Anatomy of the ADO.NET Entity Framework

Atul Adya, José A. Blakeley, Sergey Melnik, S. Muralidhar, and
the ADO.NET Team
Microsoft Corporation
One Microsoft Way, Redmond, WA 98052-6399
{adya,joseb,melnik,smurali}@microsoft.com

ABSTRACT

Traditional client-server applications relegate query and persistence operations on their data to database systems. The database system operates on data in the form of rows and tables, while the application operates on data in terms of higher-level programming language constructs (classes, structures etc.). The *impedance mismatch* in the data manipulation services between the application and the database tier was problematic even in traditional systems. With the advent of service-oriented architectures (SOA), application servers and multi-tier applications, the need for data access and manipulation services that are well-integrated with programming environments and can operate in any tier has increased tremendously.

Microsoft's ADO.NET Entity Framework is a platform for programming against data that raises the level of abstraction from the relational level to the conceptual (entity) level, and thereby significantly reduces the impedance mismatch for applications and data-centric services. This paper describes the key aspects of the Entity Framework, the overall system architecture, and the underlying technologies.

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

Modern applications require data management services in all tiers. They need to handle increasingly richer forms of data which includes not only structured business data (such as Customers and Orders), but also semi-structured and unstructured content such as email, calendars, files, and documents. These applications need to integrate data from multiple data sources as well as to collect, cleanse, transform and store this data to enable a more agile decision making process. Developers of these applications need data access, programming and development tools to increase their productivity. While relational databases have become the de facto store for most structured data, there tends to be a mismatch—the well-known *impedance mismatch* problem—between the data model (and capabilities) exposed by such databases, and the modeling capabilities and programmability needed by applications.

Two other factors also play an important part in enterprise system design. First, the data representation for applications tends to evolve differently from that of the underlying databases. Second, many systems comprise disparate database back-ends with differing degrees of capability, and consequently, the application logic in the mid-tier is responsible for reconciling these differences, and presenting a more uniform view of data. The data transformations required by applications may quickly grow complex. Implementing such transformations, especially when the underlying data needs to be updatable, is a hard problem and adds complexity to the application. A significant portion of application development—up to 40% in some cases [26]—is dedicated to writing custom data access logic to work around these problems.

The same problems exist, and are no less severe, for data-centric services. Conventional services such as query, updates, and transactions have been implemented at the logical schema (relational) level. However, the vast majority of newer services, such as replication and analysis, best operate on artifacts typically associated with a higher-level, conceptual data model. As with applications, each service typically ends up building a custom solution to this problem, and consequently, there is code duplication and little interoperability between these services.

Middleware mapping technologies such as Hibernate [8] and Oracle TopLink [33] are a popular alternative to custom data access logic. The mappings between the database and applications are expressed in a custom structure, or via schema annotations. While the mappings provide a degree of independence between the database and the application, the problem of handling multiple applications with slightly differing views of the same data, or of the needs of services which tend to be more dynamic are not well addressed by these solutions.

This paper describes the ADO.NET Entity Framework, a platform for programming against data that significantly reduces the impedance mismatch for applications and data-centric services. It differs from other systems and solutions in the following regards:

- The Entity Framework defines a rich, value-based conceptual data model (the Entity Data Model, or the EDM), and a new data manipulation language (Entity SQL) that operates on instances of this model.
- This model is made *concrete* by a runtime that includes a middleware mapping engine supporting powerful

SIGMOD'07, June 11–14, 2007, Beijing, China.

bidirectional (EDM–Relational) mapping, queries and updates.

- Applications and services may program directly against the value-based conceptual layer, or against programming-language-specific object abstractions that may be layered over the conceptual (entity) abstraction, providing object-relational mapping (ORM) functionality. We believe that an EDM conceptual abstraction is a more flexible basis for sharing data among applications and data-centric services than objects.
- Finally, the Entity Framework leverages Microsoft's new Language Integrated Query (LINQ) technologies that extend programming languages natively with query expressions to further reduce, and for some scenarios completely eliminate, the impedance mismatch for applications.

The ADO.NET Entity Framework will be part of the upcoming .NET Framework release. A community technology preview (CTP) is available at http://www.microsoft.com/downloads/.

The rest of this paper is organized as follows. Section 2 provides additional motivation for the Entity Framework. Section 3 presents the Entity Framework and the Entity Data Model. Section 4 describes programming patterns for the Entity Framework. Section 5 outlines the Object Services module. Section 6 focuses on the Mapping component of the Entity Framework, while Sections 7 and 8 explain how queries and updates are handled. Sections 9 and 10 describe the metadata subsystem and the tools components of the Entity Framework. Section 11 discusses related work, and Section 12 provides a summary of this paper.

2. MOTIVATION

This section discusses why a higher level data modeling layer has become essential for applications and data-centric services.

Today's dominant data modeling methodologies factor a data model into four main levels: Physical, Logical (Relational), Conceptual, and Programming/Presentation.

The ***physical*** model describes how data is *represented* in physical resources such as memory, wire or disk, and deals with concepts like record formats, file partitions and groups, heaps, and indexes.

A ***logical*** (relational) data model aims to capture the entire data content of the target domain using logical concepts such as tables, rows, primary-key/foreign-key constraints, and normalization. While normalization helps to achieve data consistency, increased concurrency, and better OLTP performance, it also introduces significant challenges for applications. Normalized data at the logical level is often too fragmented and application logic needs to assemble rows from multiple tables into higher level entities that more closely resemble the artifacts of the application domain.

The ***conceptual*** model captures the core information entities from the problem domain and their relationships. A well-known conceptual model is the Entity-Relationship Model introduced by Peter Chen in 1976 [13]. UML is a more recent example of a conceptual model [36]. Most applications involve a conceptual design phase early in the application development lifecycle. Unfortunately, however, the conceptual data model diagrams stay "pinned to a wall" growing increasingly disjoint from the reality of the application implementation with time. An important goal of the Entity Framework is to make the conceptual data model (embodied by the Entity Data Model, described in Section 3.2) a concrete, executable abstraction of the data platform.

The ***programming/presentation*** model describes how the entities and relationships of the conceptual model need to be manifested (presented) in different forms based on the task at hand. Some entities need to be transformed into programming language objects to implement application business logic; others need to be transformed into XML streams for web service invocations; still others need to be transformed into in-memory structures such as lists or dictionaries for the purposes of user-interface data binding. Naturally, there is no universal programming model or presentation form; thus, applications need flexible mechanisms to transform entities into the various presentation forms.

The physical, logical and programming layers described above correspond to the internal, conceptual and external levels of the ANSI/SPARC database system architecture [2]. The Entity Framework introduces a new "conceptual" level based on the EDM between the relational and the presentation levels. This new conceptual model describes data at a higher-level of abstraction than the relational model and its aim is to represent data in terms that are closer to the programming artifacts used by applications.

Most applications and data-centric services would like to reason in terms of high-level concepts such as an *Order*, not about the several tables that an order may be normalized over in a relational database schema. An order may manifest itself in multiple fashions—we believe there is no "one proper presentation model"; the real value is in providing a *concrete conceptual model*, and then being able to use that model as the basis for flexible mappings to and from various presentation models and other higher level data services.

3. THE ENTITY FRAMEWORK

Microsoft's existing ADO.NET framework is a data-access technology that enables applications to connect to data stores and manipulate data contained in them in various ways. It is part of the Microsoft .NET Framework and it is highly integrated with the rest of the Framework class library. The ADO.NET framework has two major parts: *providers* and *services*. ADO.NET *providers* are the components that know how to talk to specific data stores. Providers comprise three core pieces of functionality. *Connections* manage access to the underlying data source; *Commands* represent a command (query, procedure call, etc.) to be executed against the data source; *DataReaders* represent the result of command execution. ADO.NET *services* include provider-neutral components such as DataSet to enable offline data programming scenarios. (A DataSet is a memory-resident representation of data that provides a consistent relational programming model regardless of the data source.)

