Introducing LINQ to Relational Data

Microsoft Data Platform Development Technical Article

Writers: Elisa Flasko, Microsoft Corporation

Published: January 2008

Level: 100/200

Applies To:

LINQ to SQL

LINQ to Entities

ADO.NET Entity Framework

You can also download a Microsoft Word version of this article.

Summary: This article introduces two implementations of LINQ, LINQ to SQL and LINQ to Entities, and the key scenarios for which each was designed. (19 printed pages)

Table of Contents

Introducing LINQ to Relational Data

With the combined launch of Visual Studio 2008, SQL Server 2008, and Windows Server 2008, Microsoft is introducing five implementations of .NET Language Integrated Query (LINQ).

Of these five implementations, two specifically target access to relational databases: LINQ to SQL and LINQ to Entities. This white paper introduces these two technologies and the scenarios in which each can best be used.

Microsoft Language Integrated Query (LINQ) offers developers a new way to query data using strongly-typed queries and strongly-typed results, common across a number of disparate data types including relational databases, .NET objects, and XML. By using strongly-typed queries and results, LINQ improves developer productivity with the benefits of IntelliSense and compile-time error checking.

LINQ to SQL, released with the Visual Studio 2008, is designed to provide strongly-typed LINQ access for rapidly developed applications across the Microsoft SQL Server family of databases.

LINQ to Entities, to be released in an update to Visual Studio 2008 in the first half of 2008, is designed to provide strongly-typed LINQ access for applications requiring a more flexible Object Relational mapping, across Microsoft SQL Server and third-party databases.

What Is LINQ to SQL?

LINQ to SQL is an object-relational mapping (ORM) framework that allows the direct 1-1 mapping of a Microsoft SQL Server database to .NET classes, and query of the resulting objects using LINQ. More specifically, LINQ to SQL has been developed to target a rapid development scenario against Microsoft SQL Server where the database closely resembles the application object model and the primary concern is increased developer productivity.

Figures 1 & 2 combined with the code snippet below demonstrate a simple LINQ to SQL scenario. Figure 1 shows the LINQ to SQL mapping, and Figure 2 shows the associated database diagram, using the Northwind database.

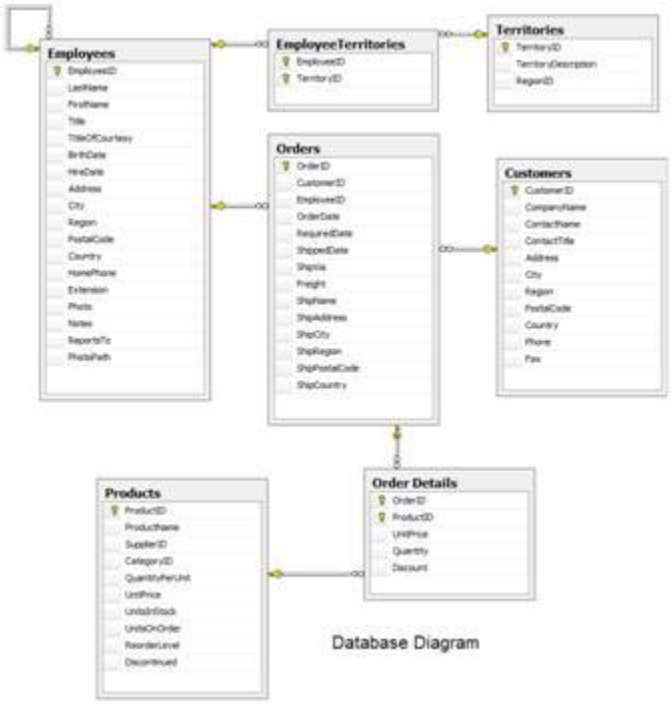

Figure 1. Database diagram for a subset of the Northwind database.

Figure 2. LINQ to SQL mapping diagram for a simple scenario using a subset of the Northwind database and the associated database diagram. Notice the use of an intermediary table to map the many-to-many relationship between Employees and Territories.

This code snippet shows a simple LINQ query against the Northwind database to retrieve all customers whose address is in London.

```
NorthwindDataContext db = new NorthwindDataContext();

var customers = from c in db.Customers

                where c.City == "London"

                select c;
```

Listing 1. Simple LINQ to SQL query

LINQ to SQL has been architected with simplicity and developer productivity in mind. APIs have been designed to "just work" for common application scenarios. Examples of this design include the ability to replace unfriendly database naming conventions with friendly names, to map SQL schema objects directly to classes in the application, to implicitly load data that has been requested but has not previously been loaded into memory, and the use of common naming conventions and partial methods to provide custom business or update logic.

