The ADO.NET Entity Framework: Making the Conceptual Level Real

José A. Blakeley, David Campbell, S. Muralidhar, Anil Nori
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399, USA
{joseb, davidc, smurali, anilnori}@microsoft.com

ABSTRACT

This paper describes the ADO.NET Entity Framework, a platform for programming against data that raises the level of abstraction from the logical (relational) level to the conceptual (entity) level, and thereby significantly reduces the impedance mismatch for applications and data services such as reporting, analysis, and replication. The conceptual data model is made real by a runtime that implements an extended relational model (the Entity Data Model aka the EDM), that embraces entities and relationships as first class concepts; a query language for the EDM; a comprehensive mapping engine that translates from the conceptual to the logical (relational) level, and a set of model-driven tools that help create entity-object, object-xml, and entity-xml transformers.

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 (customers, orders) but also XML, email, calendar, files, and documents. These applications need to integrate data residing in multiple data sources and enable end-to-end business insight by collecting, cleaning, storing, and preparing business data in forms suitable for an agile decision making process. Developers of these applications need data access, programming and development tools to increase their productivity.

This paper describes the ADO.NET Entity Framework, a platform for programming against data that significantly reduces the impedance mismatch for applications and data services such as reporting, analysis, and replication. We argue that modern applications and data services need to target a higher-level conceptual model based on entities and relationships rather than the relational model and that such a conceptual model needs to be implemented concretely in a data platform. The Entity Framework makes the conceptual data model concrete by a runtime that implements an extended relational model – the Entity Data Model, or the EDM - that embraces entities and relationships as first class concepts, a query language for the EDM, a comprehensive mapping engine that translates from the conceptual to the logical (relational) level, and a set of model-driven tools that help create entity-object, object-xml, and entity-xml transformers. The Entity Framework is part of a broader Microsoft Data Access vision supporting a family of *products and services so customers derive value from all data, birth through archival.*

Section 2 describes the physical, logical, conceptual and programming levels as well as other terms used throughout the paper. Section 3 describes the evolution of applications and data services and motivates the need for making the conceptual level central to application and data services design. Section 4 introduces the Entity Data Model and the concrete manifestation of this model in the Entity Framework. Section 5 presents a summary and conclusions.

This paper is a revised version of one that appeared in the Proceedings of the 25th International Conference on Conceptual Modeling, Tucson, AZ, USA, November 6-9, 2006, available from Springer at: http://www.springer.com/east/ home/generic/search/results?SGWID=5-40109-22-173696005-0. The authors thank Springer for their permission to publish this revised paper.

2. DATABASE MODELING LAYERS

Today's dominant information modeling methodology for producing database designs factors an information 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. The vocabulary of concepts discussed at this layer include record formats, file partitions and groups, heaps, and indexes. The physical model is typically invisible to the application - applications usually target the logical or relational data model described in the next section. Changes to the physical model should not impact application logic, but may impact application performance.

A ***logical*** data model is a complete and precise information model of the target domain. The relational model is the representation of choice for most logical data models. The concepts discussed at the logical level include tables, rows, and primary key-foreign key constraints, and normalization. While normalization helps to satisfy important application requirements such as data consistency and increased concurrency with respect to updates and OLTP performance, it also introduces significant challenges for applications. (Normalized) Data at the logical level is too fragmented and application logic needs to aggregate rows from multiple tables into higher level entities that more closely resemble the artifacts of the application domain. The conceptual level

introduced in the next section is designed to overcome these challenges.

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 [1]. UML is a more recent example of a conceptual model [2].

Most significant applications involve a conceptual design phase early in the application development lifecycle. Unfortunately, however, the conceptual data model is captured inside a database design tool that has little or no connection with the code and the relational schema used to implement the application. The database design diagrams created in the early phases of the application life cycle usually stay "pinned to a wall" growing increasingly disjoint from the reality of the application implementation with time. However, a conceptual data model can be as real, precise, and focused on the concrete "concepts" of the application domain as a logical relational model. A goal of the Microsoft Data Access vision is to make the conceptual data model (embodied by the Entity Data Model, described in Section 4.2) a concrete feature of the data platform.

