

Zeeshan Hirani

zeeshanjhirani@gmail.com

<http://weblogs.asp.net/zeeshanhirani>

If you like or dislike my work or have suggestions, send me an email at
zeeshanjhirani@gmail.com. I would love to hear your feedback!

1.	<i>Introduction to Entity Framework</i>	4
1.1	Generating Entity Data Model from the designer.....	4
1.2	Loading csdl,msl,ssdl schema files	21
1.3	Implementing IPOCO with Entity Framework.....	26
2.	<i>Modeling Entities</i>	36
2.1	Self Referencing Table	36
2.1.1	Self Referencing Table with Many to Many Association.....	36
2.1.2	Self Referencing entity with Table per Hierarchy	41
2.1.3	Using Common CTE with Self Referencing Entity.....	47
2.1.4	Many To Many association on Self referencing entity.....	52
2.2	Many to Many Mapping	55
2.2.1	Many To Many Mapping Walkthrough	56
2.2.2	Retrieving Link table for Many to Many Relation	61
2.2.3	Implementing Many to Many relationship as two 1 to Many relationship	62
2.2.4	Modeling two 1 to many relationship as Many to Many relationship.....	70
2.2.5	Mapping Many to Many table as 2 Many to Many Associations	76
2.3	Entity Splitting.....	86
2.3.1	Entity Splitting with three tables.....	86
3.	<i>Eager and Lazy Loading entities and Navigation properties</i>	91
3.1	Using Include to load Child Entities in Entity Framework	91
3.1.1	Loading EntityRef and EntityCollection Using Include	92
3.1.2	Using Include with Query Path to load related entities.....	94
3.1.3	Eagerly loading navigation properties on derived Types	97
3.1.4	Using Include with self referencing entity	100

3.1.5	Using Include with Many to Many association	104
3.1.6	Using Include at entity client layer.....	107
3.1.7	Common Pitfalls with Include operator	108
3.2	Using Load Operator to Lazy Load Collection and entity reference	113
3.3	CreateSourceQuery	120
3.3.1	CreateSourceQuery to filter associations	120
3.3.2	CreateSourceQuery to Execute Aggregate operation on Child collections.....	122
3.3.3	CreateSourceQuery to retrieve specific derived type from entity collection	123
3.4	Relationship Span	125
3.4.1	Taking Advantage of Relationship Span	125
3.4.2	Preventing Relationship span by using MergeOption.NoTracking.....	131
4.	Views	134
4.1	QueryView	134
4.1.1	Using QueryView To exclude columns and add computed columns	135
4.1.2	Using QueryView to filter collection	139
4.1.3	QueryView to map Many to Many Relationship with PlayLoad	143
4.2	DefiningQuery	155
4.2.1	Operators supported on QueryView.....	157
4.2.2	Mapping Foreign Key column to Multiple Associations Using DefiningQuery.....	158
4.2.3	Creating Dummy Defining Query to map stored procedure results	171
4.2.4	Creating Read-only Calculated Properties using Defining Query.....	175
4.2.5	Using DefiningQuery to map multiple associations to foreign key	179
5.	Inheritance.....	185
	Basics of Inheritance	185
5.1.1	Table per Type Walkthrough.....	187
5.1.2	Table per Hierarchy (Walkthrough)	193
5.1.3	Extending Table per Type with Table per Hierarchy	199
5.1.4	Extending Table per Hierarchy with Table per Type	211
5.1.5	Creating additional hierarchy for TPT using QueryView	220
5.1.6	Optimizing QueryView for Inheritance	232
5.1.7	Overriding Conditions for nested inheritance.....	235
5.1.8	Applying Conditions on Base Entity	242
5.1.9	Using Abstract entity with no table Mapping in TPH	245
5.1.10	Applying IsNull condition to Table per Hierarchy	250
5.1.11	Creating Many To 1 Association on Derived Entity.....	255
5.1.12	Table per Concrete Type.....	263
5.1.13	Mapping Column Used as a Discriminator.....	268
5.1.14	Mapping Table per Type to Foreign Key column	271
5.1.15	Using QueryView with TPH to create additional inheritance layer	279
5.1.16	Sharing Audit Fields across entities using TPC	288
5.1.17	Creating Association between Two Table Per Type entities.....	293
5.1.18	Creating Associations on Derived Entities using Table per Hierarchy	298

5.1.19	Table per Hierarchy and Table per Type Hybrid	305
5.1.20	Using multiple conditions for Table per Hierarchy	310
5.2	Linq To Sql	313
5.2.1	Table per type inheritance using Linq to Sql.....	313
5.2.2	Table per Hierarchy With Enum Using Linq To Sql.....	325
6.	Working with Objects.....	332
6.1	Using auto-generated Guids as entity key.....	332
6.2	Reading xml data type columns using EF.....	337
6.3	How does StoreGeneratedPattern work	341
6.4	Exposing EntityCollection and EntityReference properties on an entity	344
6.5	Monitoring collection changes (Add and Remove)	352
6.6	When does Association changed Event get fired.....	360
6.7	Complex Types	368
6.8	Accessing derived types from ObjectContext	376
7.	Improving Entity framework performance	379
7.1	Delay Loading Expensive Fields on a Table.....	379
7.1	GetObjectByKey vs First Operator	383
7.2	Retrieving read-only entities using MergeOption.NoTracking	387
7.3	Compiled Queries	396
7.4	Detaching entities returned from stored procedure.....	403
7.5	Improving loading time by generating store views.....	405
8.	Inserting, Updating and Deleting entities and associations	410
8.1	Assigning foreign key value without loading entity reference	410
9.	Querying with Linq to entities	414
9.1	How to do in Clause Query	414
9.2	Returning subset of collection using Paging	418
10.	Concurrency and Transactions.....	424
10.1.1	Concurrency with Table per Type	424
11.	Consuming Stored Procedures.....	428
11.1.1	Stored Procedure Returning entities	428
11.1.2	Stored Procedure Returning Scalar Types	438
11.1.3	Stored Procedure Returning Anonymous Type	441
11.1.4	Stored Procedure with Command Text Option	445

11.1.5	Stored Procedure with output parameters	450
11.1.6	Stored Procedure Returning Inheritance Hierarchy	454
12.	Mapping Crud Operations to Stored Procedure	456
12.1.1	Using EDM designer to Map Crud Operations to Stored Procedures.....	457
12.1.2	Mapping Associations to Stored Procedure.....	464
12.1.3	Deleting and Inserting Many to Many Relationship using Stored Procedures	478
12.1.4	Mapping Complex Type using Stored Procedure.....	485
12.1.5	Mapping Crud Operations To Table Per Hiearachy.....	491
12.1.6	Managing concurrency using stored procedures	501
12.2	Exploring Entity Framework Extensions	507

1. Introduction to Entity Framework

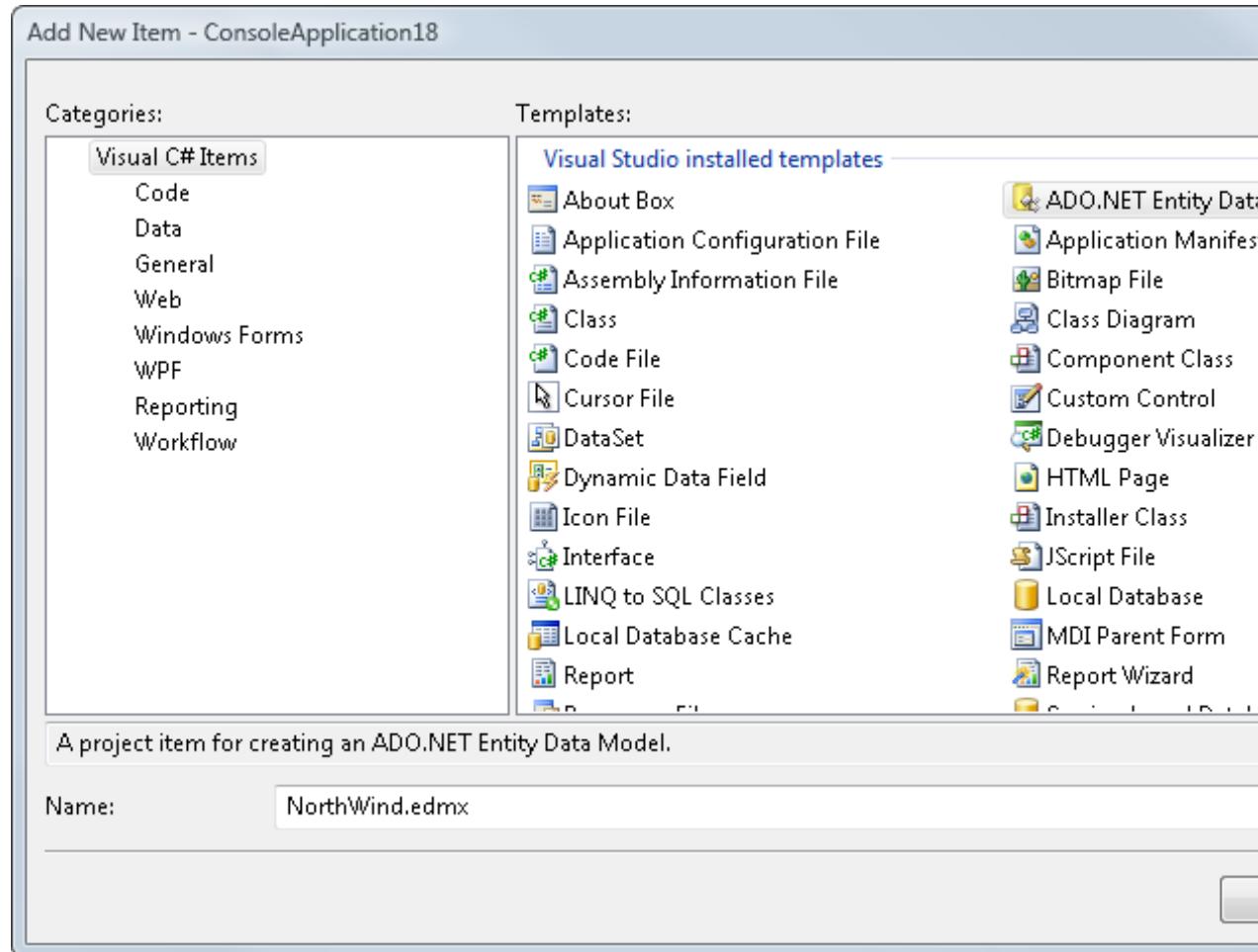
1.1 Generating Entity Data Model from the designer

Problem: You have a database and want to generate the entity data model using the database. You also would like to explore various options in the designer to customize entity classes. You also like discover how to update the model when a change is made in the database.

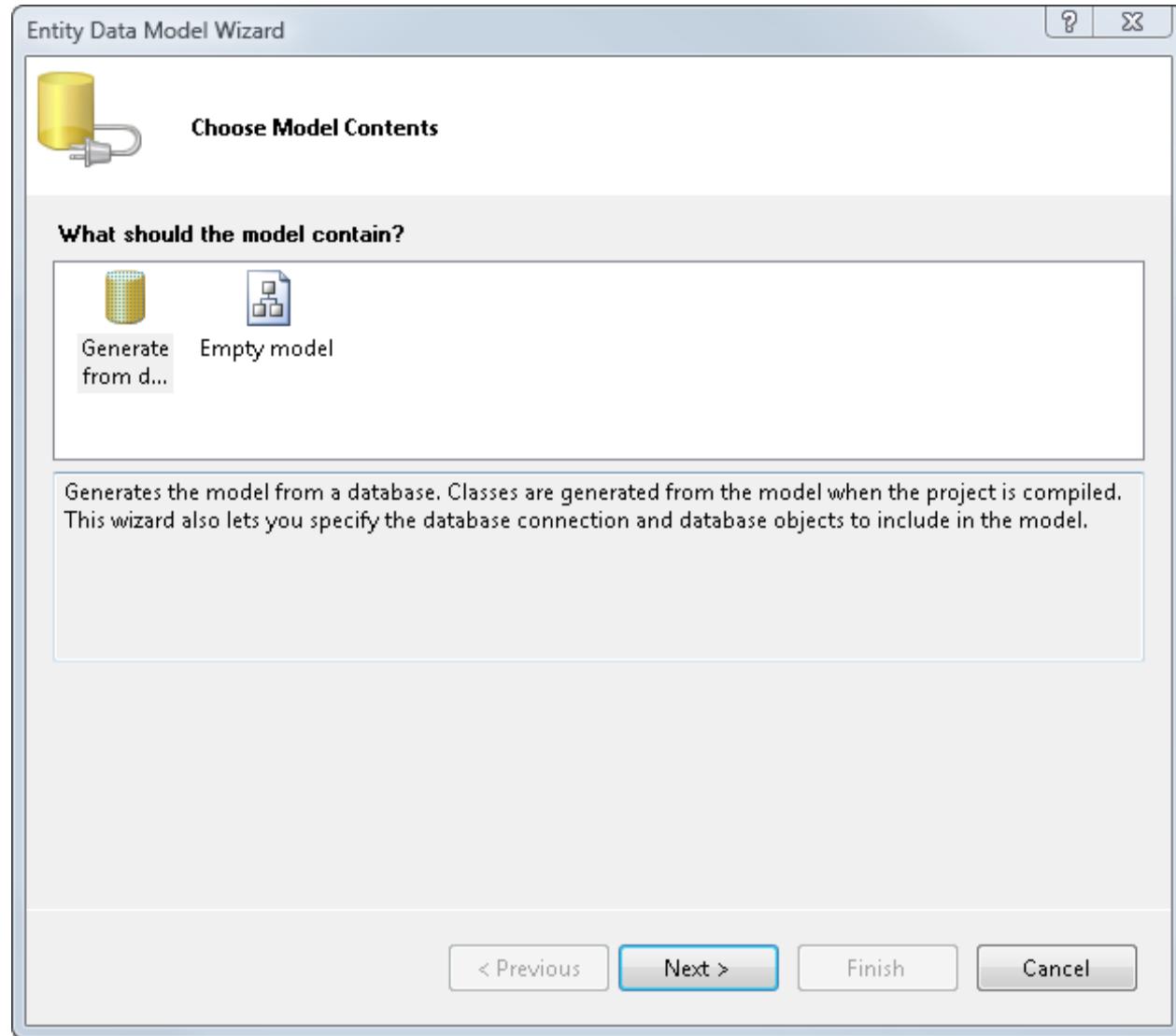
Solution: Entity framework provides an application view of the data stored in the database. Using entity framework, you can work against a conceptual model which represents your domain entities and how your business operates. The conceptual model can be mapped to 1 or more table using a mapping file. You can create a conceptual model in different ways. You can start with a conceptual model and later map the model to the database. If you already have a database that you want to get started with, you can use the database to build you conceptual model. In this walk through, I will discuss various features of the designer and how you can generate your conceptual model using database.

Discussion:

1. In your project add a new item of type Entity Data Model. I will call the model NorthWind.edmx since we are going to generate the model for NorthWind database.

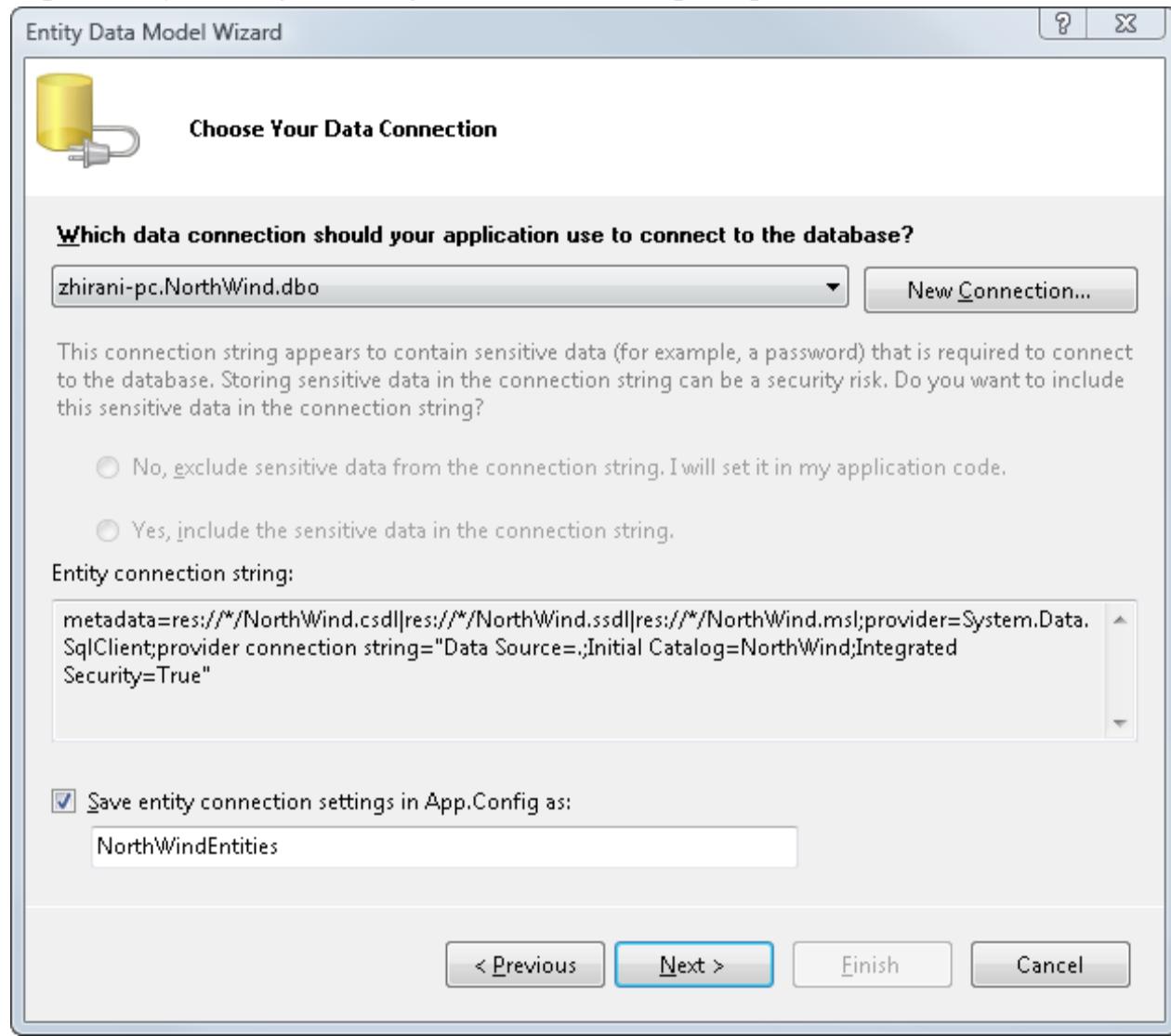


2. Select generate from database.

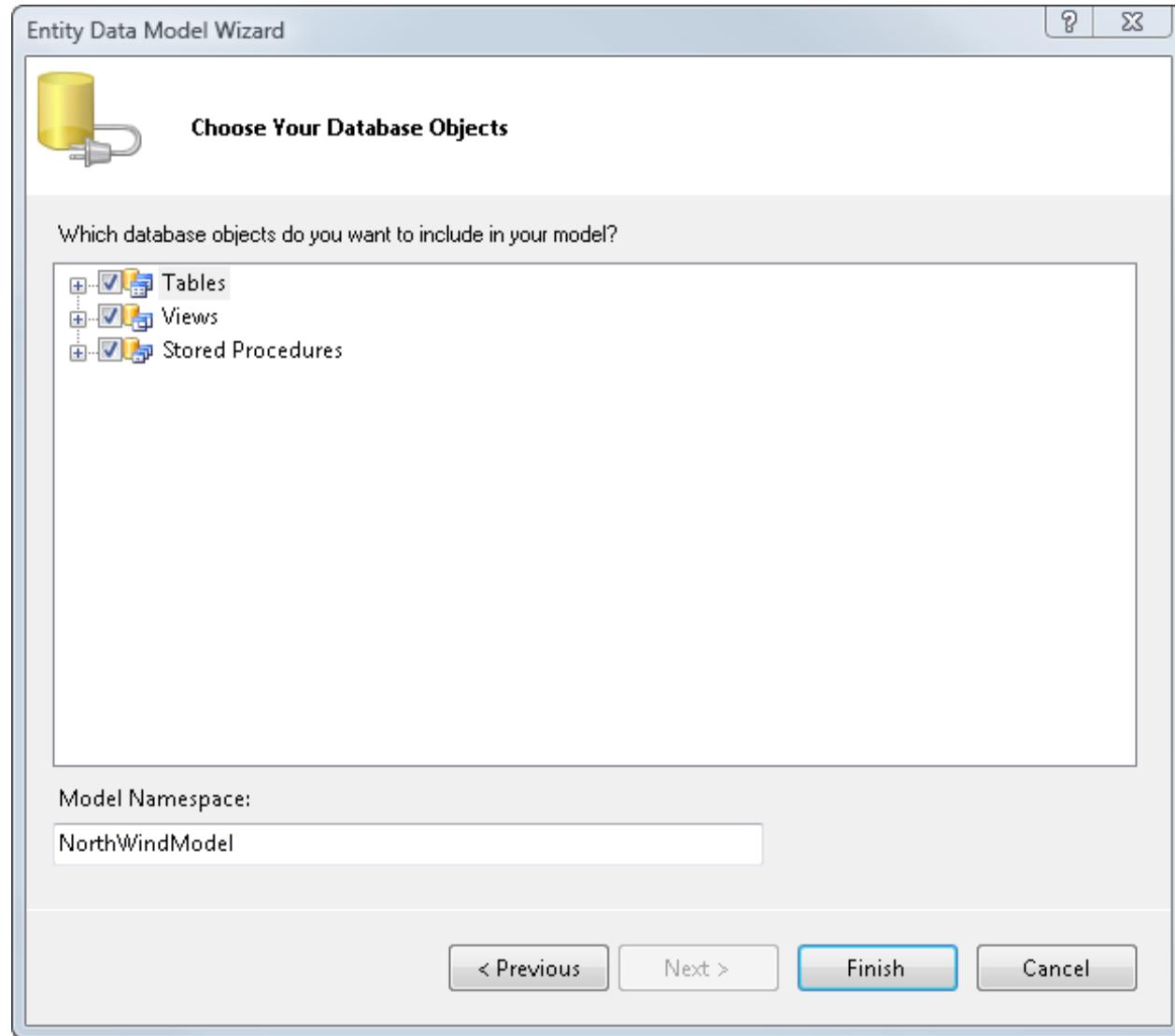


3. You can either create a new connection or select from an existing connection. After selecting the connection string you get a preview of how the connection string looks like. The connection string consists of 3 parts. First part represents the metadata file which represents the conceptual model, the data model and the mapping file which translates the conceptual model to the store model. The second part of the connection string represents the provider to use to connect to the database. In this case we are using SqlClient provider to connect to NorthWind database on sql server. The final part represents the connection string to use to connect to the database. In the connection string, you will notice that MultipleActiveResultSets is set to true. This

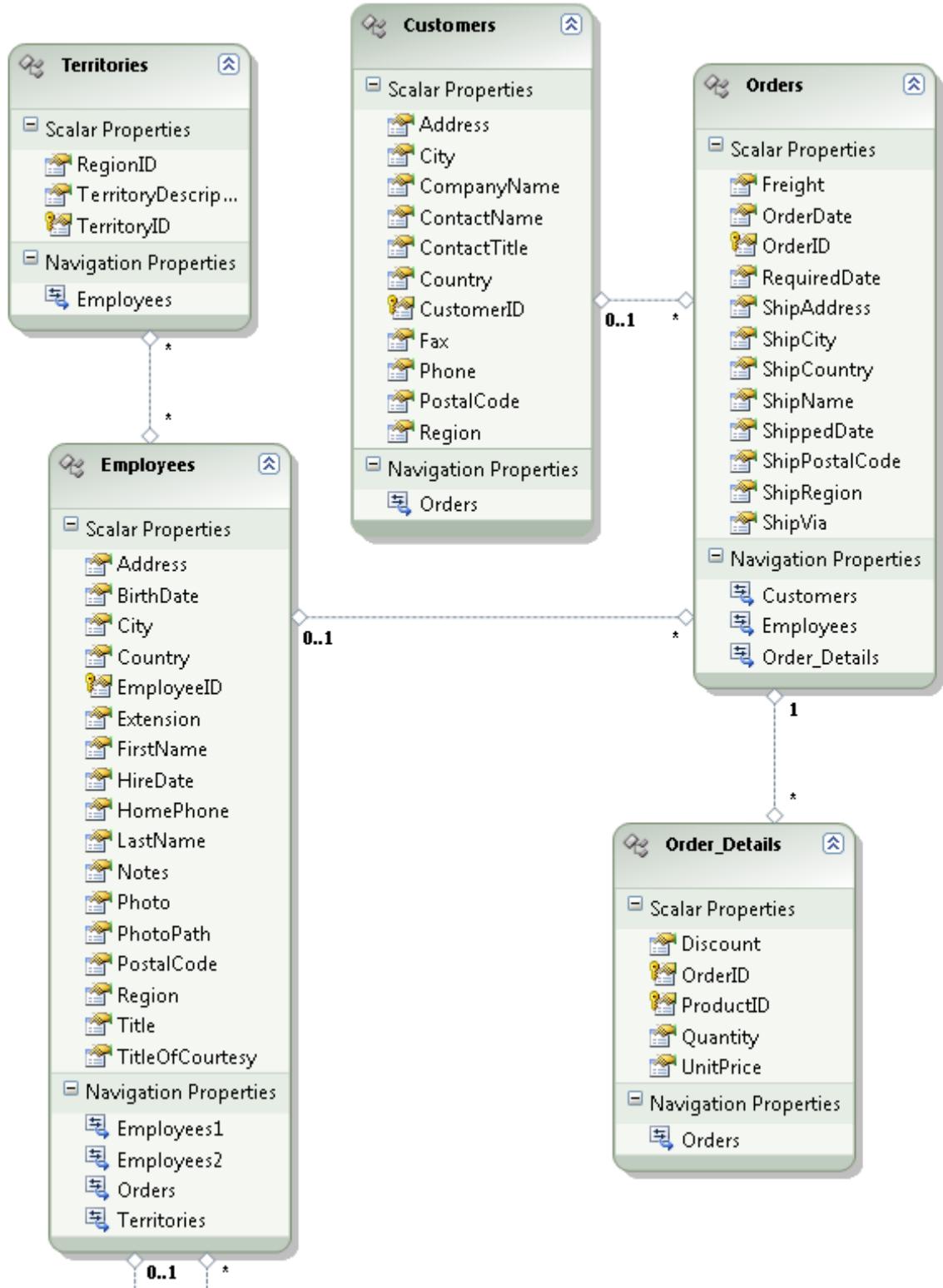
option is required for entity framework to enable multiple data readers to be read simultaneously from a single connection which is not required if you are generating the model for linq to sql classes.