The ability to replace unfriendly database naming conventions with more developer friendly names provides benefit that is pretty self explanatory, making it easier to understand the structure and meaning of data as the application is developed. These changes can be done using the LINQ to SQL designer in Visual Studio 2008, by double clicking on the entity object name, as seen in Figure 3 below.

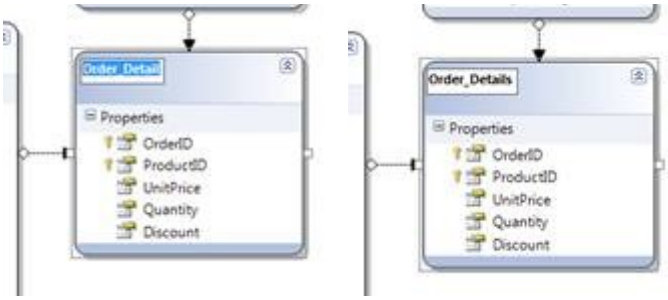

Figure 3. Changing the name of an entity object in LINQ to SQL.

As mentioned above, LINQ to SQL targets scenarios where the database closely resembles the application object model that is desired. Given this target scenario, the mapping between the SQL schema and classes in the application is a direct 1-1 mapping, meaning that a table or view being accessed from the database maps to a single class in the application, and a column being accessed maps to a property on the associated class.

By default, LINQ to SQL enables deferred loading. This means that if, for example, a query retrieves Customer data, it does not automatically pull the associated Order information into memory. When the data is requested however, LINQ to SQL uses its implicit loading capability to retrieve the data that has been requested but has not previously been loaded into memory. In Listing 1 we queried the database for all customers with an address based in London. While iterating through the resulting list of customers we would like to access the list of Orders that each customer has placed. Looking back to the original query in Listing 1, we notice that we did not retrieve any information about Orders; we simply retrieved information included in the Customer object. In Listing 2, we continue from the previous query to iterate through the customers that were returned and print out the total number Orders associated with each customer.

```
foreach (Customer c in customers)

{

     Console.WriteLine(c.CompanyName + " " + c.Orders.Count);

}
```

Listing 2. Implicit loading of Orders data that has been requested but not previously loaded into memory.

In the above example, a separate query is executed to retrieve the Orders for each Customer. If it is known in advance that we need to retrieve the orders for all customers, we can use LoadOptions to request that the associated Orders be retrieved along with the Customers, in a single request.

Beyond the simplicity of the query experience, LINQ to SQL offers additional features to improve the developer experience with regards to application logic. Partial methods, a new language feature of C# and VB in Visual Studio 2008, allow one part of a partial class to define and call methods which are invoked, if implemented in another part of the class. If the method has not been implemented; the entire method call is optimized away during compilation. By using common naming conventions in conjunction with these new partial methods and partial classes, introduced in Visual Studio 2005, LINQ to SQL allows application developers to provide custom business logic when using generated code. Using partial classes allows developers the flexibility to add methods, non-persistent members, etc., to the generated LINQ to SQL object classes. These partial methods can add logic for insert, update, and delete by simply implementing the associated partial method. Similarly, developers can use the same concepts to implement partial methods that hook up eventing in the most common scenarios, for example OnValidate, OnStatusChanging or OnStatusChanged. Example 3 shows the use of partial methods to implement custom validation on the Customer Phone property as it is changed.

```
public partial class Customer

{
```

```
    partial void OnPhoneChanging(string value)
    {
        Regex phoneNum = new Regex(@"^[2-9]\d{2}-\d{3}-\d{4}$");
        if (phoneNum.IsMatch(value) == false)
            throw new Exception("Please enter a valid Phone Number");
    }
}
```

Listing 3. Use of partial methods to implement custom validation logic.