3.1 Entity Framework – Key Functionality

The ADO.NET Entity Framework builds on the existing ADO.NET *provider* model, and adds the following functionality.

- A new conceptual data model, the *Entity Data Model* (EDM) [18], to help model conceptual schemas.
- A new data manipulation language (DML), *Entity SQL*, to query and update instances of the EDM, and a programmatic

representation of a query (canonical command trees) to communicate with different providers.
- The ability to define mappings between the conceptual schema and the logical schemas.
- An ADO.NET provider programming model against the conceptual schema.
- An object services layer to provide ORM-like functionality in all supported .NET languages.
- Integration with LINQ technology to make it easy to program against data as objects from .NET languages.

The Entity Framework currently focuses mostly on data in relational systems; future releases will tackle data from other sources as well.

3.2 The Entity Data Model

The Entity Data Model (EDM) is intended for developing rich data-centric applications. It extends the classic relational model with concepts from the E-R domain. The central concepts in the EDM are entities and associations. *Entities* represent top-level items with identity, while *Associations* are used to *relate* (or, describe relationships between) two or more entities.

An important aspect of EDM is that it is *value-based* like the relational model (and SQL), rather than *object/reference-based* like C# (CLR). Several object programming models can be easily layered on top of the EDM. Similarly, the EDM can map to one or more DBMS implementations for persistence.

The EDM and Entity SQL represent a richer data model and data manipulation language for a data platform and are intended to enable applications such as CRM and ERP, data-intensive services such as Reporting, Business Intelligence, Replication and Synchronization, and data-intensive applications to model and manipulate data at a level of structure and semantics that is closer to their needs. We discuss briefly the core concepts of the EDM; more details are available in [6].

EDM Types

An ***EntityType*** describes the structure of an entity. An entity may have one or more properties (attributes, fields) that describe the structure of the entity. Additionally, an entity type must define a *key*—a set of properties whose values uniquely identify the entity instance within a collection of entities. An EntityType may derive from (or subtype) another entity type; the EDM supports a single inheritance model. The properties of an entity may be simple or complex types. A ***SimpleType*** represents scalar (or atomic) types (e.g., integer, string), while a ***ComplexType*** represents structured properties (e.g., an Address). A ComplexType is composed of zero or more properties, which may themselves be scalar or complex type properties. An AssociationType describes an association relationship between two (or more) entity types. EDM *Schemas* provide a grouping mechanism for types—types must be defined in a schema. An application may reference multiple schemas. The namespace of the schema combined with the type name uniquely identifies the specific type in the context of an application.

Future versions of the EDM will provide support for URIs, relationships with properties, nested collections and other capabilities.

EDM Instance Model

Entity instances (or just entities) are logically contained within an ***EntitySet***. An EntitySet is a homogenous collection of entities, i.e., all entities in an EntitySet must be of the same (or derived) EntityType. An EntitySet is conceptually similar to a database table, while an entity is similar to a row of a table. (Views will be supported in a subsequent release.) An entity instance must belong to exactly one entity set. In a similar fashion, association instances are logically contained within an ***AssociationSet***. The definition of an AssociationSet scopes the relationship, i.e., it identifies the EntitySets that hold instances of the entity types that participate in the relationship. An AssociationSet is conceptually similar to a link-table in a relational database. SimpleTypes and ComplexTypes can only be instantiated as properties of an EntityType. An *EntityContainer* is a logical grouping of EntitySets and AssociationSets—akin to how a Schema is a grouping mechanism for EDM types. As with schemas, an application may reference multiple EntityContainers.

An Example EDM Schema

A sample EDM schema is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<Schema Namespace="AdventureWorks" Alias="Self" …>
  <EntityContainer Name="AdventureWorksContainer">
    <EntitySet Name="ESalesOrders"
               EntityType="Self.ESalesOrder" />
    <EntitySet Name="ESalesPersons"
               EntityType="Self.ESalesPerson" />
    <AssociationSet Name="ESalesPersonOrders"
           Association="Self.ESalesPersonOrder">
      <End Role="ESalesPerson"
           EntitySet="ESalesPersons" />
      <End Role="EOrder" EntitySet="ESalesOrders" />
    </AssociationSet>
  </EntityContainer>

  <!-- Sales Order Type Hierarchy-->
  <EntityType Name="ESalesOrder" Key="Id">
    <Property Name="Id" Type="Int32"
              Nullable="false" />
    <Property Name="AccountNum" Type="String"
              MaxLength="15" />
  </EntityType>
  <EntityType Name="EStoreSalesOrder"
              BaseType="Self.ESalesOrder">
    <Property Name="Tax" Type="Decimal"
              Precision="28" Scale="4" />
  </EntityType>

  <!-- Person EntityType -->
  <EntityType Name="ESalesPerson" Key="Id">
    <!-- Properties from SSalesPersons table-->
    <Property Name="Id" Type="Int32"
              Nullable="false" />
    <Property Name="Bonus" Type="Decimal"
              Precision="28" Scale="4" />
    <!-- Properties from SEmployees table-->
    <Property Name="Title" Type="String"
              MaxLength="50" />
    <Property Name="HireDate" Type="DateTime" />
    <!-- Properties from the SContacts table-->
    <Property Name="Name" Type="String"
              MaxLength="50" />
    <Property Name="Contact" Type="Self.ContactInfo"
              Nullable="false" />
  </EntityType>
  <ComplexType Name="ContactInfo">
    <Property Name="Email" Type="String"
              MaxLength="50" />
    <Property Name="Phone" Type="String"
              MaxLength="25" />
  </ComplexType>
```