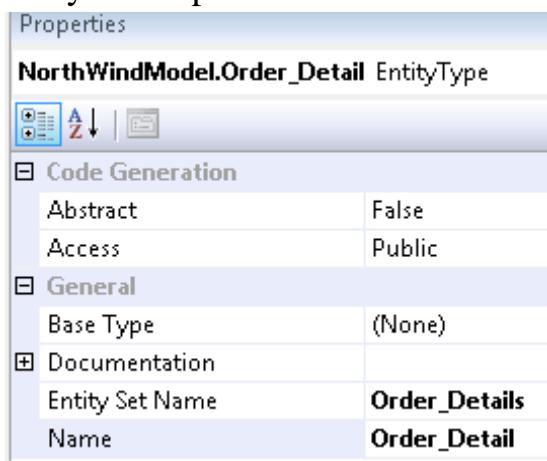
4. Choose the tables, views and stored procedures that you would like to represent and use in your conceptual model. Notice that among our list of database objects, there is no option to bring functions either scalar or table valued function. Version 1 release of the entity framework does not support functions. I will select tables, views and stored procedures. In this part of the wizard, I also get to choose the namespace where my object context will be created.



After clicking Finish, entity framework creates NorthWind.edmx file which contains entity data model. Additionally it also creates an app.config which contains the connection string to connect to the data model and the database. Usually you will create the entity data model in a business layer and the connection string will be created in app.config file inside of business layer which will not be of any use if you will add the business layer to a console, asp.net or windows application. You will have to copy the connection string from app.config to the app.config file of either the asp.net windows form or console application. Example below shows a small section of the generated entity data model.

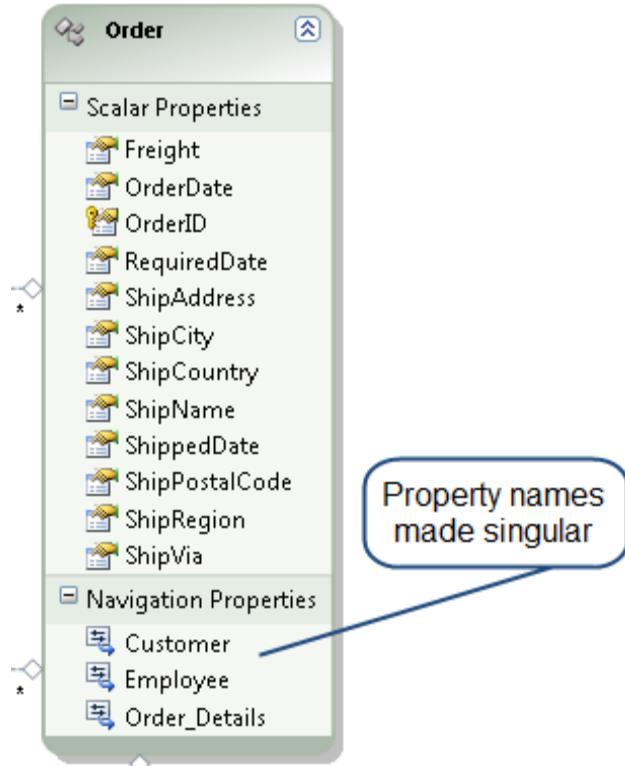


Entity data model generates different relations based on the schema defined in the database. Orders have 0 to 1 relationship with customer. The reason the relation is marked as 0 to 1 is because in the Orders table customer id column is defined as allow null so an order can optionally have 0 to 1 relationship to customer. In the case of OrderDetails, OrderId is a required field, so the generated relation is 1 order can have many OrderDetails and an OrderDetail must belong to an Order. When you create linq to sql model, the designer automatically fixes the names of entities from plural to singular. However EDM designer leaves the names as they are defined in the database. You have few options if you want to change the name of an entity. Clicking on the name of the entity in the designer twice will let you change the name. You can also right click an entity in the designer and access its properties which will also let you change the name of the entity. Example below shows the change in action.



An entity in the designer is separated in two parts. First section of an entity consists of scalar values and second section of the entity contains the navigation properties which allows an entity to access either an entity reference or entity collection. A navigation property can be 1 to 1 relation, 1 to many or many to many relationships. If you do not like the name of entity generated for navigation properties you can always select the navigation property and change the name in the properties window. For instance an order belongs to a customer and may optionally be entered by an employee. When generating the model the

navigation property is plural for both Customers and Employees for an Order. We can change the property to simply Customer and Employee to indicate it is a single customer and a single Employee as shown below.



Not only can you change the name of the navigation properties, but you can also change association of the relationship defined between two entities. For example, on the EDM model, customer and orders are related to each other with a line indicating 0 to many relationships. We can alter the relationship by selecting the relationship line and accessing its properties window. Below is the screen shot that shows how the properties window for the relationship between customer and order.

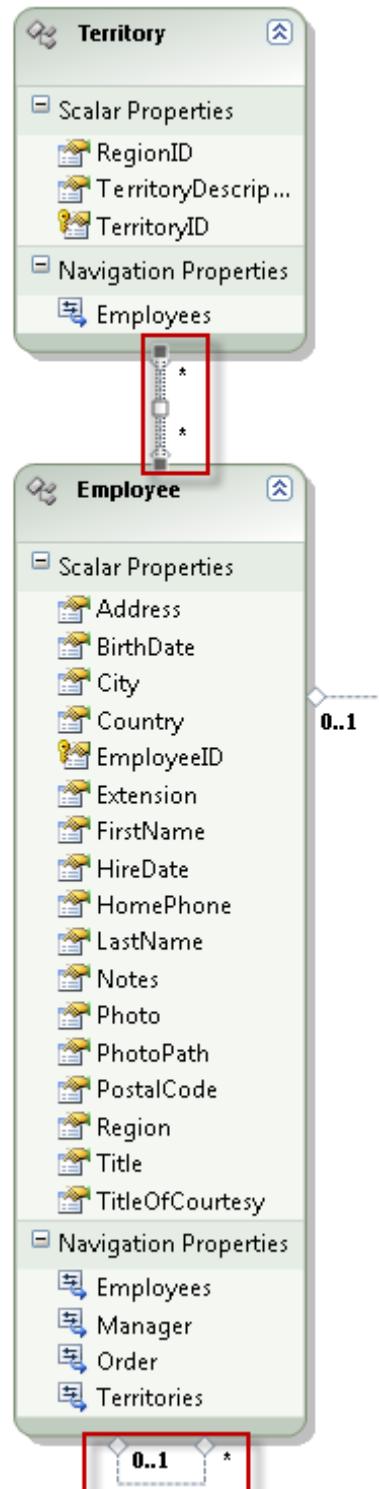
Properties	
NorthWindModel.FK_Orders_Customers Association	
Ends	
End	NorthWindModel.Order
Multiplicity	* (Many)
Role	Orders
End	NorthWindModel.Customer
Multiplicity	0..1 (Zero or One)
Role	Customers
General	
Association Set Name	FK_Orders_Customers
Documentation	
Name	FK_Orders_Customers

The screen above shows that relation has two ends. One end is the Orders which is the many side of the relationship indicated by Multiplicity. Other end of the relationship is the customer which is 0-1. If you decide that relationship inferred for Customer side is incorrect and that an order will always have a customer, you can change the multiplicity from the Customer side to 1 as shown below.

Properties	
NorthWindModel.FK_Orders_Customers Association	
Ends	
End	NorthWindModel.Order
Multiplicity	* (Many)
Role	Orders
End	NorthWindModel.Customer
Multiplicity	1 (One)
Role	Customers
General	
Association Set Name	FK_Orders_Customers
Documentation	
Name	FK_Orders_Customers

EDM designer also supports mapping many to many relationships in the designer. If two tables in the database are related to each other using a third table that only has primary columns from both tables, entity framework will automatically infer the join as many to many relationship. Opening up the relationship in the properties window, you will see that both ends of the relationship have a many relationship

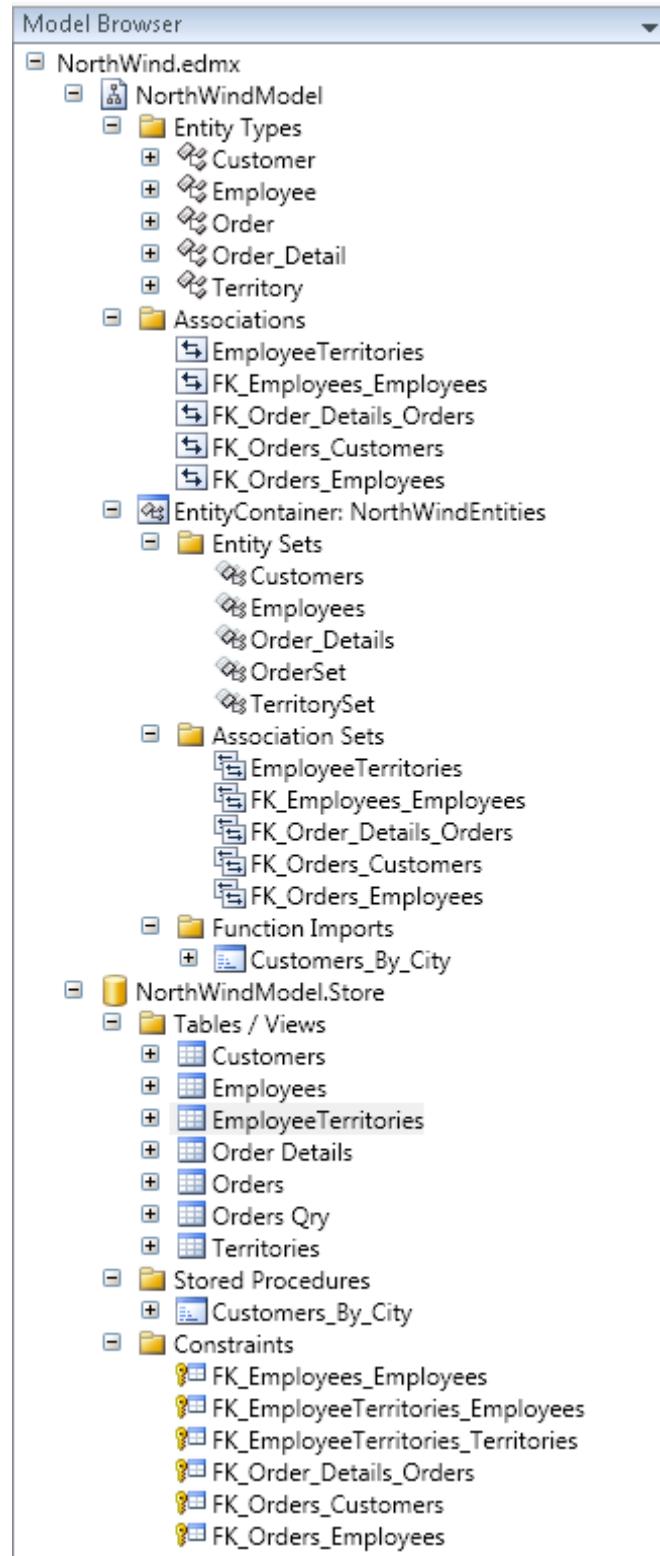
specified on the multiplicity. Screenshot below shows a many to many relationship.



Properties	
NorthWindModel.EmployeeTerritories Association	
	A Z
	X
Ends	
End	NorthWindModel.Territory
Multiplicity	* (Many)
Role	Territories
End	NorthWindModel.Employee
Multiplicity	* (Many)
Role	Employees
General	
Association Set Name	EmployeeTerritories
Documentation	
Name	EmployeeTerritories

On the screen shot above, I also have a self referencing relation in which an employee has a relation back to itself. This is a scenario where an employee reports to a manager and a manager has many employees working under him.

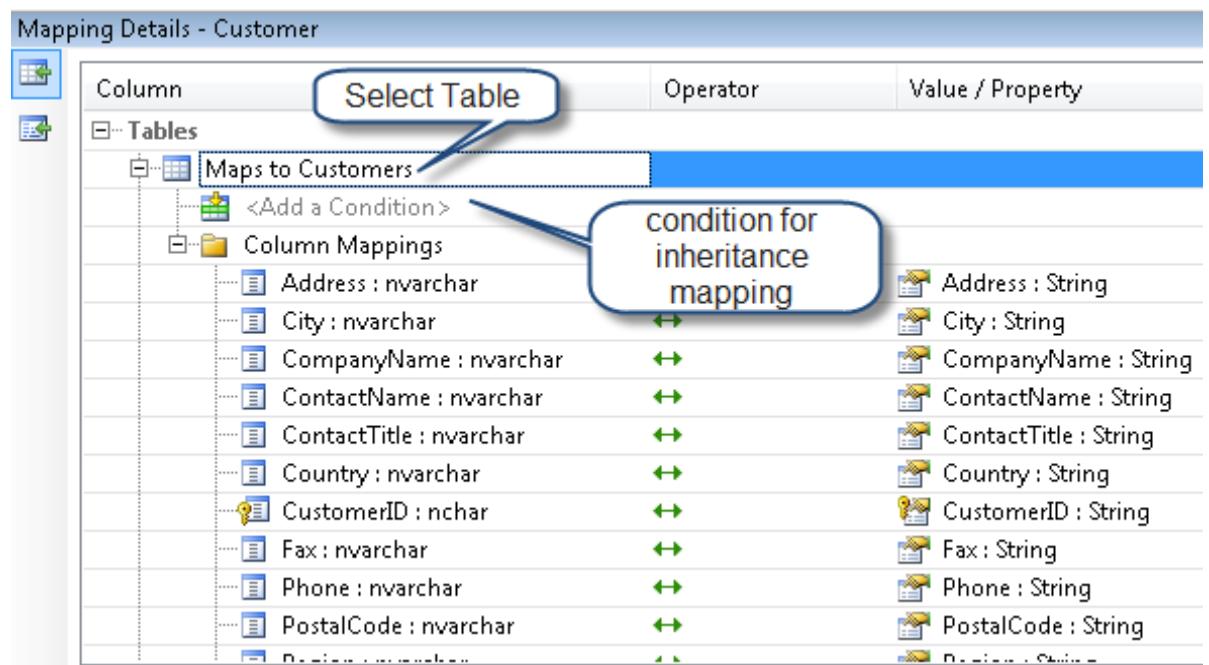
EDM designer has another window Model browser. Model browser window gives you a bigger picture of your entity data model. You can see all your entities, EntitySets, their Associations', any stored procedure imported from the database and the storage model extracted from the database. Figure below shows how model browser displays entity data model.



Model browser window has a section for Entity Types that contains all the entities declared in your model. Expanding the entities will list your

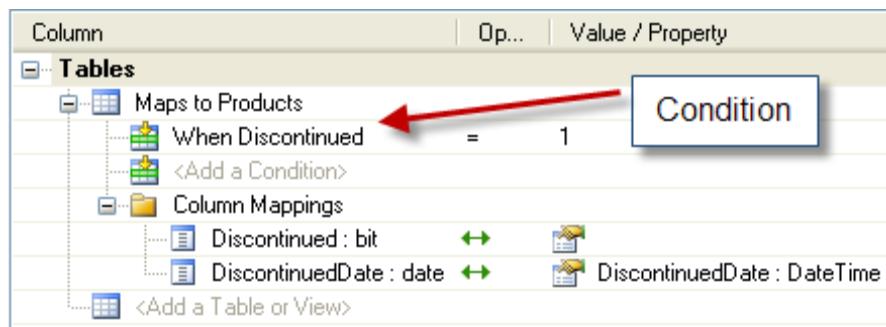
all the properties of the entity. An instance of an entity can be considered an instance of a row in a table. After Entity Types, there is a section for Associations which contains all the relationships defined in the entity data model. A Relation is a first class citizen in entity framework which has two sides. Each side of a relationship returns either an entity reference or entity collection depending on the multiplicity setup. Model browser also displays an Entity Container that contains both EntitySets and AssociationSets. An EntitySet is like a table where all entities reside in and AssociationSet is like a join defined in the database. Next node of the model is Functional Import that contains stored procedures imported in the model. Model browser also contains store node that defines various tables, views, stored procedures and constraints imported into the model.

So far we have seen entities on the entity designer. Entities need to be mapped to the store model. For this purpose there is a Mapping Detail window which you can access by selecting an entity on the designer and clicking mapping window at the bottom. Below is a screen shot of how our mapping window looks like?



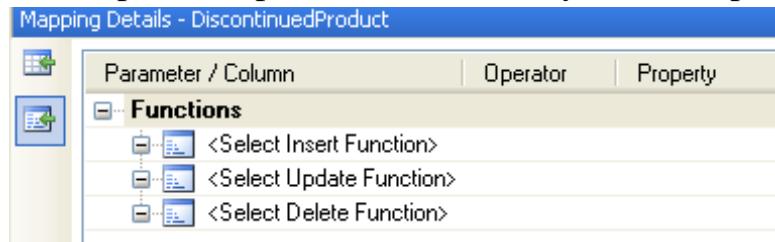
On the above mapping window, the left side of the window includes the table and its columns. The right side shows the properties of the entity selected. Mapping window allows you to map your entity to more than one table. Being able to map a single entity to multiple tables is referred as entity splitting. For instance if you have customer and customer info table in the database that have 1 to 1 association and you want to represent both tables as a single entity in your model, you can use entity splitting feature supported in mapping details window and map your entity to multiple tables. In the screen shot above, I have selected Customer and its column to map to properties on customer entity.

There is also a section on mapping details window to map an entity based on condition. Conditions are used with discriminator column to map inheritance defined on your entities to a single table in the database. For instance Products table contains discontinued products but you are representing discontinued products as a separate entity in the model. To map additional columns on Discontinued Product entity such as discontinued date back to Product table, you have to add a condition where discontinued column is true. Screen shot below shows how discontinued Product and its additional property are mapped to Product table.



Left side of the window is a section to map insert, update and delete operations performed on an entity to stored procedures. If you will take advantage of the dynamic sql statement generated by the entity framework to perform crud operations, you don't need to map stored

procedures for inserts, updates and deletes. Figure below shows screen that maps crud operations on an entity to stored procedure.



Additionally entity framework includes support for documenting code generated class files. To provide documentation for the class, you can use Summary property on the properties window for an entity. This feature is only available using entity framework designer. Linq to Sql designer does not support this feature. Screen shot below shows applying summary to Customer entity and generated class containing our summary description specified in the designer.

Documentation	
Long Description	
Summary	Customer Summary
Entity Set Name	Customers
Name	Customer

```
/// <summary>
/// Customer Summary
/// </summary>
/// <KeyProperties>
/// CustomerID
/// </KeyProperties>
[global::System.Data.Objects.Data
[global::System.Runtime.Serializa
[global::System.Serializable()]
public partial class Customer : g
{
```

Features Entity Framework designer does not support

1. Complex Types not supported.

1.2 Loading csdl,msl,ssdl schema files

Problem: You want to know different ways edmx schema files can be loaded in an application from embedding in output assembly to looking for the schema file in output directory.

Solution:

When you create an instance of `ObjectContext`, one of parameters passed to the constructor is the connection string that determines where to find edmx files and the connection string to the database. One of the portions of the connection string is a Meta data that defines the location for the csdl, msl and ssdl. The location could either be a physical location where all 3 files resides or a reference to assembly where all 3 files are embedded as a resource. Example below shows two different versions of the connection string one containing the location of the file and other points to the assembly where the schema file resides.

```
<add name="NWComplexTypeEntities"
      connectionString="metadata=ComplexType\NorthWindEFComplexType.csdl|
                        ComplexType\NorthWindEFComplexType.ssdl|
                        ComplexType\NorthWindEFComplexType.msl;" />

<add name="NorthwindEFEEntities"
      connectionString="metadata=res://*/NorthWind.csdl|
                        res://*/NorthWind.ssdl|
                        res://*/NorthWind.msl;" />
```

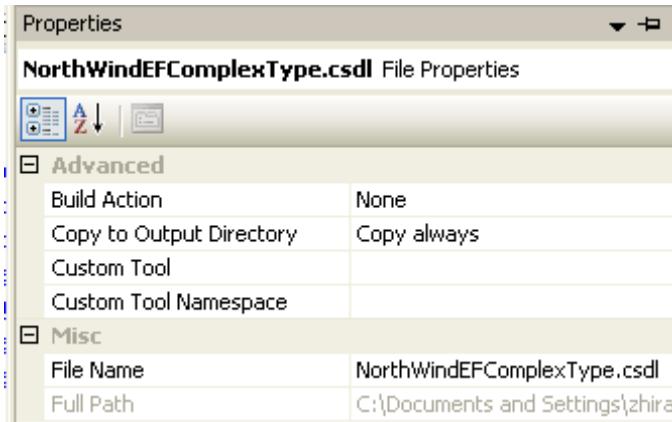
Uses
embedded
resource

Using
physical
path

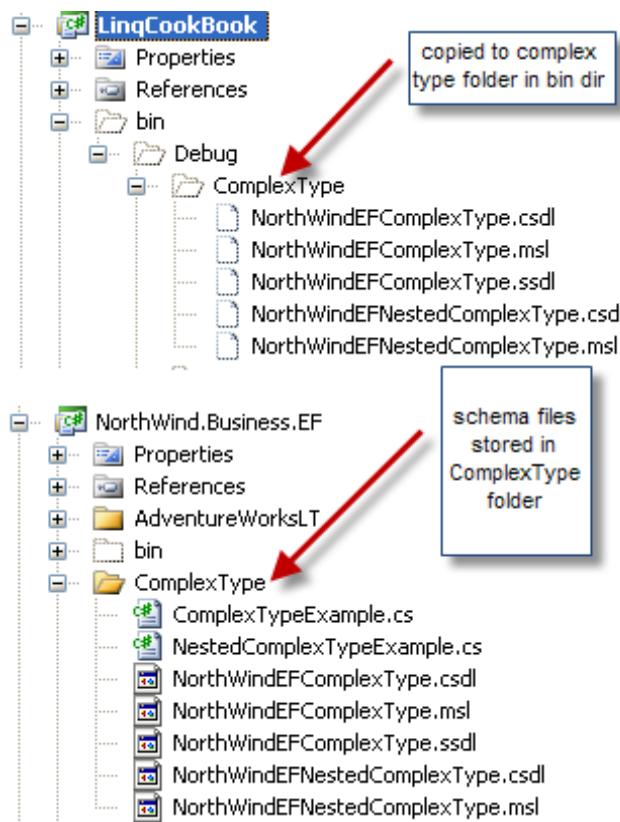
Discussion:

If you are not using entity framework designer meaning hand coding your schema files or generating the schema files using `edmgen` utility that comes with entity framework, you will start with 3 physical files in your class

library project. On the properties window of the files, you can set copy to output directory to true. Thus when you add a reference to your class library project, all 3 files will be copied to the bin directory of the consuming project whether it be a console, windows form or asp.net application. Screen shot below how to set up copy to output directory to true on a schema file.



On the above screen shot, I have set my Build Action to None and Copy to Output directory to always. Copy to Output has 3 options; Never Copy, Copy always and copy when changed. After setting the copy action to true, every time you build the project, schema files will be copied to the bin directory of the project. If schema files are not in root of the class library project, than when files gets copied over to the bin directory, the directory structure remains intact. Example below shows schema files reside in ComplexType directory of my class library project. When you build the solution, schema files are inside Complex Type folder of the bin directory of the project.



When you create an instance of `ObjectContext`, the constructor is passed the name of the connectionstring to look for in `web.config` or `app.config`. Inside the `Meta` section of the connection string, you must explicitly specify the folder name `ComplexType` created in the `bin` directory and where the schema file resides.

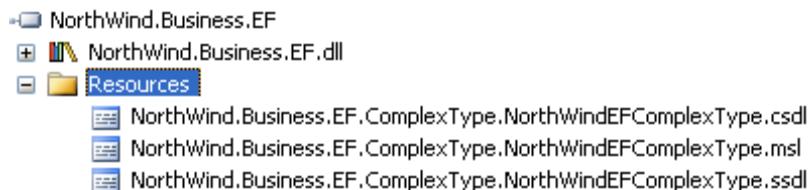
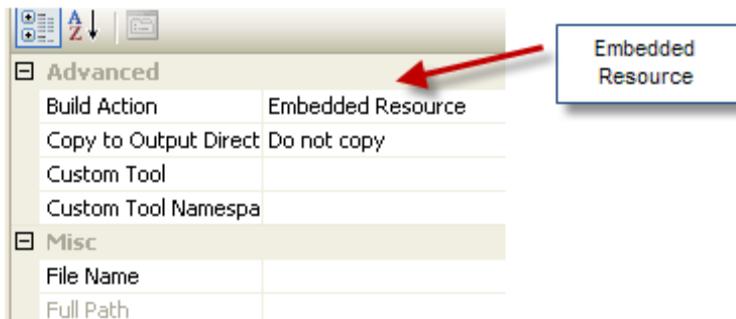
```

<add name="NWComplexTypeEntities"
connectionString="metadata=ComplexType\NorthWindEFComplexType.csdl|
ComplexType\NorthWindEFComplexType.ssdl|
ComplexType\NorthWindEFComplexType.msl;" />

```

A callout box labeled 'ComplexType Folder' has arrows pointing to the three schema file paths in the configuration code.

Other option you can take is embed all 3 files as an embedded resource in the assembly and when the assembly gets copied over in the build process to the `bin` directory, you will have all 3 schema files. Screen shot below shows setting all 3 files as embedded resource and confirming that all files are actually stored as resource for our `NorthWind.business.EF` class library project.



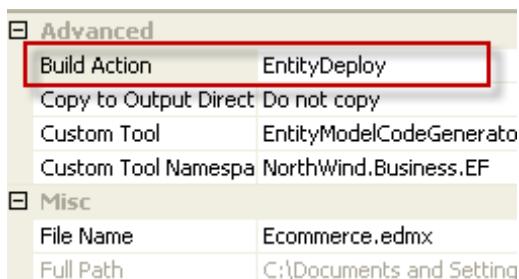
Since we have configured the schema files to reside inside the dll we need to change the connection to as follows.

```
<add name="NWComplexTypeEntities"
      connectionString="metadata=res://*/NorthWind.Business.EF.ComplexType.NorthWindEFComplexType.csdl
      |res://*/NorthWind.Business.EF.ComplexType.NorthWindEFComplexType.ssdl|
      res://*/NorthWind.Business.EF.ComplexType.NorthWindEFComplexType.msl;" />
```

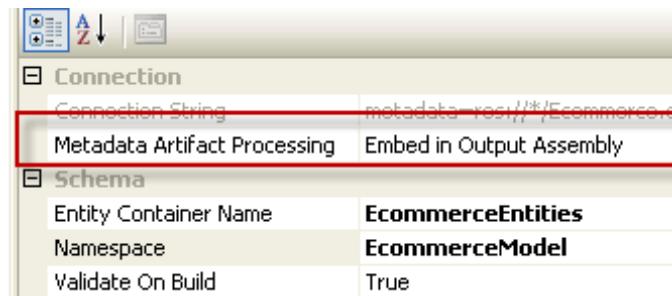
For clarity purpose I am not showing the entire connection string except how to load the schema files. I am using res to find the specified csdl, msl and ssdl in any assembly it can find. To reference our csdl, msl and ssdl, we are fully qualifying the file with the assembly name where the files are embedded as a resource. The res option has different options you can use to ease the search of finding schema files. Following table defines the different options you can use with res to search for schema files.