Once developers can leverage the query capabilities of LINQ and implement business logic in partial methods, the argument for using persistent objects becomes quite compelling. To round the story out, LINQ to SQL allows people to model inheritance hierarchies in their classes and map them to the database. Inheritance, an important feature of object-oriented programming, does not translate directly into the relational database; therefore, the ability to map inheritance from the application into the database is very important. LINQ to SQL supports one of the most common database inheritance mappings, Table per Hierarchy (TPH), where multiple classes in a hierarchy are mapped to a single table, view, stored procedure, or table valued function using a discriminator column to determine the specific type of each row/instance. In Figure 4, a single Product table in the database, maps the inheritance of DiscontinuedProduct from Product. This two-class hierarchy is mapped directly to the existing Northwind database using the discriminator column "Discontinued".

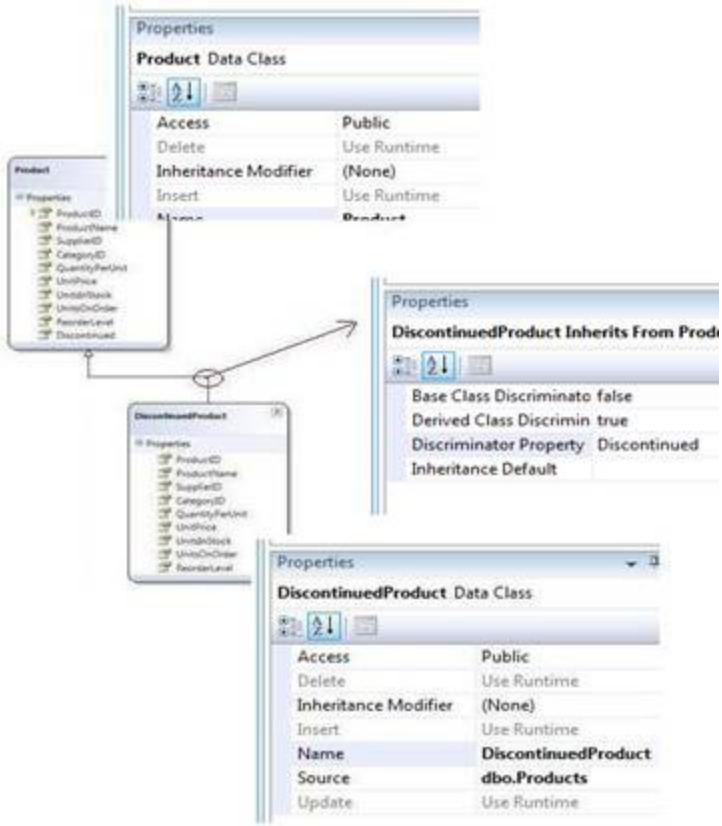

Figure 4. Inheritance in LINQ to SQL

Notice that the properties of the inheritance (right-click on the relationship arrow) specify the discriminatory property and the associated discriminator values, while the properties of DiscontinuedProduct specify the base class from which it inherits.

LINQ to SQL has been developed with a minimally intrusive object model, allowing developers to choose whether or not to make use of generated code. Rather, they can create their own classes, which do not need to be derived from any specific base class. This means that you can create classes that inherit from no base class or from a custom base class.

As with any application framework, developers must also have the ability to optimize the solution to best fit their scenario. LINQ to SQL offers a number of opportunities to optimize, including using load options to control database trips and compiled queries to amortize the overhead inherent in SQL generation. By default, LINQ to SQL enables Object Tracking, which controls the automatic change tracking and identity management of objects retrieved from the database. In some scenarios, specifically where you are accessing the data in a read-only manner, you may wish to disable Object Tracking as a performance optimization. The options for specialization of your application behavior is not local to just the API, the LINQ to SQL Designer also allows additional functionality for you to expose stored procedures and/or table valued functions as strongly typed methods on the generated DataContext, and map inserts, updates, and deletes to stored procedures if you choose not to use dynamic SQL.

Developers who are concerned about query performance can leverage the compiled queries capabilities which offer an opportunity to optimize query performance. In many applications you might have code that repeatedly executes the same query, possibly with different argument values. By default, LINQ to SQL parses the language expression each time to build the corresponding SQL statement, regardless of whether that expression has been seen previously. Compiled queries, like that seen in Listing 4, allow LINQ to SQL to avoid reparsing the expression and regenerating the SQL statement for each repeated query.

```
var customers = CompiledQuery.Compile(

                    (NorthwindDataContext context,

                     string filterCountry) =>

                        from c in context.Customers

                        where c.Orders.Count > 5

                        select c);



NorthwindDataContext db = new NorthwindDataContext();

           Console.WriteLine("Customers in the USA: ");

foreach (var row in customers(db, "USA"))

{

       Console.WriteLine(row.CompanyName);

}

Console.WriteLine();

Console.WriteLine("Customers in Spain: ");
```

```
foreach (var row in customers(db, "Spain"))

{

        Console.WriteLine(row.CompanyName);

}
```

Listing 4. An example of a simple compiled query, executed twice with varying parameters.