```
  <Association Name="ESalesPersonOrder">
    <End Role="EOrder" Type="Self.ESalesOrder"
         Multiplicity="*" />
    <End Role="ESalesPerson" Multiplicity="1"
         Type="Self.ESalesPerson" />
  </Association>
</Schema>
```

The AdventureWorks schema describes order and salesperson entities via the entity type definitions for ESalesPerson, ESalesOrder (and the EStoreSalesOrder subtype). The ESalesPersonOrder association type describes a 0..1 : N association relationship between ESalesPerson and ESalesOrder.

The AdventureWorksContainer container defines entity sets for ESalesOrders (holding instances of ESalesOrder and EStoreSalesOrder) and ESalesPersons (holding instances of ESalesPerson). Additionally, the ESalesPersonOrders association set is also defined (to hold instances of the ESalesPersonOrder association), and is scoped to the ESalesOrders and ESalesPersons entity sets.

3.3 High-Level Architecture

This section outlines the architecture of the ADO.NET Entity Framework. Its main functional components are (see Figure 1):

Data source-specific providers. The Entity Framework builds on the ADO.NET data provider model. There are specific providers for several relational, non-relational, and Web services sources, although the focus currently is on relational providers.

EntityClient provider. The EntityClient provider represents a concrete conceptual programming layer. It is a new, value-based data provider where data is accessed in terms of EDM entities and associations and is queried/updated using an entity-based SQL language (Entity SQL). The EntityClient provider includes a view mapping subsystem that supports updatable EDM views over flat relational tables. The mapping between tables and entities is specified declaratively via a mapping specification language.

Object Services and other Programming Layers. The Object Services component of the Entity Framework provides a rich object abstraction over entities, a rich set of services over these objects, and allows applications to program using familiar programming language constructs. This component provides state management services for objects (such as change tracking, supports services for navigating and loading objects and relationships, supports queries via LINQ and Entity SQL, and allows objects to be updated, and persisted.

The Entity Framework allows multiple programming layers to be plugged onto the value-based entity data services layer exposed by the EntityClient provider. The Object Services component is one such programming layer that surfaces CLR objects, and provides ORM-like functionality.

The Metadata services component manages metadata for the design time and runtime needs of the Entity Framework, and applications over the Entity Framework. All metadata associated with EDM concepts (entities, relationships (associations), EntitySets, AssociationSets), store concepts (tables, columns, constraints), and mapping concepts are exposed via metadata interfaces. The metadata component also serves as a link between the domain modeling tools which support model-driven application design.

Design and Metadata Tools. The Entity Framework integrates with domain designers to enable model-driven application development. The tools include EDM design tools, mapping design tools, code generation tools, and query modelers.

Services. Rich data-centric services such as Reporting, Synchronization and Business Analysis can be built using the Entity Framework.

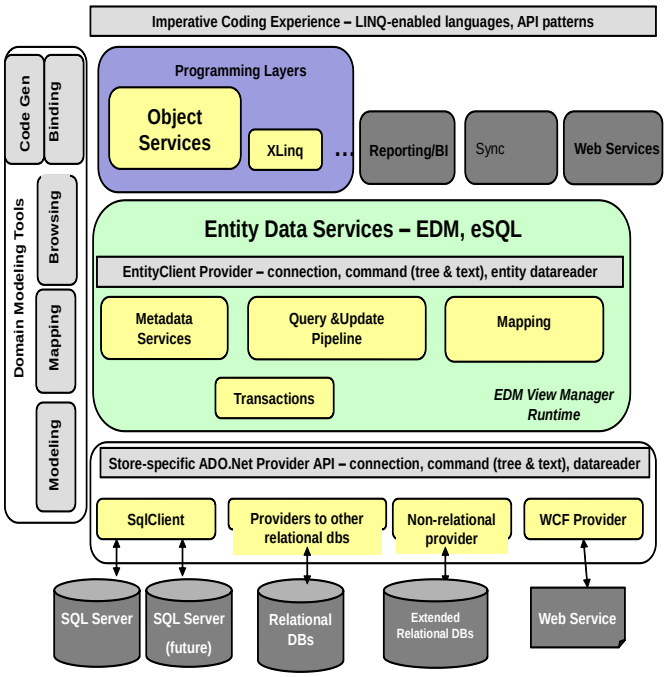

Figure 1: Entity Framework Architecture

4. PROGRAMMING PATTERNS

The ADO.NET Entity Framework together with LINQ increases application developer productivity by significantly reducing the impedance mismatch between application code and data. In this section we describe the evolution in data access programming patterns at the logical, conceptual and object abstraction layers.

Consider the following relational schema fragment based on the sample AdventureWorks database. This database consists of the following tables.

```
create table SContacts(ContactId int primary key,
    Name varchar(100), Email varchar(100),
    Phone varchar(10));
create table SEmployees(
    EmployeeId int primary key
        references SContacts(ContactId),
    Title varchar(20), HireDate date);
create table SSalesPersons(
    SalesPersonId int primary key
        references SEmployees(EmployeeId),
    Bonus int);
create table SSalesOrder(
    SalesOrderId int primary key,
    SalesPersonId int
        references SSalesPersons(SalesPersonId));
```

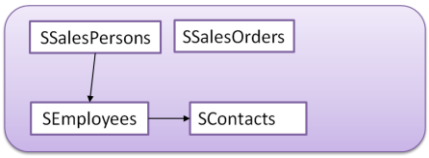

Figure 2: Sample Relational Schema

Consider an application code fragment to obtain the name, the id and the hire date for all salespersons who were hired prior to some date. There are four main shortcomings in this code fragment that have little to do with the business question that needs to be answered. First, even though the query can be stated in English very succinctly, the SQL statement is quite verbose and requires the developer to be aware of the normalized relational schema to formulate the multi-table join required to collect the appropriate columns from the SContacts, SEmployees, and SSalesPerson tables. Additionally, any change to the underlying database schemas will require corresponding changes in the code fragment below. Second, the user has to define an explicit connection to the data source. Third, since the results returned are not strongly typed, any reference to non-existing columns names will be caught only after the query has executed. Fourth, the SQL statement is a string property to the Command API and any errors in its formulation will only be caught at execution time. While this code is written using ADO.NET 2.0, the code pattern and its shortcomings applies to any other relational data access API such as ODBC, JDBC, or OLE-DB.

```
void EmpsByDate(DateTime date) {
using( SqlConnection con =
  new SqlConnection (CONN_STRING) ) {
    con.Open();
    SqlCommand cmd = con.CreateCommand();
    cmd.CommandText = @"
    SELECT SalesPersonID, FirstName, HireDate
    FROM SSalesPersons sp
      INNER JOIN SEmployees e
      ON sp.SalesPersonID = e.EmployeeID
      INNER JOIN SContacts c
      ON e.EmployeeID = c.ContactID
    WHERE e.HireDate < @date";
    cmd.Parameters.AddWithValue("@date",date);

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

The sample relational schema can be captured at the conceptual level via the EDM as illustrated in Figure 3 that defines an entity type ESalesPerson that abstracts out the fragmentation of SContacts, SEmployees, and SSalesPersons tables. It also captures the inheritance relationship between the EStoreSalesOrder and ESalesOrder entity types.

Figure 3: Sample EDM schema

The equivalent program at the conceptual layer is written as follows:

```
void EmpsByDate (DateTime date) {
using( EntityConnection con =
  new EntityConnection (CONN_STRING) ) {
    con.Open();
    EntityCommand cmd = con.CreateCommand();
    cmd.CommandText = @"
        SELECT VALUE sp
        FROM ESalesPersons sp
        WHERE sp.HireDate < @date";
    cmd.Parameters.AddWithValue ("@date",
        date);
    DbDataReader r = cmd.ExecuteReader();
    while (r.Read()) {
      Console.WriteLine("{0:d}:\t{1}",
        r["HireDate"], r["FirstName"])
} } }
```

The SQL statement has been considerably simplified—the user no longer has to know the precise database layout. Furthermore, the application logic can be isolated from changes to the underlying database schema. However, this fragment is still string-based, still does not get the benefits of programming language type-checking, and returns weakly typed results.

By adding a thin object wrapper around entities and using the Language Integrated Query (LINQ) extensions in C#, one can rewrite the equivalent function with no impedance mismatch as follows:

```
void EmpsByDate(DateTime date) {
  using (AdventureWorksDB aw =
    new AdventureWorksDB()) {
    var people = from p in aw.SalesPersons
                 where p.HireDate < date
                 select p;
    foreach (SalesPerson p in people) {
      Console.WriteLine("{0:d}\t{1}",
                p.HireDate, p.FirstName );
} } }
```

The query is simple; the application is (largely) isolated from changes to the underlying database schema; and the query is fully type-checked by the C# compiler. In addition to queries, one can interact with objects and perform regular Create, Read, Update and Delete (CRUD) operations on the objects. Examples of these are described in Section 8.