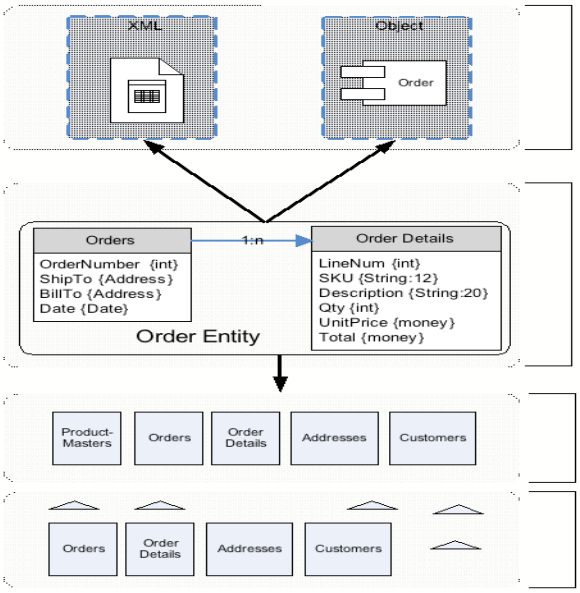

Figure 1: Physical, logical, conceptual and multiple programming and presentation views of an Order.

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.

Most developers, and most of the modern data services want to reason about high-level concepts such as an "Order" (See **Error! Reference source not found.**), not about the several tables that an order may be normalized over in a relational database schema. They want to query, secure, program, report on the order. An order may manifest itself at the presentation/programming level as a class instance in Visual Basic or C# encapsulating the state and logic associated with the order, or as an XML stream for communicating with a web service. We believe there is no "one proper presentation model"; and that the real value is in making the conceptual level real and then being able to use that model as the basis for flexible mappings to and from various presentation models and other higher level services.

3. APPLICATION AND DATA SERVICES EVOLUTION

This section describes the platform shift that motivates the need for a higher level data model and data platform. We will look at this through two perspectives: application evolution and SQL Server's evolution as a product. A key point we make in this section is that the need for rich data model is motivated not just for developing application logic but also for supporting building higher-level data services such as reporting and replication.

3.1 Application Evolution

Data-based applications 10-20 years ago were typically structured as data monoliths; closed systems with logic factored by verb-object functions that interacted with a database system at the logical schema (e.g. relational) level. A typical order entry system built around a relational database management system (RDBMS) 20 years ago would have logic partitioned around verb-object functions associated with how users interacted with the system. In fact, the user interaction model via "screens" or "forms" became the primary factoring for logic – there would be a new-order screen, and update-customer screen. The system may have also supported batch updates of SKU's, inventory, etc. The application logic was tightly bound to the logical relational schema.

Much of the data-centric logic (e.g. validation logic) is embedded within the application logic. People typically wrote batch programs to interact directly with the logical schema to perform updates. Programming languages did not support representation of high-level abstractions directly – objects did not exist. These applications can be characterized as being closed systems whose logical data consistency was maintained by application logic implemented at the logical schema level. An order was an order because the new-order logic ensured that it was.

A key reason for custom data-centric logic by applications is the well-known ***application impedance mismatch problem***. The logical schema does not match the level of abstraction of the application. Applications address this problem by developing at the data abstraction (e.g. relational) and by writing custom mapping code to bridge the gap between the application and the data abstractions. This not only leads to duplication of effort but also reduces application development productivity. In the next sections we will show how the Entity Framework and the Language Integrated Query innovations in .NET languages help to minimize this impedance mismatch.

Several significant trends have shaped the way that modern data-based applications are factored and deployed today. Chief among these are object oriented factoring, service level application composition, and higher level data services. When we think about the factoring, composition, and services from above, we can see that the conceptual entities are an important part of today's applications. It is also easy to see how these entities must be mapped to a variety of representations and bound to a variety of services. There is no one correct representation or service binding. XML, Relational and Object representations are all important but no single one will suffice.