Res://myassembly/file.ssdl	Loads ssdl file from myassembly
Res://myassembly/	Loads ssdl,csdl and msl from myassembly
Res://*/file.ssdl	Loads file.ssdl from all assemblies it can find.
Res://*/	Loads ssdl,csdl and msl from all assemblies it can find.

So far we have covered how to load schema files either from a physical location to an embedded resource in an assembly when manually maintaining the files ourselves or generating schema files using edmgen utility. So how does this process works when we are using the Edmx designer. When you use the Entity data model designer, the edmx file's build action is set to Entity Deploy. Entity Deploy is an msbuild task to deploy entity framework artifacts generated from edmx files such as our 3 files which we created manually earlier. Screen shot below shows edmx files build action set to EntityDeploy.



Also when you open up the properties window for edmx designer, the Metadata Artifacts Processing is set to Embed in Output Assembly which means to embed our 3 schema files as an embedded resource as shown below.



If you generate your model inside of asp.net or windows application as compared to using class library project, the config file will use the most liberal form of finding schema files by trying to load all 3 schema files from all assemblies it can find by using res with a * option no name specified for csdl msl and ssdl as shown below

```
res:///*
```

1.3 Implementing IPOCO with Entity Framework

Problem: You have Plain Old CLR Objects and you need to know how to use entity framework mapping feature to map your business entities and its attribute to columns in a table.

Solution: Entity framework does not support complete persistence ignorance. What I mean by persistence ignorance is the fact that you can use your objects with entity framework without knowing about the persistence provider being used. This way later if you decide to use your objects against new ado.net technology stack your business rules and code base does not need to change. All ORM solutions in the market to some degree try to get closer to concept of complete persistence ignorance but in reality you are faced with some limitations enforced by the provider that you must adhere to work with the framework. Some of the limitations include specific constructor for tracking objects, inheriting from a base class which notifies the framework of how to track the object using a specific key value. In the case of entity framework, you are required to implement 3 interfaces: IEntityWithChangeTracker, IEntityWithKey, IEntityWithRelationships. Although you are not required to implement IEntityWithKey interface but is strongly recommended as it improves performance for tracking an entity. When you use entity framework designer, all your generated classes inherit from a base class called EntityObject. EntityObject handles all details of informing the framework about different attributes of an object. EntityObject implements the 3 interfaces we mentioned earlier. Apart from these interfaces, EntityObject also implements StructuralObject which its uses to set field values for properties.

To demonstrate how to use CLR objects with entity framework, we will create Customer and Orders class and map these objects to table in the database using entity framework. After completing the mapping we will create a data context class to query for these objects and entity framework will translate the query into appropriate sql defined by our mapping file.

Below is our Customer class which includes all the attributes and interface implementation to work with entity framework.

Listing 1-1 Customer entity implementing IEntityWithChangeTracker, IEntityWithKey interfaces.

```
namespace LinqCookBook.EFUsingPOCO
{
    [EdmEntityTypeAttribute
    (NamespaceName="LinqCookBook.EFUsingPOCO", Name="Customer")]
    public class Customer : IEntityWithChangeTracker,
    IEntityWithKey
    {
        // IEntity tracker is required to participate in
        // change tracking.

        private IEntityChangeTracker changetracker;
        public void SetChangeTracker(IEntityChangeTracker
changeTracker)
        {
            this.changetracker = changeTracker;
        }

        protected void ReportPropertyChanging(string
propertyname)
        {
            if (changetracker != null)
            {

changetracker.EntityMemberChanging(propertyname);
            }
        }

        protected void ReportPropertyChanged(string
propertyname)
        {
            if (changetracker != null)
            {

changetracker.EntityMemberChanged(propertyname);
            }
        }

        EntityKey entitykey;
        public System.Data.EntityKey EntityKey
        {
            get
        }
    }
}
```

```
        {
            return entitykey;
        }
        set
        {
            entitykey = value;
        }
    }

    string customerid;
    [EdmScalarPropertyAttribute
        (EntityKeyProperty=true,IsNullable=false)]
    public string CustomerID
    {
        get{return customerid;}
        set
        {
            ReportPropertyChanging("CustomerID");
            customerid = value;
            ReportPropertyChanged("CustomerID");
        }
    }
    string contacttitle;
    [EdmScalarPropertyAttribute]
    public string ContactTitle
    {
        get
        {
            return contacttitle;
        }
        set
        {
            ReportPropertyChanging("ContactTitle");
            contacttitle = value;
            ReportPropertyChanged("ContactTitle");
        }
    }
    string companyname;
    [EdmScalarPropertyAttribute(IsRequired=false)]
    public string CompanyName
    {
        get{return companyname;}
        set
        {
            companyname = value;
        }
    }
}
```

```
    }  
}
```

Discussion:

In the Listing 1-1, I have marked the class with EdmEntityTypeAttribute which tells entity framework that customer class in an entity.

EdmEntityTypeAttribute attribute takes two parameters. First parameter represents the namespace where the entity resides and second parameter specifies the name of the entity. Next I implement 2 interfaces IEntityWithChangeTracker and IEntityWithKey which entity framework requires for persistence and tracking of customer class. The first interface IEntityWithChangeTracker is used to enable change tracking on the customer object. When you implement the interface you provide reference to IEntityChangeTracker by implementing the SetChangeTracker method. ChangeTracker is then by properties on the object to report changes. Listing 1-2 implements SetChangeTracker method on IEntityChangeTracker interface.

Listing 1-2

```
private IEntityChangeTracker changetracker;  
public void SetChangeTracker(IEntityChangeTracker  
changeTracker)  
{  
    this.changetracker = changeTracker;  
}
```

In the above code, I am setting the reference of my private variable changetracker to the value passed in the parameter of SetChangeTracker method. The reason I am keeping the reference of change tracker in my class is, when any of the property on my object gets changed, I will simply call changetracker.EntityMemberchanging passing in the name of the property to report changes. Since we want to notify the framework both before and after the property changes we will create two methods that will call entitymemberchanging and entitymemberchanged in their respective methods as follows.

Listing 1-3

```
protected void ReportPropertyChanging(string
propertyname)
{
    if (changetracker != null)
    {

changetracker.EntityMemberChanging(propertyname);
    }
}
protected void ReportPropertyChanged(string
propertyname)
{
    if (changetracker != null)
    {

changetracker.EntityMemberChanged(propertyname);
    }
}
```

In listing 1-3, I have declared two methods called ReportPropertyChanging and ReportPropertyChanged. ReportPropertyChanging first checks to see if we have a reference to changetracker. If the reference is not null we pass in the name of property that is about change by calling EntityMemberChanging on the changetracker object. Similarly when the property is changed successfully we call ReportPropertyChanged method which ultimately calls EntityMemberChanged on changetracker to notify that property has changed.

Next interface Customer class implements is IEntityWithKey which exposes an entity key to object services. Entity Key is used by object services to identity and track objects. If you do not implement this interface, you would see considerable decrease in performance. In Listing 1-4, we are implementing IEntityWithKey interface by exposing a getter and setter that sets the value entity key private variable.

Listing 1-4 implementing IEntityWithKey interface

```
EntityKey entitykey;
```

```

public System.Data.EntityKey EntityKey
{
    get
    {
        return entitykey;
    }
    set
    {
        entitykey = value;
    }
}

```

If your object is going to have navigation properties or relationship to other properties, you are also required to implement an interface called IEntityTypeWithRelationships which I have about earlier.

IEntityTypeWithRelationships interface is used by a class to get reference to the relationship manager object. You then use that reference to access relatedCollection such as orders for a customer. Similarly if you have an order class, you can access the customer reference by using relation manager object. To keep the example simple and illustrate bare minimum requirements to use an object with entity framework, I am not exposing any relations in my customer object class. In future recipes I will cover how to use relationship manager to access other entities related to Customer entity.

Apart from implementing the two interfaces, Customer class exposes 3 properties CustomerId, ContactName and CompanyName. To expose these properties to entity framework, we are marking the properties with EdmScalarPropertyAttribute as shown in listing 1-5

Listing 1-5

```

string customerid;
[EdmScalarPropertyAttribute
    (EntityKeyProperty=true, IsNullable=false)]
public string CustomerID
{
    get{return customerid;}
    set
    {
        ReportPropertyChanging("CustomerID");
        customerid = value;
        ReportPropertyChanged("CustomerID");
    }
}

```

In Listing 1-5, I am exposing my customerid property by attributing the property with EdmScalarPropertyAttribute. I am also passing two additional parameters to the attribute. First parameter EntityKeyProperty is set to true to tell that customerid is the key property on the customer object. Since cutomerid cannot be null, I am also setting isnullable to false. The other properties on the Customer object also work the same way so I am not going to cover them. In the setter of all the 3 properties, I am calling ReportingPropertyChanging and Changed to notify change tracker when a property is changing and has changed. This information is useful for the framework to identity the state of the object.

Once I have defined my customer object, I have to create 3 xml files. First file will be our conceptual schema which defines how our customer object and its properties. Second file would be our schema file that defines customer table in the database meaning what columns the table has, its datatype and what is the primary key column. The third file will be our mapping file which defines how to map our customer object to customer table in the database. Listing 1-6 shows conceptual schema of our customer object represented in xml format.

Listing 1-6 Customer object defined in NorthWindEFModel.csdl conceptual model

Listing 1-7

```
<?xml version="1.0" encoding="utf-8" ?>
<Schema Namespace="LinqCookBook.EFUsingPOCO" Alias="Self"
xmlns="http://schemas.microsoft.com/ado/2006/04/edm">
    <EntityContainer Name="NorthwindEntities">
        <EntityType Name="Customers"
EntityType="LinqCookBook.EFUsingPOCO.Customer" />
    </EntityContainer>
    <EntityType Name="Customer" >
        <Key>
            <PropertyRef Name="CustomerID" />
        </Key>
        <Property Name="CustomerID" Type="String"
Nullable="false" />
        <Property Name="ContactTitle" Type="String" />
        <Property Name="CompanyName" Type="String"
Nullable="false" />
    </EntityType>
</Schema>
```

In listing 1-6, we start with EntityContainer that defines the namespace where customer object reside. Entity container has EntitySet which defines the entities that we are going to be exposing through our objectcontext. In our case we are going to be exposing Customers which would be of Customer type. When we work on building our ObjectContext, you will see that we use Customers to query for customer object. Next we define our entity in EntityType attribute by defining the key property in the customer table followed by other properties with their data type and whether the property can allow null or not.

Next we define our customer schema in NorthwindModel.ssdl schema file as shown in listing 1-7. Schema file in listing 1-7 contains an entity set with Name being the table name and entity type being the type for customer. Than using the Entitytype element, I am defining my customer table with the column information, column's data type, length, and whether the column is nullable or not. I am also defining the primary key column in the customer table by using key element and specifying CustomerId as the PropertyRef.

Listing 1-7

```
<?xml version="1.0" encoding="utf-8" ?>
<Schema Namespace="NorthwindEFModel.Store"
    Alias="Self"

    xmlns="http://schemas.microsoft.com/ado/2006/04/edm/ssdl"
        Provider="System.Data.SqlClient"
        ProviderManifestToken="2008">
    <EntityContainer Name="dbo">
        <EntitySet Name="Customers"
EntityType="NorthwindEFModel.Store.Customers" />
    </EntityContainer>
    <EntityType Name="Customers">
        <Key>
            <PropertyRef Name="CustomerID" />
        </Key>
        <Property Name="CustomerID" Type="nchar"
Nullable="false" MaxLength="5" />
        <Property Name="CompanyName" Type="nvarchar"
Nullable="false" MaxLength="40" />
        <Property Name="ContactTitle" Type="nvarchar"
MaxLength="30" />
    </EntityType>
</Schema>
```

Once I have defined the storage model, I have to create a mapping file which defines how customer entity maps to customer table. Listing 1-8 shows the mapping file I have created. Mapping file has an attribute CdmEntityContainer that defines the namespace where my entities reside. Within in the EntityTypeMapping, I am specifying my customer type with TypeName maps to Customers table defined by StoreEntitySet. Inside the MappingFragment, I am mapping scalar properties on customer entity to column name in customer table.

Listing 1-8

```

<Mapping Space="C-S" xmlns="urn:schemas-microsoft-
com:windows:storage:mapping:CS">
  <EntityContainerMapping
    StorageEntityContainer="dbo"
    CdmEntityContainer="NorthwindEntities">
    <EntitySetMapping Name="Customers">
      <EntityTypeMapping TypeName=
LinqCookBook.EFUsingPOCO.Customer">
        <MappingFragment StoreEntitySet="Customers">
          <ScalarProperty Name="CustomerID"
ColumnName="CustomerID" />
          <ScalarProperty Name="CompanyName"
ColumnName="CompanyName" />
          <ScalarProperty Name="ContactTitle"
ColumnName="ContactTitle" />
        </MappingFragment>
      </EntityTypeMapping>
    </EntitySetMapping>
  </EntityContainerMapping>

</Mapping>

```

Now that we have defined our conceptual modal, mapping and storage model, we need to create our datacontext class which can talk to the model we have defined in the xml file. Listing 1-9 shows the code for NorthwindEntities class. In the example, I am creating NorthWindEntities class which inherits from ObjectContext. ObjectContext class is responsible for querying and working with entity data as objects. The constructor of the class takes two parameters. First parameter represents the name of connectionstring defined in either web.config or app.config. For this example, I have defined the connectionstring inside app.config as follows

```

<add name="nwefpoco"

```

```

connectionString="provider=System.Data.SqlClient;metadata=EF
UsingPOCO\Tables\NorthwindEFModel.csdl|EFUsingPOCO\Tables\NorthwindEFModel.msl|EFUsingPOCO\Tables\NorthwindEFModel.ss
dl;provider connection string="Data
Source=.\\SQLEXPRESS;AttachDbFilename=|DataDirectory|\\DB\\Nort
hwindEF.mdf;Integrated Security=True;User
Instance=True;MultipleActiveResultSets=True";
providerName="System.Data.EntityClient"  />

```

Second parameter defines the name of the entitycontainer we defined in our conceptual model. Now to query for customers, we are exposing a public property Customers which does lazy loading to load the customers by calling CreateQuery passing in the name of the entityset defined in our conceptual model.

Listing 1-9

```

namespace LinqCookBook.EFUsingPOCO
{
    public class NorthwindEntities : ObjectContext
    {
        public NorthwindEntities()
            : base("name=nwefpoco", "NorthwindEntities") { }
        ObjectQuery<Customer> customers;
        public ObjectQuery<Customer> Customers
        {
            get
            {
                if (customers == null)
                {
                    this.customers =
base.CreateQuery<Customer>("[Customers]");
                }
                return customers;
            }
        }
    }
}

```

To test if we can retrieve Customer objects from ObjectContext, we can write a simple query that retrieves Customers with Contact Title of Sales Representative as shown below.

```

public static void CustomersWithSalesRepresentative()
{
    var db = new NorthwindEntities();

```

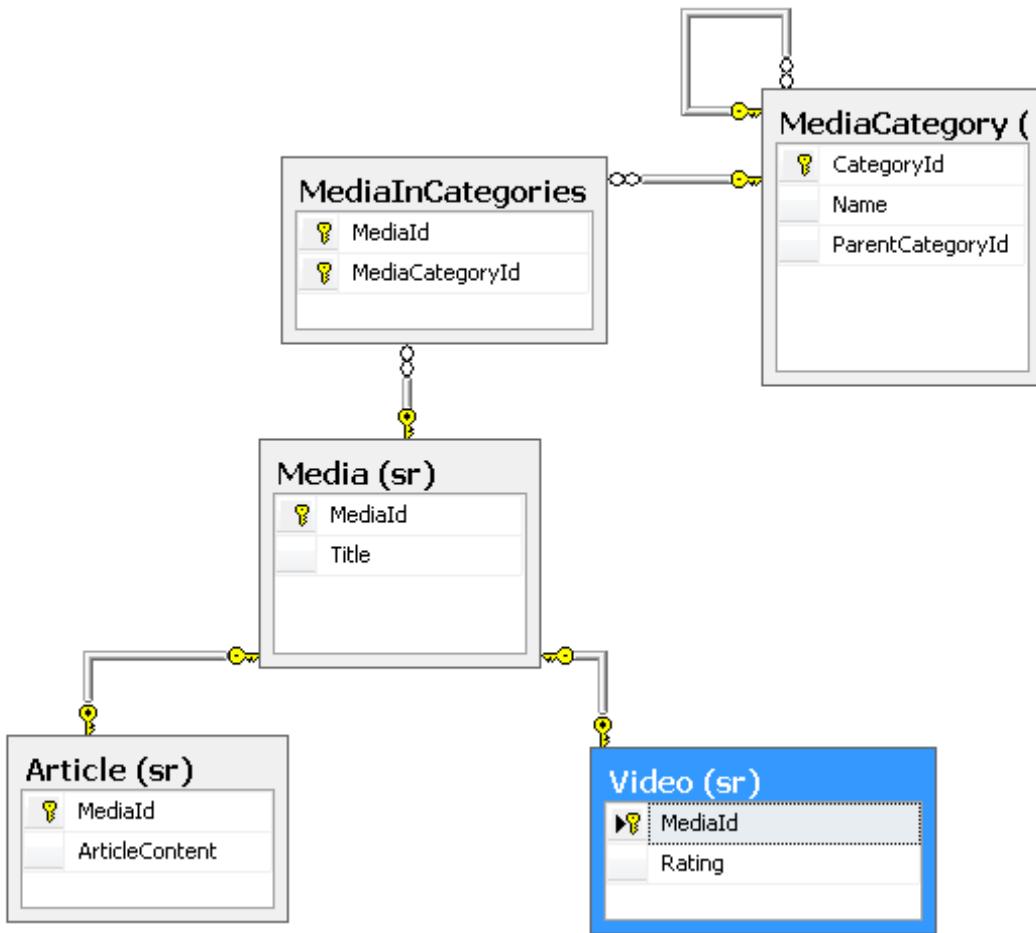
```
    var custs = db.Customers.Where(c =>
c.ContactTitle == "Sales Representative");
    foreach (var cus in custs)
    {
        Console.WriteLine("cust {0}",
cus.CustomerID);
    }
}
```

2. Modeling Entities

2.1 Self Referencing Table

2.1.1 Self Referencing Table with Many to Many Association

Problem: Figure below shows the database diagram for different types of Media belong to many categories.

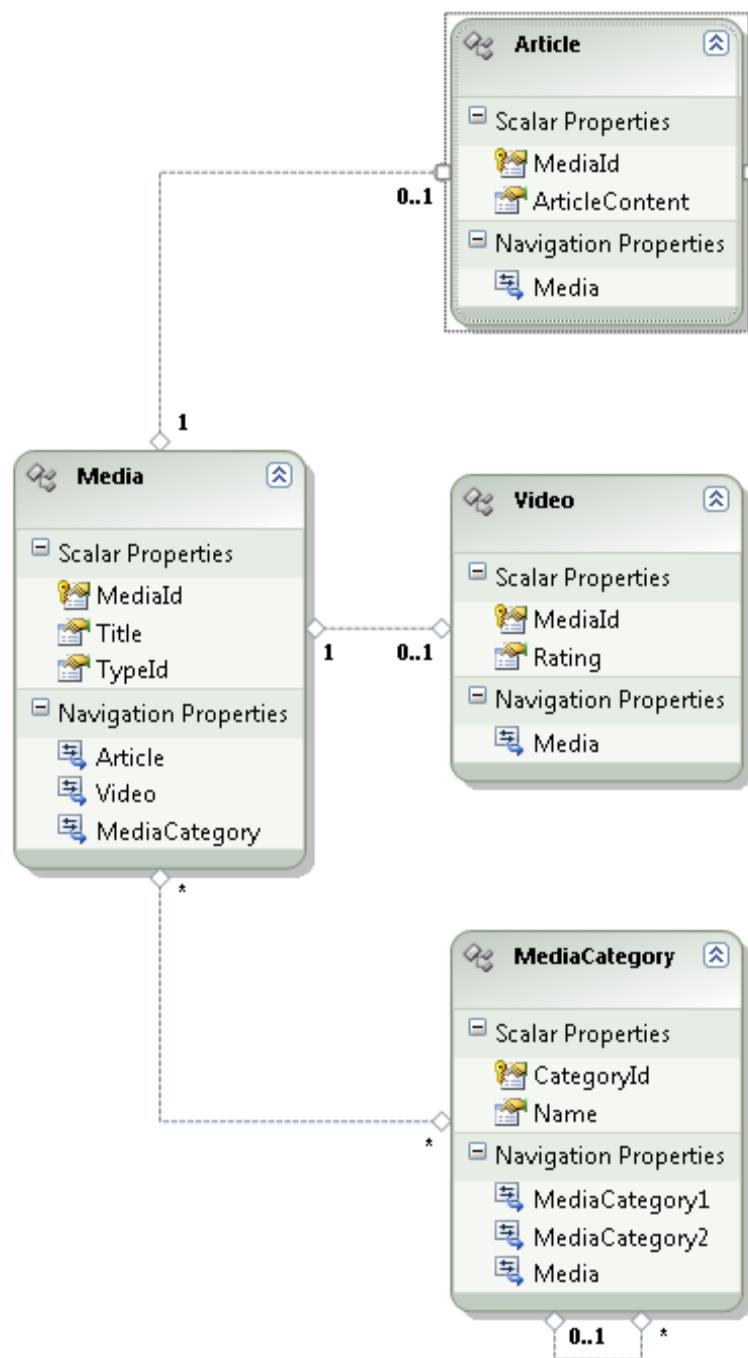


On the figure above, we have **MediaCategories** table which has Categories for various types of Media including Videos and Articles. Each Category can have subcategories which are identified by **ParentCategoryId** column in the **MediaCategory** table. This makes **MediaCategory** a self referencing table where Categories and SubCategories are stored in 1 table and to find out the parent category for a category, we have to look at **ParentCategoryId**. When the **ParentCategoryId** is null, we are at root Category. **Media** table contains common fields across both Articles and Videos. Fields specific to Video and Articles are stored in **Article** and **Video** table. You want to import the above table structure using Self referencing entity which would allow us to get Subcategories for a given Category. Additionally each Category should expose a navigation property **Medias** which should be collection containing Articles and Videos.

Solution: When a table with self referencing relationship is imported into EDM, entity framework automatically creates an association to the entity itself. Since the names generated by the designer for the navigation properties are obscure, you will have to change the names of the relationship. To import the above table structure use entity framework designer. The wizard will create an association to MediaCategory itself. In addition, Many to Many relationship between MediaCategory and Media entity will be created because MediaInCategories is a join table with no payloads, EF will ignore this table. Since the Media table will contain two types of Media, extend the Article and Video class to inherit from Media entity and configure the mappings for both entities using mapping window.

Discussion: EF is aware of self referencing relationship, therefore when we import the above table structure, EF will create a self referencing relationship. To ensure that navigation properties are readable, change the name of the navigation properties. Steps below outline the process of importing the above table structure.

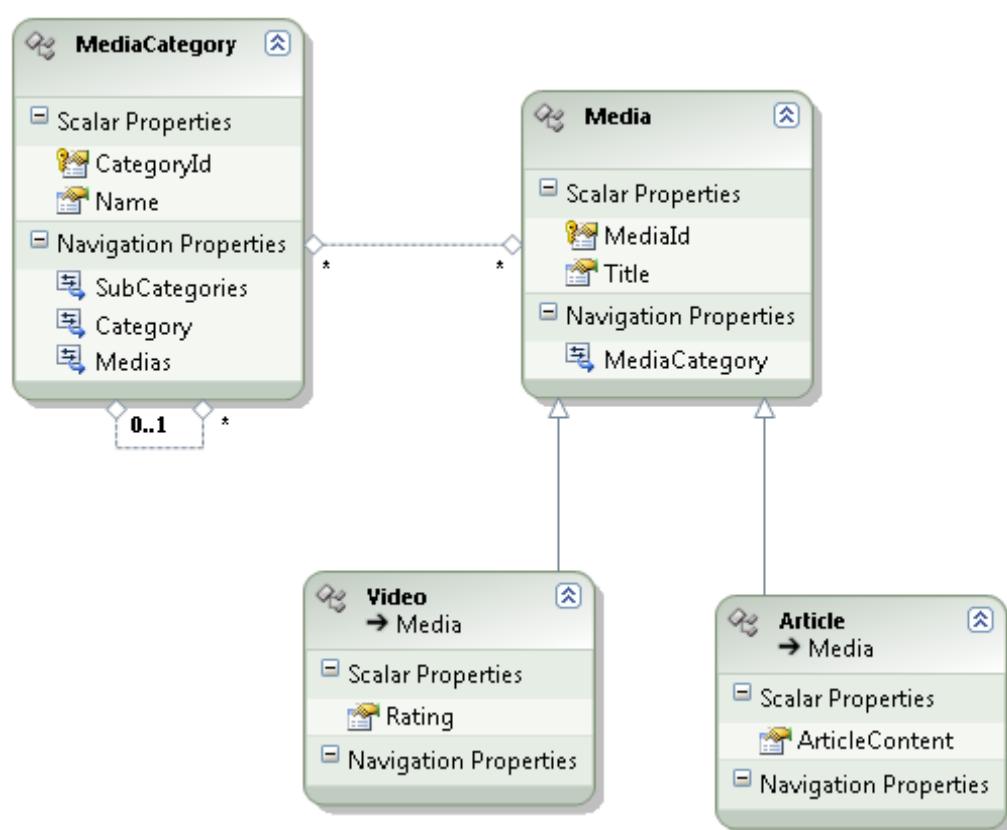
1. Import MediaCategory, MediaInCategories, Media, Article and Video using Entity Data Model Wizard. Figure below shows the model created by the wizard.



Notice that designer created a self relationship to Media entity and created a Many To Many relationship with Media entity by omitting MediaInCategory link table.

2. Change MediaCategory1 navigation property to MediaSubCategories. The reason we are changing MediaCategory1 is because it is the many side of the relationship.

3. Remove Article and Video association from Media. Make entity Media entity abstract because in code, we will either instantiate Article or Video entity.
 4. Make sure Video and Article entity derive from Media. Remove MediaId from Media and Article entity because we will use MediaId inherited from Media entity.
 5. Configure mapping for Video entity where MediaId column maps to MediaId from Media entity.
 6. Configure mapping for Article entity where MediaId column maps to MediaId from Media entity.
- The completed entity model should like below.