5. OBJECT SERVICES

The Object Services component is a programming/presentation layer over the conceptual (entity) layer. It houses several components that facilitate the interaction between the programming language and the value-based conceptual layer entities. We expect one object service to exist per programming language runtime (e.g., .NET, Java). Currently, we support the .NET CLR which enables programs in any .NET language to interact with the Entity Framework. Object Services comprises the following major components:

The ***ObjectContext*** class houses the database connection, metadata workspace, object state manager, and object materializer. This class includes an object query interface

ObjectQuery<T> to enable the formulation of queries in either Entity SQL or LINQ syntax, and returns strongly-typed object results as an *ObjectReader<T>*. The ObjectContext also exposes query and update (i.e., *SaveChanges*) object-level interfaces between the programming language layer and the conceptual layer. The ***Object state manager*** has three main functions: (a) cache query results and manage policies to merge objects from overlapping query results, (b) track in-memory changes, and (c) construct the change list input to the processing infrastructure (see Sec. 8). The object state manager maintains the state of each entity in the cache—*detached* (from the cache), *added*, *unchanged*, *modified*, and *deleted*—and tracks their state transitions. The ***Object materializer*** performs the transformations during query and update between entity values from the conceptual layer and the corresponding CLR objects.

6. MAPPING

The backbone of a general-purpose data access layer such as the ADO.NET Entity Framework is a *mapping* that establishes a relationship between the application data and the data stored in the database. An application queries and updates data at the object or conceptual level and these operations are translated to the store via the mapping. There are a number of technical challenges that have to be addressed by any mapping solution. It is relatively straightforward to build an ORM that uses a one-to-one mapping to expose each row in a relational table as an object, especially if no declarative data manipulation is required. However, as more complex mappings, set-based operations, performance, multi-DBMS-vendor support, and other requirements weigh in, ad hoc solutions quickly grow out of hand.

6.1 Problem: Updates via Mappings

The problem of accessing data via mappings can be modeled in terms of "views", i.e., the objects/entities in the client layer can be considered as rich views over the table rows. However, it is well known that only a limited class of views is updateable, e.g., commercial database systems do not allow updates to multiple tables in views containing joins or unions. The work in [16] observed that finding a unique update translation over even quite simple views is rarely possible due to the intrinsic under-specification of the update behavior by a view. Subsequent research has shown that teasing out the update semantics from views is hard and can require significant user expertise [4]. However, for mapping-driven data access, it is imperative that there exists a well-defined translation of every update to the view.

Furthermore, in mapping-driven scenarios, the updatability requirement goes beyond a single view. For example, a business application that manipulates Customer and Order entities effectively performs operations against two views. Sometimes a consistent application state can only be achieved by updating several views simultaneously. Case-by-case translation of such updates may yield a combinatorial explosion of the update logic. Delegating its implementation to application developers is extremely unsatisfactory because it requires them to manually tackle one of the most complicated parts of data access.

6.2 The ADO.NET Mapping Approach

The ADO.NET Entity Framework supports an innovative mapping architecture that aims to address the above challenges. It exploits the following ideas:

- Specification: Mappings are specified using a *declarative language* that has well-defined semantics and puts a wide range of mapping scenarios within reach of non-expert users.
- Compilation: Mappings are compiled into *bidirectional views*, called query and update views, that drive query and update processing in the runtime engine.
- Execution: Update translation is done using a general mechanism that leverages *materialized view maintenance*, a robust database technology [5][20]. Query translation uses view unfolding.

The new mapping architecture enables building a powerful stack of mapping-driven technologies in a principled, future-proof way. Moreover, it opens up interesting research directions of immediate practical relevance. The following subsections illustrate the specification and compilation of mappings. A more detailed description is in [32]. Query execution and updates are considered in Sections 7 and 8.

6.3 Specification of Mappings

A mapping is specified using a set of mapping fragments. Each mapping fragment is a constraint of the form $Q_{Entities} = Q_{Tables}$ where $Q_{Entities}$ is a query over the entity schema (on the application side) and Q_{Tables} is a query over the database schema (on the store side). A mapping fragment describes how a portion of entity data corresponds to a portion of relational data. That is, a mapping fragment is an elementary unit of specification that can be understood independently of other fragments.

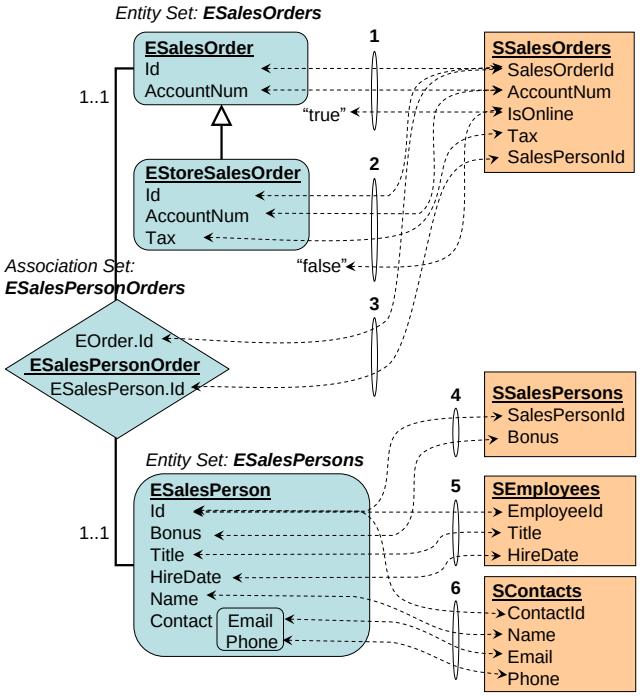

Figure 4: Mapping between an entity schema (left) and a database schema (right)

To illustrate, consider the sample mapping scenario in Figure 4. The mapping can be defined using an XML file or a graphical tool. The entity schema corresponds to the one in Section 3.2. On

the store side there are four tables, SSalesOrders, SSalesPersons, SEmployees, and SContacts.

The mapping is represented in terms of queries on the entity schema and the relational schema as shown in Figure 5.

SELECT o.Id, o.AccountNum FROM **ESalesOrders** o WHERE o IS OF (ONLY ESalesOrder)	=	SELECT SalesOrderId, AccountNum FROM **SSalesOrders** WHERE IsOnline = "true"
SELECT o.Id, o.AccountNum, o.Tax FROM **ESalesOrders** o WHERE o IS OF EStoreSalesOrder	=	SELECT SalesOrderId, AccountNum, Tax FROM **SSalesOrders** WHERE IsOnline = "false"
SELECT o.EOrder.Id, o.ESalesPerson.Id FROM **ESalesPersonOrders o**	=	SELECT SalesOrderId, SalesPersonId FROM **SSalesOrders**
SELECT p.Id, p.Bonus FROM **ESalesPersons** p	=	SELECT SalesPersonId, Bonus FROM **SSalesPersons**
SELECT p.Id, p.Title, p.HireDate FROM **ESalesPersons** p	=	SELECT EmployeeId, Title, HireDate FROM **SEmployees**
SELECT p.Id, p.Name, p.Contact.Email, p.Contact.Phone FROM **ESalesPersons** p	=	SELECT ContactId, Name, Email, Phone FROM **SContacts**

Figure 5: Representation of the mapping in Figure 4 as pairs of queries

Fragment 1 says that the set of (Id, AccountNum) values for all entities of exact type ESalesOrder in ESalesOrders is identical to the set of (SalesOrderId, AccountNum) values retrieved from the SSalesOrders table for which IsOnline is true. Fragment 2 is similar. Fragment 3 maps the association set ESalesPersonOrders to the SSalesOrders table and says that each association entry corresponds to the primary key, foreign key pair for each row in this table. Fragments 4, 5, and 6 say that the entities in the ESalesPersons entity set are split across three tables SSalesPersons, SContacts, SEmployees.

6.4 Bidirectional Views

The mappings are compiled into bidirectional Entity SQL views that drive the runtime. The *query views* express entities in terms of tables, while the *update views* express tables in terms of entities.