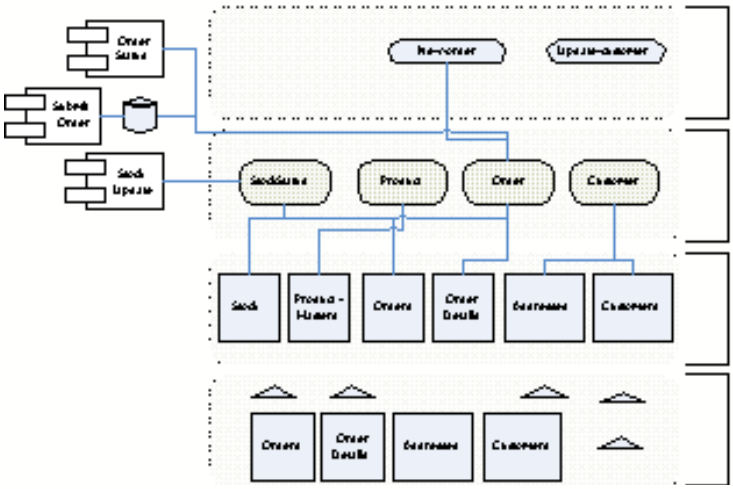

Figure 2 Order Entry System circa 2005

Consider a "StockNotifications" application which deals with concepts like Customer Order, Product, and Stock. How do we make them real and use our conceptual understanding of them throughout the system whether they are stored in a multi-dimensional database for analytics, in a durable queue between systems, in a mid-tier cache; a business object, etc.

Figure 2 captures the essence of this issue by focusing on several entities in our order entry system. Note that conceptual level entities have become real. Also note that the conceptual entities are communicating with and mapping to various logical schema formats, e.g. relational for the persistent storage, messages for the durable message queue on the Submit Order service, and perhaps XML for the Stock Update and Order Status web services.

3.2 SQL Server Evolution

The data services provided by a "data platform" 20 years ago were minimal and focused around the logical schema in an RDBMS. These services included query & update, atomic transactions, and bulk operations such as backup and load/extract.

SQL Server itself is evolving from a traditional RDBMS to a *complete data platform* that provides a number of high value data services over entities realized at the conceptual schema level. While providing services such as reporting, analysis, and data integration in a single product and realizing synergy among them was a conscious business strategy, the means to achieve these services and the resultant ways of describing the entities they operate over happened more organically – many times in response to problems recognized in trying to provide higher level data services over the logical schema level. There are two good examples of the need for concrete entity representation for services now provided within SQL Server: *logical records* for merge replication, and the *semantic model* for report builder.

Early versions of merge **replication** in SQL Server provided for multi-master replication of individual rows. In this early mode, rows can be updated independently by multiple agents; changes can conflict; and various conflict resolution mechanisms are provided with the model. This row-centric service had a fundamental flaw – it did not capture the fact that there is an implicit consistency guarantee around entities as they flow between systems. To address this flaw, the replication service introduced "logical records" as a way to describe and define consistency boundaries across entities comprised of multiple related rows at the logical schema level. "Logical records" are defined in the part of the SQL catalog associated with merge replication. There is no proper design-time tool experience to define a "logical record" such as an Order that includes its Order Details – applications do it through a series of stored procedure invocations.

Report Builder (RB) is another example of SQL Server providing a data service at the conceptual entity level. Since it operates at the logical schema level though, writing reports requires knowing how to compose queries at the logical schema level – e.g. creating an order status report requires knowing how to write the join across the several tables that make up an order. End users and analysts, however, want to write reports directly over Customers, Orders, Sales, etc. Thus, the SQL Server team created a means to describe and map conceptual entities to the logical schema layer we call the Semantic Model Definition Language (SMDL).

These are just two of a number of mapping services provided within SQL Server – the Unified Dimensional Model (UDM)

provides a multi-dimensional view abstraction over several logical data models. A Data Source View (DSV), on which the BI tools work, also provides conceptual view mapping technology.

