Next-Generation Data Access: Making the Conceptual Level Real

José Blakeley, David Campbell, Jim Gray, S. Muralidhar, Anil Nori

June 2006

Applies to:
ADO.NET
.NET Language Integrated Query (LINQ)
SQL Server

Summary: Eliminate the impedance mismatch for both applications and data services like reporting, analysis, and replication offered as part of the SQL Server product by raising the level of abstraction from the logical (relational) level to the conceptual (entity) level. (31 printed pages)

Contents

Abstract
Introduction
Impedance Mismatch
Database Modeling Layers
Data Services Evolution
The Vision
Making the Conceptual Level Real
Entity Framework Architecture
References

Abstract

Significant technology and industry trends have fundamentally changed the way that applications are being built. Line of business (LOB) applications that were constructed as monoliths around a relational database system 10-20 years ago must now connect with other systems and produce and consume data from a variety of disparate sources. Business processes have moved from semi-automated to autonomous. Service oriented architectures (SOA) introduce new consistency and coordination requirements. Higher level data services, such as reporting, data mining, analysis, synchronization, and complex integration have moved from esoteric to mainstream.

A common theme throughout all modern application architectures is a need to transform data from one form to another to have it in the right form for the task at hand. Today's applications sport a number of data transformers. A common transformation usually encapsulated as a proprietary data access layer inside applications is designed to minimize the impedance mismatch between application objects and relational rows. However, other mappings to navigate object-xml, and relational-xml exist. This impedance mismatch is not unique to applications. As SQL Server has evolved as a product, it has had to add a number of these modeling and mapping mechanisms across the services it provides within the product. Most of these mappings are produced in a point-to-point fashion and each requires a different means to describe the point-to-point transformation.

A fundamental insight is that most traditional data centric services such as query, replication, ETL, have been implemented at the logical schema level. However, the vast majority of new data centric services best operate on artifacts typically associated with a conceptual data model. The essence of our data platform vision is to elevate Microsoft's data services, across several products, from their respective logical schema levels to the conceptual schema level. Reifying the conceptual schema layer allows us to create services around common abstractions and share tooling, definition, and models across the majority of our data services. We will demonstrate in this paper how this shift will profoundly impact our ability to provide value across our entire application platform.

This paper focuses on programming against data and how by raising the level of abstraction from the logical (relational) level to the conceptual (entity) level we can eliminate the impedance mismatch for both applications and data services like reporting, analysis, and replication offered as part of the SQL Server product. The conceptual data model is made real by the creation of an extended relational model, called the entity data model (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, entity-xml transformers. Collectively, all these services are called the Entity Framework. ADO.NET, the Entity Framework, and .NET Language Integrated Query (LINQ) innovations in C# and Visual Basic represent a next-generation data access platform for Microsoft.

Introduction

The Microsoft Data Access vision supports a family of ***products and services so customers derive value from all data, birth through archival.*** While the vision statement does not include explicit verbiage, the goal of the vision is to provide products and services for data across all tiers of the application (solution). Such a complete data platform must have the following characteristics:

Data in All Tiers. A complete data platform provides data management and data access services everywhere. In the client-server world, "everywhere" includes data services on the client and the data server; in the enterprise world, "everywhere" includes the data server tier, the app server (mid) tier, and the client tier; the mobile world includes the mobile device tier;

and the next generation web (cloud) world includes data in the shared web space.

All types of Data. Increasingly, applications incorporate a variety of data—e.g. XML, email, calendar, files, documents, and structured business data. The Microsoft Data Access vision supports an integrated store vision that can store and manage all of this data, secure it, search and query it, analyze it, share it, synchronize it, etc. Such an integrated store includes the core data management capabilities and a platform for application development.

Uniform Data Access. While applications in different tiers require different kinds of data management services, they all expect (require) significant uniformity in application development environment (programming models and tools). Often, the same application may be deployed across multiple tiers (e.g. on the devices and on the desktops) and it is highly desirable to develop once and deploy on different tiers. In addition, as application scale needs increase and the application moves up tiers, it must be possible to upsize the database (e.g. from SQL Everywhere to SQL Express to SQL Server), without requiring (significant) application changes. Support for uniform application development requires: Rich Data Modeling to match the required abstractions for application data, Rich and Consistent Programming environment, and Tools for all data.

End-to-End Business Insight. End-to-end business insight is all about enabling better decision making. It is about the technology that can enable our customers to collect, clean, store and prepare their business data for the decision making process. It is also about the experiences that the Business Users and Information Workers will have when accessing, analyzing, visualizing and reporting on the data while gathering the information necessary for their decisions.

Ubiquitous Data Services. Applications invest significant effort in (custom) development of services like data security, synchronization, serialization for data exchange (or for web services), analytics, and reporting over data in all the tiers, over abstractions that are close to the applications' perspective. The Microsoft Data Access vision expects offering such services, in a uniform manner, across data in all tiers.

Rich "Abilities". Data is a key asset, whether it is an enterprise, corporate, or a home user. Customers want their data to be always available; it must be secured; the access must be performant; their applications must scale and supportable. The Microsoft Data Access vision presumes rich "abilities" on all data. In addition, it provides ease of management of all data, thereby significantly minimizing the TCO of the data.

Impedance Mismatch

A key issue addressed by the next-generation data access platform is the well-known *application impedance mismatch problem*. Consider the way in which data access applications are written today. Data access code has not changed significantly in the last 10-15 years. The data access patterns introduced in the ODBC are still present in OLE-DB, JDBC, and ADO.NET. Here is an example of ADO.NET today, similar examples can be written in other APIs.

```
class DataAccess

{

 static void GetNewOrders(DateTime date, int qty) {
 using (SqlConnection con =

 new SqlConnection(Settings.Default.NWDB)) {
             con.Open();

 SqlCommand cmd = con.CreateCommand();
             cmd.CommandText = @"
                  SELECT o.OrderDate, o.OrderID, SUM(d.Quantity) as Total
                  FROM Orders AS o
                  LEFT JOIN [Order Details] AS d ON o.OrderID = d.OrderID
                  WHERE o.OrderDate >= @date
                  GROUP BY o.OrderID
                  HAVING Total >= 1000";

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

 DbDataReader r = cmd.ExecuteReader();
 while (r.Read()) {
 Console.WriteLine("{0:d}:\t{1}:\t{2}", r["OrderDate"],

                         r["OrderID"], r["Total"]);
             }
         }
     }
}
```

This code pesents several inconveniences to the developer. The query is expressed by text strings opaque to the programming language. The query includes a left-outer join needed to assemble rows from the normalized orders and the order details tables and is not directly related to the business request. Results are returned in untyped data records. A more elegant code that leverages the Entity Framework in ADO.NET as well as the language integration innovations in .NET Language Integrated Query (LINQ) would be:

```
class DataAccess
{

 static void GetNewOrders(DateTime date, int qty) {

 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);

           }
       }
   }
}
```

While .NET applications like to access data as CLR objects, most data services like reporting and replication prefer to access data as entity values. Such data services would formulate dynamic queries over entities as follows:

```
MapCommand cmd = con.CreateCommand();
cmd.CommandText = @"
    SELECT o.OrderDate, o.OrderId, o.Total
    FROM  (SELECT o.OrderId,
                SUM(o.OrderDetails..Quantity) AS Total,
            FROM Orders AS o
             WHERE o.OrderDate >= @date) AS o
    WHERE o.Total >= 1000";
cmd.Parameters.AddWithValue("@date", date);
```

There is a strong need for applications and data services to manipulate data at the conceptual level and to have mechanisms built into their development frameworks to easily transform these concepts into objects that are more natural for the application to reason with.

