today are data-centric applications. Companies usually have several database server systems managing their business and operational information, and several applications built on top of them. Application developers in IT departments and software development companies face the challenge of creating and evolving data-centric applications in a fast and cost-effective manner.

Some of these challenges can be addressed with tools that offer object-to-relational mapping (ORM) capabilities [4], although the majority of these tools are focused primarily on binding relational data to objects in specific programming languages and are not well-suited for general-purpose database development.

The ADO.NET Entity Framework [1] is a technology designed to elevate the level of abstraction at which application developers work when creating and maintaining data-centric applications. To achieve this goal, it focuses on three main areas: a) a higher-level data model for applications to operate on, b) an object services layer that exposes the application data through an object-oriented interface and processes create/read/update/delete operations on objects, and c) support for the language-integrated query (LINQ [5]) mechanism in the upcoming version of C# and Visual Basic.

Today, most enterprise data is stored in relational databases. The Entity Framework provides a flexible mechanism for mapping higher-level application models to existing relational schemas. It supports various persistence strategies and helps build new applications on top of legacy databases. A prerelease of the Entity Framework is available for download.

SIGMOD'07, June 11–14, 2007, Beijing, China.
ACM 978-1-59593-686-8/07/0006.

2. WHAT IS DEMONSTRATED

We demonstrate how the ADO.NET Entity Framework addresses various real-world issues that application developers often encounter when creating data-centric applications.

2.1 Working with a higher-level data model

In this section we demonstrate how a higher-level data model can help express the application semantics more explicitly. We start with the relational database schema shown in Figure 1.

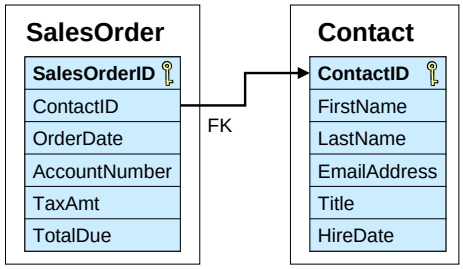

Figure 1: Sample relational schema

Consider a hypothetical application that manages two kinds of sales orders: online orders and those from retail stores. A traditional way of discriminating the orders stored in a relational table is by convention. For example, if TaxAmt is null then the table row represents an online order. When the application wants to list all the store sales orders, it can use code like this (in C# and a previous version of ADO.NET):

```
void PrintOrders(DateTime date) {
  using(SqlConnection con = new
            SqlConnection(CONNSTRING_SQL)) {
    con.Open();

    SqlCommand cmd = con.CreateCommand();
    cmd.CommandText = @"
      SELECT o.OrderDate, o.TotalDue
      FROM SalesOrder AS o
      WHERE o.TaxAmt IS NOT NULL
      AND o.OrderDate > @date";
```

```
    cmd.Parameters.AddWithValue("date",
                                date);

    DbDataReader r = cmd.ExecuteReader(
          CommandBehavior.SequentialAccess);
    while(r.Read())
      Console.WriteLine("{0:d}:\t{1}",
            r["OrderDate"], r["TotalDue"]);
  }
}
```

While the SQL query shown above is relatively simple, its semantics is not obvious. Specifically, "o.TaxAmt IS NOT NULL" actually means "a store sales order"; that meaning needs to be documented externally as it cannot be derived from the query without the appropriate context.

The ADO.NET Entity Framework operates on a higher-level entity-relationship model called the Entity Data Model (EDM) [3], where Entities are a first-class concept of the system. An entity in EDM is a structure with a key. It can surface in various ways in the programming model (objects, rows/columns, etc.). Entities are instances of Entity Types, and are contained in Entity Sets (somewhat analogous to tables). Entity Types can be derived from other Entity Types enabling structural inheritance. EDM also has an explicit concept of an Association that goes beyond a foreign key constraint in relational schemas and can be used to navigate between entities in queries and in other contexts.