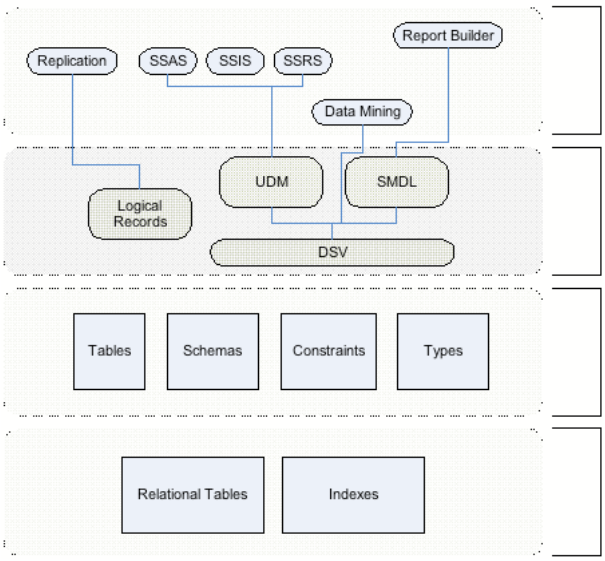

Figure 3: SQL Server 2005

A key observation is that several higher-level data services in the SQL Server product are increasingly delivering their services at the conceptual schema level. Currently, each of these services has a separate tool to describe conceptual entities and map them down to the underlying logical schema level. **Figure 3** illustrates the evolution of SQL Server into a data platform with many high value data services and multiple means to map conceptual entities to their underlying logical schemata.

4. ENTITY FRAMEWORK

This section describes the ADO.NET Entity Framework that makes the conceptual level real. We start with the rationale that led us to the development of an Entity Data Model (EDM) followed by an overview of the EDM. We present an architectural description of the entity framework implementing a runtime supporting the EDM, a query language, and mapping. We conclude the section with a description of the development process around the EDM.

4.1 Why a new model?

The Entity Data Model (EDM) is intended for developing rich data-centric applications. The obvious question that arises is: "why not use (or extend) one of these established data models? There are at least four other modern candidates for such a data model:

- The SQL data model (tables, columns, keys, referential integrity constraints...). SQL99 extends this core model to include object relational features (user defined-, structured-, and distinct-types, methods, typed tables, refs…).
- The CLR data model (classes, fields, methods, properties, value, and Ref types, collections…)
- The XSD model based on XML Infoset (Atomic-, list-, and union-types, primitive- and derived-types, token, ID, IDREF, ENTITY…)
- The UML data model (classes, objects, associations, generalizations, attributes, operations, aggregations…)

The overall reason is that we need something that maps cleanly to both the CLR and to relational databases like SQL Server, for programmability and persistence respectively. None of the other candidates has all the needed facilities for both. The CLR is an object-oriented, imperative-programming runtime, and has no native data model or notions of integrity constraints, relationships, or persistence. SQL99 lacks data modeling concepts like relationships, and does not have good programming language integration. The XSD specification does not support concepts like keys, relationships, and persistence. In addition, the full XSD specification is complex and has awkward mapping to both the runtime and to relational database models. The UML is too general: it requires application developers to add precise semantics, especially for persistence.

The EDM has been designed to map downward cleanly to both the CLR and to a relational database, and upward to a specialization of UML. Designers can work with concepts familiar from UML, which can be compiled in phases to XML, CLR programs, and SQL.

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). One or more object programming models can be easily supported on top of EDM. Similarly, the EDM can mapped to one or more relational DBMS implementations for persistence.

4.2 EDM Overview

The EDM extends the classic relational model with concepts from E-R modeling. The central concepts in the EDM are entities and relationships. *Entities* represent top-level objects with independent existence and identity, while *Relationships* are used to *relate* (or, describe relationships between) two or more entities.

4.2.1 Types

An ***EntityType*** describes the definition of an entity. An entity typically is a top-level object with independent existence. An entity has a *payload* - zero or more properties 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 its container. EntityTypes may derive from (or subtype) other entity types. EDM supports a single inheritance model.

The properties of an entity may be simple or complex types. A ***SimpleType*** (or a PrimitiveType) represents scalar (or atomic) types (e.g. integer, string), while a ***ComplexType*** can be used to represent structured properties (e.g. an Address). A ComplexType is composed of zero or more properties, which may themselves be scalar or complex type properties.