Update views may be somewhat counterintuitive because they specify persistent data in terms of virtual constructs, but as we show later, they can be leveraged for supporting updates in an elegant way. The generated views 'respect' the mapping in a well-defined sense and have the following properties (note that the presentation is slightly simplified—in particular, the persistent state is not completely determined by the virtual state):

- Entities = QueryViews(Tables)
- Tables = UpdateViews(Entities)
- Entities = QueryViews(UpdateViews(Entities))

The last condition is the *roundtripping criterion*, which ensures that all entity data can be persisted and reassembled from the database in a lossless fashion. The mapping compiler included in the Entity Framework guarantees that the generated views satisfy the roundtripping criterion. It raises an error if no such views can be produced from the input mapping.

Figure 6 shows the query and update views generated by the mapping compiler for the mapping in Figure 5. In general, the views are significantly more complex than the input mapping, as they explicitly specify the required data transformations. For example, in QV_1 the ESalesOrders entity set is constructed from the SSalesOrders table so that either an ESalesOrder or an EStoreSalesOrder is instantiated depending on whether or not the IsOnline flag is true. To reassemble the ESalesPersons entity set from the relational tables, one needs to perform a join between SSalesPersons, SEmployees, and SContacts tables (QV_3).

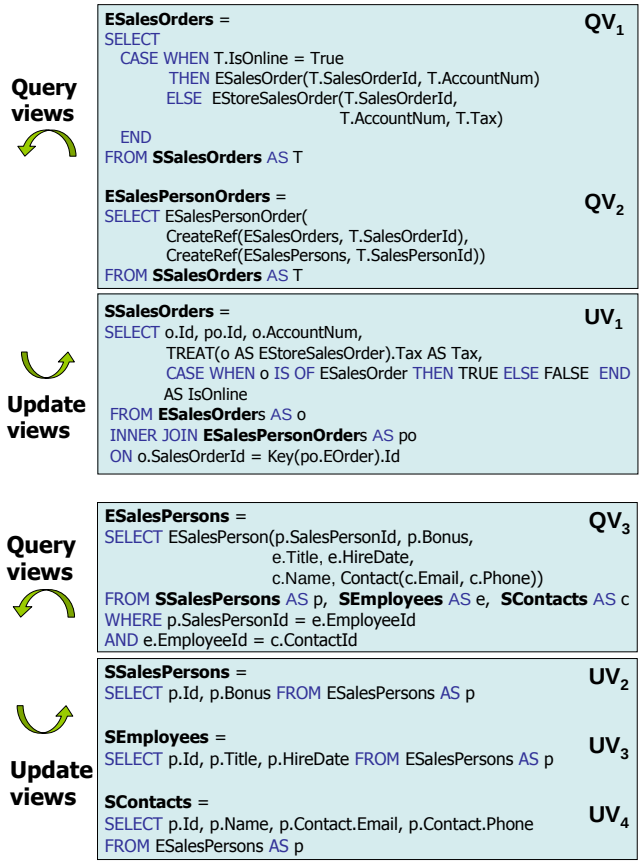

Figure 6: Bidirectional views for mappings in Figure 5

Writing query and update views by hand that satisfy the roundtripping criterion is tricky and requires significant database expertise; therefore, currently the Entity Framework only accepts the views produced by the built-in mapping compiler.

6.5 Mapping Compiler

The Entity Framework contains a mapping compiler that generates the query and update views from the EDM schema, the store schema, and the mapping (the metadata artifacts are discussed in Section 9). These views are consumed by the query and update pipelines. The compiler can be invoked either at design time or at runtime when the first query is executed against the EDM schema. The view generation algorithms used in the compiler are based on the answering-queries-using-views techniques for exact rewritings [21].

7. QUERY PROCESSING

7.1 Query Languages

The Entity Framework is designed to work with multiple query languages. In this paper we focus on Entity SQL and LINQ.

Entity SQL

Entity SQL is a derivative of SQL designed to query and manipulate EDM instances. Entity SQL extends standard SQL in the following ways.

- *Native support for EDM constructs* (entities, associations, complex types etc.): constructors, member accessors, type interrogation, relationship navigation, nest/unnest etc.
- *Namespaces*. Entity SQL uses namespaces as a grouping construct for types and functions (similar to XQuery and other programming languages).
- *Extensible functions*. Entity SQL supports no built-in functions. All functions (*min*, *max*, *substring*, etc.) are defined externally in a namespace, and *imported* into a query, usually from the underlying store.

The Entity Framework supports Entity SQL as the query language at the EntityClient provider layer, and in the Object Services component. A sample Entity SQL query is shown in Section 4.

Language Integrated Query (LINQ)

Language-integrated query [28], or LINQ for short, is an innovation in .NET programming languages that introduces query-related constructs to mainstream programming languages such as C# and Visual Basic. The query expressions are not processed by an external tool or language pre-processor but instead are first-class expressions of the languages themselves. LINQ allows *query expressions* to benefit from the rich metadata, compile-time syntax checking, static typing and IntelliSense [23] that was previously available only to imperative code. LINQ defines a set of general-purpose *standard query operators* that allow traversal, filter, join, projection, sorting and grouping operations to be expressed in a direct yet declarative way in any .NET-based programming language. C# and Visual Basic also support query comprehensions, i.e., language syntax extensions that leverage the standard query operators. An example query using LINQ in C# is shown in Section 4.

7.2 Canonical Command Trees

Canonical Command Trees, or simply command trees, are the programmatic (tree) representation of all queries in the Entity Framework. Queries expressed via Entity SQL or LINQ are first parsed and converted into command trees; all subsequent processing is performed on the command trees. The Entity Framework also allows queries to be dynamically constructed (or edited) via command tree construction/edit APIs. Command trees may represent queries, inserts, updates, deletes, and procedure calls. A command tree is composed of one or more Expressions. An Expression simply represents some computation. The Entity Framework provides a variety of expressions including constants, parameters, arithmetic operations, relational operations (projection, filter, joins etc.), function calls and so on. Finally, command trees are used as the means of communication for queries between the EntityClient provider and the underlying store-specific provider.

7.3 Query Pipeline

Query execution in the Entity Framework is delegated to the data stores. The query processing infrastructure of the Entity Framework is responsible for breaking down an Entity SQL or LINQ query into one or more elementary, relational-only queries that can be evaluated by the underlying store, along with additional assembly information, which is used to reshape the flat results of the simpler queries into the richer EDM structures.

The Entity Framework assumes that stores must support capabilities similar to that of SQL Server 2000. Queries are broken down into simpler flat-relational queries that fit this profile. Future releases of the Entity Framework will allow stores to take on larger parts of query processing.

A typical query is processed as follows.

- *Syntax and Semantic Analysis.* An Entity SQL query is first parsed and semantically analyzed using information from the Metadata services component. LINQ queries are parsed and analyzed as part of the appropriate language compiler.
- *Conversion to a Canonical Command Tree.* The query is now converted into a command tree, regardless of how it was originally expressed, and validated.
- *Mapping View Unfolding.* Queries in the Entity Framework target the conceptual (EDM) schemas. These queries must be translated to reference the underlying database tables and views instead. This process—referred to as mapping view unfolding—is analogous to the view unfolding mechanism in database systems. The mappings between the EDM schema and the database schema are compiled into query and update views. The query view is then unfolded in the user query; the query now targets the database tables and views.
- *Structured Type Elimination.* All references to structured types are now eliminated from the query, and added to the reassembly information (to guide result assembly). This includes references to type constructors, member accessors, and type interrogation expressions.
- *Projection Pruning.* The query is analyzed, and unreferenced expressions in the query are eliminated.
- *Nest Pull-up*. Any nesting operations in the query are bubbled up to the top leaving behind a basic relational substrate. The original query may now be broken down into one or more simpler (relational) queries.
- *Transformations*. A set of heuristic transformations are applied to simplify the query. These include filter pushdowns, apply-to-join conversions, case expression folding, etc. Redundant joins (self-joins, primary-key, foreign-key joins) are eliminated at this stage. Note that the query processing infrastructure here does not perform any cost-based optimization.
- *Translation into Provider-Specific Commands*. The query (i.e., command tree) is now handed off to providers to produce a provider-specific command, possibly in the providers' native SQL dialect. We refer to this step as SQLGen.
- *Command Execution*. The provider commands are executed.
- *Result Assembly*. The results (DataReaders) from the providers are then reshaped into the appropriate form using the assembly information gathered earlier, and a single DataReader is returned to the caller.
- *Object Materialization*. For queries issued via the Object Services component, the results are then materialized into the appropriate programming language objects.