Now that the model is complete we can write some queries against the model to return data. If want to access all the top level categories and the total Articles and videos associated with the category, we can write the following query.

```
var db = new MediaSelfRefEntities();
```

```

var cats = from cat in db.MediaCategories.Include("Medias")
           where cat.Category == null /* parent category is null */
*/
           && cat.Medias.Count() > 0
           select cat;

foreach (var cat in cats)
{
    Console.WriteLine("Name:{0} Articles:{1} Videos:{2}",
                      cat.Name,
                      cat.Medias.OfType<Article>().Count(),
                      cat.Medias.OfType<Video>().Count());
}

```

On the code above I am retrieving the top level category by checking if parent category is null. If the parent or root category is null, the category is a root level category. I am also filtering the categories to ones that have some kind media associated with it. Along with Category I am also retrieving the Media association by using Include operator. Since Media is comprised of Article and Video entity, the collection may contain both types of Media. To explicitly find the articles and videos in a given category, I am using OfType operator. Figure below shows the output on the console window.

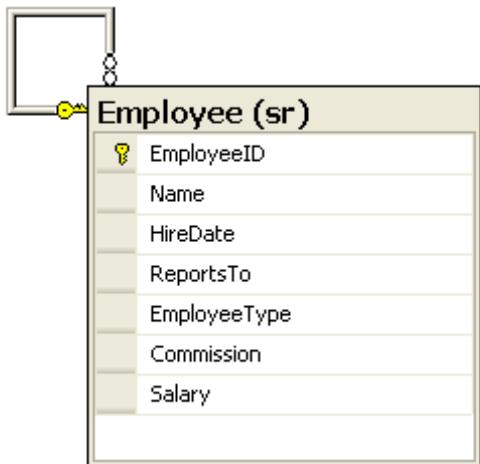
```

C:\Windows\system32\cmd.exe
Name:Ammunition Articles:4 Videos:1
Name:Gun Gear Articles:15 Videos:10
Name:Hunting Articles:5 Videos:0
Name:Knives Articles:3 Videos:2
Name:Optics Articles:2 Videos:0
Name:Fun Stuff Articles:14 Videos:10
Name:Gun History Articles:4 Videos:0
Press any key to continue . . .

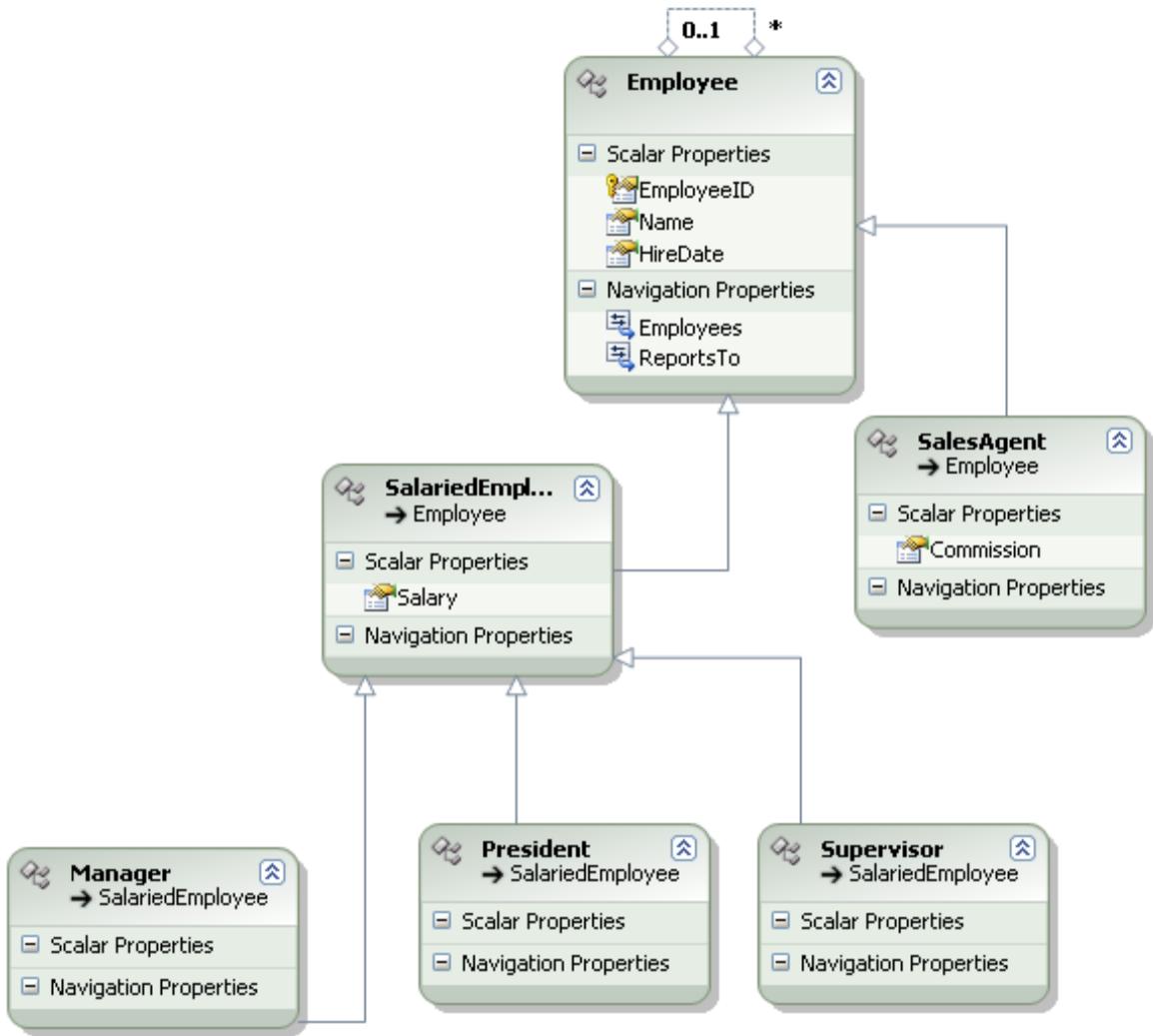
```

2.1.2 Self Referencing entity with Table per Hierarchy

Problem: Figure below shows the database diagram for Employee table that contain Employees with different role.



The above employee table contains Employee with different roles. An employee could be President, Manager, Supervisor or a SalesAgent identified by the EmployeeType column. Each Employee reports to an employee identified by ReportTo EmployeeId; for instance a sales agent would report to a Supervisor and a supervisor would report to Manager and Manager would report to the President. You want to import the above table schema using Table per Hierarchy and each Employee should have a navigation property ReportsTo that tells which Employee, the employee reports to. Completed entity data model should look as follows.

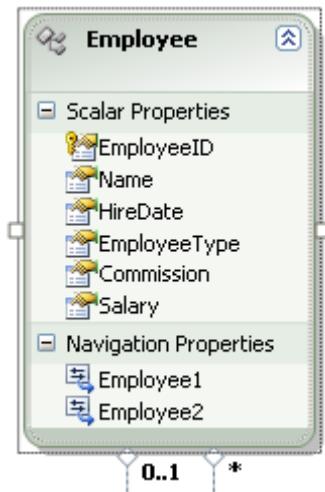


Solution:

Discussion:

Steps below outline the process of importing Employee table as a self referencing entity with Table Per Hierarchy for each type of Employee defined by Employee Type.

1. Import Employee table into EDM wizard. Figure below shows the model created by the wizard.

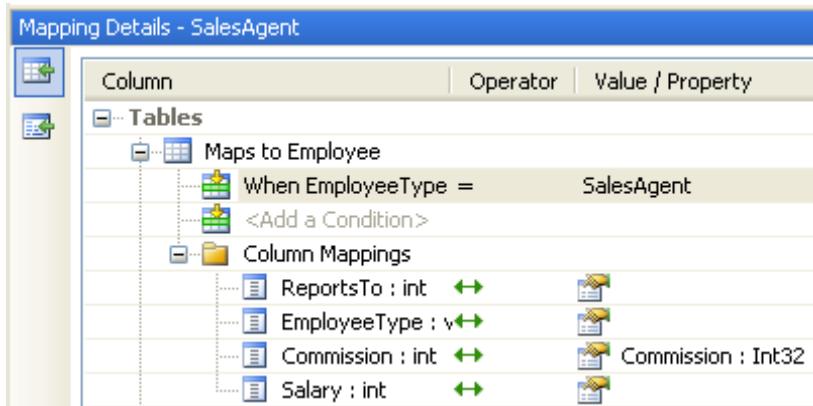


Change the name of Employee1 on the figure to Employees as it is the Many side of the relationship and change Employee2 to ReportsTo. Also change the names of the roles for the self referencing association created by the designer. Call Many side of the role as Employees and 1 side of the role as ReportTo. Figure below shows the name for the association.



Make Employee entity abstract as we will not create an instance of Employee entity directly and will only serve as a base class. Also remove EmployeeType property because EmployeeType column will be used as a discriminator column for inheritance.

2. Create SalesAgent entity deriving from Employee entity. Move commission column from Employee entity to SalesAgent entity and map Commission property to Commission column in Employee table. Set the condition for EmployeeType to SalesAgent.
- Figure below shows the mapping for SalesAgent entity.



3. Create SalariedEmployee and move Salary property from Employee entity to SalariedEmployee entity. Make salaried Employee abstract since it will serve as base class for President, Manager and Supervisor entity. Additionally it allows us to declare Salary property once and use it in all derived types. However when it comes to mapping Salary to the Salary column in the database, it has to be done for each entity deriving from Salaried Employee.
4. Create Manager Entity deriving from SalariedEmployee. Set the table mapping to Employee table and condition where EmployeeType equal to Manager. You won't be able to map properties to columns using the designer because we added SalariedEmployee entity that does not map to any table. In later steps we will edit the edmx in xml and map the properties to columns manually.
5. Create President and Supervisor entity deriving from SalariedEmployee and map both entities to Employee table. For President set EmployeeType condition to President and condition of Supervisor for Supervisor entity.

6. Open up edmx file in xml and configure property mappings for EmployeeId and Salary column for all entities deriving from SalariedEmployee. Code below shows the mapping.

```

<EntityTypeMapping TypeName="IsTypeOf(SFTPHModel.Manager)">
    <MappingFragment StoreEntitySet="Employee" >
        <ScalarProperty Name="EmployeeID"
ColumnName="EmployeeID" />
            <ScalarProperty Name="Salary" ColumnName="Salary" />
        <Condition ColumnName="EmployeeType" Value="Manager" />
    </MappingFragment>
</EntityTypeMapping>
<EntityTypeMapping TypeName="IsTypeOf(SFTPHModel.President)">
    <MappingFragment StoreEntitySet="Employee" >
        <ScalarProperty Name="EmployeeID"
ColumnName="EmployeeID" />
            <ScalarProperty Name="Salary" ColumnName="Salary" />
        <Condition ColumnName="EmployeeType"
Value="President" />
    </MappingFragment>
</EntityTypeMapping>
<EntityTypeMapping TypeName="IsTypeOf(SFTPHModel.Supervisor)">
    <MappingFragment StoreEntitySet="Employee" >
        <ScalarProperty Name="EmployeeID"
ColumnName="EmployeeID" />
            <ScalarProperty Name="Salary" ColumnName="Salary" />
        <Condition ColumnName="EmployeeType"
Value="Supervisor" />
    </MappingFragment>
</EntityTypeMapping>

```

On the code below, I am mapping EmployeeId and Salary property to each entity deriving from SalariedEmployee. The reason we are doing it individually is because SalariedEmployee only serves as abstract class on our conceptual model and does not have any table mapping on the database.

To test the model, we can retrieve our top level employee which is the president. An immediate child for the president would be manager, which will have children of supervisor and that will have SalesAgents working under him. Code below shows how we retrieve four level deep hierarchies.

```

var db = new SFTPHEntities();
var president = db.Employees
    .Include("Employees.Employees.Employees")

```

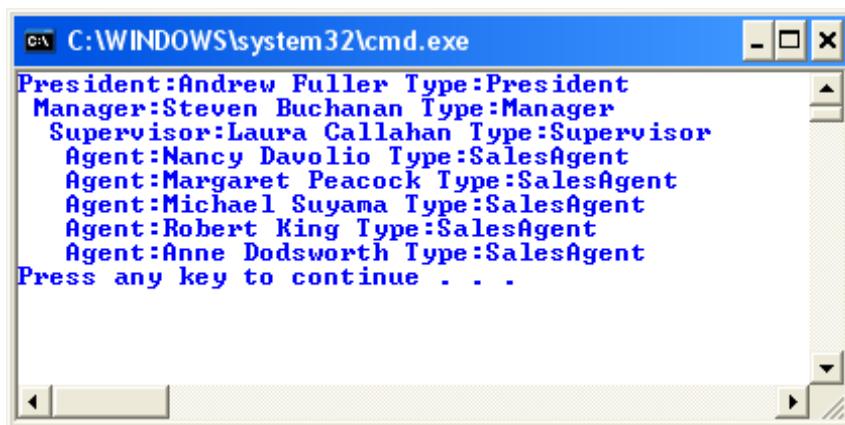
```

        .Where(e => e.ReportsTo == null).First();

    Console.WriteLine("President:{0} Type:{1}", president.Name,
president.GetType().Name);
    var manager = president.Employees.First();
    Console.WriteLine(" Manager:{0} Type:{1}", manager.Name,
manager.GetType().Name);
    var supervisor = manager.Employees.First();
    Console.WriteLine(" Supervisor:{0} Type:{1}", supervisor.Name,
supervisor.GetType().Name);
    foreach (var agent in supervisor.Employees)
    {
        Console.WriteLine(" Agent:{0} Type:{1}", agent.Name,
agent.GetType().Name);
    }
}

```

On the code above, I have included 3 includes for Employees. First include for employee will return the manager reporting to the president. Second Include will return supervisor reporting to the manager and third include would return agents reporting to the supervisor. The query returns the top level employee by checking to see if the employee has no one to report as president does not report to anyone. Next I retrieve the manager for the president and prints it information to the console window. Then for the manager entity, I retrieve its supervisor followed by agents for the supervisors. The screenshot below shows the results printed on the console window.



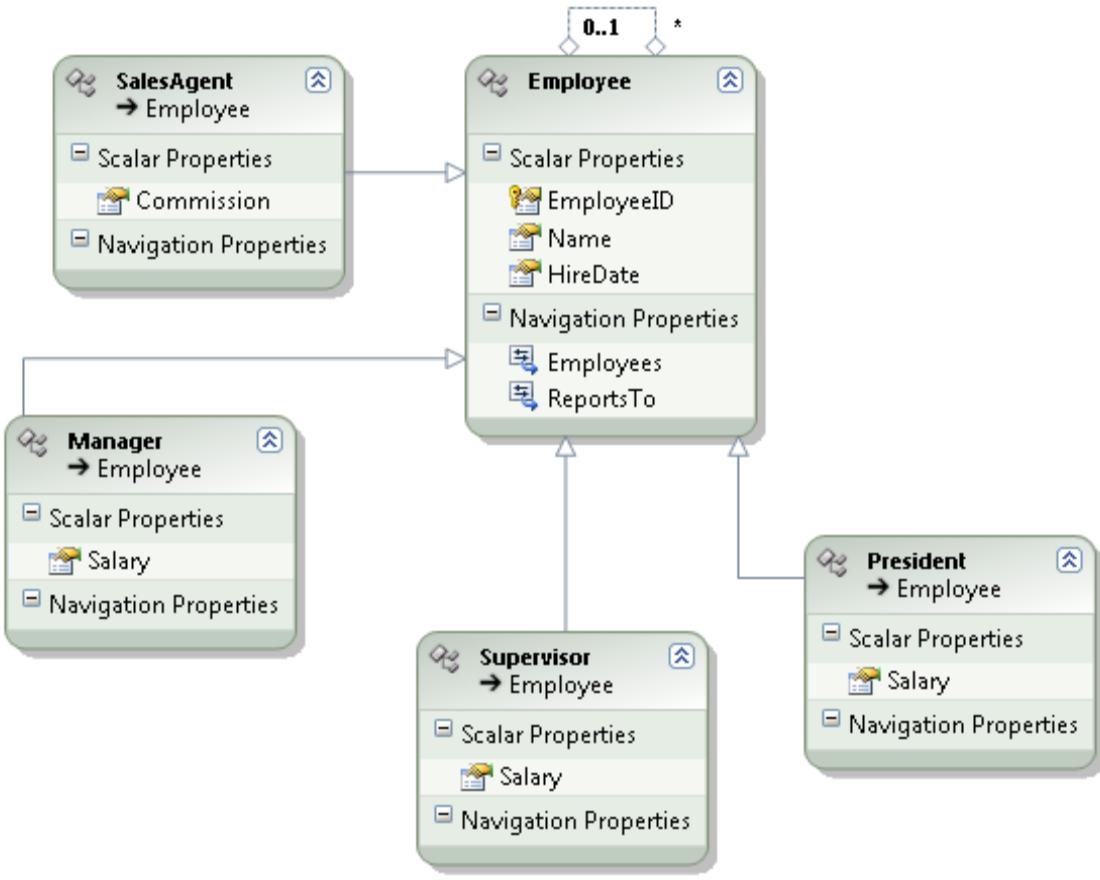
```

President:Andrew Fuller Type:President
Manager:Steven Buchanan Type:Manager
Supervisor:Laura Callahan Type:Supervisor
Agent:Nancy Davolio Type:SalesAgent
Agent:Margaret Peacock Type:SalesAgent
Agent:Michael Suyama Type:SalesAgent
Agent:Robert King Type:SalesAgent
Agent:Anne Dodsworth Type:SalesAgent
Press any key to continue . . .

```

2.1.3 Using Common CTE with Self Referencing Entity

Problem: Figure below shows the conceptual model we created earlier in problem 3.1.3 with slight modifications.



To recap the above model, an Employee can be a SalesAgent, Manager, Supervisor or a President. Each employee except the President reports to an employee above it which can be identified by **ReportsTo** navigation property. In business terms a SalesAgent reports to a Supervisor, a Supervisor reports to Manager and a Manager ReportsTo the president. Given an instance of president entity, you want to return all the employees that directly or indirectly report to the president. This requires that we traverse an infinite dept for the **ReportsTo** relationship. The end result should be a collection of Employees containing SalesAgent, Manager, Supervisor and President.

Solution: EF does not support recursive queries that can return all employees that indirectly reports to the President. The **ReportsTo** navigation property only returns the immediate employees that report to presidents which are Managers. If we want our result to also include employees that report to Manager and so on, we need to make use of Common Table Expressions. CTE is a feature introduced in Sql server 2005 that allows recursive queries

until you reach to the end of the list. To use CTE, we need to create a stored procedure on the database that takes an EmployeeId and returns all employees that directly and indirectly reports to the employeeid passed in. The stored procedure then needs to be imported into the store model. To use the stored procedure inside our ObjectLayer, we need to use FunctionImport to import the stored procedure into the conceptual model and set the return type to Employee entity. Since the employees returned by the stored procedure would contain different types of employees ranging from Manager, Supervisor and SalesAgent, we need to map each row returned by the stored procedure to appropriate derived entity based on the EmployeeType column. After setting up the mapping, we can call the method created on the ObjectContext that takes in the EmployeeId and returns a list of Employees that directly or indirectly reports to the that employee.

Discussion: Steps below outline the process of returning recursive queries for a self referencing entity.

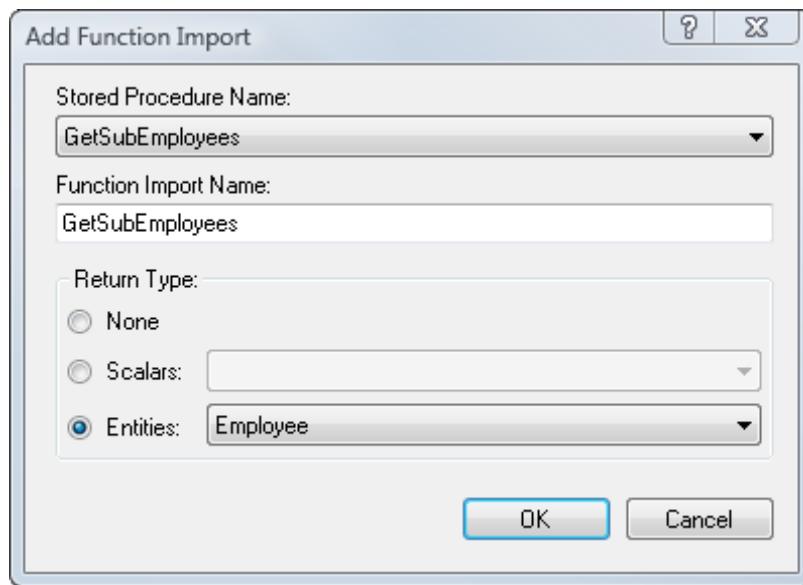
1. To return all employees that directly or indirectly reports to a Employee we need to create a function on the store model that uses Common Table Expression to recursively go through each Employee's ReportsTo column and find its ReportsTo until we reach to the last record in the list. Code below shows the function we need to define on the store model.

```
<Function Name="GetSubEmployees" IsComposable="false">
    <CommandText>
        with emps as
        (
            select e1.*
            from sr.Employee e1
            where EmployeeId = @EmployeeId
            union all
            select e2.*
            from emps join sr.Employee e2 on emps.EmployeeId =
e2.ReportsTo
        )
        select * from emps where employeeid != @EmployeeId
    </CommandText>
    <Parameter Name="EmployeeId" Type="int" Mode="In" />
</Function>
```

We could have created a stored procedure and referenced the stored procedure inside of the function but for demo purposes, it is easier to declare the sql

inline using CommandText option. Beware if you use this technique, your function will get overwritten the next time you try to update the model from the database.

2. Import the stored procedure on the store model into the conceptual model by using Function Import option which can be accessed by right clicking the stored procedure on the model browser and choose Create Function Import. On the dialog, set the return type for the method to Employee entity. Figure below shows the values inputted on the dialog.



3. Since stored procedure returns various derived types of Employees, we need to map each record returned from the stored procedure to its appropriate derived typed. Mapping a derived type to a result returned from the stored procedure cannot be accomplished using the designer. To perform mapping open the edmx file in xml and find the section for the function import and modify the function import as follows.

```
<FunctionImportMapping FunctionImportName="GetSubEmployees"
FunctionName="EcommerceModel.Store.GetSubEmployees">
    <ResultMapping>
        <EntityTypeMapping
TypeName="SelfReferencing.SalesAgent">
            <Condition ColumnName="EmployeeType"
Value="SalesAgent" />
        </EntityTypeMapping>
```

```

        <EntityTypeMapping
TypeName="SelfReferencing.Manager">
    <Condition ColumnName="EmployeeType"
Value="Manager" />
</EntityTypeMapping>
<EntityTypeMapping
TypeName="SelfReferencing.Supervisor">
    <Condition ColumnName="EmployeeType"
Value="Supervisor" />
</EntityTypeMapping>
<EntityTypeMapping
TypeName="SelfReferencing.President">
    <Condition ColumnName="EmployeeType"
Value="President" />
</EntityTypeMapping>
</ResultMapping>
</FunctionImportMapping>

```

On the above code, based on the value on the EmployeeType returning I am mapping the record to a particular derived entity. For instance if EmployeeType has a value of SalesAgent, then we need to create an instance of SalesAgent entity.

Code below test the above recursive method we have created on the ObjectContext.

```

var db = new SFCTEPHEntities();
    var president = db.Employees.First(e => e.ReportsTo == null);
    var emps = db.GetSubEmployees(president.EmployeeID);
    Console.WriteLine("All Employees working under president\r\n");
    foreach (var emp in emps)
    {
        Console.WriteLine("Name:{0} Type:{1}", emp.Name,
emp.GetType().Name);
    }

```

First I am getting the root Employee, president and using its EmployeeId I call GetSubEmployees to get all the employees that work below him. For each Employee that works below him, I am printing Employee Name and the type of Employee using GetType. Since we have configured the entity Type mapping for stored procedure, we will observe from the results on the console window that results would include SalesAgent, Manager and Supervisor. Figure below shows the result on the console window.

```

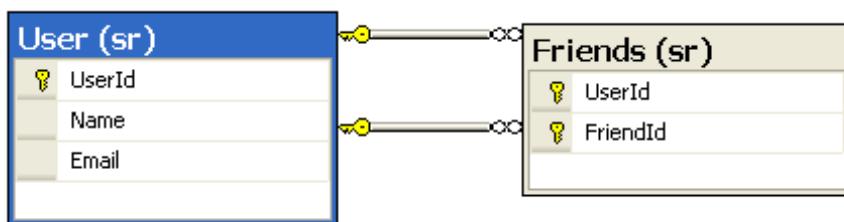
C:\Windows\system32\cmd.exe
All Employees working under president
Name:Steven Buchanan Type:Manager
Name:Laura Callahan Type:Supervisor
Name:Nancy Davolio Type:SalesAgent
Name:Margaret Peacock Type:SalesAgent
Name:Michael Suyama Type:SalesAgent
Name:Robert King Type:SalesAgent
Name:Anne Dodsworth Type:SalesAgent
Press any key to continue . . .

```

Note: In V1 of EF, stored procedure mapping does not support base type that is marked abstract. To overcome this limitation like we did in our case, ensure that base entity Employee is not marked abstract and map it to a dummy discriminator value for EmployeeType that is not defined on the database.

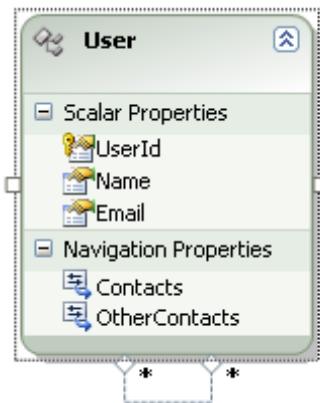
2.1.4 Many To Many association on Self referencing entity

Problem: Figure below shows the database diagram of Linked In Profile where a User has Many Friends and each Friend can know Many users. This is a Many to Many Relationship on User table defined using Friend as a link table.



You want to import the above model as a User entity which has a Many to Many relationship to itself.

Solution: When we import the above model using EF model wizard, EF will create a User entity and also create an M-M relation to the User entity by picking up that there is a link table between User-Friends and User. Figure below shows the model configured for the above table structure.



With the current conceptual model shown above, if I wanted to find all my contacts, I need to use Contacts navigation property. Contacts navigation property returns all the contacts I have made. But if I am the contact for someone else then he also becomes my contact as well. To access those Contacts, I can use OtherContacts navigation property to return Users who made me their contact.

Discussion:

1. Import User and Friends table using EDM wizard. When the wizard completes, you will have a User entity that has a M-M association to itself. Figure below shows the model created by the wizard.



2. Change the navigation property User1 to Contacts and User2 to OtherContacts. Both navigation property should return a collection of Contacts because the both ends of the association refer back to User entity.