A ***Relationship*** type is a specialized entity type that describes relationships between two (or more) entity types. Initially, the EDM supports one kind of relationship, namely ***Association***, which models peer-to-peer entity relationships (e.g., Supplier-Part). Containment parent-child relationships (e.g. Order-Line) are modeled as associations with cascading

actions. The key for a relationship type is usually, but not necessarily, the concatenated keys of the entity types participating in the relationship. Relationships – especially many-to-many relationships - may optionally contain properties of the relationship itself.

EDM ***Schemas*** provide a grouping mechanism for types – types must be defined in a schema.

In addition to the types above, the EDM supports *transient* types in the form of RowTypes and CollectionTypes. These occur mostly in the context of query operations (e.g., projections, joins). A *RowType* is an anonymous type that is structurally similar to a *ComplexType***.** A RowType's structure depends on the sequence of typed and named members that it is comprised of. A rowtype has no identity and cannot be inherited from. Instances of the same row type are equivalent if the corresponding members (in order) are respectively equivalent. Rows have no behavior beyond their structure. A *CollectionType* represents a homogenous collection of objects.

4.2.2 Primitive Types

The EDM is a data model, not a type system. The EDM defines shaping constructs (entity types etc.), but the actual types (and their semantics) are defined by the hosting environment. The EDM does define a set of abstract (or template) primitive types, and a set of associated facets, that enable the abstract primitive types to represent primitive types of the hosting environment (SqlServer databases, the CLR, etc.). These abstract types are proxies for the real primitive types defined by the host, and the semantics of operations over these types are entirely governed by the host.

4.2.3 Instances

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 entity instance must belong to exactly one entity set. In a similar fashion, relationship instances are logically contained within a ***RelationshipSet***. The definition of a RelationshipSet scopes the relationship, that is, it identifies the EntitySets that hold instances of the entity types that participate in the relationship. SimpleTypes and ComplexTypes can only be instantiated as properties of entity instances.

An ***EntityContainer*** is a logical grouping of EntitySets and RelationshipSets – akin to how a Schema is a grouping mechanism for EDM types.

4.2.4 Examples

```
<?xml version="1.0"?>
<Schema Namespace="CNorthwindSchema"
        xmlns="urn:schemas-microsoft-com:windows:storage">
<!–
Typical Entity definition, has identity and some members
-->
  <EntityType Name="Product" Key="ProductID">
    <Property Name="ProductID" Type="System.Int32" />
    <Property Name="ProductName" Type="System.String"
          Size="max" />
    ...
  </EntityType>

<!–
A derived product
-->
  <EntityType Name="DiscontinuedProduct" BaseType="Product">
    <Property Name="DiscReason" Type="System.String"
          Size="max" />
  </EntityType>

<!–
A complex type defines structure but no identity. It can be
used inline
in 0 or more Entity definitions
-->
   <ComplexType Name="CtAddress" >
      <Property Name="Address" Type="System.String"
          Size="max" />
      <Property Name="City" Type="System.String"
          Size="max" />
      <Property Name="PostalCode" Type="System.String"
          Size="max" />
      ...
   </ComplexType>
<!–
A Customer Entity
-->
   <EntityType Name="Customer" Key="CustomerID">
          <!– Address is a member which references a
          complextype -->
      <Property Name="Address" Type="CNorthwind.CtAddress" />
      <Property Name="CustomerID" Type="System.String"
          Size="max" />
   </EntityType>
<!–
An example of an association between Product [defined above]
and
OrderDetails [not shown for sake of brevity]
-->
  <Association Name="Order_DetailsProducts">
    <End Name="Product" Type="Product" Multiplicity="1" />
    <End Name="Order_Details" Type="OrderDetail"
          Multiplicity="*" />
  </Association>

</Schema>

<!–
The Entity Container defines the logical encapsulation of
EntitySets (sets of (possibly) polymorphic instances of a
type) and
AssociationSets (logical link tables for relating two or more
entity instances)
-->
  <EntityContainer Name="CNorthwind">
    <Using Namespace="CNorthwindSchema" />

    <EntitySet Name="Products" EntityType="Product" />
    <EntitySet Name="Customers" EntityType="Customer" />
    <EntitySet Name="Order_Details"
          EntityType="OrderDetail" />
    <EntitySet Name="Orders" EntityType="Order" />

    <AssociationSet Name="Order_DetailsProductsSet"
          Association="Order_DetailsProducts">
      <End Name="Product" EntitySet="Products" />
      <End Name="Order_Details" EntitySet="Order_Details"/>
    </AssociationSet>
  </EntityContainer>
```