7.4 SQLGen

As mentioned in the previous section, query execution is delegated to the underlying store. The query must first be translated into a form that is appropriate for the store. However, different stores support different dialects of SQL, and it is infeasible for the Entity Framework to natively support all of them. The query pipeline hands over a query in the form of a command tree to the store provider. The store provider must

translate the command tree into a native command. This is usually accomplished by translating the command tree into the provider's native SQL dialect—hence the term SQLGen for this phase. The resulting command can then be executed to produce the relevant results. In addition to working against various versions of SQL Server, the Entity Framework is being integrated with various third-party ADO.NET providers for DB2, Oracle, and MySQL.

8. UPDATE PROCESSING

This section describes how update processing is performed in the ADO.NET Entity Framework. There are two phases to update processing, compile-time and runtime. In Section 6.4 we described the process of compiling the mapping specification into a collection of view expressions. This section describes how these view expressions are exploited at runtime to translate the object modifications performed at the object layer (or Entity SQL DML updates at the EDM layer) into equivalent SQL updates at the relational layer.

8.1 Updates via View Maintenance

One of the key insights exploited in the ADO.NET mapping architecture is that materialized view maintenance algorithms can be leveraged to propagate updates through bidirectional views. This process is illustrated in Figure 7.

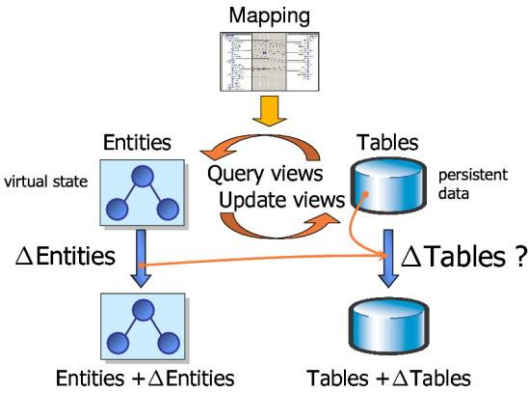

Figure 7: Bidirectional views and update translation

Tables inside a database hold persistent data. An EntityContainer represents a virtual state of this persistent data since typically only a tiny fraction of the entities in the EntitySets are materialized on the client. The goal is to translate an update ΔEntities on the virtual state of Entities into an update ΔTables on the persistent state of Tables. This can be done using the following two steps:

1. *View maintenance:*
   ΔTables = ΔUpdateViews(Entities, ΔEntities)
2. *View unfolding:*
   ΔTables = ΔUpdateViews(QueryViews(Tables), ΔEntities)

In Step 1, view maintenance algorithms are applied to update views. This produces a set of delta expressions, ΔUpdateViews, which tell us how to obtain ΔTables from ΔEntities and a snapshot of Entities. Since the latter is not fully materialized on the client, in Step 2 view unfolding is used to combine the delta expressions with query views. Together, these steps generate an expression that takes as input the initial database state and the update to entities, and computes the update to the database.

This approach yields a clean, uniform algorithm that works for both object-at-a-time and set-based updates (i.e., those expressed using data manipulation statements), and leverages robust database technology. In practice, Step 1 is often sufficient for update translation since many updates do not directly depend on the current database state; in those situations we have ΔTables = ΔUpdateViews(ΔEntities). If ΔEntities is given as a set of object-at-a-time modifications on cached entities, then Step 1 can be further optimized by executing view maintenance algorithms directly on the modified entities rather than computing the ΔUpdateViews expression.

8.2 Translating Updates on Objects

To illustrate the approach outlined above, consider the following example which gives a bonus and promotion to eligible salespeople who have been with the company for at least 5 years.

```
using(AdventureWorksDB aw =
        new AdventureWorksDB()) {
  // People hired more than 5 years ago
  var people = from p in aw.SalesPeople
               where p.HireDate <
               DateTime.Today.AddYears(-5)
               select p;

  foreach(SalesPerson p in people) {
    if(HRWebService.ReadyForPromotion(p)) {
      p.Bonus += 10;
      p.Title = "Senior Sales Representative";
    }
  }
  aw.SaveChanges(); // push changes to DB
}
```

AdventureWorksDB is a tool-generated class that derives from a generic ObjectContext class (described in Section 5). The above code fragment describes an update that modifies the title and bonus properties of ESalesPerson objects which are stored in the SEmployees and SSalesPersons tables, respectively. The process of transforming the object updates into the corresponding table updates triggered by the call to the SaveChanges method consists of the following four steps:

Change List Generation. A list of changes per entity set is created from the object cache. Updates are represented as lists of deleted and inserted elements. Added objects become inserts. Deleted objects become deletes.

Value Expression Propagation. This step takes the list of changes and the update views (kept in the metadata workspace) and, using incremental materialized view maintenance expressions ΔUpdateViews, transforms the list of object changes into a sequence of algebraic base table insert and delete expressions against the underlying affected tables. For this example, the relevant update views are UV_2 and UV_3 shown in Figure 6. These views are simple project-select queries, so applying view maintenance rules is straightforward. We obtain the following ΔUpdateViews expressions, which are the same for insertions (Δ^+) and deletions (Δ^-):

ΔSSalesPersons = SELECT p.Id, p.Bonus
FROM ΔESalesPersons AS p

ΔSEmployees = SELECT p.Id, p.Title
FROM ΔESalesPersons AS p

ΔSContacts = SELECT p.Id, p.Name, p.Contact.Email,
p.Contact.Phone FROM ΔESalesPersons AS p

Suppose the loop shown above updated the entity E_{old} = ESalesPersons(1, 20, "", "Alice", Contact("a@sales", NULL)) to E_{new} = ESalesPersons(1, 30, "Senior …", "Alice", Contact("a@sales", NULL)). Then, the initial delta is Δ^{+}ESalesOrders = $\{E_{new}\}$ for insertions and Δ^{-}ESalesOrders = $\{E_{old}\}$ for deletions. We obtain Δ^{+}SSalesPersons = {(1, 30)}, Δ^{-}SSalesPersons = {(1, 20)}. The computed insertions and deletions on the SSalesPersons table are then combined into a single update that sets the Bonus value to 30. The deltas on SEmployees are computed analogously. For SContacts, we get Δ^{+}SContacts = Δ^{-}SContacts, so no update is required.

In addition to computing the deltas on the affected base tables, this phase is responsible for (a) the correct ordering in which the table updates must be performed, taking into consideration referential integrity constraints, (b) retrieval of store-generated keys needed prior to submitting the final updates to the database, and (c) gathering the information for optimistic concurrency control.