When Do I Use LINQ to SQL?

The primary scenario for using LINQ to SQL is when building applications with a rapid development cycle and a simple one-to-one object to relational mapping against the Microsoft SQL Server family of databases. In other words, when building an application whose object model is structured very similarly to the existing database structure, or when a database for the application does not yet exist and there is no predisposition against creating a database schema that mirrors the object model; you can use LINQ to SQL to map a subset of tables directly to classes, with the required columns from each table represented as properties on the corresponding class. Usually in these scenarios, the database has not and/or will not be heavily normalized.

I want to	LINQ to SQL is applicable
Use an ORM solution and my database is 1:1 with my object model	✓
Use an ORM solution with inheritance hierarchies that are stored in a single table	✓
Use my own plain CLR classes instead of using generated classes or deriving from a base class or implementing an interface	✓
Leverage LINQ as the way I write queries	✓
Use an ORM but I want something that is very performant and where I can optimize performance through stored procedures and compiled queries	✓

Table 1. LINQ to SQL Scenarios

What Is LINQ to Entities?

LINQ to Entities, like LINQ to SQL is a LINQ implementation providing access to relational data, but with some key differences.

LINQ to Entities is, specifically, a part of the ADO.NET Entity Framework which allows LINQ query capabilities. The Entity

Framework is the evolution of ADO.NET that allows developers to program in terms of the standard ADO.NET abstraction or in terms of persistent objects (ORM) and is built upon the standard ADO.NET Provider model which allows access to third party databases. The Entity Framework introduces a new set of services around the Entity Data Model (EDM) (a medium for defining domain models for an application). This set of services includes the following:

- Domain Modeling Capabilities and the ability to define conceptual, data store agnostic models
- Object Relational Mapping Capabilities (full CRUD and state management scenarios)
- Database independent Query Capabilities using LINQ or Entity SQL

More than a simple Object Relational Mapping (ORM) tool, the ADO.NET Entity Framework and LINQ to Entities allow developers to work against a conceptual model with a very flexible mapping and the ability to accommodate a high degree of divergence from the underlying data store. For further discussion of the Entity Framework and EDM, please see the Data Platform Development Center (http://msdn.microsoft.com/data).

Figure 5 below shows a simple LINQ to Entities EDM diagram, using the same subset of the Northwind database seen in Figure 1.

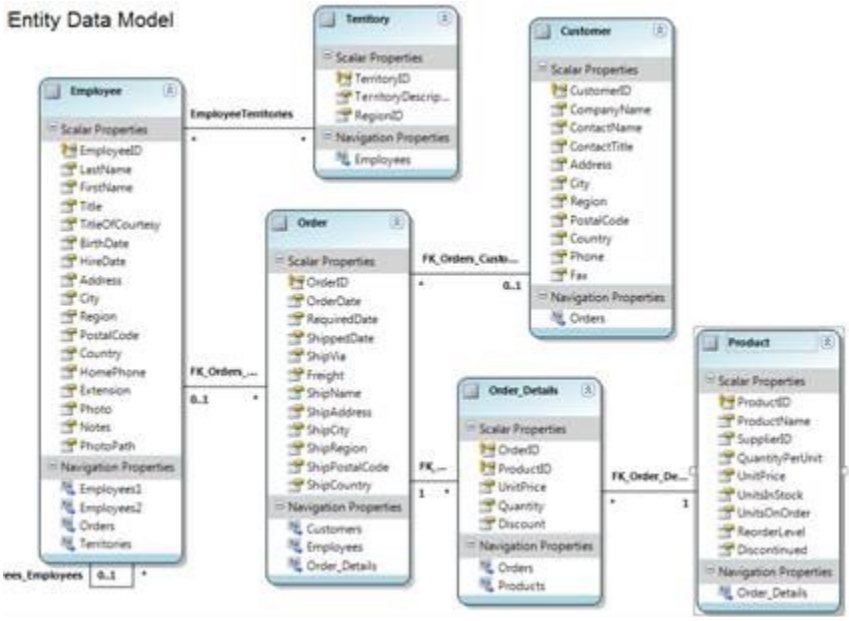

Figure 5. LINQ to Entities mapping diagram corresponding to the Northwind database subset in Figure 1. Notice the directly mapped many-to-many relationship between Employees and Territories without an intermediary table.