While the innovations introduced by the Entity Framework in ADO.NET and LINQ bring an exciting new world to application developers by solving the long-standing impedance mismatch problem our responsibility as platform vendor is to introduce these innovations in an *evolutionary* manner that preserves existing code investments in ADO.NET 2.0. Specifically, we will ensure that no rewriting of existing applications is needed to leverage some of the benefits of the Entity Framework such as rich conceptual model, data independence of ADO.NET code from logical (relational) schema, and an entity query language firmly rooted in SQL. We will ensure ADO.NET applications can opportunistically adopt capabilities of the Entity Framework when appropriate. An important design goal of the Entity Framework is to provide fine-grained control and extensibility to the developer. This means we do not provide an 80% solution that hits a brick wall and instead allow applications written against the Entity Framework to reach lower level ADO.NET components when needed. The integration of LINQ on top of entities allows ADO.NET developers to enjoy the innovations of LINQ and the powerful relational-to-entity mapping capabilities offered by the Entity Framework.

Database Modeling Layers

In order to describe the higher level modeling capabilities of the next-generation data platform we need to introduce some basic database design concepts. Today's dominant information modeling methodology for producing database designs factors an information model into four main levels: physical, logical (relational), conceptual, and programming or presentation levels described.

Physical Model Level

The physical model describes how data is *represented* in physical resources such as memory, wire or disk. The concepts discussed in this layer include record formats, file partitions and groups, heaps, and indexes. The physical model is described in this document primarily for completeness since it is typically invisible to the application. Changes to the physical model usually do not impact the application logic, and manifest themselves only in the way the application performs. Applications usually target the logical or relational data model described next.

Logical Model Level

A logical data model is a complete and precise information model of the target domain. The relational model, due to its dominance, is typically the target representation of a logical data model. The concepts discussed at the logical level include tables, rows, and primary key-foreign key constraints. This is the layer where database normalization concepts (i.e., vertical partitioning of data) are used. In fact, it is not uncommon to find relational schemas that display a high degree of normalization. The logical model level has been the target of relational database applications for the last 20 years. The mapping between the logical and physical levels is entirely outside the realm of the application and is performed implicitly by the relational database system. The ability to isolate applications written against the logical level from changes at the physical level &150; such as adding or dropping indexes and physical reorganization of records—is referred to as *data independence* and is considered one of the main benefits of the relational model. In a relational database system, the SQL query language operates over tables and views but does not (typically) have to know about the various indexes available at the physical layer.

The sometimes high degree of normalization found in relational schemas today helps to satisfy important application requirements such as data consistency and increased concurrency with respect to updates and OLTP performance. However, normalization introduces challenges to applications in that data at the logical level is too fragmented and the 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.

Conceptual Model Level

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 [CHEN76]. UML is a more recent example of a conceptual model [UML].

Most significant applications involve a conceptual design phase early in the application development lifecycle. Today, many people interpret "conceptual" as "abstract" because the conceptual data model is captured inside a database design tool that has no connection with the code and the logical relational schema used to implement the application. The database design diagrams 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. There is no reason why a conceptual model could not be embodied concretely by a database system. A goal of the Microsoft Data Access vision is to make the conceptual data model a concrete feature of the data platform.

Most people associate the task of transforming a conceptual model into a logical model as Normal Form transformations into a logical *relational* model. This doesn't necessarily need to be the case as we'll show later. Just like relational systems provide data independence between the logical and physical levels, a system implementing a conceptual model can provide data independence between the conceptual and logical levels. The isolation of applications targeting the conceptual level from logical level changes is highly desirable. In a recent survey conducted by the Microsoft Developer Division the need to isolate applications from changes to the (relational) logical level was one of the top ranked feature requests.

The conceptual model in the data platform is embodied by the Entity Data Model (EDM) [**EDM**]. The central concepts in the EDM are entities and relationships. *Entities* are instances of Entity Types (e.g., Customer, Employee) which are richly structured records with a key. An entity key is formed from a subset of properties of the Entity Type. The key (e.g., CustId, EmpId) is a fundamental concept to uniquely identify and update entity instances and to allow entity instances to participate in relationships. Entities are grouped in Entity Sets (i.e., Customers is a set of Customer instances). *Relationships* relate entity instances and are instances of Relationship Types (e.g., Employee *WorksFor* Department). Relationships are grouped in Relationship Sets.

The EDM works in conjunction with the eSQL query language [**ESQL**], which is an evolution of SQL designed to enable set-oriented, declarative queries and updates over entities and relationships. EDM can work with other query languages as well. EDM and eSQL together represent a conceptual data model and query language for the data platform and have been designed to enable business applications such as CRM and ERP, data services such as reporting, analysis, and synchronization, and applications to model and manipulate data at a level of abstraction and semantics closer to their needs. The EDM is a value-based conceptual model. It does not incorporate behaviors of any kind. It is also the basis for the programming/presentation level described in the next section.

Programming and Presentation Level

The entities and relationships of the conceptual model usually need to be manifested in different forms based on the task at hand. Some entities need to be transformed into objects amenable to the programming language implementing the application business logic. Some entities need to be transformed into XML streams as a serialization format for web services invocations. Other entities need to be transformed into in-memory structures such as lists, dictionaries, or data tables for the purposes of UI 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.

Often, proponents of a particular presentation or programming model will argue that their particular "presentation" view is the one truth. 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 mapping to and from various presentation models and other higher level services. Most developers, and most of our modern services as we will point out later, want to reason about high-level concepts such as an "Order" (See Figure 1) not about the several tables that it is normalized over in a relational database schema. They want to query, secure, program, report on the order. Most developers implicitly *think* about the order when they design their application. It may be in their head, a UML diagram, or their whiteboard. 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.

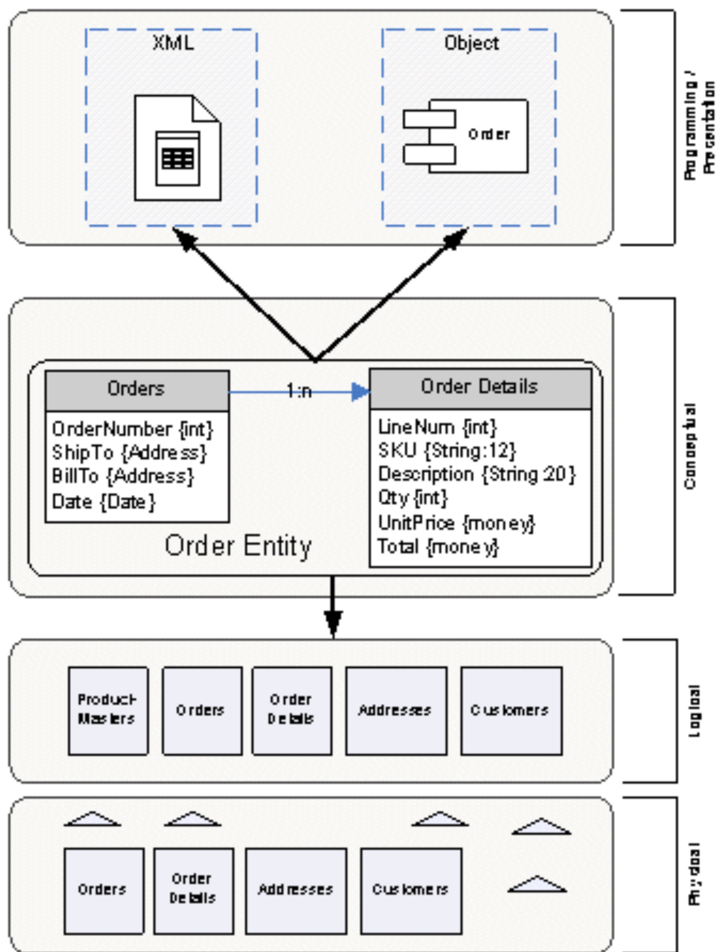

Figure 1. Physical, logical, conceptual and multiple programming and presentation views of an Order. (Click on the image for a larger picture)

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. The key point here is that the impedance mismatch problem illustrated in Section 2 is not unique to applications, but it is also a challenge in building higher-level data services such as reporting and replication.