To test the above Self referencing User entity with M-M association to itself, we can create a user and add contacts to the user. Then using a second datacontext we can query for all contacts for a user by using both navigation properties Contact and OtherContacts. Code below creates multiple contacts for a user.

```
var db = new MMSFEntities();
    var user = new User
    {
        Name = "Zeeshan",
        Contacts =
        {
            new User
            {
                Name = "Chuck",
                Contacts = {new User{Name="Kirk"}}
            },
            new User
            {
                Name="Larry"
            }
        }
    };
db.AddToUsers(user);
db.SaveChanges();
```

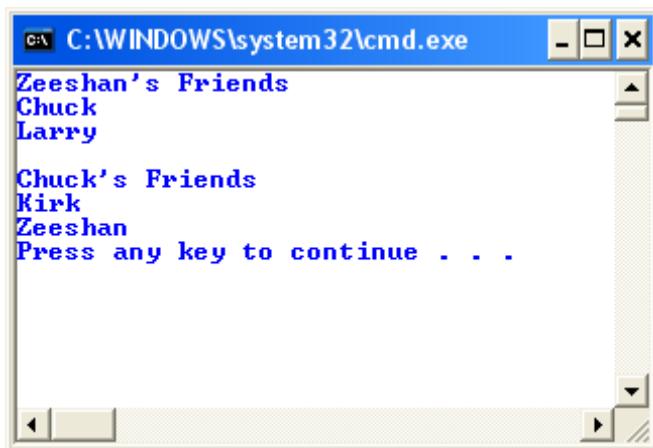
On the above code, user Zeeshan has two contacts, Chuck and Larry. However Chuck has one Contact Kirk but because Chuck became a contact for Zeeshan, Zeeshan should be considered as one of OtherContacts for Chuck. Now using the second datacontext, we can access the contacts for Zeeshan as follows.

```
var db2 = new MMSFEntities();
    var zeeshan = db2.Users.Include("Contacts").First(u => u.Name ==
"Zeeshan");
    Console.WriteLine("Zeeshan's Friends");
    foreach (var User in zeeshan.Contacts)
    {
        Console.WriteLine(User.Name);
    }
```

To access all contacts for Zeeshan, we are eagerly loading Contacts navigation property for Zeeshan and printing the result to console window. However to access all contacts for Chuck, we have to eagerly load Contacts and OtherContacts navigation property. Code below retrieves all contacts for Chuck.

```
var chuck = db2.Users.Include("Contacts").Include("OtherContacts").First(u =>
u.Name == "Chuck");
    //chuck's friend is Kirk
    Console.WriteLine("\r\nChuck's Friends");
    foreach (var User in chuck.Contacts)
    {
        Console.WriteLine(User.Name);
    }
    //zeeshan has Chuck as his contact which makes zeeshan as Chuck's
otherContact.
    foreach (var User in chuck.OtherContacts)
    {
        Console.WriteLine(User.Name);
    }
```

On the above code, we can see that when we access Contacts navigation property for chuck, we only get Kirk because Chuck only made Krik as his contact. However since the user Zeeshan made Chuck his contact, we can access Zeeshan through Chuck's OtherContact navigation property to find out who made Chuck as his contact. Screenshot below shows the final result printed on the console window which shows that Chuck has two contacts; Kirk and Zeeshan.



```
C:\WINDOWS\system32\cmd.exe
Zeeshan's Friends
Chuck
Larry

Chuck's Friends
Kirk
Zeeshan
Press any key to continue . . .
```

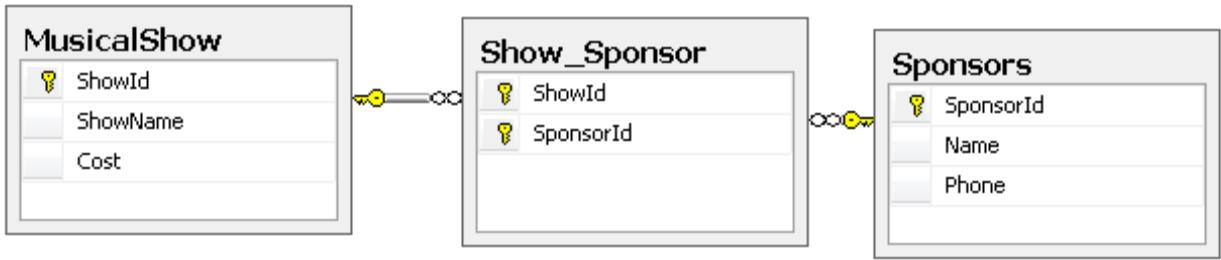
2.2 Many to Many Mapping

2.2.1 Many To Many Mapping Walkthrough

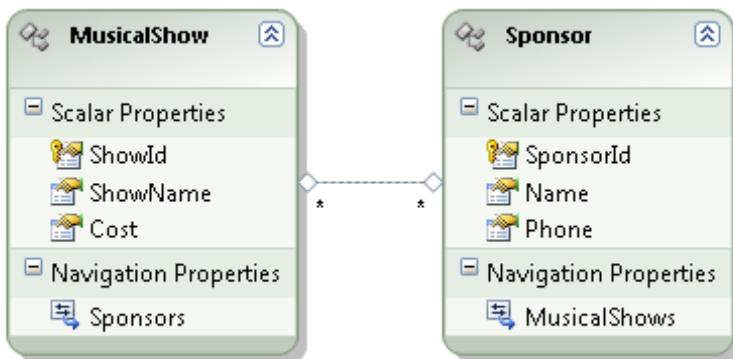
Problem: You have defined 3 tables in the database called MusicalShow, Sponsor and Show_Sponsor. A sponsor can contribute to many shows and a show can have many sponsors. You want to use the entity framework designer to map link table as a many to many relationship. You also want to know how to insert and delete entities relationship between two entities having many to many relationships.

Solution: To map a link table to a many to many relationship in entity framework, the link table cannot have payloads. If there is additional info that you are tracking about the relationship in the link table, entity framework has to create a third entity so you can access additional properties and also be able to insert and update those values for the link table. In our example, Show_Sponsor table does not contain additional properties other than primary keys from Sponsor and MusicalShow table. Therefore when we import the model using the update wizard, entity framework recognizes that link table does not contain any payloads and automatically removes the link table by representing relationship between MusicalShow and Sponsor as Many To Many.

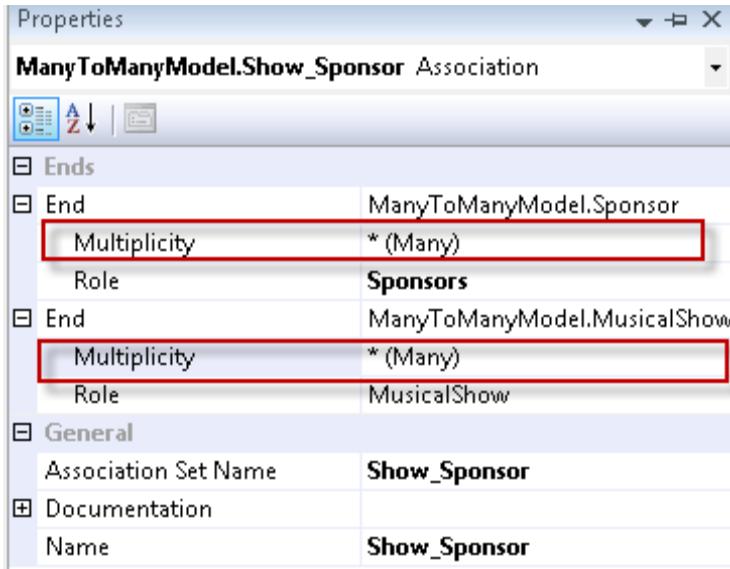
Discussion: To map a link table as many to many relationships, we have to create 3 tables in the database MusicalShow, Sponsor and Show_Sponsor. The link table Show_Sponsor cannot have any additional properties apart from the primary keys from Sponsor and MusicalShow table. If there are additional fields found in the link table, then entity framework designer will import the tables as 3 entities instead of two entities with many to many relationships. Screen shot below database diagram that shows the relationship between the tables.



On the above database diagram, Show_Sponsor acts as link table between MusicalShow and Sponsor table and to make sure that a show cannot have the same sponsor repeated twice, I have defined ShowId and SponsorId to be the primary key of the link table. When we import the tables using Update Model from database wizard, entity framework gets rid of the link table and shows MusicalShow and Sponsor to have many to many relationships. Screen shot below shows the many to many relationships between Show and Sponsor.



When we import the model, we get two entities MusicalShow and Sponsors. MusicalShow has a navigation property Sponsors that lets you access all the sponsors for a given MusicalShow. Similarly if you have sponsor entity, you can access all the musicalshows the sponsor has contributed to by using the MusicalShows navigation property. Both navigation properties are exposed as an EntityCollection. The association line between MusicalShow and Sponsor is a many to many which means both ends of the association have a multiplicity of Many. To see that clearly we can select the association line and look in the property window to see the End role for each side of the association set. Screen shot below shows the properties windows for many to many association set.



Now that we have modeled our entities, we can program against these entities to retrieve many side of relationship using eager loading or lazy loading. In the code below I am using Include and Load operator to load many side of relationship for sponsors.

```
var db = new ManyToManyEntities();
    var miller = db.Sponsors.First(s => s.Name ==
"Miller Lite");
        //lazy load the miller
        if (!miller.MusicalShows.IsLoaded)
        {
            miller.MusicalShows.Load();
        }
        Console.WriteLine("Lazy Loading");
        Console.WriteLine("Shows for miller lite " +
miller.MusicalShows.Count());

        //eager loading
        var db2 = new ManyToManyEntities();
        var miller2 =
db2.Sponsors.Include("MusicalShows").First(s => s.Name ==
"Miller Lite");

        Console.WriteLine("Eager Loading");
        Console.WriteLine("Shows for miller lite " +
miller2.MusicalShows.Count());
```

On the above code, to load the MusicalShows entity collection, I am calling Load on MusicalShows if it is not already loaded. This process is called lazy loading of entities. Similarly to eager load MusicalShows, I can use the include operator with Sponsor to indicate that when I bring sponsor, also retrieve MusicalShows for the sponsor. It is important to mention at this point that EF does not support relationship span for many to many relationship. Relationship span is a feature which allows EF framework to fix relationships of objects that are loaded separately. For instance if I load customers and orders separately and if the customers in the object context have an association to any of the orders tracked in ObjectContext, Ef framework will fix the relationship automatically and associate those orders to customer. With Many to Many relationships, retrieving the many side of the relationship can be an expensive operation as it requires an extra join to link table and is not take care by the framework. In the code below, I have retrieved the shows and the sponsor separately and when I access the shows for the sponsor, I get a return value of 0 which indicates that EF did not fix the navigation relationship for me although both sides of relationship were loaded in the objectcontext.

```
var db = new ManyToManyEntities();
        var miller = db.Sponsors.First(s => s.Name ==
"Miller Lite");
        var shows = (from show in db.MusicalShows
                    from sponsor in show.Sponsors
                    where sponsor.Name == "Miller Lite"
                    select show
                    ).ToList();
Console.WriteLine("using relationship span");
Console.WriteLine("shows for miller sponsor " +
miller.MusicalShows.Count());
```

To delete and add relationship between show and a sponsor we can make use of Add Delete method exposed on entity collection for both sides of the navigation. Example below shows how to add and delete MusicalShows from a sponsor.

```
var db = new ManyToManyEntities();
        var show1 = new MusicalShow { ShowName = "Johnny and
the Sprites" };
```

```

        var show2 = new MusicalShow { ShowName = "Sesame
Street" };

        var miller = db.Sponsors.First(s => s.Name ==
"Miller Lite");
        //add show1 and show2 to miller
        miller.MusicalShows.Add(show1);
        miller.MusicalShows.Add(show2);

        db.SaveChanges();
        //to delete show relationship with miller
        //sponsor simply remove it from collection
        miller.MusicalShows.Remove(show1);

        //deleting the show automatically removes the
relationship
        //from the sponsor.
        db.DeleteObject(show2);
        db.SaveChanges();
    
```

On the above code, I am creating two shows and adding it to MusicalShows navigation property of miller sponsor followed by SaveChanges. To remove a show from a sponsor, I am using Remove method on MusicalShow navigation property. When I call Remove I am simply removing the relationship between the show and sponsor, it does not actually remove the show from the database. Another way to remove a relationship is by deleting the entity itself. Like in the above example, I am deleting show2 and since show2 has relationship with Miller sponsor, EF framework understands that it does not make sense to have a relation with an entity that is marked for deletion. Therefore when SaveChanges is called, EF also issues a delete command to delete the relationship. EF will only issue the command to delete a relationship when the relationship is loaded and tracked by the ObjectContext. If the relationship between sponsor and show is not tracked by Object Context, deleting show would cause database to throw foreign key violation constraints because it will be still tied to a sponsor.

As discussed earlier, if the link table carries the responsibility of capturing data specific to the relationship, entity framework wouldn't be able to transform relationship into many to many associations. This is because associations currently do not support properties other than having roles on

both ends of an association. This is something which EF team is considering in the next release of Entity framework.

2.2.2 Retrieving Link table for Many to Many Relation

Problem: You have Show and Sponsor entity that are defined using Many to Many relationship. To access Shows for a sponsor, we use navigation relationship. However you want to just access the relationship between Shows and Sponsor without bringing additional columns available on both Show and the Sponsor. The problem is link table which contains the relationship is not exposed by the entity framework. You want to know how to retrieve only relationship between shows and sponsors without the load of additional columns.

Solution: When you define a link table between two tables, entity framework by default represents the relationship as many to many. This makes it easy to retrieve both sides of the relationship. However if we just care of about the relationship without additional columns, there is no direct support from entity framework because link table is not exposed. With a simple linq query containing nested from clause, you can retrieve only the relationship information and return the results as anonymous class.

Discussion: To retrieve the show and the sponsor without any hierarchy, we have to flatten the list. You can accomplish this either by using SelectMany operator in a method syntax or use nested from clauses to flatten the hierarchy. Code below shows two different ways of retrieving only the relationship between Show and Sponsor without extra columns.

```
var db = new ManyToManyEntities();
    var showsponsor = from sponsor in db.Sponsors
                      from show in sponsor.MusicalShows
                      select new { show.ShowId,
sponsor.SponsorId };

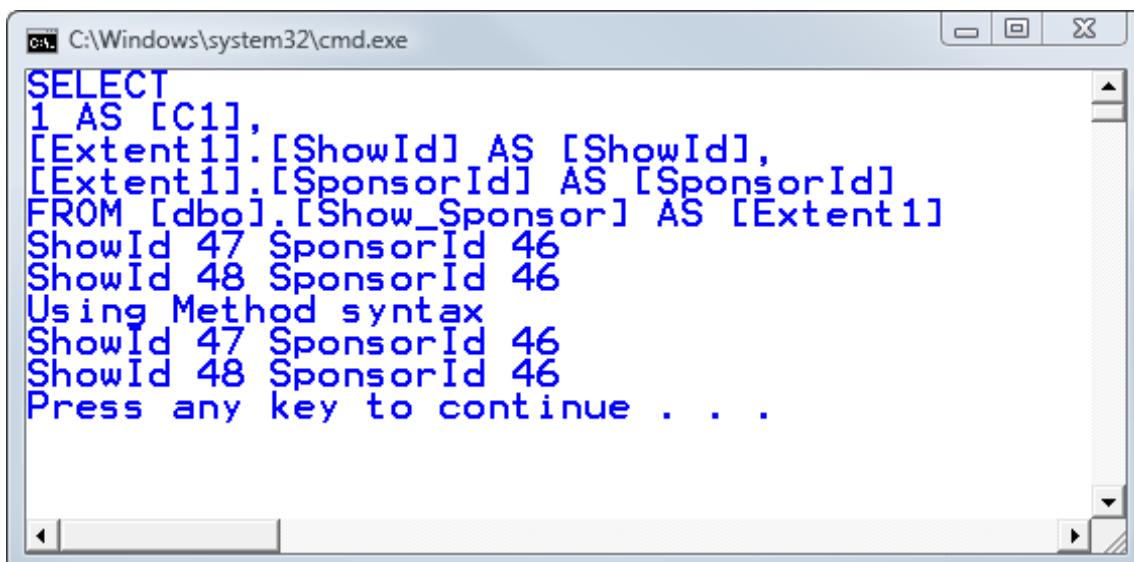
Console.WriteLine(showsponsor.AsObjectQuery().ToTraceString());
    showsponsor.ToList().ForEach(
        sp =>
            Console.WriteLine("ShowId {0} SponsorId
{1}", sp.ShowId, sp.SponsorId));
```

```

//using method syntax
var showsp = db.Sponsors.
    SelectMany(s => s.MusicalShows,
        (sponsor, show) => new {
    sponsor.SponsorId, show.ShowId });
    Console.WriteLine("Using Method syntax");
    showsp.ToList().ForEach(
        sp =>
            Console.WriteLine("ShowId {0} SponsorId
{1}", sp.ShowId, sp.SponsorId));

```

On the above code, I am only retrieving the ShowId and SponsorId for every relationship. First option uses the query syntax and second option uses method syntax by making use of SelectMany operator. To confirm that query only returns the relationship column, I am also printing the sql query send to the database. Screen shot below shows the results of along with the query sent to the database.



The screenshot shows a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The window displays the following text:

```

SELECT
1 AS [C1],
[Extent1].[ShowId] AS [ShowId],
[Extent1].[SponsorId] AS [SponsorId]
FROM [dbo].[Show_Sponsor] AS [Extent1]
ShowId 47 SponsorId 46
ShowId 48 SponsorId 46
Using Method syntax
ShowId 47 SponsorId 46
ShowId 48 SponsorId 46
Press any key to continue . . .

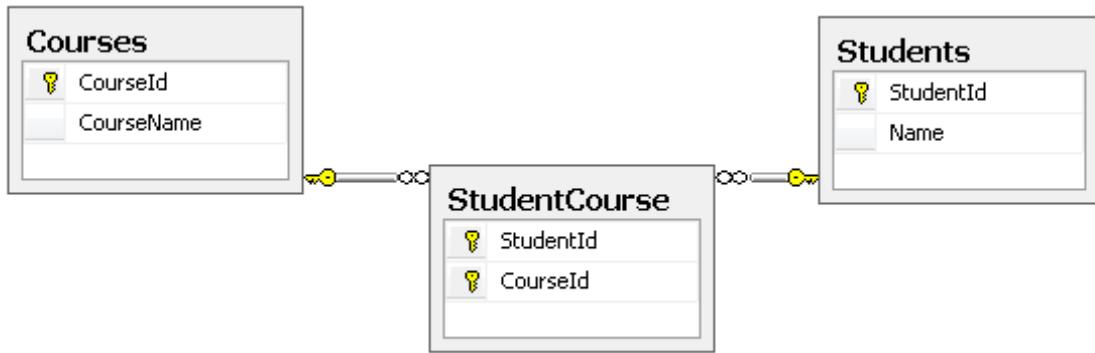
```

2.2.3 Implementing Many to Many relationship as two 1 to Many relationships

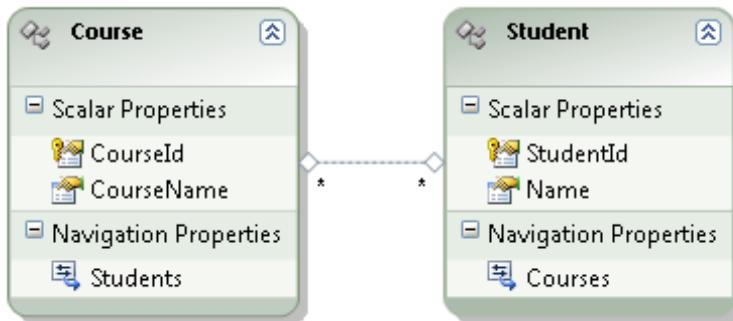
Problem: You have defined Student, Course and StudentCourse link table in the database. Student can be in many courses and a course can have many students. Currently relationship table does not have any payloads but you want to make sure that as the application grows and there is a need for having additional columns to track about the relationship, you do not have to change your entity data model and all the code against it. When you import the table into EDMX, course and student entity are related by Many to Many relationship. You want to change the representation into two 1 to many relationship so in future when the link table needs an additional column, the conceptual schema can accommodate that.

Solution: By default when you import two tables joined by a link table that does not have any payloads, entity framework will remove the link table and create many to many association between the entities. If you want to keep the link table intact because later down you will have additional columns that pertain to the relationship, you can edit the model generated by the designer and introduce the link table. Although with the introduction of link table, the query would get complicated as now you would have to travel the link table to get to the many side of the entity. Customizing the model generated by the designer is fully supported.

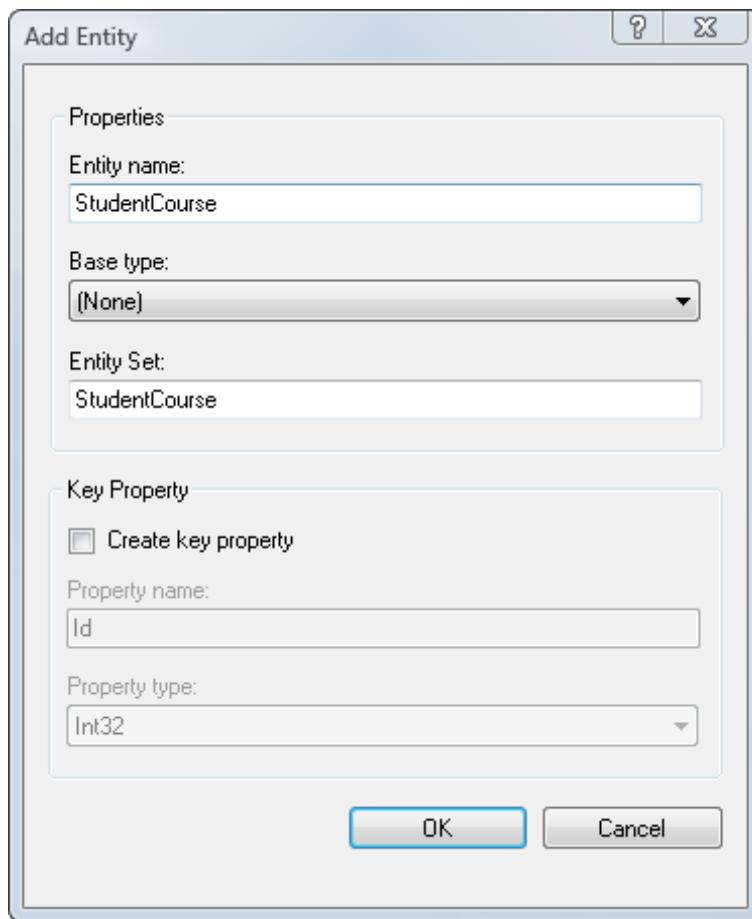
Discussion: In this walk through we will go through the steps of how to configure a many to many relationship into two 1 to many relationship by customizing the model after it is generated. To get started we will create three tables Student, Course and StudentCourse. StudentCourse will be our link table that defines what students will be enrolled in a given course. Screen shot below shows database diagram for tables and how they are related to each other.



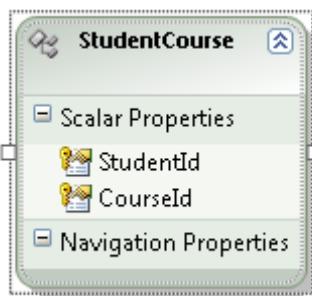
After creating our tables we can import the tables into EDM. When we import the model, entity framework gets rid of the link table and creates an association between Student and Courses with both ends having a multiplicity of Many. Screen shot below shows how the model is currently represented.



To change the model, we will first delete the many to many relationship link and a new entity called StudentCourse that has 2 properties CourseId and StudentId mark both of them as entity key. You can either create a property by right clicking StudentCourse entity and choosing Add a Scalar Property. Since I already have both these entities available on Course and Student entity, I will go ahead and copy the properties and paste it on my StudentCourse entity. Screen shot below shows how StudentCourse entity look like.

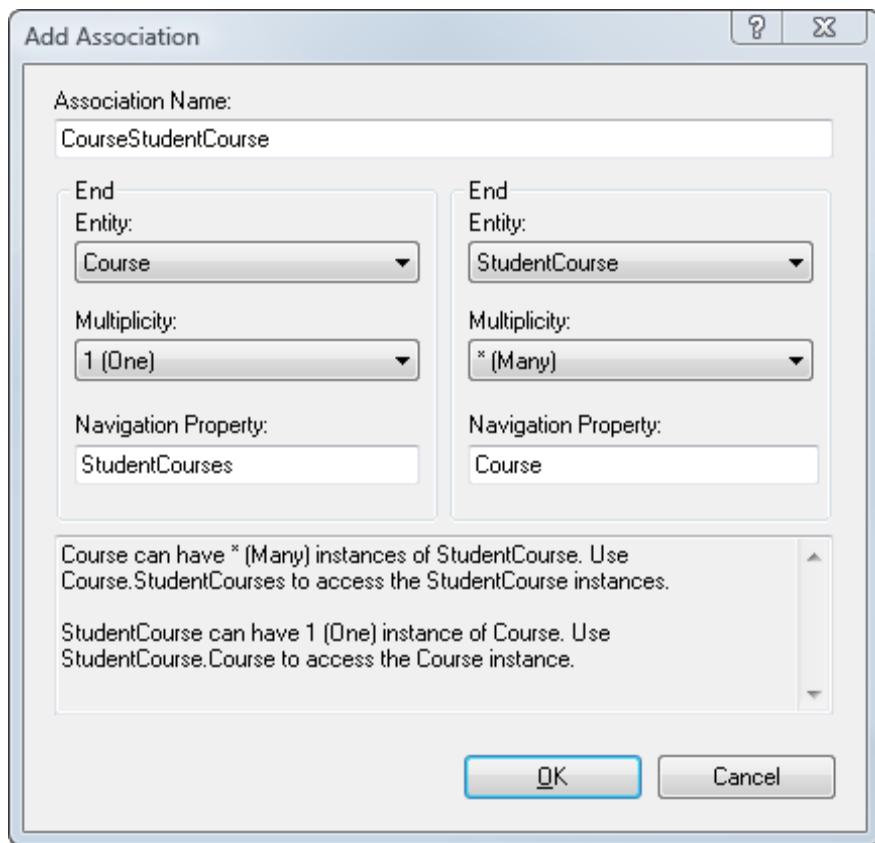


Creating Student Course entity.