4.3 Entity Framework Architecture

This section briefly describes the architecture of the Entity Framework being built as part of ADO.NET. The main functional components of the ADO.NET Entity Framework (see Error! Reference source not found.) are:

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.

EntityClient provider. The Entity Framework includes a new data provider, the EntityClient provider. This provider houses the services implementing the mapping transformation from conceptual to logical constructs. The EntityClient provider is a value-based, outside-the-store view runtime where data is accessed in terms of EDM entities and relationships and queried/updated using an entity-based SQL language (eSQL). The EntityClient provider includes the following services:

- **EDM/eSQL**. The EntityClient provider processes and exposes data in terms of the EDM values. Queries and updates are formulated using eSQL. They are processed through the query and update pipeline engines which incorporate mapping transformations and knowledge about the specific capabilities of the data sources.
- **Mapping**. View mapping, one of the key services of the EntityClient provider, is the subsystem that implements bidirectional (read and write) views that allow applications to manipulate data in terms of entities and relationships rather than rows and tables. The mapping from tables to entities is specified declaratively through a mapping definition language.
- **Store-specific bridge**. The bridge component is a service that supports the query execution capabilities of the query pipeline and coordinates the generation of queries using provider specific syntax.
- **Metadata services**. The metadata service supports all metadata discovery activities of the components running inside the EntityClient provider. All metadata associated with EDM concepts (entities, relationships, entitysets, relationshipsets), store concepts (tables, columns, constraints), and mapping concepts are exposed via metadata interfaces. The metadata services component also serves as a link between the domain modeling tools which support model-driven application design.
- **Transactions**. The EntityClient provider integrates with the transactional capabilities of the underlying stores.
- **API**. The API of the EntityClient provider follows the ADO.NET provider model based on Connection, Command, and DataReader objects. The EntityClient provider accepts commands in the form of eSQL text or canonical trees and produces DataReader objects as results.

Occasionally Connected Components. The Entity Framework enhances the well established disconnected programming model of the ADO.NET DataSet. In addition to enhancing the programming experiences around the typed and un-typed DataSets, the Entity Framework embraces the EDM to provide rich disconnected experiences around cached collections of entities and entitysets.

Embedded Database. The Entity Framework encompasses the capabilities of a low-memory footprint, embeddable database engine to enrich the services for applications that need rich middle-tier caching and disconnected programming experiences.

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

Programming Layers. ADO.NET 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. There are multiple mechanisms by which a programming layer may interact with the entity framework. One of the important mechanisms is LINQ expression trees.

Services. Rich SQL data services such as reporting, replication, business analysis will be built on top of the Entity Framework.