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 level. Let's take an order entry system as an example.

Order Entry Circa 1985

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. See Figure 2.

The key point is that the application logic was tightly bound to the logical relational schema. The new-order screen would reference an existing customer, assemble the order information from products, orders and order details. Submission of the screen would begin a transaction and invoke logic to insert a new order in the "orders" table containing date, order status, customer ID, ship-to, bill-to, etc. The new-order routine would also insert a number of line items into the order-details table. Many first generation RDBMS did not support foreign key constraint checking at the database level so the application logic had to ensure that every new order had a valid customer ID or that deleting of an order-header row required deleting all associated order-details. The batch processes to update master data such as SKU and pricing information also worked at the logical schema level. People typically wrote batch programs to interact directly with the logical schema to perform these updates. One other important point to make is that, 20 years ago, virtually all of the database services (insert, delete, query, bulk load) were also built at the logical schema level. Programming languages did not support representation of high-level abstractions directly—objects did not exist.

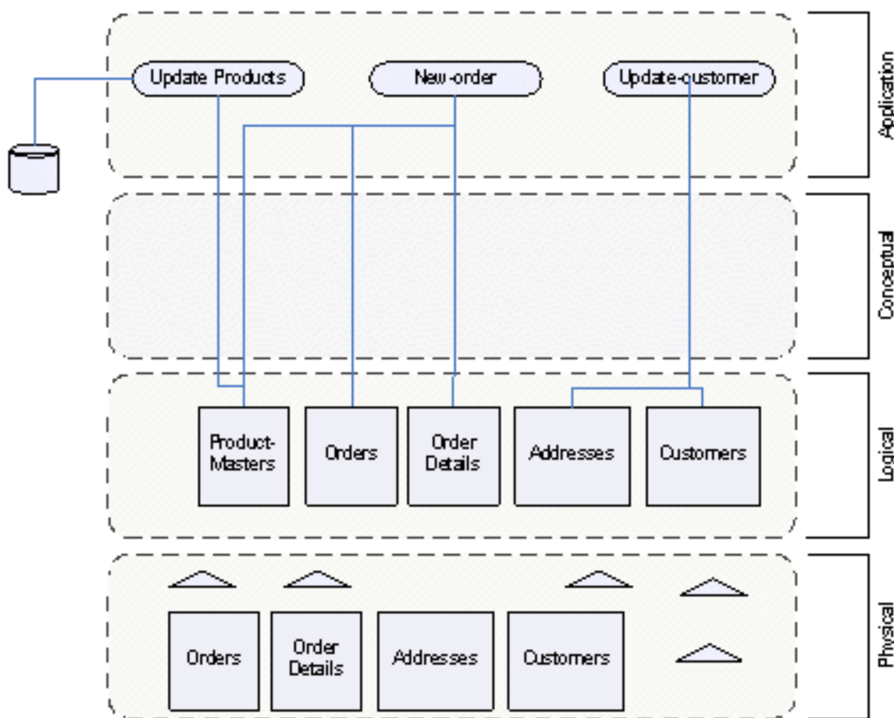

Figure 2. Order Entry System circa 1985 (Click on the image for a larger picture)

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.

Order Entry Circa 2005

Several significant trends have shaped the way that modern data-based applications are factored and deployed. Chief among these are object oriented factoring, service level application composition, and higher level data services.

Object Oriented Factoring

Object oriented methodologies replaced verb-object functional factoring by associating logic with "Business Objects". Business objects typically follow the conceptual "entity" factoring that would have been created as part of a conceptual schema model. Entities such as "Customers", "Orders", "ProductMaster" become business objects. Relationships between business object entities, captured during conceptual schema design, are implemented as methods on business objects. Thus, the "Customer" business object has a method called "GetOpenOrders" which returns a collection of that customer's open orders. With the advent of business objects, an additional abstraction layer was added in the application and, rather than having free variables collected in a new-order screen be sent to the database as columns in rows, the business object maintained the state of the customer entity which could be retrieved or persisted to the database implicitly or explicitly.

Service Level Application Composition

An order entry application 20 years ago was typically implemented as a single monolith around an RDBMS. Reference and resource data, such as pricing, product master, and true inventory were generally updated at the logical schema level via batch jobs that extracted the data from other systems into flat files and then loaded them into the order entry system via bulk load utilities at the logical schema level.

Today's order entry system is more likely composed of services provided by other systems. These services are invoked on an as needed basis. Furthermore, the order entry system must respond to service requests and events initiated by other systems. Let's look at inventory management. Instead of getting a batch update of inventory levels from another system on a daily basis, it is more likely that an order entry system will send a service request to a distribution system to check on inventory levels. Or, an order management system might simply receive stock level notifications, such as "In Stock" or "Backordered" from the distribution system. These stock level notifications might be fairly stateful such as an "inventory low" notification that lets the order management system know that there are only 3 days of inventory available at current forecast sales. Or a "Backordered" stock notification could indicate when the next shipment is due to arrive at the distribution facility. A key point is that "StockNotification" is a conceptual entity. In our order entry system, a "StockNotification" may be resident in memory as a business object; stored in a durable queue between systems; saved in a database for subsequent analytics to evaluate supplier performance; or serialized as XML as a web service "noun", etc. In all of these forms "StockNotification" still contains the same conceptual structure; however, its logical schema; physical representation, and what services can be bound to its current representation depend on the current use context. What this means is that a data centric view of these conceptual entities offers an opportunity to abstract and provide additional platform services over them. In essence, the data representing the state of a concept in a modern application needs to be mapped into various representations and bound to various services throughout the application.

View of Today's Order Entry System

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. When we first design the system, we think about "StockNotifications". 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 3 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.