In this figure you can see that the object model is not mapped directly, one-to-one, to the database, rather we have bypassed the intermediary table used to represent the many-to-many relationship between Employees and Territories. Although you can map many-to-many relationships in both LINQ to SQL and LINQ to Entities, LINQ to Entities allows a direct mapping of many-to-many relationships with no intermediary class, while LINQ to SQL requires that an intermediary class map one-to-many to each of the classes that are party to the many-to-many relationship.

Listing 5 below shows the same simple LINQ query used in Listing 1 for LINQ to SQL, used against the Entity Data Model shown in Figure 5.

```
using (NorthwindEntities nw = new NorthwindEntities())

{
```

```
        var cusotmers = from c in nw.Customers
                        where c.City == "London"
                        select c;
}
```

Listing 5. Simple LINQ to Entities query

The similarities between the two LINQ implementations for this simple query highlight the benefit of LINQ creating a query language that remains consistent across data stores.

Microsoft designed the ADO.NET Entity Framework, and in turn LINQ to Entities, to enable flexible and more complex mappings, ideal in the scenario where it is not possible or ideal for the object model to match the database schema. The mapping flexibility available with the ADO.NET Entity Framework allows the database and application(s) to evolve separately and makes development against highly normalized databases simpler and easier to understand. When a change is made in the database schema, the application is insulated from the change by the Entity Framework, and there is no requirement to rewrite portions of the application, but rather the mapping files can simply be updated to accommodate the database change. In Figures 6 & 7, a change is made to the database splitting the Employees table to separate personal information that is only accessible to HR and information that is available in the Employee Address Book. If existing applications are not going to make use of these changes we simply need to update the mapping to account for this change.

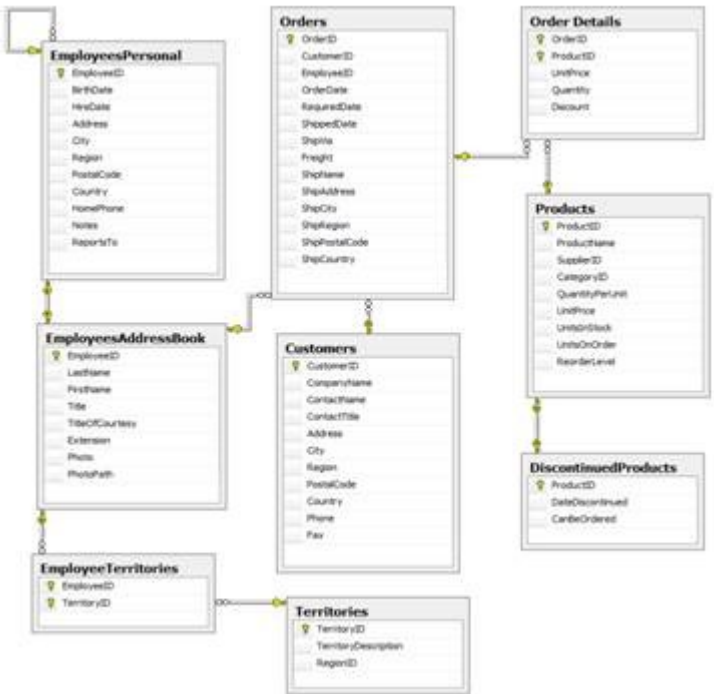

Figure 6. Database Model for the new modified Northwind database subset. In this modified database, we have split the Employees table into two tables one containing Employee information that is only available to HR and another containing information that is available to all employees in the Employee Address Book. We have also used a different inheritance mapping, Table per Subclass, which we will discuss later.