One possible representation of the relational sales data in EDM terms is shown in Figure 2. The EDM schema contains roughly the same elements as the relational schema shown in Figure 1, but uses inheritance to model SalesOrders in general and StoreSalesOrders as a subtype. The EDM schema also introduces an association between SalesOrder and Contact. The SalesDB element shown in Figure 2 represents the container for the entity sets Contacts (which contains instances of Contact) and SalesOrders (which contains instances of SalesOrder and its subtype StoreSalesOrder).

In addition to the EDM schema, the system needs information that describes how the various elements of the EDM schema *map* to the underlying relational database that contains the data. That is done through a *mapping specification*. The mapping is specified using an XML file that can be authored by hand or using a visual

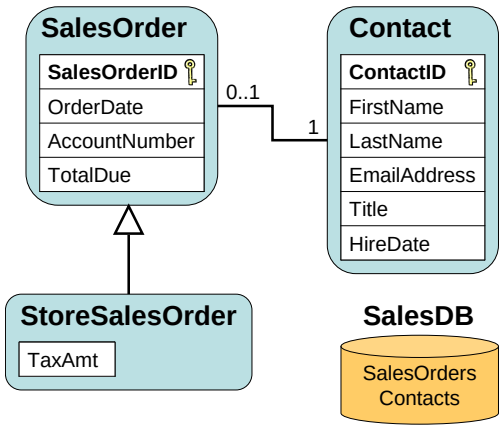

Figure 2: Sample EDM schema

tool, such as the one described in [2], and is compiled into data transformations that drive query and update processing [6].

For this example, we define a mapping specification that maps contacts using a trivial 1:1 mapping. It maps sales orders using a discriminating condition that tells the system that those rows with a null value in TaxAmt should be mapped to instances of the SalesOrder type, and the rest to instances of the StoreSalesOrder type.

Now the developer can target the conceptual model in the application. One way of incrementally adopting this model while leveraging the coding patterns established in previous versions of ADO.NET is by using the *EntityClient provider*, an ADO.NET data-access provider that operates at the conceptual level and uses *Entity SQL* (a SQL dialect) as its query language. For example (in C# along with the ADO.NET Entity Framework):

```
void PrintOrders(DateTime date) {
  using(EntityConnection con = new
          EntityConnection(CONNSTRING_EDM)) {
    con.Open();

    EntityCommand cmd = con.CreateCommand();
    cmd.CommandText = @"
      SELECT o.OrderDate, o.TotalDue
      FROM Sales.SalesDB.SalesOrders AS o
      WHERE o IS OF (Sales.StoreSalesOrder)
      AND o.OrderDate > @date";
    cmd.Parameters.AddWithValue("date",
                                date);

    DbDataReader r = cmd.ExecuteReader(
          CommandBehavior.SequentialAccess);
    while(r.Read())
      Console.WriteLine("{0:d}:\t{1}",
            r["OrderDate"], r["TotalDue"]);
  }
}
```

Note that the usage pattern for the new API is identical to that of previous releases, helping with the learning curve. Also, the developer's intent can now be clearly understood from the formulation of the query; specifically, the query asks for sales orders *o* such that "*o* IS OF (Sales.StoreSalesOrder)", i.e., for orders that are instances of the StoreSalesOrder entity type.

2.2 Isolation from schema changes

Here we demonstrate how having a rich mapping layer between the application's conceptual model and the database schema introduces a new level of data independence and helps with schema evolution in certain scenarios.

While some ORMs can already do this to varying degrees, they focus on mapping to objects; in contrast, the Entity Framework provides a general mechanism that can be used regardless of the choice of data access interface.

We present an example of database refactoring that affects the schema in a way that would break a traditional database application, and show how the mapping infrastructure can help avoid a code change in the application.

Suppose that the Contact table was a very large table and had a large number of contacts, not all of them sales people; in order to increase the row density in the contacts table and reduce the disk I/O for certain workloads, the database administrator decides to

vertically partition the table by adding a new table SalesPerson with a foreign key into Contact, which preserves the existing keys.