SQL DML or Stored Procedure Calls Generation. This step transforms the list of inserted and deleted deltas plus additional annotations related to concurrency handling into a sequence of SQL DML statements or stored procedure calls. In this example, the update statements generated for the affected salesperson are:

```
BEGIN TRANSACTION
UPDATE [dbo].[SSalesPersons] SET [Bonus]=30
WHERE [SalesPersonID]=1
UPDATE [dbo].[SEmployees]
SET [Title]= N'Senior Sales Representative'
WHERE [EmployeeID]=1
END TRANSACTION
```

Cache Synchronization. Once updates have been performed, the state of the cache is synchronized with the new state of the database. Thus, if necessary, a query-processing step is performed to transform the new modified relational state to its corresponding entity and object state.

9. METADATA

The metadata subsystem is analogous to a database catalog, and is designed to satisfy the design-time and runtime metadata needs of the Entity Framework.

9.1 Metadata Artifacts

The key metadata artifacts are the following:

Conceptual Schema (CSDL files)*:* The conceptual schema is usually defined in a CSDL file (Conceptual Schema Definition Language) and contains the EDM types (entity types, associations) and entity sets that describes the application's conceptual view of the data.

Store Schema (SSDL files)*:* The store schema information (tables, columns, keys etc.) are also expressed in terms of EDM constructs (EntitySets, properties, keys). Usually, these are defined in an SSDL (Store Schema Definition Language) file.

C-S Mapping Specification (MSL file)*:* The mapping between the conceptual schema and the store schema is captured in a mapping specification, typically in an MSL file (Mapping Specification Language). This specification is used by the mapping compiler to produce the query and update views.

Provider Manifest*:* The Provider Manifest is a description of functionality supported by each provider, and includes information about the supported primitive types and built-in functions.

In addition to these artifacts, the metadata subsystem also keeps track of the generated object classes, and the mappings between these and the corresponding conceptual entity types.

9.2 Metadata Services Architecture

The metadata consumed by the Entity Framework comes from different sources in different formats. The metadata subsystem is built over a set of unified low-level metadata interfaces that allow the metadata runtime to work independently of the details of the different metadata persistent formats/sources.

The metadata subsystem includes the following components. The **metadata cache** caches metadata retrieved from different sources, and provides consumers a common API to retrieve and manipulate the metadata. Since the metadata may be represented in different forms, and stored in different locations, the metadata subsystem supports a loader interface. **Metadata loaders** implement the loader interface, and are responsible for loading the metadata from the appropriate source (CSDL/SSDL files etc.). A **metadata workspace** aggregates several pieces of metadata to provide the complete set of metadata for an application. A metadata workspace usually contains information about the conceptual model, the store schema, the object classes, and the mappings between these constructs.

10. TOOLS

The Entity Framework includes a collection of design-time tools to increase development productivity.

Model designer: One of the early steps in the development of an application is the definition of a conceptual model. The Entity Framework allows application designers and analysts to describe the main concepts of their application in terms of entities and relationships. The model designer is a tool that allows this conceptual modeling task to be performed interactively. The artifacts of the design are captured directly in the Metadata component which may persist its state in the database. The model designer can also generate and consume model descriptions (specified via CSDL), and can synthesize EDM models from relational metadata.

Mapping designer: Once an EDM model has been designed, the developer needs to specify how a conceptual model maps to a relational database. The mapping designer helps developers describe how entities and relationships map to tables and columns in the database. It visualizes the mapping expressions specified declaratively as equalities of Entity SQL queries. These expressions become the input to the mapping compilation component which generates the query and update views.

Code generation: The Entity Framework includes a set of code generation tools that take EDM models as input and produce strongly-typed CLR classes for entity types. The code generation tools can also generate a strongly-typed object context (e.g., AdventureWorksDB) which exposes strongly typed collections for all entity and relationship sets defined by the model (e.g., ObjectQuery<SalesPerson>).

11. RELATED WORK

Bridging applications and databases is a longstanding problem [3][15]. Researchers and practitioners attacked it in a number of ways. In 1996, Carey and DeWitt [11] outlined why many

technologies, including object-oriented databases and persistent programming languages, did not gain wide acceptance due to limitations in query and update processing, transaction throughput, scalability, etc. They speculated that object-relational databases would dominate in 2006. Indeed, DB2 [10] and Oracle [27] database systems include a built-in object layer that uses a hardwired O/R mapping on top of a conventional relational engine. However, the O/R features offered by these systems appear to be rarely used for storing enterprise data [19], with the exception of multimedia and spatial data types. Among the reasons are data and vendor independence, the cost of migrating legacy databases, scale-out difficulties when business logic runs inside the database instead of the middle tier, and insufficient integration with programming languages [37].

Since mid 1990's, client-side data mapping layers have gained popularity, fueled by the growth of Internet applications. A core function of such a layer is to provide an updatable view that exposes a data model closely aligned with the application's data model, driven by an explicit mapping. Many commercial products and open source projects have emerged to offer these capabilities. Virtually every enterprise framework provides a client-side persistence layer (e.g., EJB in J2EE). Most packaged business applications, such as ERP and CRM applications, incorporate proprietary data access interfaces (e.g., BAPI in SAP R/3).

The J2EE framework includes a number of persistence solutions, such as JDO [24], EJB [17], etc. JDO is more of an object-persistence solution with support for persistence of Java classes, a query language JDOQL to query over data, class-generation tools etc. The EJB 3.0 standard specifies a more conventional ORM solution, with support for mapping Java classes onto persistent stores, a query language (Java Persistence Query Language), schema design tools, and so on. The Entity Framework is similar to the EJB specification in that it allows for existing database schemas to be mapped onto classes. Unlike both EJB and JDO, however, the Entity Framework defines a concrete value-based conceptual layer, and allows applications to program directly against the conceptual layer. The Entity Framework also allows applications to be written against programming language abstractions in any of the supported .NET languages. Finally, the Entity Framework supports language-integrated queries (LINQ).

One widely used open source ORM framework for Java is Hibernate [8] (and its .NET implementation, NHibernate). The latest release of Hibernate supports a number of inheritance mapping scenarios, optimistic concurrency control, and comprehensive object services, and conforms to the EJB 3.0 standard. On the commercial side, popular ORMs include Oracle TopLink [33] on Java and LLBLGen [29] on the .NET platform. These and other ORMs we know of are tightly coupled with the object models of their target programming languages.

The Service Data Object (SDO) [35] specification allows applications to access heterogeneous data in a uniform fashion. SDO defines the notion of a disconnected *data graph* – this data graph is the unit of transfer between data sources and client applications. The data graph may be manipulated directly, or may be mapped into programming language abstractions (via JAXB and other similar mechanisms). *Data mediator services* provide a common data graph abstraction over disparate data sources. Many of these concepts are similar to those of the Entity Framework – data mediators (providers), language-neutrality etc. Unlike SDO's data graphs, however, the Entity Framework is based on a high-level data model (EDM), and allows applications to program and query against the EDM.

BEA's AquaLogic Data Services Platform (ALDSP) [9] is based on SDO, and uses XML Schema for modeling application data. The XML data is assembled using XQuery from databases and web services. ALDSP's runtime supports queries over multiple data sources and performs client-side query optimization. The updates are performed as view updates on XQuery views. If an update does not have a unique translation, the developer needs to override the update logic using imperative code.

Today's client-side mapping layers offer widely varying degrees of capability, robustness, and total cost of ownership. Typically, the mapping between the application and database artifacts used by ORMs has vague semantics and drives case-by-case reasoning. A scenario-driven implementation limits the range of supported mappings and often yields a fragile runtime that is difficult to extend. Few data access solutions leverage data transformation techniques developed by the database community, and often rely on ad hoc solutions for query and update translation.