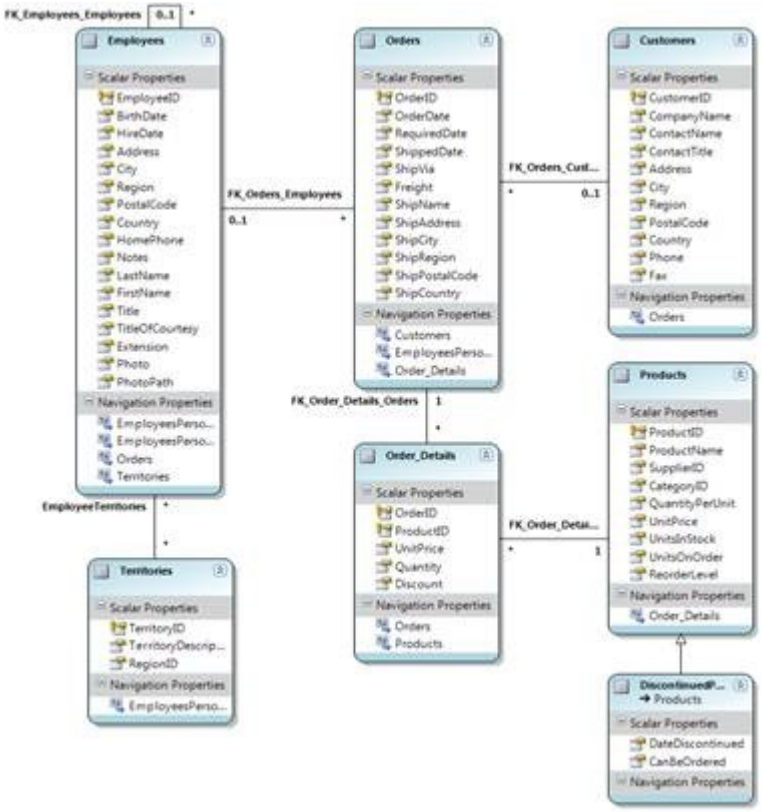

Figure 7. Entity Data Model Diagram for the modified Northwind database subset shown in Figure 6.

Listing 6 shows a query against a single entity, mapped to multiple tables in the database. The same query directly against the database would require the knowledge that Employee information is split between two tables and a join of those two tables in the query.

```
using (NorthwindModEntities nw = new NorthwindModEntities())            {

       var q = from e in nw.Employees

               select e;

       foreach (Employees emp in q)

       {

               Console.WriteLine(emp.FirstName + ", " + emp.LastName + ", Hire Date: " +
emp.HireDate.ToString());

       }

}
```

Listing 6. Querying a single Entity mapped to two tables in the database.

As discussed earlier in this article, LINQ to SQL allows one of the most common inheritance scenarios to be mapped, Table per Hierarchy. LINQ to Entities and the ADO.NET Entity Framework also allow the mapping of two other types of inheritance. In addition to Table per Hierarchy (as supported by LINQ to SQL), the Entity Framework supports:

- The mapping of Table per Concrete Type, a separate table for each class or type in the hierarchy.
- The mapping of Table per Subclass, a hybrid approach using a shared table for information about the base type and separate tables for information about the derived types seen in Figure 8.

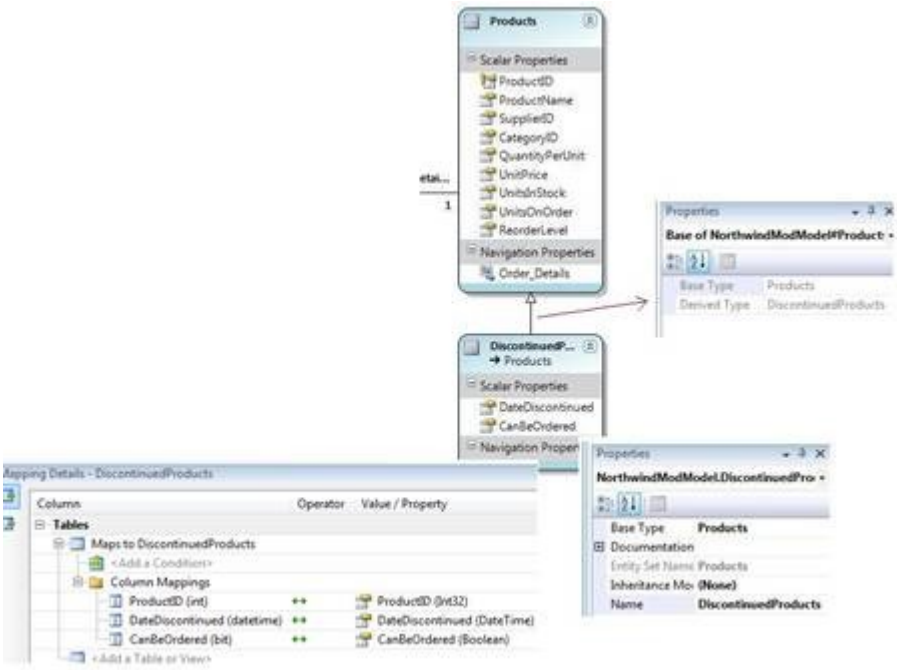

Figure 8. Mapping Table per Subclass inheritance with the Entity Data Model.