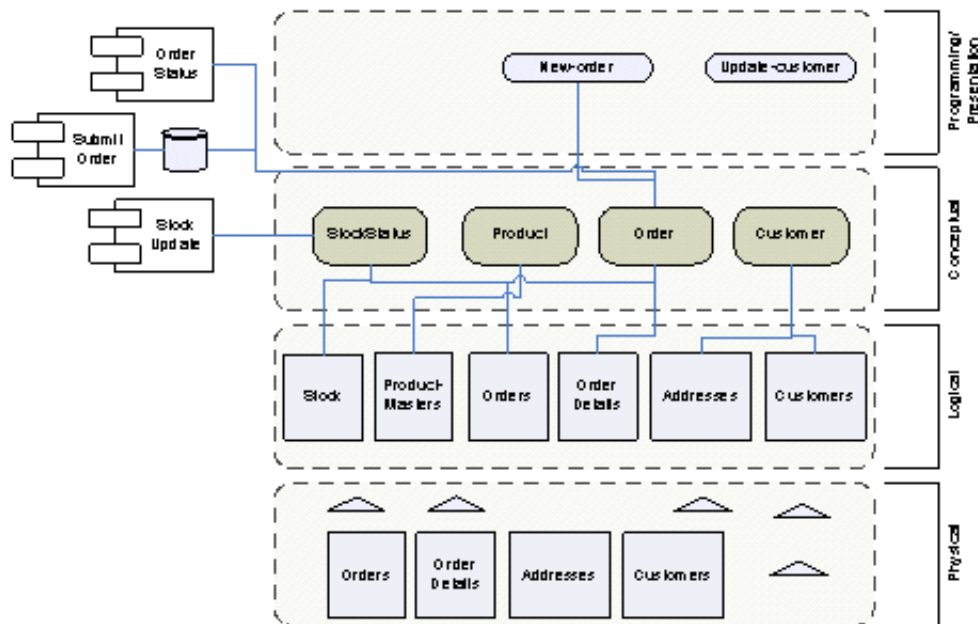

Figure 3. Order Entry System circa 2005 (Click on the image for a larger picture)

SQL Server Evolution

The data services provided by the "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 amongst 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 great 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. In a relational database system the ACID guarantees of the concurrency control system are what prevent chaos. If an order consists of a single order-header and 5 order details, isolation provided by the concurrency control system prevents other agents on the system from seeing or processing the order in an inconsistent state when say only 3 of the 5 order details have been inserted. In the midst of a transaction, the database is often in a logically inconsistent state—for example when only 3 of the 5 order details are inserted or money has been debited from one account before being credited to another in a transfer. Other agents in the system only see the consistent "before state" or "after state". Since merge replication operated at the logical schema or row level it could not capture that the data representing a new order, while inserted in a single isolated transaction on one system, was to be installed in an isolated fashion when replicated. So, if the replication transport failed after transferring 3 out of 5 order details, the order processing system picking up and processing the inconsistent new order had no way of knowing that it was not yet complete. 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 not a 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. SQL Server Reporting Services (SSRS) has added incredible value to the SQL product. 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. Once SSRS was released, there was incredible demand for an end user report authoring environment that didn't require a developer to fire up Visual Studio and author SQL queries for reports. End users and analysts want to write reports directly over Customers, Orders, Sales, etc. They are business people who think at the business concept, or "domain", level and want to express their queries at this level rather than at the logical schema level. Virtually all "end user" and "English query" reporting environments require this. 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.

The key point is that one key driver in SQL Server's success is that it is delivering higher level data services at the conceptual schema level. Currently, each of these services has a separate tools to describe conceptual entities and map them down to the underlying logical schema level. If you have a Customer in your problem domain, you need to define it one way for merge replication, another way for report builder, and so on.

Figure 4 shows SQL Server as it existed in Version 6.0—essentially a traditional RDBMS with all data centric services provided at the logical schema level. **Figure** 5 demonstrates the evolution of SQL Server into a data platform with many high value data services and multiple means to reify and map conceptual entities to their underlying logical schemata.