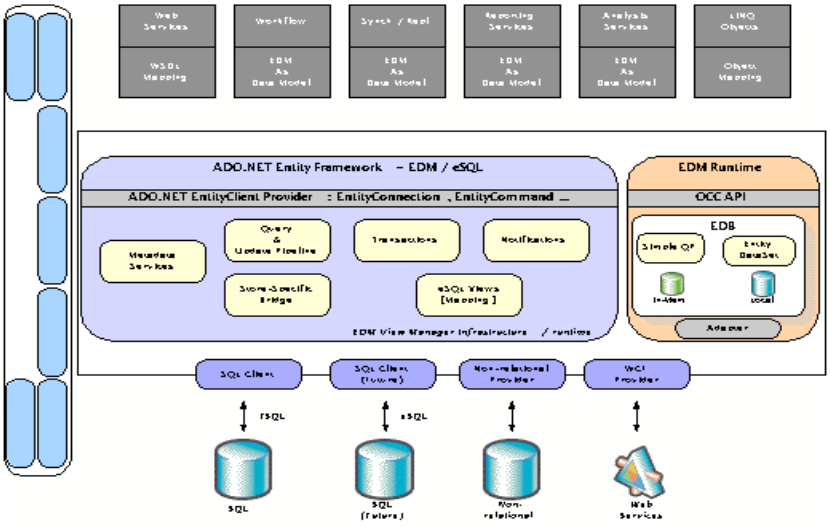

Figure 4 Entity Framework Architecture

4.4 Making the Conceptual Level real

This section outlines how one may define a conceptual model and work against it. We use a modified version of the Northwind database for familiarity.

4.4.1 Build the conceptual model

The first step is to define one's conceptual model. The EDM allows you to describe the model in terms of entities and relationships. The model may be defined explicitly by hand writing the XML serialized form of the model as shown

above. Alternately, a graphical EDM designer tool may be used.

4.4.2 *Apply the mapping*

After we define the EDM conceptual model, we identify a target store, and then map the conceptual model to the target store's logical schema model. As with the conceptual EDM, one can hand write an explicit mapping or use a mapping tool. For example, the Northwind store may stripe data across multiple tables (the vertical partitioning strategy); however, applications would want to reason about the data as a single entity without the need for joins or knowledge of the relational model. The mapping layers isolate the application from knowledge of the store's schemas.

4.4.3 *Automatically Generated Classes*

Having the conceptual level is indeed sufficient for many applications as it provides a domain model that is live within the context of a comfortable pattern (ADO.NET commands, connections and data readers) and allows for late bound scenarios. Many applications, however, prefer an object programming layer (See **Figure 5**). This can be facilitated through code generation driven from the EDM description. For increased flexibility and data independence between the object and conceptual level, a mapping may be defined between classes and the conceptual model. The mapping between classes and the conceptual model is a straightforward member-wise mapping. This enables applications built against these classes to be reused against other versions of the conceptual model, provided a legal map can be defined.

4.4.4 *Using Objects*

One can interact with objects and perform regular Create, Read, Update and Delete (CRUD) operations on the objects. The example below demonstrates the use of Language Integrated Query (LINQ) to identify all orders that are newer than a given date

```
class DataAccess
{
 static void GetNewOrders(DateTime date) {
  using (NorthWindDB nw =
          new NorthWindDB ()) {
   var orders = from o in nw.Orders
          where o.OrderDate > date
          select new {o.orderID, o.OrderDate,
                Total = o.OrderLines.Sum(
            l => l.Quantity);

foreach (SalesOrder o in orders) {
    Console.WriteLine("{0:d}\t{1}\t{2}",
      o.OrderDate, o.OrderId, o.Total);
   }
  }
}
```

4.4.5 *Using Values*

There are many ISVs, framework and data services developers who just prefer to work against a .NET data provider; the EntityClient Provider is intended for such usage scenarios. The EntityClient Provider has a connection and a command and returns a DbDataReader when one invokes EntityCommand.ExecuteReader(). An example of a query using the EntityCommandCommand is as follows:

```
public void DoValueQueries(DateTime date)
{
 using (EntityConnection conn =
          new EntityConnection (connString))
 {
  conn.Open();
  EntityCommand command =
          conn.CreateCommand();
  command.CommandText =
          @"select value e from Employees as e
    where e.HireDate > @HireDate";
  command.Parameters.Add(
          new EntityParameter ("HireDate",
                date));
  DbDataReader reader =
          command.ExecuteReader();
  while(reader.Read()) {
   //--- process record
  }
 }
}
```

5. SUMMARY AND CONCLUSION

Significant application and database technology trends require richer services at the conceptual rather than at the logical schema level. The Entity Framework provides a broad data platform with a rich and concrete conceptual schema to enable new applications and data services. The data platform includes the following components:

1. Entity Framework. A value-based runtime that implements an extended relational model - EDM - that embraces entities and relationships as first class concepts, a query language for the EDM, and a comprehensive mapping engine from the conceptual to the logical (relational) level.
2. Comprehensive programming model. We need programming model innovations that bridge the gap between different data representations (XML, relational, objects). In fact, by developing programming languages and APIs at the conceptual level, we will be able to liberate the programmer from the impedance mismatches that exist among different logical models. Programming language extensions such as Linq [5] provide richer, declarative programming models across different data representations.