StudentCourse entity with StudentId and CourseId as EntityKey

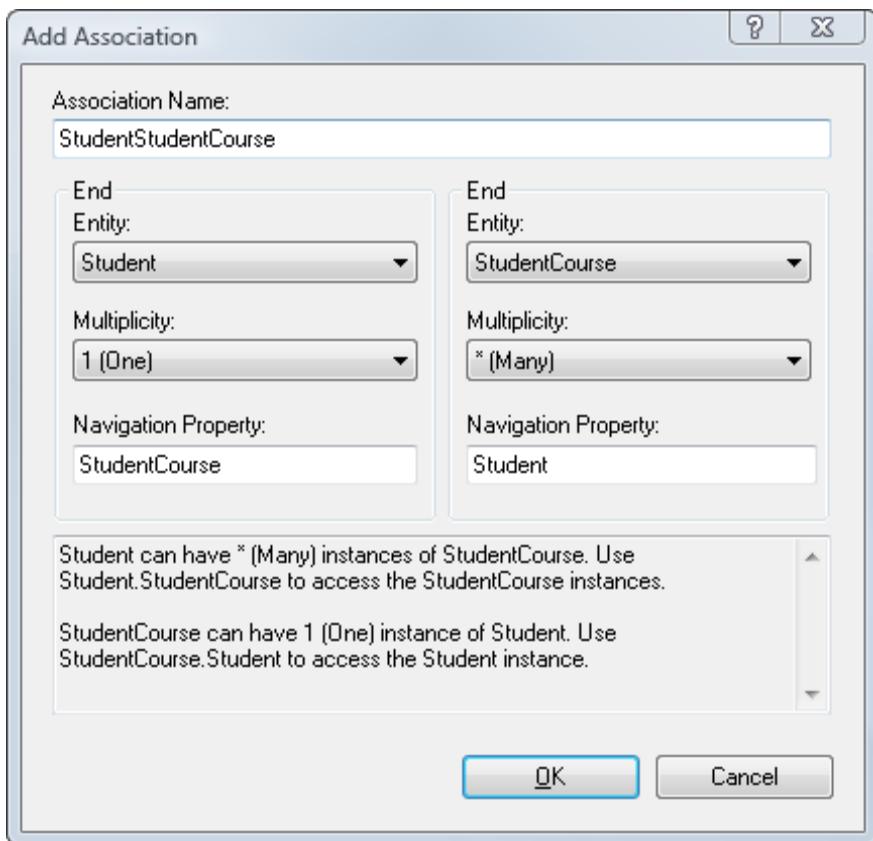
The next step is to create associations between Course and StudentCourse. To create the association, right click Course entity, select Add and choose Association. On the association window that opens up we will define how StudentCourse and Course are related to each other. Screen shot below shows how I have configured both sides of the association.



On the example above, one side of my association has an entity type of Course with multiplicity of 1. This means that a single course can have many StudentCourses that can be accessed by our navigation relationship called StuentCourses. The other side of association contains StudentCourse which has multiplicity of many as 1 student can be enrolled in Many courses. To navigate to Course entity from Stduent entity, we are exposing Course navigation property. So far we have only configured the conceptual model, we need to map the association to the relation defined on the database. To map the association right click the association and select Table mapping. Since the association is between StudentCourse and Course, we will select StudentCourse as the table to map. When we select StudentCourse table, EDM designer automatically picks up the entity keys for both sides of the association and assigns it to the association set defined on the store model. Screen shot below shows how we have configured the StudentCourse and Course association.

Property	Operator	Column
Association		
Maps to StudentCourse		
Course		
CourseId : Int32	↔	CourseId : int
StudentCourse		
StudentId : Int32	↔	StudentId : int
CourseId : Int32	↔	CourseId : int

Next we need to create an association between StudentCourse and Student. To do that we will right click on student select Add and choose Association. For one side of the association we will select with a multiplicity of 1. On student we will expose StudentCourses navigation property to access StudentCourses entity. Similarly StudentCourses will be the Many side of the relationship having multiplicity of Many and will use Student navigation property to access Student instance. Screen shot below show how we have configured the association between StudentCourse and Student entity.



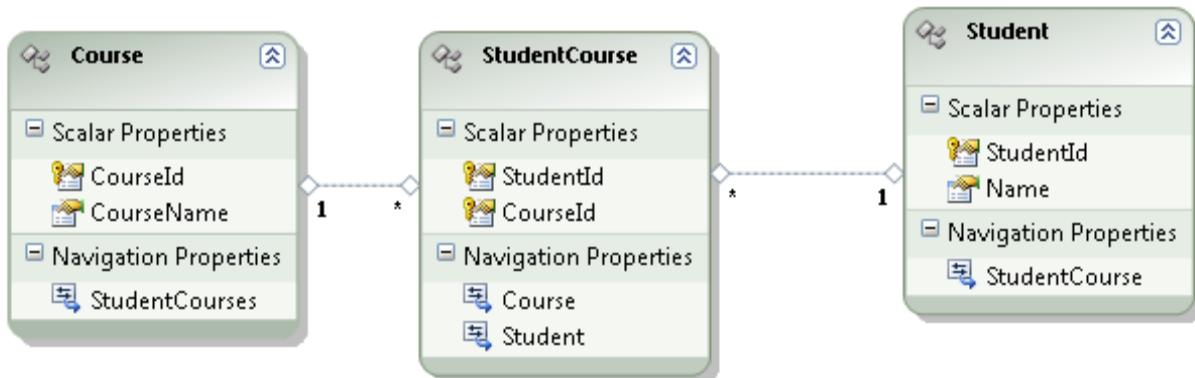
After creating the association we have to map the association to the association defined on the store model. To complete this process we will right click the association we created and choose table mapping. On the table mapping window, select StudentCourse table and designer will pre fill the mapping with entity keys from both sides of the association as shown below in the screen shot.

We are not done because we need to also map the new entity StudentCourse that we created earlier. To map StudentCourse entity to a table, right click the entity and select table mapping. On the table mapping window choose StudentCourse table and the designer will map the properties on StudentCourse entity with StudentCourse table on the database. Screen shot below shows the StudentCourse entity mapping.

Mapping Details - StudentCourse

Column	Operator	Value / Property
Tables		
Maps to StudentCourse		
<Add a Condition>		
Column Mappings		
StudentId : int	↔	StudentId : Int32
CourseId : int	↔	CourseId : Int32
<Add a Table or View>		

Now if you try to validate the model, you should not get any errors. Screen shot below shows the completed EDM model transformed from many to many to two 1 to many association.

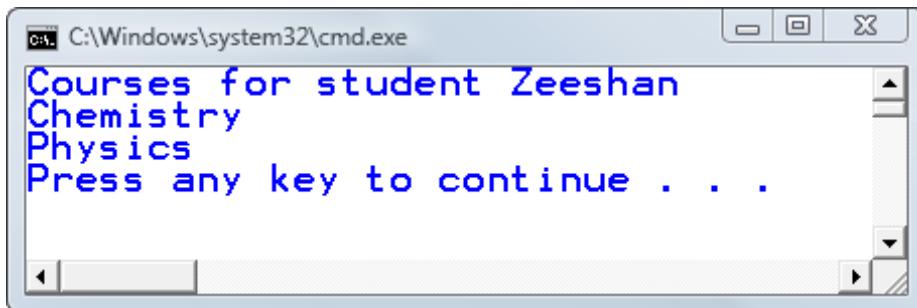


To query against the model would be little extra steps because to get to a course for a student we have to travel an additional entity `StudentCourse` and access its `Course` property to access course instance. Example below shows how to access courses for a student.

```

var db = new ManyToManyEntities();
var st =
db.Students.Include("StudentCourse.Course").First(s => s.Name ==
"Zeeshan");
foreach (var stcourse in st.StudentCourse)
{
    Console.WriteLine(stcourse.Course.CourseName);
}
  
```

On the above code, I retrieve student Zeeshan and since I will be accessing all its relationship, I am loading all these relations ahead of time by using Include operator. Include method allows me to specify a path which can be any level deep. After getting reference to student instance, I loop through all the StudentCourses and for each StudentCourse I access its Course property and print the Name of the course the student is enrolled in. Screen shot below shows the output on the console window.



```
C:\Windows\system32\cmd.exe
Courses for student Zeeshan
Chemistry
Physics
Press any key to continue . . .
```

2.2.4 Modeling two 1 to many relationship as Many to Many relationship

Problem: You have defined 3 tables in the database; Club, Members and Membership. A club can have many members and a member can be part of many clubs. This relationship is stored in Membership table. Membership table carries a payload column MemberDues that stores how much Dues a member has towards a club. Because Membership table contains additional columns apart from the primary keys from Club and Members table, entity framework models the relationship as two 1 to Many associations. This is good for inserts and updates but for querying purpose to know what clubs a member is associated with, extra joins with MemberShip table, makes the query harder to read and adds noise to the code. You want your entity data model to support Many to Many relationship in addition to two 1 to many relationship. For querying purpose you want to leverage the Many to Many relationships but Inserts and Updates can still be processed using a separate Membership link table.

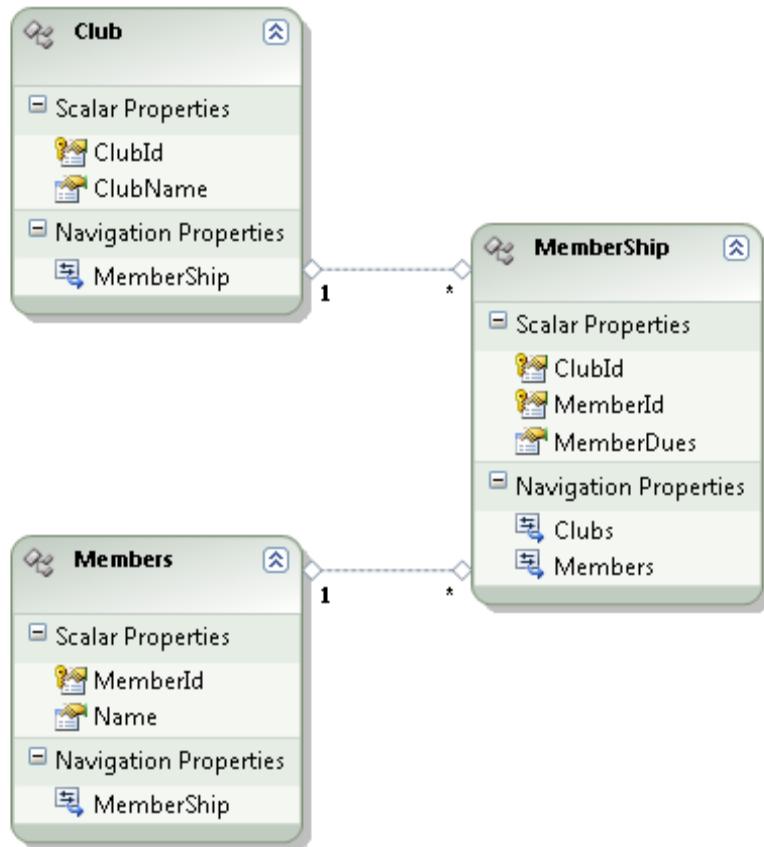
Solution: When we import Club and Members table, entity framework does not get rid of the link table because it has additional columns that are required for the relationship and cannot be specified on insert and update if link entity

did not exists. For querying purpose, we can also create an additional association that has Many To Many relationship between Club and Members that is void of additional columns. To accomplish that we have to manually modify the SSDL file as this is not supported in the designer. In the ssdl model, we have to create an EntityType that contains only the primary keys from Club and Members entity. In addition we have to define an entityset that will use a DefiningQuery to only retrieve the columns we mentioned in the entity type earlier and also map the results of the query to the EntityType. After creating EntityType and EntitySet, we can go into the designer and create a Many To Many association between Clubs and Members and map the association to the entityset we created in the ssdl.

Discussion: In this walk through we will define a Many to Many association between Club and Member in addition to two 1 to many associations created by the designer when we import the tables into our model. To get started we will create 3 tables Clubs, Members and Membership table. Screen shot below shows the database diagram after creating the tables and relationship between them.



After creating the tables, we will import the model into EDM by using Update Model from the database. Screen shot below shows the entity data model created by EF when we import the tables.



The next step is to create a new entity type called ClubMember with only primary keys from Club and MemberShip table which is ClubId and MemberShipId. The entity would be part of an entityset which we will call ClubMembers and data for the entityset would come from a defining query. Now you must be wondering why we cannot simply create an association between club and members and map the association to existing Membership table we imported earlier. The reason is because you cannot map two entities to a single row in a table. Since we already have mapping for a record in membership table that was created when the model got imported as two 1 to many relationship, we cannot reuse that. When you try to map the Many to Many associations to the same Membership table, edmx validation would raise the following error.

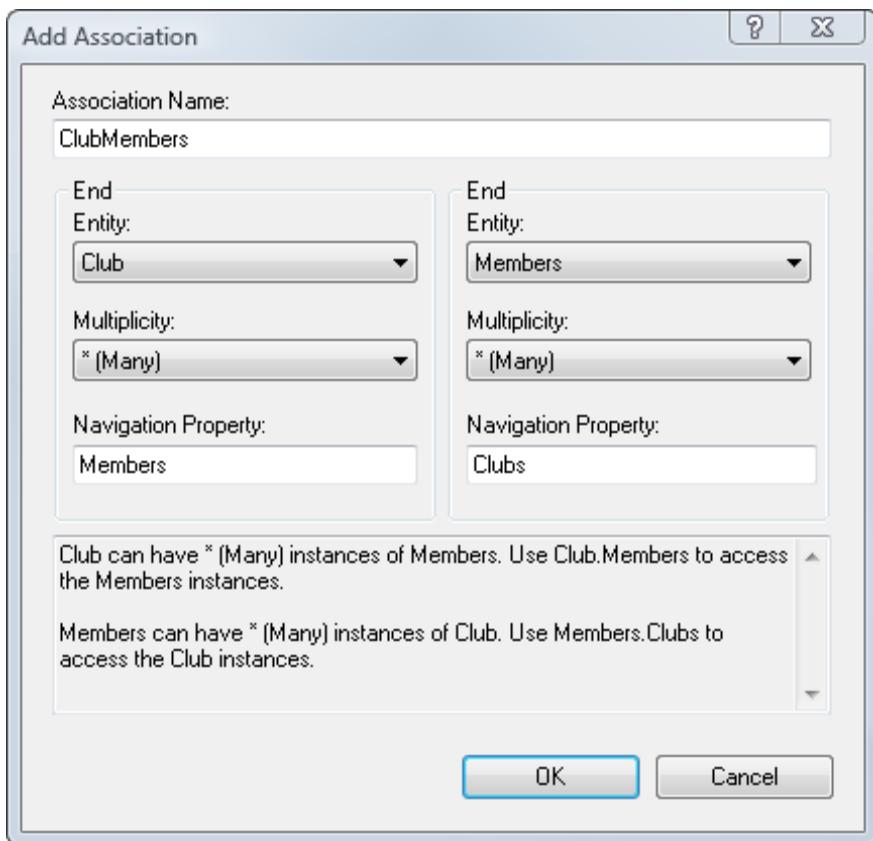
Error 3034: Problem in Mapping Fragments. Two entities with possibly different keys are mapped to the same row.

SSDL below shows the EntityType and EntitySet created that will later be used in mapping Many To Many associations between club and Members.

```
<EntitySet Name="ClubMember" EntityType="Self.ClubMember">
    <DefiningQuery>
        SELECT ClubId, MemberId from
        Membership
    </DefiningQuery>
</EntitySet>

<EntityType Name="ClubMember">
    <Key>
        <PropertyRef Name="ClubId" />
        <PropertyRef Name="MemberId" />
    </Key>
    <Property Name="ClubId" Type="int"
    Nullable="false" />
    <Property Name="MemberId" Type="int"
    Nullable="false" />
</EntityType>
```

On ssdl above, ClubMember only contains entity keys from Club and Members table. The results of ClubMember entity set comes from a defining query which only retrieves ClubId and MembershipId columns and is mapped to ClubMember entity type. Since we are editing the SSDL model manually, it is important to know that if you try to update the model again, you will lose all your changes as the ssdl model gets overwritten. In the next release of EF, the update model wizard would preserve changes made to ssdl model even after updating the model from the database. After creating entity type and entityset we can go back to the designer and add Many to Many association between Club and Member by right clicking Club entity, selecting Add and choosing association. Screen shot shown below captures the association we have created.

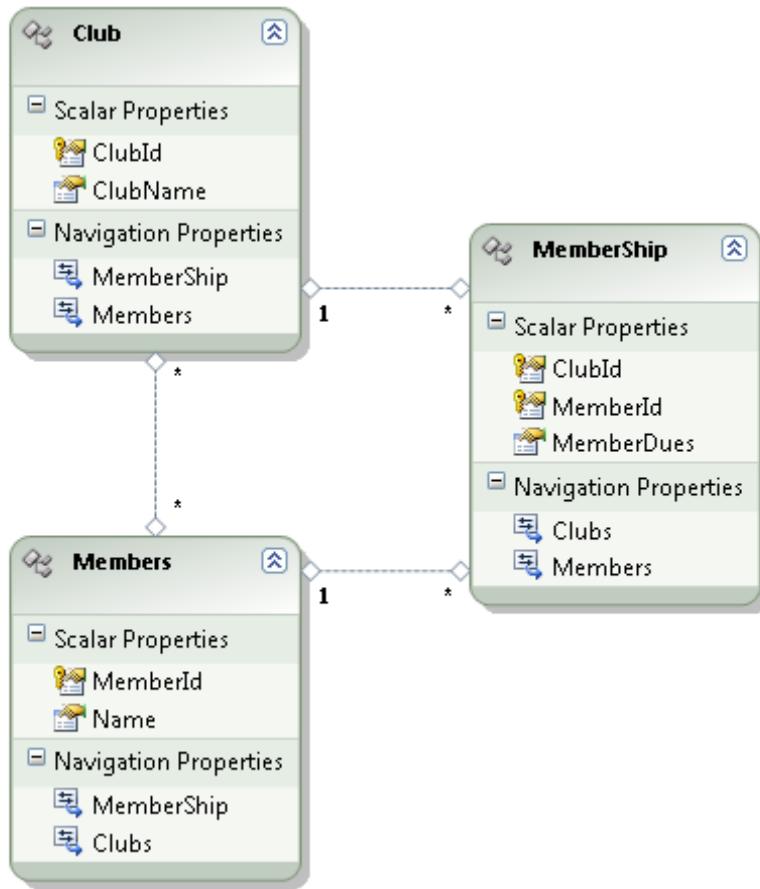


The above association has Club and Members as both sides of the association with a multiplicity of Many. To access the Members for Club from Club entity, we will use Members navigation property. Similarly to access Clubs collection from member, we will use Clubs navigation property. Next step is to map the association to the virtual table we created earlier on the stored. To do that, right click the Many to Many associations we created earlier and select table mapping. When you select ClubMember table, the designer auto populates the fields with entity keys defined on both ends of the association.

Screen shot below shows how we mapped many to many associations.

Property	Operator	Column
Association		
Maps to ClubMember		
Club		
ClubId : Int32	↔	ClubId : int
Members		
MemberId : Int32	↔	MemberId : int

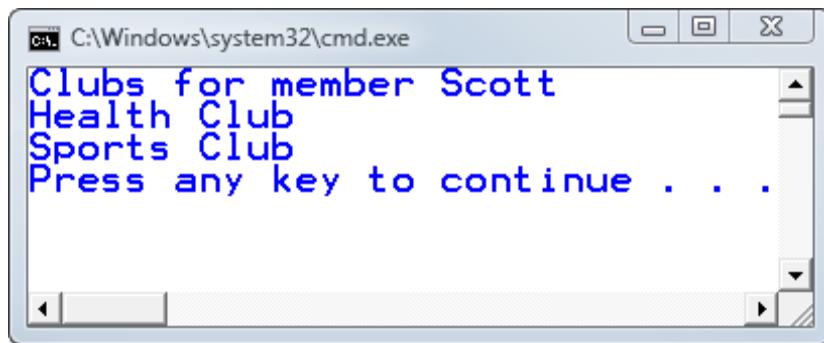
Screen shot below shows the completed model with 2 to 1 many associations and 1 read only Many to Many association between club and member.



To confirm the mapping is configured correctly we can write a simple query that accesses clubs a member is associated with without using the Membership intermediary table. Codes below uses include to eagerly load

Clubs for a member. Using the Clubs property, I loop through the club collection and print the club name to the output window. Screen shot below shows the output to console window.

```
var db = new ManyToManyEntities();
    var scott = db.Members.Include("Clubs").First(m =>
m.Name == "Scott");
    foreach (var club in scott.Clubs)
{
    Console.WriteLine(club.ClubName);
}
```



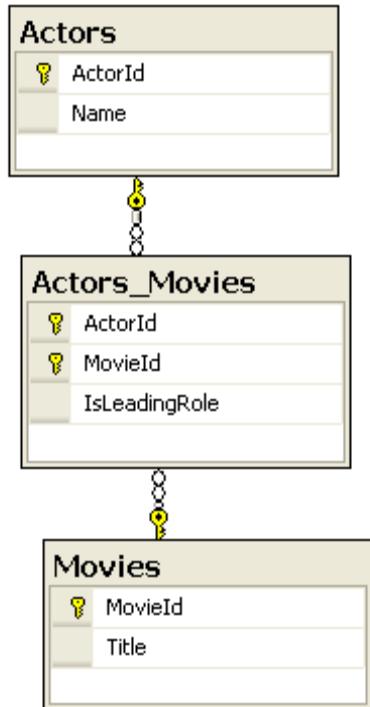
Exposing Many To Many Relationship as EntityReference

2.2.5 Mapping Many to Many table as 2 Many to Many Associations

Problem: You have 3 tables in the database called Actors, Movies and ActorMovie link table. An actor can be part of many movies and a movie can have many actors. The link table also contains an additional column IsLeadingRole which determines if an actor is playing a leading role in the movie or not. You want to represent the Many To Many table as two types of associations in EDM. An Actor should have two types of Navigation to Movie entity; MoviesWithLeadingRole and MoviesWithSupportingArtist. MoviesWithLeadingRole will return Movies where the actor has a leading role. MoviesWithSupportingArtist will return Movies where the actor is a supporting actor.

Solution: To map the link table as two different Entityset, we need to create two views, one view will return ActorMovie table where IsLeadingRole is 1 and other view will return ActorMovie table where IsLeadingRole is 0. Next step is to import the views, Actor and Movie table into entity data model and then create two many to many association between actor and Movies. One association would map to a view where IsLeadingRole equal to 1 and other association would map to view where IsLeadingRole is 1. Since views does not support inserts and updates, we will have to create separate stored procedure which will be responsible for inserting into our link table.

Discussion: First step is creating the 3 tables Actor, Movies and ActorMovie link table. For Actor table define ActorId as the primary key. For Movies table, define MovieId as the primary key and ActorMovie link table will have ActorId and MovieId as the primary key. In addition ActorMovie link table will also contain IsLeadingRole bit field that tells if the actor has a leading role in the movie or not. Database diagram below shows how the relationship between tables looks like.



Since we want to maintain two different types of movie collection for an actor; one with leading role and other with supporting roles, we need to create two views that return appropriate data from Actor_Movies table. Views shown below achieve our requirement.

```
create view dbo.MoviesWithLeadingActor
as
select ActorId,MovieId from Actors_Movies where IsLeadingRole = 1

create view dbo.MoviesWithSupportingActor
as
select ActorId,MovieId from Actors_Movies where IsLeadingRole = 0
```

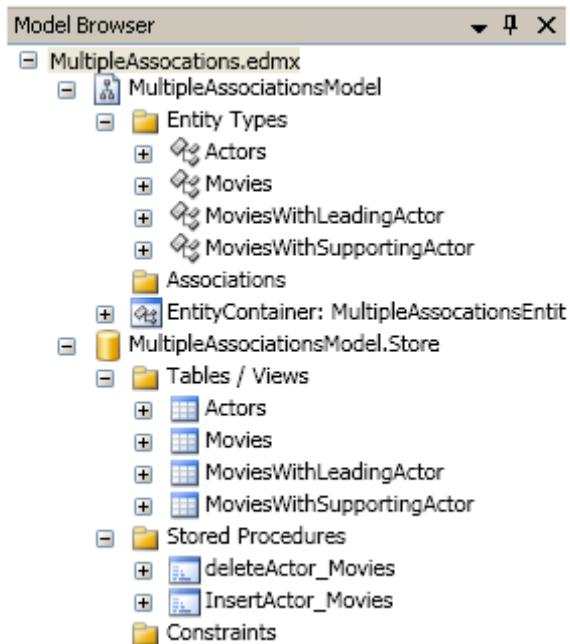
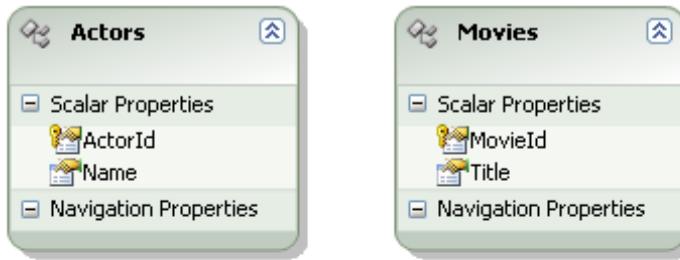
Later we will create two many to many associations on the EDM and map them to our view. Since views do not support insert and deletes, we need to create two stored procedures that will be responsible for inserting and deleting into the Actor_Movies link table. Stored procedures below serve our need.

```
create proc dbo.InsertActor_Movies
(@ActorId int,@MovieId int,@IsLeadingRole bit)
as
begin
insert into Actors_Movies values (@ActorId,@MovieId,@IsLeadingRole)
end

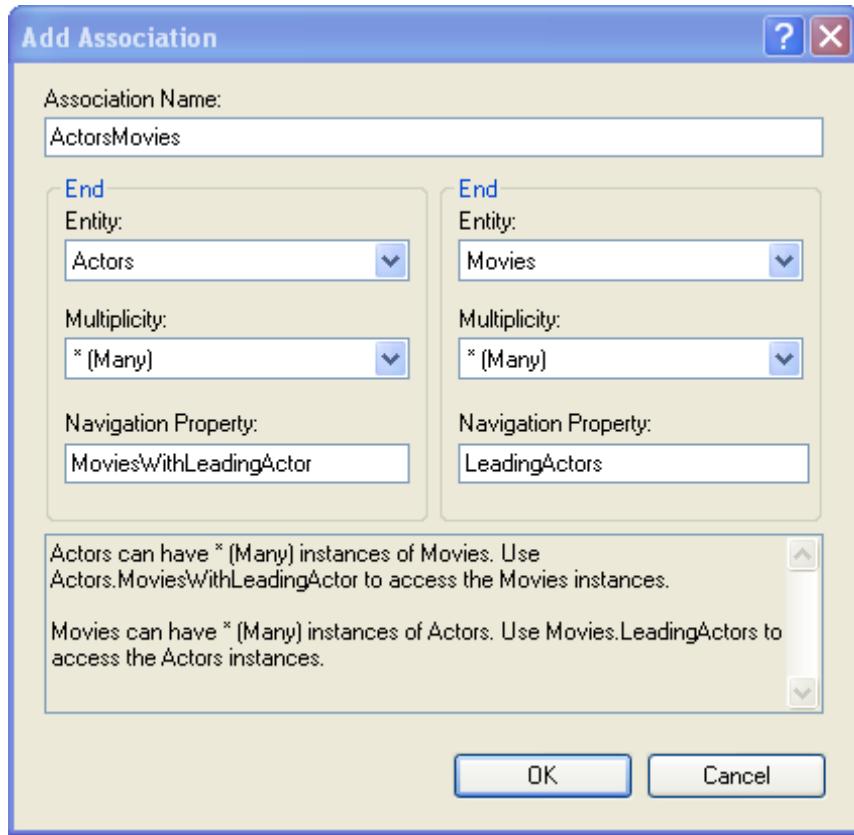
create proc dbo.deleteActor_Movies
(@ActorId int,@MovieId int)
as
begin
delete Actors_Movies where ActorId =@ActorId and MovieId =@MovieId
end
```

Since Many To Many table does not have a concept of Update, we only need stored procedure for Insert and Update cases. The insert stored procedure takes ActorId, MovieId and IsLeadingRole to insert into Actor_Movies table. To delete an entry from Actor_Movies table we are passing ActorId and MovieId since they are primary key columns in the link table.