An application using the EDM schema shown in Figure 2 can use the following query to find the names of the sales people hired after a given date, regardless of how they are mapped to the underlying tables:

```
SELECT c.FirstName, c.LastName
FROM Sales.SalesDB.Contact AS c
WHERE c.HireDate > @date
```

After the database refactoring the developer has to adjust the mapping specification to tell the system that now the Contact entities are created by joining the Contact and SalesPerson tables and extracting the desired properties from these tables.

Note that the conceptual schema and the application code (including queries) were not affected at all by this change. The system will exploit the new mapping to query and update the refactored tables. No database views or triggers need to be created or modified.

2.3 Presenting data as objects

The examples shown in Sections 2.1 and 2.2 execute queries against the conceptual model and return values as rows and columns using the DataReader API construct. We now discuss how to use .NET objects instead of rows and columns to represent entities.

The ADO.NET Entity Framework provides an Object Services layer, which enables developers to use regular .NET objects to interact with the data, both for retrieval and updates. The tools included with the Entity Framework automatically generate .NET classes to represent each declared Entity Type.

There are many options for incrementally layering the object services on top of the EntityClient provider; in the interest of brevity, we show here the simplest option where EntityClient is used internally and is set up automatically by the system. To follow the running example, the code excerpt below is equivalent to the one used in Section 2.1 and obtains sales orders that were posted via a retail store after a certain date:

```
void PrintOrders(DateTime date) {
  using(SalesDB db = new SalesDB()) {

    ObjectQuery<SalesOrder> orders =
           db.CreateQuery<SalesOrder>(@"
   SELECT VALUE o
   FROM Sales.SalesDB.SalesOrders AS o
   WHERE o IS OF (Sales.StoreSalesOrder)
   AND o.OrderDate > @date",
     new ObjectParameter("date", date));

    foreach(SalesOrder o in orders)
        Console.WriteLine("{0:d}:\t{1}",
              o.OrderDate, o.TotalDue);
  }
}
```

The "VALUE" keyword in the select clause eliminates the row-wrapper that otherwise would be generated by the system if we simply had "o" in the projection list.

While this version of the code has identical functionality to the earlier one, the actual code has significantly fewer database-specific constructs; it does not explicitly create and initialize a connection, nor does it need configuration information inside the program. All of this information is captured during code generation and stored in external configuration files. Also, the query results are .NET objects and not rows and columns.

2.4 Language-integrated query

Most current data-access libraries used in commercial applications expect SQL queries as strings. Having SQL queries be represented as strings means that the compiler cannot help the developer with compile-time checking of syntactic and semantic correctness like it does for the rest of the program.

Language-integrated query [5], or LINQ for short, is an innovation in the programming languages space that introduces query-related constructs to mainstream programming languages such as C# and Visual Basic. The query constructs are not processed by an external tool but instead are first-class type-checked expressions of the language itself.

The ADO.NET Entity Framework is fully integrated with LINQ. Developers can formulate queries against the conceptual model using the language constructs for writing queries. For example:

```
void PrintOrders(DateTime date) {
  using(SalesDB db = new SalesDB()) {

    var orders = from o in db.SalesOrders
                 where o is StoreSalesOrder
                       && o.OrderDate > date
                 select o;

    foreach(SalesOrder o in orders)
      Console.WriteLine("{0:d}:\t{1}",
                  o.OrderDate, o.TotalDue);
  }
}
```

In this example the query is expressed using the constructs of the C# language so the compiler can verify the syntax and the semantic correctness of the query during compilation.

3. ACKNOWLEDGEMENTS

We are grateful to the dozens of engineers on the ADO.NET team who helped build the technology that we demonstrate here.

4. REFERENCES

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

[2] P. A. Bernstein, S. Melnik, J. E. Churchill. Incremental Schema Matching. In *VLDB*, 2006

[3] J. A. Blakeley, S. Muralidhar, A. Nori. The ADO.NET Entity Framework: Making the Conceptual Level Real. In *ER*, 2006

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

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

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