Similar to LINQ to SQL, LINQ to Entities uses partial classes and partial methods to allow update and business logic to be easily added to generated code. Using partial classes and partial methods allows developers the flexibility to add methods, non-persistent members, etc., to the generated Entity Framework object classes, and for the addition of custom logic for insert, update, and delete by simply implementing the associated partial method. Similarly, developers can use the same concepts to implement partial methods that hook up eventing in the most common scenarios, for example OnStatusChanging or OnStatusChanged. Listing 7 shows the use of partial methods to implement custom validation on the Customer Phone property as it is changed.

```
public partial class Customers

{

        partial void OnPhoneChanging(string value)

        {

             Regex phoneNum = new Regex(@"^[2-9]\d{2}-\d{3}-\d{4}$");

             if (phoneNum.IsMatch(value) == false)

                 throw new Exception("Please enter a valid Phone Number");

        }

}
```

Listing 7. Use of partial methods to implement custom validation logic.

Due to the explicit nature of LINQ to Entities, developers also have the ability to optimize the solution to best fit their scenario. In Listing 2 for LINQ to SQL, when we queried for Customer data, Order information was not automatically pulled into memory, but rather was only pulled into memory only when the Order information was accessed. In LINQ to Entities, the developer has full control over the number of database round trips by explicitly specifying when to load such information from the database. Navigating to associated information that has not yet been retrieved from the database will not cause an additional database trip, but rather will only return the information if it was explicitly requested with the first query or with a new query, as seen in Listing 8.

```
using (NorthwindModEntities nw = new NorthwindModEntities())

{

   var q = from cus in nw.Customers

           where cus.City == "London"

           select cus;

// Loop through customers and print out orders since January 1, 1998

   foreach (Customers customer in q)

   {

       Console.WriteLine(customer.CompanyName + ": " );

// Note line below to explicitly load all orders for each customer

       customer.Orders.Load();

       foreach (Orders order in customer.Orders.Where(o => o.OrderDate > new
DateTime(1998, 1, 1)))

       {

           Console.WriteLine("\t{0},{1}", order.OrderID, order.OrderDate);

       }

   }

   Console.ReadLine();

}
```

Listing 8. Explicit loading of information to control number of database roundtrips.

Like LINQ to SQL, LINQ to Entities enables Object Tracking by default. In some scenarios, specifically where you are accessing the data in a read-only manner, you may wish to disable Object Tracking as a performance optimization.

LINQ to Entities also provides the ability to expose stored procedures as strongly typed methods on the generated ObjectContext, and map inserts, updates, and deletes to stored procedures if you choose not to use dynamic SQL.

When do I use LINQ to Entities?

The primary scenario targeted by LINQ to Entities is a flexible and more complex mapping scenario, often seen in the enterprise, where the application is accessing data stored in Microsoft SQL Server or other-third party databases.

In other words, the database in these scenarios contains a physical data structure that could be significantly different from what you expect your object model to look like. Often in these scenarios, the database is not owned or controlled by the application developer(s), but rather owned by a DBA or other third party, possibly preventing application developers from making any changes to the database and requiring them to adapt quickly to database changes that they may not have been aware of.

I want to	**LINQ to Entities is applicable**
Write applications that can target different database engines in addition to Microsoft SQL Server	✓
Define domain models for my application and use these as the basis for my persistence layer.	✓
Use an ORM solution where my classes may be 1:1 with the database or may have a very different structure from the database schema	✓
Use an ORM solution with inheritance hierarchies that may have alternative storage schemes (single table for the hierarchy, single table for each class, single table for all data related to specific type)	✓
Leverage LINQ as the way I write queries and have the query work in a database vendor agnostic manner.	✓
Use an ORM but I want something that is very performant and where I can optimize performance through stored procedures and compiled queries	✓

Table 2. LINQ to Entities Scenarios