The next step is to import Actor and Movies table, our two views, and two stored procedure we created for inserts and delete to link table. Screen shots below shows how EDM model and model browser window looks like after importing objects from the database.



First step is to delete the two entities `MoviesWithLeadingActor` and `MoviesWithSupportingActor` which are based on views. Next step is to create Many to Many association between Actor and Movies. This association will represent a collection containing only movies where the actor has leading role. To perform this operation, right click on Actors and Add Association. You will get an association dialog. Screen shot below shows the values we filled for the association.



On the Add Association dialog above, I have indicated that both ends of the association will have a multiplicity of Many. To access Movies collection from actor we will use MoviesWithLeadingActor navigation property. This is because we will map this association to MoviesWithLeadingActor view. After clicking okay on the dialog, you model should look like this.

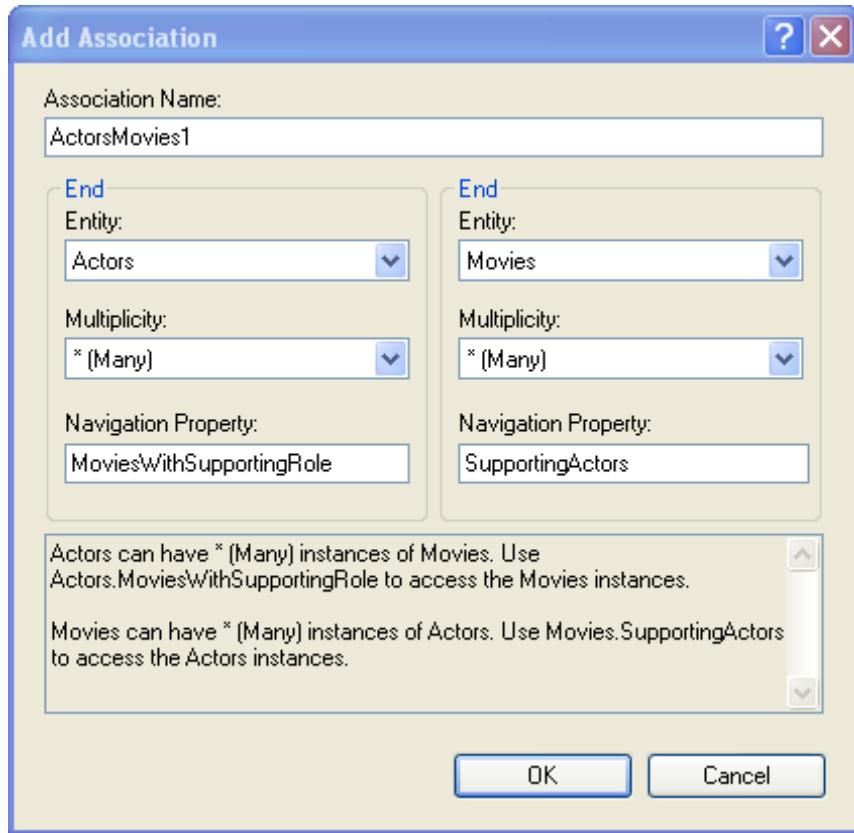


To map the association we created, right clicking on the association line and select table mapping. On the table mapping select MoviesWithLeadingActor view and the designer would map the columns to properties on the association automatically.

Mapping Details - ActorsMovies

Property	Operator	Column
Association		
Maps to MoviesWithLeadingActor		
Actors	ActorId : Int32	ActorId : int
Movies	Moviefld : Int32	Moviefld : int

Similarly to create our second association right click on actors, select add association and make both ends of the association as Many To Many and name the navigation property that goes from Actor to Movies as MoviesWithSupportingActor. Screen shot below shows the values we filled for the association.



To map the association, right click the association line and select table mapping. Map the association to `MoviesWithSupportingActor` view as shown below.

Property	Operator	Column
Association		
Maps to MoviesWithSupportingActor		
Actors		
ActorId : Int32	↔	ActorId : int
Movies		
MovieId : Int32	↔	MovieId : int

If you try to validate the model, you will not get any errors. However when we try to add items to the two collections created earlier, we will get an exception because the associations are mapped to views and views are not updatable. We need to map the associations to Insert and Delete stored procedures imported earlier. Since mapping associations to stored procedure

is not supported by the designer, we have to manually edit the msdl and specify insert and delete stored procedure. If we look at the insert stored procedure created earlier, it has an additional parameter called IsLeadingRole for which we cannot specify any mapping as it is not returned from our view definition. To be able to use the same stored procedure for inserting either ActorsWithLeadingRoles and ActorsWithSupportingRoles associations, we need to leverage the CommandText property of the function definition on the sddl and specify default values for IsLeadingRole depending on the association being inserted. Example below shows the updated version of the function that inserts LeadingActors and SupportingActors relationship.

```

<Function Name="InsertLeadingActor_Movies" Aggregate="false"
BuiltIn="false" NiladicFunction="false" IsComposable="false"
ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo">
    <CommandText>
        exec dbo.InsertActor_Movies @ActorId = @ActorId,@MovieId =
@MovieId,@IsLeadingRole = 1
    </CommandText>
    <Parameter Name="ActorId" Type="int" Mode="In" />
    <Parameter Name="MovieId" Type="int" Mode="In" />
</Function>
<Function Name="InsertSupporingActor_Movies" Aggregate="false"
BuiltIn="false" NiladicFunction="false" IsComposable="false"
ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo">
    <CommandText>
        exec dbo.InsertActor_Movies @ActorId = @ActorId,@MovieId =
@MovieId,@IsLeadingRole = 0
    </CommandText>
    <Parameter Name="ActorId" Type="int" Mode="In" />
    <Parameter Name="MovieId" Type="int" Mode="In" />
</Function>

```

On the above function, we got rid of the additional parameter because we do not have any mapping for IsLeadingRole. Instead we created two functions that basically call the same stored procedure in the database with different default values for IsLeadingRole.

We then need to map the stored procedures to insert and delete function of associationSetMapping for both associations.

We need to map our association to Inserts and Delete stored procedure modified above. Mapping stored procedure to associations is currently not supported by the designer, so we will have to go into msdl and manually

specify the Insert and Delete functions for both the associations. Msdl below shows the ModificationFunctionMapping section for both associations.

Notice for LeadingActor association, we are calling

InsertLeadingActor_Movies stored procedure and for SupportingActor association, we are calling InsertSupporingActor_Movies stored procedure.

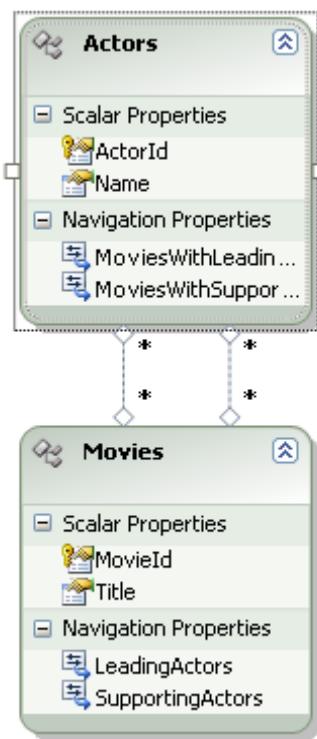
```
<AssociationSetMapping Name="ActorsMovies"
  TypeName="MultipleAssociationsModel.ActorsMovies"
  StoreEntitySet="MoviesWithLeadingActor">
  <EndProperty Name="Movies">
    <ScalarProperty Name="MovieId" ColumnName="MovieId" />
/></EndProperty>
  <EndProperty Name="Actors">
    <ScalarProperty Name="ActorId" ColumnName="ActorId" />
/></EndProperty>
  <ModificationFunctionMapping>
    <InsertFunction
      FunctionName="MultipleAssociationsModel.Store.InsertLeadingActor_Movies" >
      <EndProperty Name="Movies">
        <ScalarProperty Name="MovieId" ParameterName="MovieId" />
      </EndProperty>
      <EndProperty Name="Actors">
        <ScalarProperty Name="ActorId" ParameterName="ActorId" />
      </EndProperty>
    </InsertFunction>
    <DeleteFunction
      FunctionName="MultipleAssociationsModel.Store.deleteActor_Movies">
      <EndProperty Name="Movies">
        <ScalarProperty Name="MovieId" ParameterName="MovieId" />
      </EndProperty>
      <EndProperty Name="Actors">
        <ScalarProperty Name="ActorId" ParameterName="ActorId" />
      </EndProperty>
    </DeleteFunction>
  </ModificationFunctionMapping>
</AssociationSetMapping>
<AssociationSetMapping Name="ActorsMovies1"
  TypeName="MultipleAssociationsModel.ActorsMovies1"
  StoreEntitySet="MoviesWithSupportingActor">
  <EndProperty Name="Movies">
    <ScalarProperty Name="MovieId" ColumnName="MovieId" />
/></EndProperty>
  <EndProperty Name="Actors">
    <ScalarProperty Name="ActorId" ColumnName="ActorId" />
/></EndProperty>
  <ModificationFunctionMapping>
    <InsertFunction
      FunctionName="MultipleAssociationsModel.Store.InsertSupporingActor_Movies" >
      <EndProperty Name="Movies">
        <ScalarProperty Name="MovieId" ParameterName="MovieId" />
      </EndProperty>
      <EndProperty Name="Actors">
        <ScalarProperty Name="ActorId" ParameterName="ActorId" />
      </EndProperty>
```

```

        </InsertFunction>
        <DeleteFunction
    FunctionName="MultipleAssociationsModel.Store.deleteActor_Movies">
        <EndProperty Name="Movies">
            <ScalarProperty Name="MovieId" ParameterName="MovieId" />
        </EndProperty>
        <EndProperty Name="Actors">
            <ScalarProperty Name="ActorId" ParameterName="ActorId" />
        </EndProperty>
    </DeleteFunction>
</ModificationFunctionMapping>

```

This completes our modeling of Many to much relationship as two Many To Many associations. The final entity entity data model looks like this



Now we can program against the model by adding the movie to right collection depending on if the actor participated in a leading role or supporting role. When we retrieve the movies, both collections would have their correct types of movies filled. Code below creates three movies and adds two movies to MoviesWithLeadingActor collection and one movie to MoviesWithSupportingActor. Then using a different datacontext, I am retrieving the collection count to ensure that I have two movies with LeadingActor and 1 one movie with SupportingActor. When I am retrieving

the actor, I am using Include operator to load both collections ahead of time rather than call load separately on each collection to do lazy loading.

```
var db = new MultipleAssociationsEntities();
    var george = new Actors { Name = "George Clooney" };

    var Syriana = new Movies { Title = "Syriana" };
    var BabyTalk = new Movies { Title = "Baby Talk" };
    var Roseanne = new Movies { Title = "Roseanne" };

    db.AddToMovies(Syriana);
    db.AddToMovies(BabyTalk);
    db.AddToMovies(Roseanne);

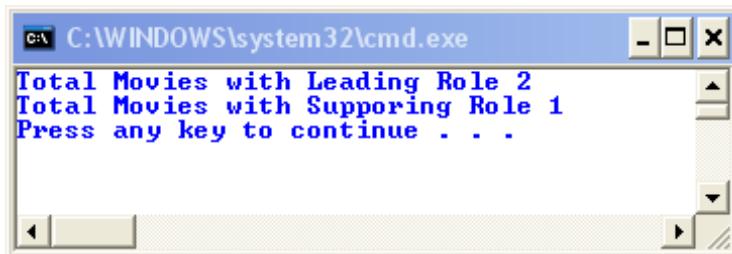
    george.MoviesWithLeadingActor.Add(BabyTalk);
    george.MoviesWithLeadingActor.Add(Syriana);
    george.MoviesWithSupportingRole.Add(Roseanne);

    db.SaveChanges();

    //query using a different datacontext.
    var dbl = new MultipleAssociationsEntities();
    var georgel =
db1.Actors.Include("MoviesWithLeadingRole").Include("MoviesWithSupportingArti
st").First(a => a.Name == "George Clooney");

    Console.WriteLine("Total Movies with Leading Role " +
georgel.MoviesWithLeadingActor.Count());
    Console.WriteLine("Total Movies with Supporting Role " +
george.MoviesWithSupportingRole.Count());
```

Output from the code executed is shown below.

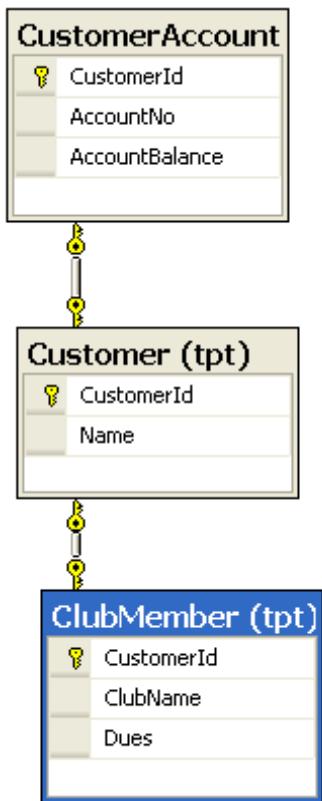


```
C:\WINDOWS\system32\cmd.exe
Total Movies with Leading Role 2
Total Movies with Supporting Role 1
Press any key to continue . . .
```

2.3 Entity Splitting

2.3.1 Entity Splitting with three tables

Problem: Figure below shows the database diagram for Customer and its additional info in the related tables.

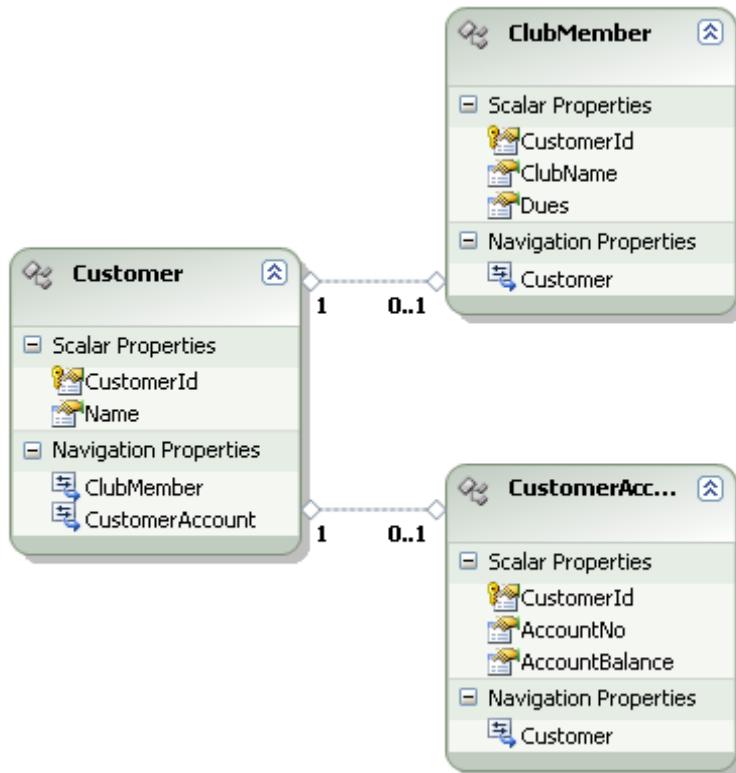


On the above database relationship, Customer has an addition table CustomerAccount that shows details for Customer's account. In addition if the customer is a ClubMember, ClubMember table would reveal the ClubName and Dues the Customer has towards the Club. Since your User interface requires all these fields to be always available, you want to expose all three tables as one single entity. You want to map the above structure to a single entity Customer using entity data model.

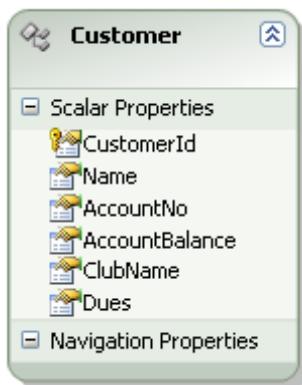
Solution: To implement the above table structure as a single entity, we need to leverage entity splitting feature of EF. Entity splitting allows you to map a single entity to two or more tables. All tables participating in entity splitting must share a common primary key which means that one table would generate the primary key and other tables would use that value as their primary key. To accomplish entity splitting open up the mapping window and select additional tables and map the properties on the entity to columns defined in other tables.

Discussion: Steps below outline the process of achieving entity splitting to expose multiple tables as a single entity.

1. Import CustomerAccount, Customer and ClubMember table using entity Model Wizard. Figure below shows the model after finishing the wizard.



2. Since we only want to have a single entity that is mapped to all 3 tables, move ClubName, Dues, AccountNumber and AccountBalance to Customer entity and delete ClubMember and CustomerAccount entity created by the designer. Figure below shows the updated model.



3. Map the additional fields moved from ClubMember and CustomerAccount entity. To map the properties, right click on Customer entity and choose mapping window. On the mapping window select Clubmember table and the designer would auto map ClubName and Dues to properties defined on Customer entity. Similarly select CustomerAccount table and EF would auto map AccountNo and AccountBalance properties to columns on CustomerAccount table. Figure below show the updated mapping configured for Customer entity.

Column	Operator	Value / Property
Tables		
Maps to Customer		
Column Mappings		
CustomerId : int	↔	CustomerId : Int32
Name : varchar	↔	Name : String
Maps to ClubMember		
Column Mappings		
CustomerId : int	↔	CustomerId : Int32
ClubName : varchar	↔	ClubName : String
Dues : int	↔	Dues : Int32
Maps to CustomerAccount		
Column Mappings		
CustomerId : int	↔	CustomerId : Int32
AccountNo : varchar	↔	AccountNo : String
AccountBalance : int	↔	AccountBalance : Int32
<Add a Table or View>		

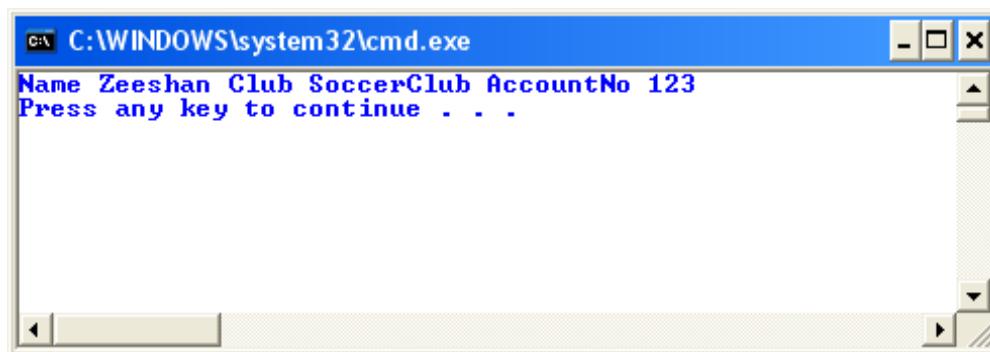
To use the model created above, we need to create an instance of Customer entity and assign values to all properties on the Customer entity and save the entity to the database. From a programming perspective, query an entity that is mapped to a single table to an entity that is mapped to two more tables is same experience. However you should consider using entity splitting only when most scenarios require access to all values defined on other related table for the Customer. Because every time we query for customer EF applies a join to all 3 tables to build a customer entity. If most scenarios only require fetching columns from customer table, then it might be better to

leave related tables as association and only load the tables when needed. On the code below, I am creating an instance of Customer entity and saving it to the database. Using second dacontext, I am retrieving the Customer entity and printing a column from each table to confirm that EF inserted record into all 3 tables.

```
var db = new ESEntities();
    var customer = new Customer
    {
        Name= "Zeeshan",
        AccountNo= "123",
        AccountBalance = 500,
        ClubName= "SoccerClub",
        Dues=10
    };
    db.AddToCustomers(customer);
    db.SaveChanges();

    var db2 = new ESEntities();
    var cust = db2.Customers.First(c => c.Name == "Zeeshan");
    Console.WriteLine("Name {0} Club {1} AccountNo
{2}", cust.Name, cust.ClubName, cust.AccountNo);
```

Figure below shows the result from the console window.



3. Eager and Lazy Loading entities and Navigation properties

3.1 Using Include to load Child Entities in Entity Framework

Ef is explicit about only loading items that you have requested. If you request for a Customer entity and Customer entity has orders associated with it, EF will not load the Orders entitycollection on behalf for you. There are two ways to load related navigation properties of an entity. You can either use Load or Include operator. You would use Load operator to lazy load a certain navigation property. However in a case where along with Customer, you also want to retrieve its Orders, you can Include operator. Include operator avoids an additional database roundtrip, by fetching Customer and its Orders collection in one single query.

Code below shows an Example of using Include operator.

```
var db = new NorthwindEFEntities();
        var custs = db.Customers.Include("Orders");
        foreach (var cus in custs)
        {
            Console.WriteLine(cus.ContactName + " Orders:" +
cus.Orders.Count());
        }
```

On the above code, we are retrieving all the Orders for the every customer in the list. If we did not use Include operator and tried to access Orders collection, the return value we will get is zero.

If you overuse Include to load quite a few navigation properties, there is a possibility that the entire query may time out. The reason is, for every Include, EF has to include additional join and this can cause the query to get fairly complex. To ensure correct retrieval of data, EF de-normalizes the query by flattening the hierarchy which can lead to more columns being projected on the select clause of the query. This will cause the width of the result set to increase and also lead to redundant and duplicate data brought

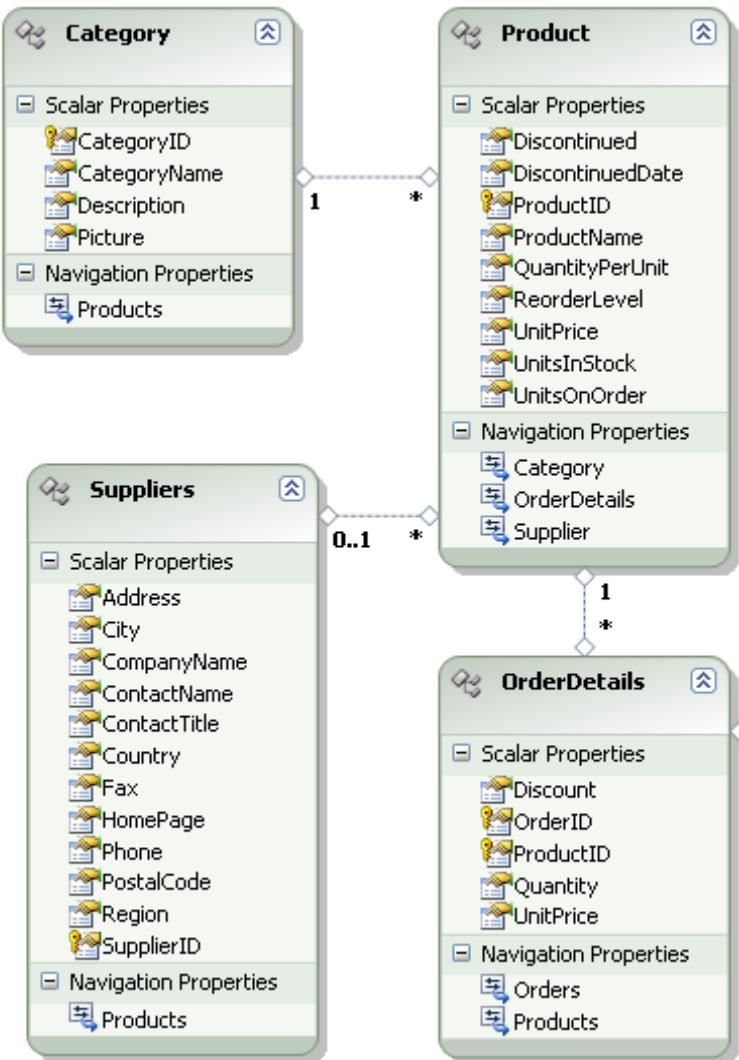
from the database into application server memory. Eventually EF when it tries to create objects, will remove the duplication but the amount of data it needs to process to remove the duplication can be more overhead then simply bring each navigation property separately by calling Load. If the entity you are including is an entity ref meaning the original entity is many side of many to 1 association, there won't be any duplication of data but including a navigation property that is a many side of the association will always result in redundant data brought from the database. However it is important to evaluate your case and identify if having a single round trip for the entire query gives you better performance in comparison to bring data in small chunks using Load method.

When a query uses Include to load navigation properties, EF under the covers rewrites the query into a projection which has in one column containing the original entity type and other columns contains rest of the related entities. Additionally it stores the Meta data about how the query needs to be mapped back to the originally entity requested and fix up the relationship between the original entity and the related entities when data is brought over from the database. This ensures that data is not returned as DbDataRecord where the first column contains original entity and rest of the columns has related entities.

In examples below we will discuss using Include operator to eagerly load entity collection and entity reference.

3.1.1 Loading EntityRef and EntityCollection Using Include

Problem: Figure below shows the EDM model for Product entity which has associations to Category, supplier and the OrderDetails for each Product.



You want to get products that have a UnitPrice greater than 50 and along with the Products, you want to eagerly load its Category, OrderDetails and Supplier reference.