Database research has contributed many powerful techniques that can be leveraged for building persistence layers. And yet, there are significant gaps. Among the most critical ones is supporting updates through mappings. Compared to queries, updates are far more difficult to deal with as they need to preserve data consistency across mappings, may trigger business rules, and so on. Updates through database views are intrinsically hard: even for very simple views finding a unique update translation is rarely possible [16]. As a consequence, commercial database systems and data access products offer very limited support for updatable views. Recently, researchers turned to alternative approaches, such as bidirectional transformations [7]. One of the key innovations in the Entity Framework is a technique for propagating updates incrementally using view maintenance algorithms [5][20] applied to bidirectional views. Another one is a novel mechanism for obtaining such views by compiling them from mappings [32], which allows describing complex mapping scenarios in an elegant way.

Traditionally, conceptual modeling has been limited to database and application design, reverse-engineering, and schema translation. Many design tools use UML [36]. Only very recently conceptual modeling started penetrating industry-strength data mapping solutions. For example, the concept of entities and relationships surfaces both in ALDSP and EJB 3.0. ALDSP overlays E-R-style relationships on top of complex-typed XML data, while EJB 3.0 allows specifying relationships between objects using class annotations. The Entity Framework goes one step further by making conceptual modeling its core development paradigm, and layering other data-centric services on top of it.

Schema mapping techniques are used in many data integration products, such as Microsoft BizTalk Server [30], IBM Rational Data Architect [22], and ETL tools. These products allow developers to design data transformations or compile them from mappings [34] to translate e-commerce messages or load data warehouses. The Entity Framework combines a mapping compiler, a data transformation engine based on bidirectional views, ORM-style object services, and language-integrated queries to offer a comprehensive data access solution.

12. CONCLUSIONS AND OUTLOOK

This paper presented a detailed overview of the ADO.NET Entity Framework. The Entity Framework allows applications and data-centric services to operate at a higher level of abstraction than relational tables via a new Entity Data Model and rich mapping support between conceptual schemas and database schemas.

The Entity Framework enables general-purpose database development against conceptual schemas. It leverages the well-known .NET data provider model, and allows building data-centric services like reporting and replication on top of the conceptual layer. Furthermore, the Entity Framework provides ORM-style functionality using an object services layer that exposes objects as thin wrappers around entities.

Together, the Entity Framework and Language Integrated Query in .NET promise to significantly reduce the impedance mismatch for applications and data services, provide a clean separation between a conceptual layer and an object layer, and introduce a new level of data independence between applications and databases.

In future releases, we plan to enhance the capabilities of the Entity Framework along multiple dimensions such as broader integration with XML/XSD, mapping, and querying; richer web services support; and a more comprehensive set of design and mapping tools.

13. ACKNOWLEDMENTS

The ADO.NET Entity Framework is a large team effort. We would like to thank all members of the ADO.NET team, in particular, Mark Ashton, Brian Beckman, Kawarjit Bedi, Phil Bernstein, David Campbell, Pablo Castro, Simon Cavanagh, Andy Conrad, Samuel Druker, Kati Iceva, Britt Johnston, Asad Khan, Nick Kline, Vamsi Kuppa, Kirk Lang, Tim Mallalieu, Srikanth Mandadi, Colin Meek, Anil Nori, Lance Olson, Rick Olson, Pratik Patel, Shyam Pather, Michael Pizzo, Daniel Simmons, Steve Starck, Ed Triou, Fabio Valbuena, Naveen Valluri, Jason Wilcox, and many more, who helped build this product.

14. REFERENCES

[1] ADO.NET, http://msdn.microsoft.com/data/ref/adonetnext/

[2] ANSI/X3/SPARC Study Group on Data Base Management Systems. Interim Report. *FDT (ACM SIGMOD bulletin)* 7, No. 2, 1975.

[3] Atkinson, M. P., Buneman, P. Types and Persistence in Database Programming Languages. *ACM Comput. Surv.*, 19(2):105–190, 1987

[4] Barsalou, T., Keller, A. M., Siambela, N., Wiederhold, G. Updating Relational Databases through Object-Based Views. *SIGMOD*, 1991.

[5] Blakeley, J.A., Larson, P., Tompa, F.W. Efficiently Updating Materialized Views, *SIGMOD*, 1986.

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

[7] Bohannon, A., Pierce, B. C., Vaughan, J. A. Relational Lenses: A Language for Updatable Views. *PODS*, 2006.

[8] Bauer, C., King, G. *Java Persistence with Hibernate*, Manning Publications, Nov. 2006.

[9] Carey, M. J. et al. Data Delivery in a Service Oriented World: The BEA AquaLogic Data Services Platform, *SIGMOD* 2006.

[10] Carey, M. J., Chamberlin, D. D., Narayanan, S., Vance, B., Doole, D., Rielau, S., Swagerman, R., Mattos, N. M. O-O, What Have They Done to DB2? *VLDB*, 1999.

[11] Carey, M. J., DeWitt, D. J. Of Objects and Databases: A Decade of Turmoil. *VLDB*, 1996.

[12] Castro, P., Melnik, S., Adya, A. ADO.NET Entity Framework: Raising the Level of Abstraction in Data Programming. *SIGMOD* (demo), 2007.

[13] Chen, P. The Entity-Relationship Model—Toward a Unified View of Data, *ACM Trans. on Database Syst.*, 1(1), 1976.

[14] Common Language Runtime (CLR), http://msdn2.microsoft.com/en-us/library/ms131047.aspx

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

[16] Dayal, U., Bernstein, P. A. On the Updatability of Relational Views. *VLDB*, 1978.

[17] EJB 3.0, http://java.sun.com/products/ejb/

[18] Entity Data Model, http://msdn2.microsoft.com/en-us/library/aa697428(vs.80).aspx

[19] Grimes, S. Object/Relational Reality Check. *Database Programming & Design (DBPD)*, 11(7), July 1998.

[20] Gupta, A., Mumick, I.S. *Materialized Views: Techniques, Implementations, and Applications*, The MIT Press, 1999.

[21] Halevy, A. Y. Answering Queries Using Views: A Survey. *VLDB Journal*, 10(4):270–294, 2001.

[22] IBM Rational Data Architect. www.ibm.com/software/data/integration/rda/

[23] IntelliSense, http://msdn2.microsoft.com/en-us/library/hcw1s69b(vs.71).aspx

[24] JDO (Java Data Objects), http://java.sun.com/products/jdo/

[25] Keller, A.M., R. Jensen, S. Agrawal. Persistence Software: Bridging Object-Oriented Programming and Relational Databases. *SIGMOD*, 1993.

[26] Keene, C. Data Services for Next-Generation SOAs. *SOA WebServices Journal*, 4(12), 2004

[27] Krishnamurthy, V., Banerjee, S., Nori, A. Bringing Object-Relational Technology to Mainstream. *SIGMOD*, 1999.

[28] The LINQ Project. http://msdn.microsoft.com/library/en-us/dndotnet/html/linqprojectovw.asp

[29] LLBLGen Pro O/R Mapper – Generator. www.llblgen.com

[30] Microsoft BizTalk Server. www.microsoft.com/biztalk/

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

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

[33] Oracle TopLink. www.oracle.com/technology/products/ias/toplink/

[34] Roth, M., Hernandez, M. A., Coulthard, P., Yan, L., Popa, L., Ho, H. C.-T., Salter, C. C. XML Mapping Technology: Making Connections in an XML-Centric World. *IBM Systems J.*, 45(2), 2006.

[35] SDO, http://www.osoa.org/display/Main/Service+Data+Objects+Home

[36] Unified Modeling Language. http://www.uml.org/

[37] Zhang, W., Ritter, N. The Real Benefits of Object-Relational DB-Technology for Object-Oriented Software Development. In *BNCOD*, 2001.