3. Data services targeting the conceptual level. Examples include Synchronization/ Replication, Reporting, and Security.

4. Design-time tools. Data modeling tools today produce models that are largely abstract. They are used sometimes to produce a logical or physical design for a relational database implementation. We envision design-time tools that are used to: (a) build EDM models, (b) map EDM models to logical (relational) as well as other programming and presentation representations, and (c) semantics tools where you may introduce synonyms, aliases, translation and other semantic adornments for natural language and end user query.

6. ACKNOWLEDGMENTS

We would like to thank all members of the ADO.NET team for their contributions to building this system.

7. REFERENCES

[1] Chen, P. *The Entity-Relationship Model—toward a unified view of data*, ACM Transactions on Database Systems, Vol. 1, Issue 1, March 1976, pp. 9-36.

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

[3] Microsoft. The ADO.Net Entity Framework Overview. http://msdn.microsoft.com/data/default.aspx?pull=/library/en-us/dnvs05/html/ADONETEnFrmOvw.asp, June 2006.

[4] Blakeley, J.A., Campbell, D., Gray, J., Muralidhar, S., Nori, A.. Next-Generation Data Access: Making the Conceptual Level Real. http://msdn.microsoft.com/data/default.aspx?pull=/library/en-us/dnvs05/html/nxtgenda.asp, June 2006.

[5] Microsoft. The Linq Project. http://msdn.microsoft.com/data/ref/linq/default.aspx.

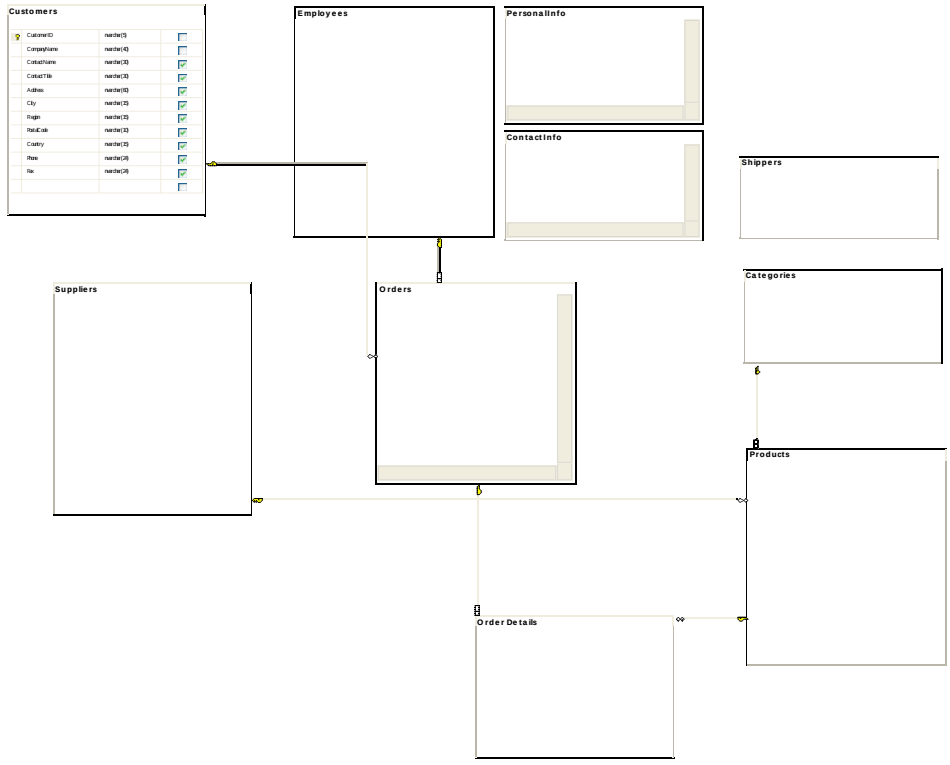

Figure 5: Entity Data Model for Northwind.