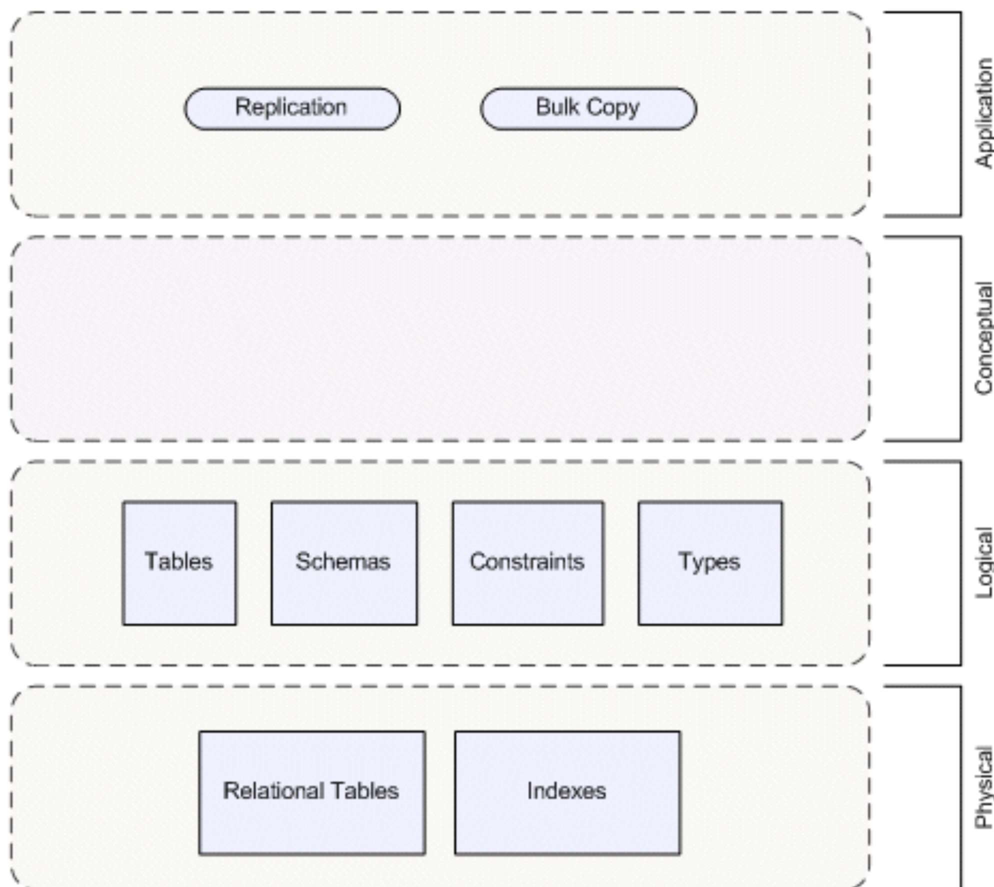

Figure 4. SQL Server 1995

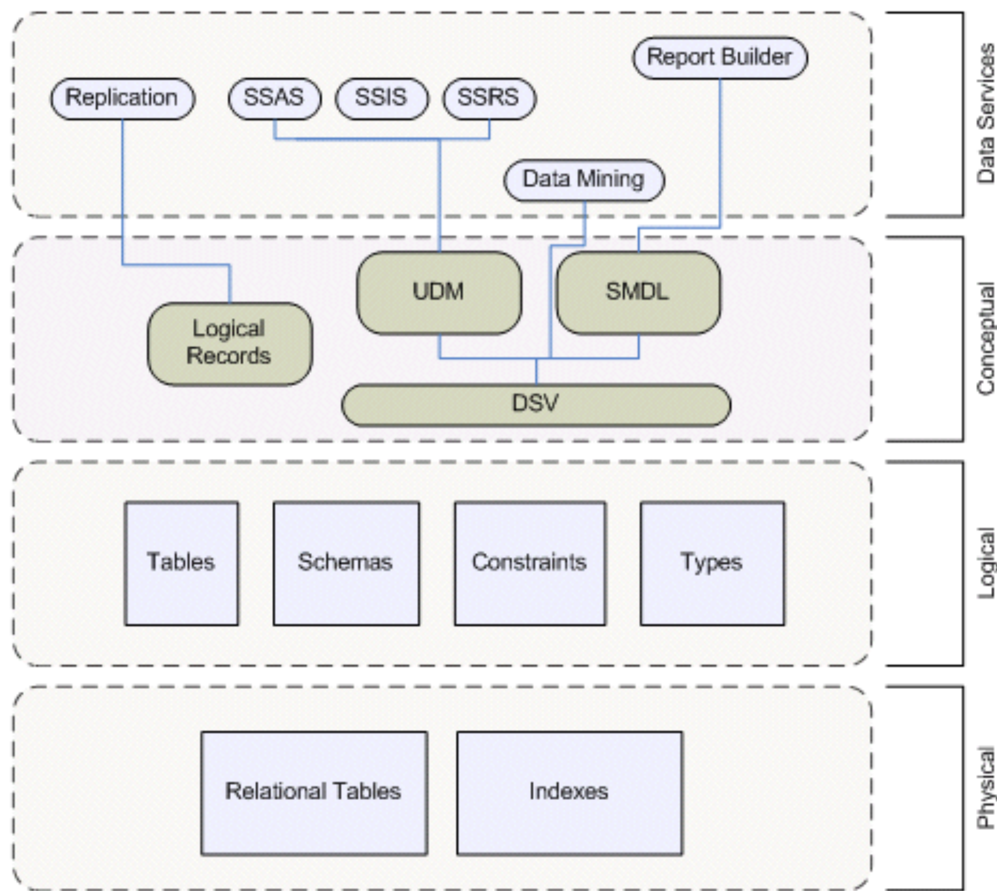

Figure 5. SQL Server 2005

Other Data Services

While we have motivated the need for making the conceptual level real from the perspective of the SQL data platform it should be apparent that this mapping from real services at the conceptual level to any number of logical and programming/presentation representations is occurring in multiple areas. Consider the web services serialization "Data Contract" which maps between a CLR object state representation to its serialized XML form. If one does "contract first" services development, the nouns in the service contact are really state representations of entities—think back to the example of the "StockNotification". This trend will continue with:

- Database "search". Many are working on performing "search" over structured storage—essentially trying to bridge the gap between query languages and search expressions. There are two vexing questions when trying to implement database search over a logical relational schema:
- What does one return for a result? Returning disassociated addresses when matching on 98052 as a ZIP code is not very useful. How then do we return something of value when a search expression of "customer zip 98052" is presented? How about "Pizza zip 98052"? Note that making the conceptual level real can both help understand the semantics of the search expression and help shape what is returned as a result.
- How does one implement a security scheme over database search? We must assume that the search expression domain is over the entire database. How then do we implement security on what we uncover in the database? Authorizing based upon access to the underlying logical schema only goes so far. The real liberation comes if we are able to raise the authorization scheme up from the logical schema level to the conceptual level—only there can we secure what matters: Customers, Orders and ProductMasters.
- Office documents as a content model: Office 12 takes a major step in moving from a presentation format model to a content model. Thus, an Office document can be seen as an entity container bringing together the worlds of structured, semi-structured, and unstructured data. If we had a real conceptual model for an insurance domain, we could create an InfoPath form for claims processing by dragging entities from a designer into an InfoPath design surface. An adjuster can go to a claim scene and populate a form that both creates and references entities such as: PolicyHolder, Claim, Policy, Claimant, etc. They could also capture unstructured data such as a sketch of the scene, photographs of the damage, etc. This rich document can be "Submitted" into an application platform that has a rich conceptual notion of PolicyHolder, Claim, Date, Location, etc. Services can be built directly on top of these concepts —it is easy for someone to write a report listing Incidents occurring at a particular Location over the last year. The claim information can be easily extracted from the document and used as the noun in a web service that initiates a new claim process. Note that both data-centric document consumption into the platform and data-centric document production from the platform are enabled in this new world.

The Vision

So, what then is the vision for the next-generation data access? Namely, to acknowledge that significant technology and application trends are moving us towards providing richer services at the conceptual rather than at the logical schema level and to seize this as an opportunity to provide a broad platform around making the conceptual schema level real in a way that allows us to realize extensive platform value on top of it.

What services should be elevated? Briefly:

1. **Data Modeling**—we need to provide a data model that can define the structure essence of entities in a way that allows them to be represented in a variety of logical representations. This model must have a concrete representation that allows us to manipulate data at this level, and to build real design-time tools and runtime support around it. The platform needs to have a set-based query and update language for entities and relationships. Note that the entity data model is a conceptual model exposing values not objects. This is extremely important because the data platform needs to manage data at the highest possible semantic level without tying itself to the particular mechanisms of a presentation layer. This enables other data services such as reporting, replication, and analysis to leverage the mapping and query infrastructure that supports the conceptual level.
2. **Mapping**—having a concrete entity model allows us to build out a series of mappings to a variety of logical and presentation representations. Certainly there will be default mappings but part of our platform value will be in providing tools to map to and from foreign representations whether it is to federate product catalogs, normalize purchase order forms into our internal canonical standard, or to provide heterogeneous synchronization to a variety of operational systems as a Master Data Management solution.
3. **Design-time tools**—Entity based tools today produce models that are mainly destined for the plotter so that large pictures of a modeling effort can be enshrined on wall somewhere to quickly fall out of date. Some people use them to produce a logical or physical design for a relational database implementation but that's about as far as it goes. Our data access tooling should be layered and factored to meet a number of needs over a base entity model:
   a. **Base entity tooling**—this is used to design entities as well as their relationships.
   b. **Mapping**—tooling to map from a conceptual entity into a number of logical and (and perhaps physical) representations.
   c. **Semantic tooling**—can be built for higher level semantic services such as SQL Server Report builder where you may introduce synonyms, aliases, translation and other semantic adornments for natural language and end user query.
4. **Transformation runtime**—once we can describe and encode mappings, we can build machinery to automatically map entities into multiple representations. We can retrieve a "customer" from the database; realize it as an object; render it as XML for exposure through a web service all in a declarative fashion driven by the mapping descriptions.
5. **Comprehensive programming model**—we need programming models that bridge the gap between different logical 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. Further, these programming models should support development of application business logic that can run inside or outside the data store depending on deployment, performance, and scalability requirements of the application.
6. Data services targeting the conceptual level. Some examples include:
   a. **Synchronization**—many entity synchronization services can be made platform level services through this vision.
   b. **Security**—we will want to build out security services at the entity level.
   c. **Report builder**—this service is already taking steps in this direction through the semantic data model (SMDL).
   d. **Administration**—beyond security, operations like archive can benefit from a conceptual perspective.

The force motivating the move towards conceptual level services should be clear at this point. We can see evidence all around from the stovepipes we are building to model, tool, and transform from conceptual entities to logical and physical representations. The Data Access vision that we are proposing seeks to establish a significant platform shift by unifying these stovepipes in a coordinated, cross-product, multi-release quest.

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.

Build the Conceptual Model

The first step is to define one's conceptual model. The EDM represents a formal, design and run-time expression of such a model. The EDM allows you to describe the model in terms of entities and relationships. Ideally, there shall be two ways to define a model from scratch. One may define the model explicitly by hand writing the XML serialized form of the model as shown below. The other through a graphical EDM designer tool.

```xml
<?xml version="1.0"?>

<Schema Namespace="CNorthwind"
   xmlns="urn:schemas-microsoft-com:windows:storage">

<!-
Typical Entity definition, has identitiy [the key] and a some members
-->
  <EntityType Name="Product" Key="ProductID">
    <Property Name="ProductID" Type="System.Int32" Nullable="false" />
    <Property Name="ProductName" Type="System.String"
         Nullable="false" Size="max" />
    <Property Name="QuantityPerUnit" Type="System.String"
         Nullable="false" Size="max" />
    <Property Name="ReorderLevel" Type="System.Int16" Nullable="false" />

    <Property Name="UnitPrice" Type="System.Decimal" Nullable="false" />
    <Property Name="UnitsInStock" Type="System.Int16" Nullable="false" />
    <Property Name="UnitsOnOrder" Type="System.Int16" Nullable="false" />
  </EntityType>

<!-
A derived product, we can map TPH, TPC, TPT
-->
  <EntityType Name="DiscontinuedProduct" BaseType="Product">

    <Property Name="DiscReason" Type="System.String"
         Nullable="false" Size="max" />
  </EntityType>

<!-
A complex type defines structure but no identity. I can
   be used inline in 0 or more Entity definitions
-->
   <ComplexType Name="CtAddress" >
      <Property Name="Address" Type="System.String"
         Nullable="false" Size="max" />
      <Property Name="City" Type="System.String"
         Nullable="false" Size="max" />

      <Property Name="PostalCode" Type="System.String"
         Nullable="false" Size="max" />
      <Property Name="Region" Type="System.String"
         Nullable="false" Size="max" />
      <Property Name="Fax" Type="System.String"
         Nullable="false" Size="max" />
      <Property Name="Country" Type="System.String"
         Nullable="false" Size="max" />
      <Property Name="Phone" Type="System.String"
         Nullable="false" Size="max" />
   </ComplexType>

   <EntityType Name="Customer" Key="CustomerID">
<!- Address is a member which references a complex type inline -->
      <Property Name="Address" Type="CNorthwind.CtAddress"
         Nullable="false" />
      <Property Name="CompanyName" Type="System.String"
         Nullable="false" Size="max" />
      <Property Name="ContactName" Type="System.String"
         Nullable="false" Size="max" />
      <Property Name="ContactTitle" Type="System.String"
         Nullable="false" Size="max" />
      <Property Name="CustomerID" Type="System.String"
         Nullable="false" 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>


<!-
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)
-->
  <EntityContainerType Name="CNorthwind">

    <Property Name="Products" Type="EntitySet(Product)" />
    <Property Name="Customers" Type="EntitySet(Customer)" />
    <Property Name="Order_Details" Type="EntitySet(OrderDetail)" />
    <Property Name="Orders" Type="EntitySet(Order)" />

    <Property Name="Order_DetailsOrders"
            Type="RelationshipSet(Order_DetailsOrders)">
      <End Name="Order" Extent="Orders" />

      <End Name="Order_Details" Extent="Order_Details" />
    </Property>
```

```
    <Property Name="Order_DetailsProducts"
            Type="RelationshipSet(Order_DetailsProducts)">
      <End Name="Product" Extent="Products" />
      <End Name="Order_Details" Extent="Order_Details"/>
    </Property>

  </EntityContainerType>

</Schema>
```

Apply the Mapping

Once one has an EDM conceptual model, providing that one has a target store already defined, we can map to the target store's logical schema model.

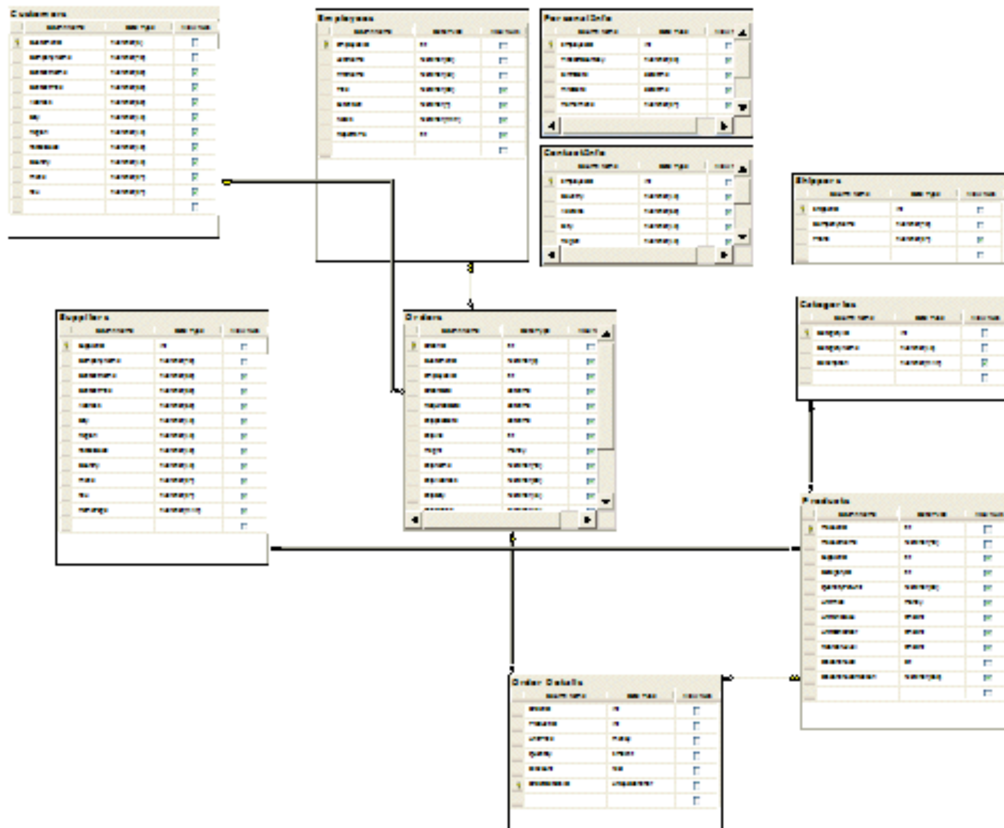

Figure 6. Entity Data Model for Northwind (Click on the image for a larger picture)

Of course this model can be expressed in SQL DDL, for instance the Employees Table may look like:

```
CREATE TABLE [dbo].[Employees](
   [EmployeeID] [int] NOT NULL,
   [LastName] [nvarchar](20)
         COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
   [FirstName] [nvarchar](10)
         COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
   [Title] [nvarchar](30) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
   [Extension] [nvarchar](4) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
   [Notes] [nvarchar](max) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
   [ReportsTo] [int] NULL,
 CONSTRAINT [PK_Employees] PRIMARY KEY CLUSTERED
(
   [EmployeeID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
```

As with the conceptual EDM, one can hand write an explicit mapping or use a mapping tool. Figure 7 is a representation of a mapping design tool in action mapping the above EDM conceptual model to a modified version of the Northwind logical schema..

The items on the left represent the constructs in the EDM model. The items on the right represent the constructs in the store's logical model. In this example we can see one of the motivations for modifying Northwind. The desire was to reflect a common strategy for vertically partitioning data in separate tables in the store. Although one does this, ideally one would want to reason about the data as a single entity without the need for joins or knowledge of the logical model. An entity like Employee can be defined and mapped across multiple tables in the store. Following is the serialized form of the mapping, note that we map at the EntitySet level and allow table fragments which express the tables and join conditions that make up an entity with in a given EntitySet:

```
<EntitySetMapping cdm:Name='Employees'>
```

```
      <EntityTypeMapping cdm:TypeName='IsTypeOf(CNorthwind.Employee)'>
        <TableMappingFragment cdm:TableName='ContactInfo'>
          <EntityKey>
            <ScalarProperty cdm:Name='EmployeeID'
                        cdm:ColumnName='EmployeeID' />

          </EntityKey>
          <ScalarProperty cdm:Name='Address' cdm:ColumnName='Address' />
          <ScalarProperty cdm:Name='City' cdm:ColumnName='City' />
          <ScalarProperty cdm:Name='Country' cdm:ColumnName='Country' />
          <ScalarProperty cdm:Name='PostalCode'
                        cdm:ColumnName='PostalCode' />
          <ScalarProperty cdm:Name='Region' cdm:ColumnName='Region' />

        </TableMappingFragment>
      </EntityTypeMapping>
      <EntityTypeMapping cdm:TypeName='IsTypeOf(CNorthwind.Employee)'>
        <TableMappingFragment cdm:TableName='Employees'>
          <EntityKey>
            <ScalarProperty cdm:Name='EmployeeID'
                        cdm:ColumnName='EmployeeID' />

          </EntityKey>
          <ScalarProperty cdm:Name='Extension'
                        cdm:ColumnName='Extension' />
          <ScalarProperty cdm:Name='FirstName'
                        cdm:ColumnName='FirstName' />
          <ScalarProperty cdm:Name='LastName' cdm:ColumnName='LastName' />
          <ScalarProperty cdm:Name='Notes' cdm:ColumnName='Notes' />
          <ScalarProperty cdm:Name='ReportsTo'
                        cdm:ColumnName='ReportsTo' />

          <ScalarProperty cdm:Name='Title' cdm:ColumnName='Title' />
        </TableMappingFragment>
      </EntityTypeMapping>
      <EntityTypeMapping cdm:TypeName='IsTypeOf(CNorthwind.Employee)'>
        <TableMappingFragment cdm:TableName='PersonalInfo'>
          <EntityKey>

            <ScalarProperty cdm:Name='EmployeeID'
                        cdm:ColumnName='EmployeeID' />
          </EntityKey>
          <ScalarProperty cdm:Name='BirthDate'
                        cdm:ColumnName='BirthDate' />
          <ScalarProperty cdm:Name='HireDate' cdm:ColumnName='HireDate' />
          <ScalarProperty cdm:Name='HomePhone'
                        cdm:ColumnName='HomePhone' />
          <ScalarProperty cdm:Name='Photo' cdm:ColumnName='Photo' />

          <ScalarProperty cdm:Name='TitleOfCourtesy'
                        cdm:ColumnName='TitleOfCourtesy' />
        </TableMappingFragment>
      </EntityTypeMapping>
    </EntitySetMapping>
```

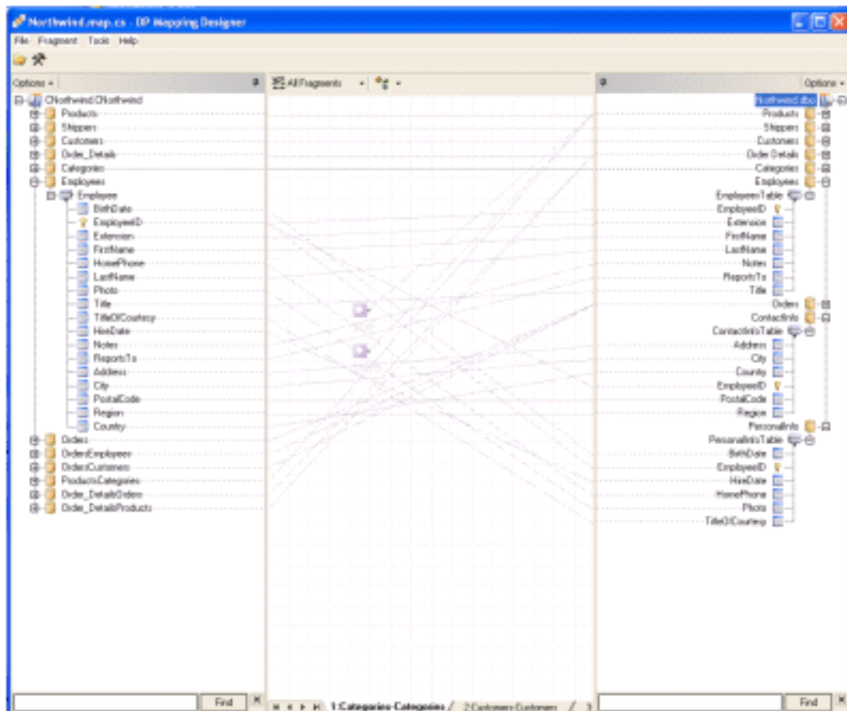

Figure 7. A conceptual to logical mapping tool (Click on the image for a larger picture)

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 great late bound scenarios since return values are self describing. Many applications, however, prefer an object programming layer. This can be facilitated through code generation driven from the EDM description. Within Visual Studio, one can leverage an ADO.NET aware build task to generate the corresponding code. For increased flexibility and data independence between the object and conceptual level, there may be another level of mapping between classes and the conceptual model. This would enable applications built against these classes to be reused against versions of the conceptual model provided a legal map can be defined. For the mapping between classes and the conceptual model the constraints are minimal and are along the lines of preserving identity and fidelity of round trips. The following diagram illustrates the classes from same solution.

Figure 8. Automatically generated classes representing a programming level (Click on the image for a larger picture)

The mapping between classes and the conceptual model is a straightforward member-wise mapping, a snippet of it is shown below:

```
<ObjectMapping cdm:CLRTypeName="CNorthwind.Product"
      cdm:CdmTypeName="CNorthwind.Product">

  <MemberMapping cdm:CLRMember="ProductID" cdm:CLRAlias="ProductID"
      cdm:CdmMember="ProductID" />
  <MemberMapping cdm:CLRMember="ProductName" cdm:CLRAlias="ProductName"
      cdm:CdmMember="ProductName" />
  <MemberMapping cdm:CLRMember="QuantityPerUnit"
      cdm:CLRAlias="QuantityPerUnit"
      cdm:CdmMember="QuantityPerUnit" />
  <MemberMapping cdm:CLRMember="ReorderLevel" cdm:CLRAlias="ReorderLevel"
      cdm:CdmMember="ReorderLevel" />
  <MemberMapping cdm:CLRMember="UnitPrice" cdm:CLRAlias="UnitPrice"
      cdm:CdmMember="UnitPrice" />
  <MemberMapping cdm:CLRMember="UnitsInStock" cdm:CLRAlias="UnitsInStock"
      cdm:CdmMember="UnitsInStock" />

  <MemberMapping cdm:CLRMember="UnitsOnOrder" cdm:CLRAlias="UnitsOnOrder"
      cdm:CdmMember="UnitsOnOrder" />
</ObjectMapping>
<ObjectMapping cdm:CLRTypeName="CNorthwind.Shipper"
      cdm:CdmTypeName="CNorthwind.Shipper">
  <MemberMapping cdm:CLRMember="CompanyName" cdm:CLRAlias="CompanyName"
      cdm:CdmMember="CompanyName" />
  <MemberMapping cdm:CLRMember="Phone" cdm:CLRAlias="Phone"
      cdm:CdmMember="Phone" />
  <MemberMapping cdm:CLRMember="ShipperID" cdm:CLRAlias="ShipperID"
      cdm:CdmMember="ShipperID" />
</ObjectMapping>
```

Using Objects

One can interact with objects and perform regular Create Read Update Delete (CRUD) operations on the objects, for example here we query for all Employees hired after a given point in time:

```
public void DoObjectQueries(DateTime date)
    {
         //--- get a connection
           MapConnection conn =
               new MapConnectionFactory().GetMapConnection();
           Northwind nw = new Northwind(conn,conn.MetadataWorkspace);
```

```
var employees = from e in nw.Employees where e.HireDate > date

select e;
        foreach( Employee e in employees )
         {
               Console.WriteLine(e.FirstName);
         }

   }
```

Similarly we could give each person a promotion and then persist the change to the store:

```
          MapConnection conn =
              new MapConnectionFactory().GetMapConnection();
          Northwind nw = new Northwind(conn,conn.MetadataWorkspace);

var employees = from e in nw.Employees where e.HireDate > date

select e;
        foreach( Employee e in employees )
         {
               Console.WriteLine(e.FirstName);
              e.Title = "manager";
        }
        nw.SaveChanges();
```

In this case the interesting part of this interaction is the SaveChanges() call on the northwind instance. The northwind instance is a specialization of ObjectContext and provides a top-level context for state management and the like. In this particular case instances that are retrieved from the store are cached by default and when we invoke save changes the changes for the type (which is cached here) are pushed to the store through the update pipeline.

Using Values

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

```
public void DoObjectQueries(DateTime date)
{
    //--- get a connection
    using (MapConnection conn =
          new MapConnectionFactory().GetMapConnection())
    {
        conn.Open();
        MapCommand command = conn.CreateCommand();
        command.CommandText = @"
                Select value e from Employees
                 as e where e.HireDate>@HireDate";
        command.Parameters.Add(new MapParameter("HireDate",date));
        DbDataReader reader = command.ExecuteReader();
        while(reader.Read())
        {
            //--- do something interesting here
        }
    }

}
```

Entity Framework Architecture

This section briefly describes the architecture of the Entity Framework being built as part of ADO.NET. A detailed description of the architecture can be found in [**ARCH**]. The main functional components of the ADO.NET Entity Framework (see **Figure** 1) are:

Data source-specific providers. The Entity Framework builds on the ADO.NET data provider model. SqlClient is the storage-specific provider for the Microsoft SQL Server database products including SQL Server 2000 and SQL Server 2005. WCFClient is a future provider to enable access to data from Web Services. In terms of interfaces, the ADO.NET provider model contains Connection, Command, and DataReader objects. A new SqlGen service in the Bridge (mentioned below) generates store-specific SQL text from canonical commands.

Map provider. The Entity Framework includes a new data provider, the Map provider. This provider houses the services implementing the mapping transformation from conceptual to logical constructs. The Map provider represents a value-based, client-side view runtime where data is accessed in terms of EDM entities and relationships and queried using the eSQL language. The Map provider includes the following services:

EDM/eSQL. The Map provider processes and exposes data in terms of the EDM values. Queries and updates are formulated using an entity-based SQL language called eSQL.

Mapping. View mapping, one of the key services of the Map provider, is the subsystem that implements bidirectional 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.

Query and update pipelines. Queries and update requests are specified to the Map provider via its Command object

either as eSQL text or as canonical command trees.

Store-specific bridge. The bridge component is a service that supports the query execution capabilities of the query pipeline. The bridge takes a command tree as input and produces an equivalent command tree (or command trees) in terms of query capabilities supported by the underlying store as output.

Metadata services. The metadata service supports all metadata discovery activities of the components running inside the Map 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 Map provider integrates with the transactional capabilities of the underlying stores.

API. The API of the Map provider follows the ADO.NET provider model based on Connection, Command, and DataReader objects. Like other store-specific providers, the Map provider accepts commands in the form of eSQL text or canonical trees. The results of commands are returned as DataReader objects

Occasionally Connected Components. The Entity Framework enhances the well established disconnected programming model introduced by 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 Data Platform will include 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. The embedded database will include a simple query processor and non-authoritative persistence capabilities to enable large middle-tier data caches.

Design and Metadata Tools. The data platform integrates with domain designers to enable model-driven application development. The tools include EDM, mapping, and query modelers. Note that mapping tools driven from the EDM has created a series of tool generated maps to map from a user defined entity such as an Asset or Customer into a series of logical schema and presentation formats associated with various platform services. These could represent a Sharepoint List definition, a binding for a business object, a merge replication logical record, or an XML representation for use in a web service or workflow. In any case, the entity need only be defined a single time and tooling associated with the data platform can generate the appropriate logical schema definitions.

Programming Layers. ADO.NET allows multiple programming layers to be plugged onto the value-based entity data services layer exposed by the Map 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. In the future, we expect other programming surfaces to be built on top of the entity services.

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

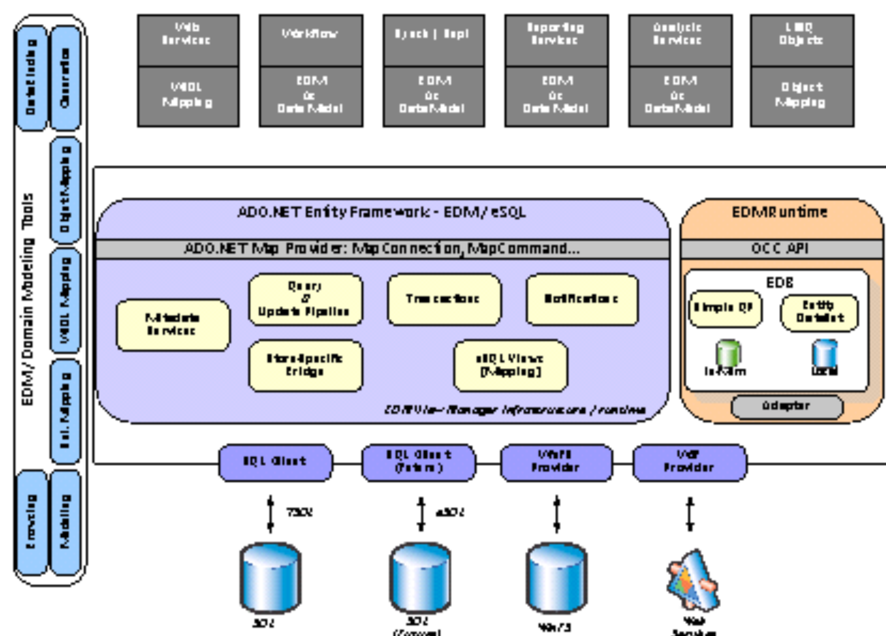

Figure 9. Entity Framework Architecture (Click on the image for a larger picture)

References

[CHEN76] Chen, Peter Pin-Shan. *The Entity-Relationship Model—toward a unified view of data*, ACM Transactions on Database Systems, Vol. 1, Issue 1, March 1976, pp. 9-36.

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

[EDM] Entity Data Model. ADO.NET Technical Preview, June 2006

[ARCH] ADO.NET Entity Framework Architecture. ADO.NET Technical Preview, June 2006.

[ADO.NET] ADO.NET Tech Preview Overview, June 2006.

© Microsoft Corporation. All rights reserved.

© 2011 Microsoft. All rights reserved. Terms of Use | Trademarks | Privacy Statement