Solution: To load Category, OrderDetails and Supplier navigation properties for products that have unit Price greater than 50 dollars, we need to use Include operator three times for every navigation property on Product entity. Code below uses multiple includes to all three navigation properties and prints the result to the console window.

```

var db = new NorthwindEFEntities();
    var prods = from p in
db.Products.Include("Category").Include("Supplier").Include("OrderDetails")
    
```

```

        where p.UnitPrice > 50.0M
        select p;
    foreach (var prod in prods)
    {

Console.WriteLine("ProdID:{0}\tCategory:{1}\tSupplier:{2}\tODS Count:{3}",
prod.ProductID, prod.Category.CategoryName, prod.Supplier.ContactName,
prod.OrderDetails.Count());
    }

```

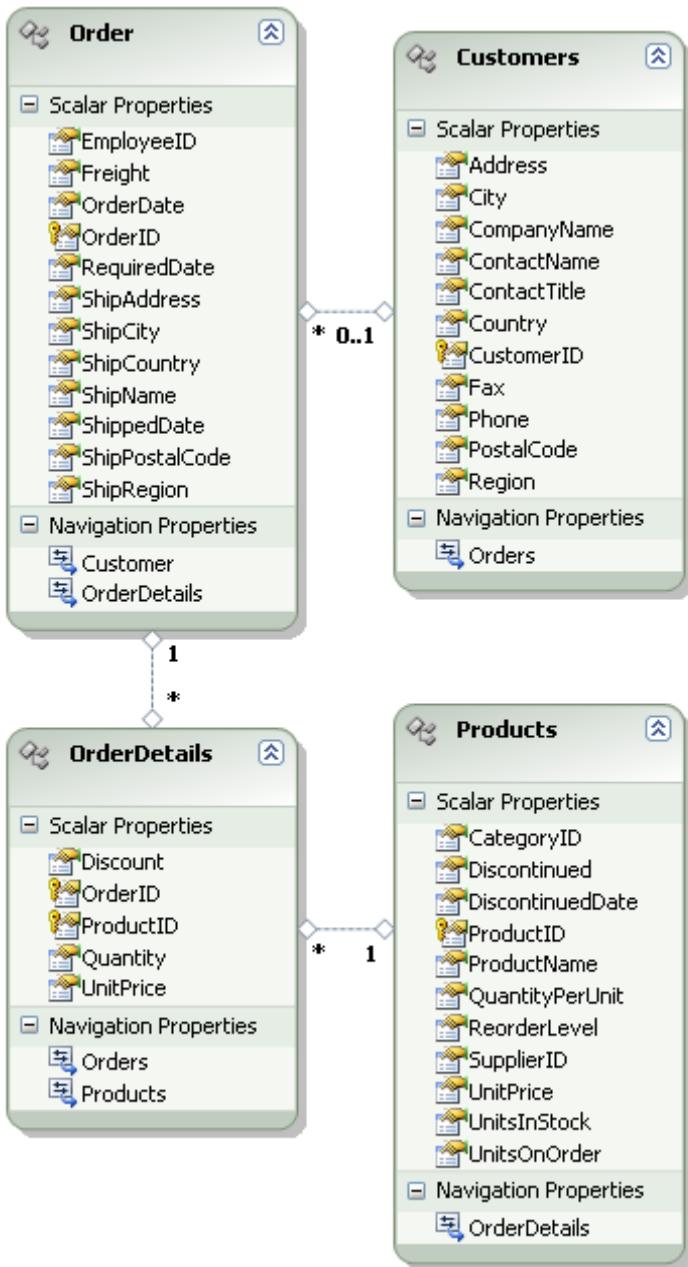
Figure below shows the Category and their related navigation properties printed on the console window.

ProdID	Category	Supplier	ODS Count
9	Meat/Poultry	Yoshi Nagase	5
18	Seafood	Ian Devling	27
20	Confections	Peter Wilson	16
29	Meat/Poultry	Martin Bein	32
38	Beverages	Guylène Nodier	24
51	Produce	Wendy Mackenzie	39
59	Dairy Products	Eliane Noz	54

Discussion: Entity framework allows you to use Include to eagerly load different types of association. On the above example, if we look at the conceptual model, we will notice that Product has 1 to many relationship with OrderDetails resulting in OrderDetails returning an entity collection. Product has Many to 1 association with Category and Supplier with Supplier end of the association is optional. When there is a Many to 1 association, EF exposes an entity reference for the navigation property; hence Product entity has two additional properties CategoryReference and SupplierrReference. Loading entity reference or an entityCollection is exactly the same using Include operator.

3.1.2 Using Include with Query Path to load related entities

Problem: Figure below shows EDM model for Orders and its related and its related entities.



Based on EDM model above you want to load top 2 orders which were shipped city of London. For each of these orders, you want to load its Customer, its OrderDetails and for each OrderDetails load its product information.

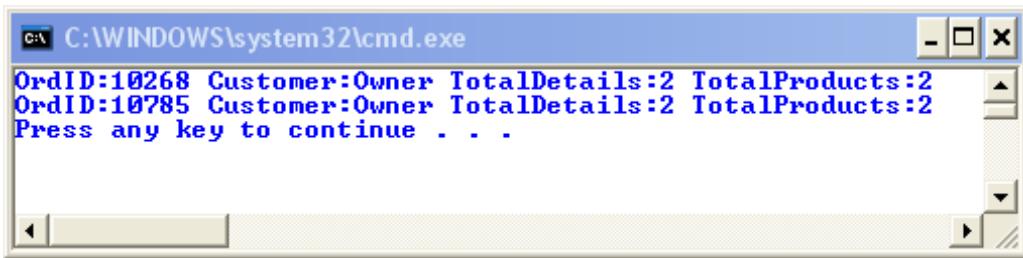
Solution: To load related entities for Order entity, we will use Include operator. Since Customer entity is a navigation property on the Order entity, it will only require adding an include of Customers. However if we want to load Products which is a navigation property of OrderDetails entity which is a navigation

property of Order, we will have use a query path. Query path allows us to load object graph defined by the path specified as string inside the include operator. For instance to load OrderDetails and its product information for given Order, we can use OrderDetails.Products.

Discussion: When you specify query path, you can go as many level deep in the object graph as you want. However this would lead to a fairly complex query which may result in a timeout in the execution of the query. Currently in v1, Ef does not have the smartness to identify that query is quite big and it may be more efficient to break the execution into two separate queries and then execute to achieve better performance. Additionally an entity can have several includes and each include can contain a query path or just a simple navigation property. Code below shows how to load related entities for Order entity defined on the above model.

```
var db = new LazyLoadingEntities();
    var orders = from o in
db.Orders.Include("Customer").Include("OrderDetails.Products")
    where o.ShipCity == "Caracas"
    select o;
foreach (var order in orders)
{
    Console.WriteLine("OrdID:{0} Customer:{1} TotalDetails:{2}
TotalProducts:{3}",
        order.OrderID, order.Customer.ContactTitle,
        order.OrderDetails.Count(),
        order.OrderDetails.Select(od =>
od.Products.ProductID).Count()
    );
}
```

On the code above I am getting all orders shipped in the city of Carcas. Additionally I am loading each order's customer, OrderDetails and for each OrderDetail loading its product. On the console window I am printing OrderId, Customer's ContactTitle, count of OrderDetails for an order and for each order the count of distinct products ordered. Figure below shows the result on printed on the console window.

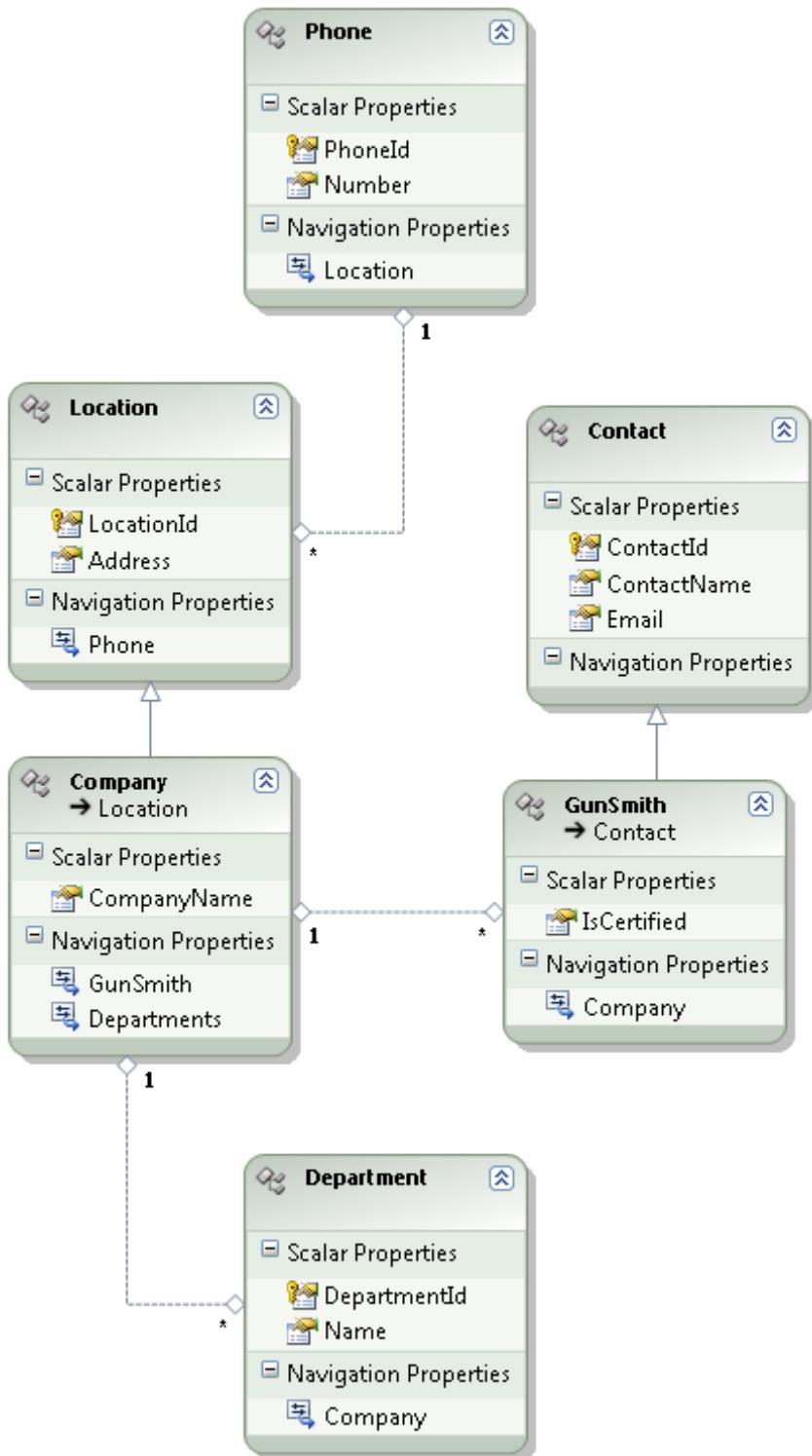


A screenshot of a Windows Command Prompt window titled "C:\WINDOWS\system32\cmd.exe". The window contains the following text output:

```
OrdID:10268 Customer:Owner TotalDetails:2 TotalProducts:2
OrdID:10785 Customer:Owner TotalDetails:2 TotalProducts:2
Press any key to continue . . .
```

3.1.3 Eagerly loading navigation properties on derived Types

Problem: Figure below shows the EDM model for Gunsmith, its associated company and Phone entity.



On the above conceptual mode, GunSmith extends Contact entity. A Gunsmith belongs to a Company. The Company extends a Location entity

which has an association to Phone entity. In addition a company has Many departments represented by 1 to Many association between Company and Departments. You want to retrieve gunsmith, the company he belongs to, the company's phone and all the departments the company has in one single query.

Solution: EF does not expose derived entities on the objectcontext, instead it exposes an entityset called Contacts. So if want to eagerly load Company navigation property on GunSmith entity, we cannot use an include statement like below

Db.Contacts.Include("Company")

The above query will lead to runtime error saying there is no Company property available on Contact. If we look at the model we can confirm that surely there is no Company navigation property on Contact. Company property is available on a type of Contact, GunSmith. Therefore to eagerly load Company, we need to use OfType operator first to reach to Gunsmith and then call Include for Company. While we are at the Company entity we can use query path twice, first to load the Departments for the company and second to load the Phone for the company exposed on the base entity location.

Discussion: The behavior of Include is same regardless if the entity is deriving from another entity. If you want to eagerly load a navigation property that is available on a derived entity, you have to use OfType operator to first reach to derived entity and then call Include to load the navigation property. Calling include at the base entity would lead to runtime saying that a navigation property being accessed is not available. Codes below demonstrate eager loading gunsmith, company, company's department and company's phone entity.

```
var db = new IncludeTPTEntities();
    var gunsmith =
db.Contacts.OfType<GunSmith>().Include( "Company.Phone" ).Include( "Company.Depa
rtments" ).First();
    Console.WriteLine( "Name:{0} Company:{1} Phone:{2}
Departments:{3}" ,
                    gunsmith.ContactName, gunsmith.Company.CompanyName,
                    gunsmith.Company.Phone.Number,
                    gunsmith.Company.Departments.Count() );
```

The code above uses two query paths. First query path eagerly load Company and its Phone and second query path eagerly loads Company and its entire department. Notice that both query paths are applied after the OfType operator because OfType operator allows the query to reach to Gunsmith of Type Contact. If the query did not use OfType operator EF would not know how to access company because it is exposed on the derived type.

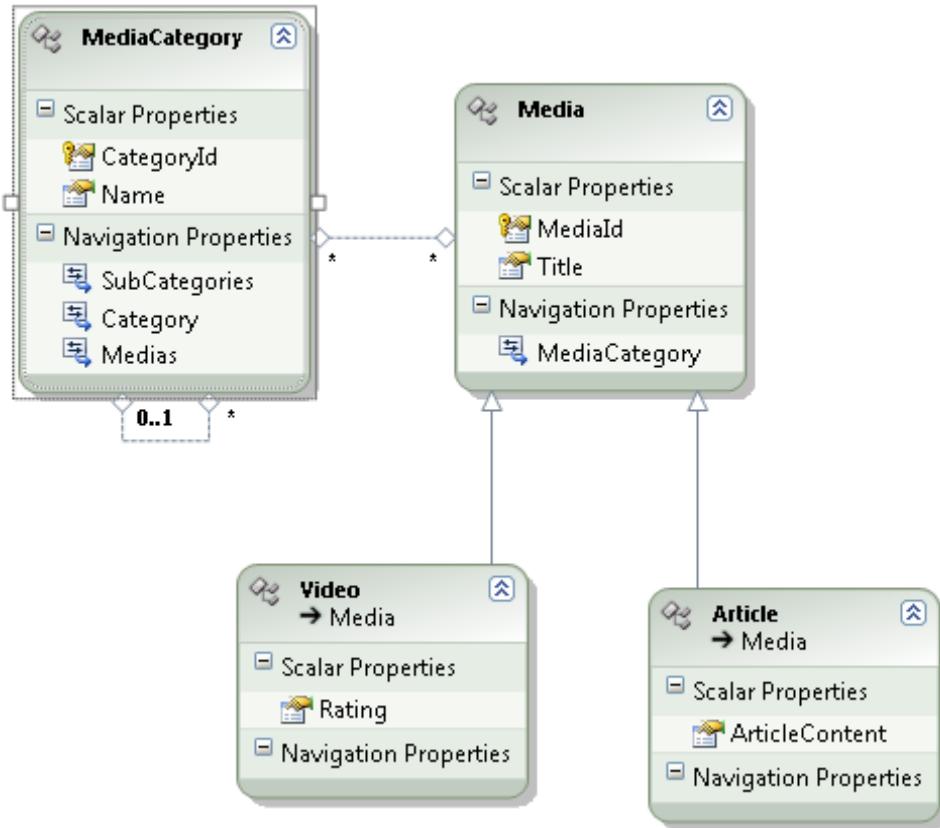
Figure below shows the output of the above query on the console window.



A screenshot of a Windows command prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The window contains the following text:
Name : Zeeshan Company : Food LTD Phone : 123-456-9899 Department : Sales
Press any key to continue . . .

3.1.4 Using Include with self referencing entity

Problem: Figure below shows EDM model for MediaCategories and the media associated with those categories.



On the above model, a category can have subcategories and at each category you can different types of media associated to that category. Types of media associated to a category could be an Article or Video. In your database you have a category GunHistory which has articles and videos associated to it. In addition gunhistory category also has subcategories those categories also have media associated it. You want to load gunhistory and its medias and also all the subcategories for gunhistory and its media as well in a single database request.

Solution: To accomplish the above requirement based on the conceptual model shown above, we have to use two includes. First include would eagerly load all the medias for gunhistory. Second include would make use of query path to load subcategories for the gunhistory and its Medias by using `Subcategories`. Medias. When we include `Media` navigation property the result would contain both types of Media, Article and Video. If we want to access properties specific to each type of media we can either cast the `Media`

to specific derived type or use the OfType operator passing the generic T which would represent a specific derived type.

Discussion: Code below shows Include statements required to eagerly load Medias for gunhistory category, its subcategory and its Medias as well.

```
var db = new MediaSelfRefEntities();
    var gunhistory = db.MediaCategories
        .Include("Medias")
        .Include("SubCategories.Medias")
        .First(mc => mc.Name == "Gun History");

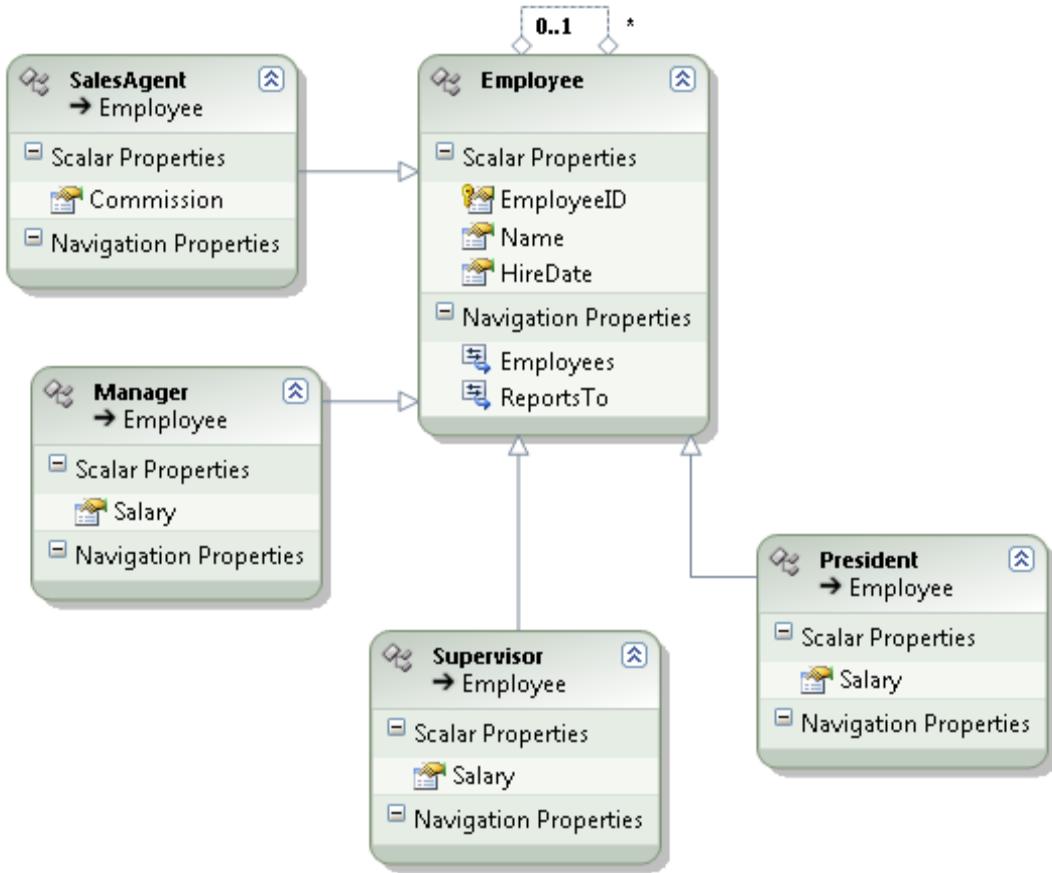
    Console.WriteLine("Category:{0} Total
Medias:{1}", gunhistory.Name, gunhistory.Medias.Count());
    Console.WriteLine("    SubCategories");
    foreach (var subcategory in gunhistory.SubCategories)
    {
        Console.WriteLine("        Category:{0}
TotalMedias:{1}", subcategory.Name, subcategory.Medias.Count());
    }
```

On the above code, as discussed I have two includes; first include retrieves all the Medias associated with gunhistory category and the second include retrieves both SubCategories for gunhistory and all its medias. To confirm the result I am printing the name of the category and Total Medias in that category. In addition I am looping through the subcategories for gunhistory and also printing the sub category name and the Medias in those subcategories. Figure below shows the result printed on the console window.

```
C:\Windows\system32\cmd.exe
Category:Gun History Total Medias:4
SubCategories
Category:SubMachine Guns TotalMedias:68
Category:Rifles TotalMedias:14
Category:Shotguns TotalMedias:0
Category:Handguns TotalMedias:0
Press any key to continue . . .
```

On the above example MediaCategory is a self referencing entity and we only went one level deep in the hierarchy. From EF perspective there is no limit to the depth of tree that you can travel. Although you reach to a point where going that many levels deep would create an enormous query that may not be an optimal way to execute on the database. Just to proof that there is no limit

to the dept you can use Include with self referencing entity, I have created another entity data model which has data at several depths.



On the above Employee model, an employee reports to an employee above it using the **ReportsTo** navigation property. An Employee can of four different types, **SalesAgent**, **Manager**, **Supervisor** and **President**. A **SalesAgent** reports to a **Supervisor**, a **Supervisor** reports to **Manager** and **Manager** reports to the president of the company. So given an entity employee of type **president** and you want to access all levels of tree, you would have to include three times to get to the last level of employee which is the **SalesAgent**. Code below shows how the **Include** would look like.

```

var db = new SFCTEPHEntities();
var president = db.Employees
    .Include("Employees.Employees.Employees")
    .First(e => e.ReportsTo == null);
Console.WriteLine("President:{0}", president.Name);
var manager = president.Employees.OfType<SFCTE.Manager>().First();
Console.WriteLine("Manager:{0}", manager.Name);
  
```

```

var supervisor =
manager.Employees.OfType<SFCTE.Supervisor>().First();
Console.WriteLine("Supervisor:{0}",supervisor.Name);
foreach (var agent in
supervisor.Employees.OfType<SFCTE.SalesAgent>())
{
    Console.WriteLine("SalesAgent:{0}",agent.Name);
}

```

On the above code, I am using a querypath inside of Include operator. From a president level, employees could be three levels deep. The first Employees will load Managers reporting to the President. Second Employees will load supervisors reporting to manager and the last Employees navigation property will load all the salesagent reporting to the supervisor. The rest of the code accesses each level and prints employees on all levels to the console window. Figure below shows the result on the console window.

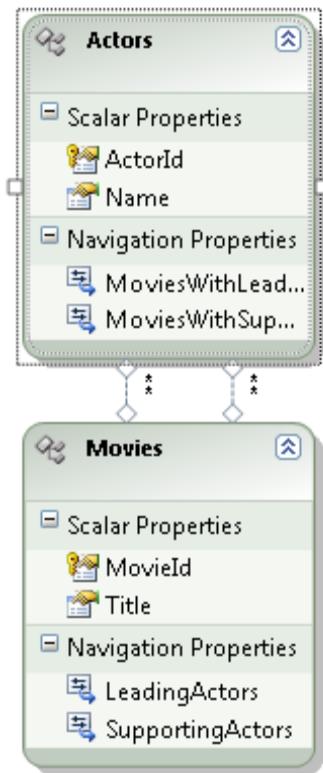
```

President:Andrew Fuller
Manager:Steven Buchanan
Supervisor:Laura Callahan
SalesAgent:Nancy Davolio
SalesAgent:Margaret Peacock
SalesAgent:Michael Suyama
SalesAgent:Robert King
SalesAgent:Anne Dodsworth
Press any key to continue . .

```

3.1.5 Using Include with Many to Many association

Problem: Figure below shows the EDM model with relationship between Actor and Movies entity.



An actor can participate in many movies in a leading role or a supporting artist. To access the movies for the actor where the actor participated as leading role, we can use **MoviesWithLeadingRole** navigation property. Similarly movies with supporting artist for the actor can be accessed using **MoviesWithSupportingArtist**. Both navigation properties are a many to relationship with **Movies** entity. For a given actor you want to access all the movies the actor participated in whether it be in leading role or as a supporting artist.

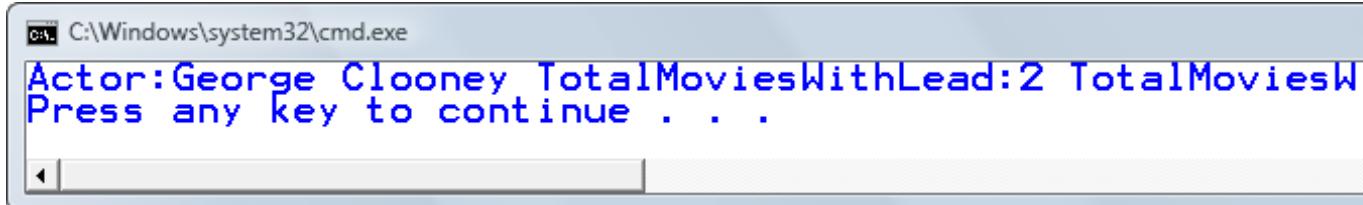
Solution: Using **Include** with Many to Many relationship is not any different than with any other types of relationship offered in EF. To access both types of movies, we will have to use **include** multiple times. Beware that using with Many to Many relationship could turn to be an expensive because both includes would require a join to the link table under the covers to access the many side of the relationship.

Discussion: When you multiple includes to the same table EF creates a union statement bring both results together in the same query. As mentioned earlier beware that an **Include** on M-M relationship could be expensive as it would

require an extra join cost which 1-M and 1-M relationship won't incur. Code below shows the linq query required to retrieve both types of movies for an actor.

```
var db = new MultipleAssociationsEntities();
    var actor = db.Actors
        .Include("MoviesWithLeadingActor")
        .Include("MoviesWithSupportingRole").First();
    Console.WriteLine("Actor:{0} TotalMoviesWithLead:{1}
TotalMoviesWithSupport:{2}",
    actor.Name, actor.MoviesWithLeadingActor.Count(), actor.MoviesWithSupportin
gRole.Count());
```

On the code above, having multiple includes ensure that I am immediately bring both types of movies for the actor and printing the result to the console window. Figure below shows the result on the console window.



```
C:\Windows\system32\cmd.exe
Actor:George Clooney TotalMoviesWithLead:2 TotalMoviesWithSupport:1
Press any key to continue . . .
```

Just for the sake of understanding the complexity of using Include with M-M relationship, I am also display the sql generated by EF when eager loading multiple M-M relationship.

```
SELECT
  ...
FROM  ( SELECT
  ...
FROM  ( SELECT TOP (1)
      [Extent1].[ActorId] AS [ActorId],
      [Extent1].[Name] AS [Name],
      1 AS [C1]
      FROM [dbo].[Actors] AS [Extent1] ) AS [Limit1]
LEFT OUTER JOIN  (SELECT
      [Extent2].[ActorId] AS [ActorId],
      [Extent3].[MovieId] AS [MovieId],
      [Extent3].[Title] AS [Title],
      1 AS [C1]
      FROM  (SELECT
      [MoviesWithLeadingActor].[ActorId] AS [ActorId],
      [MoviesWithLeadingActor].[MovieId] AS [MovieId]
      FROM [dbo].[MoviesWithLeadingActor] AS [MoviesWithLeadingActor]) AS
[Extent2]
```

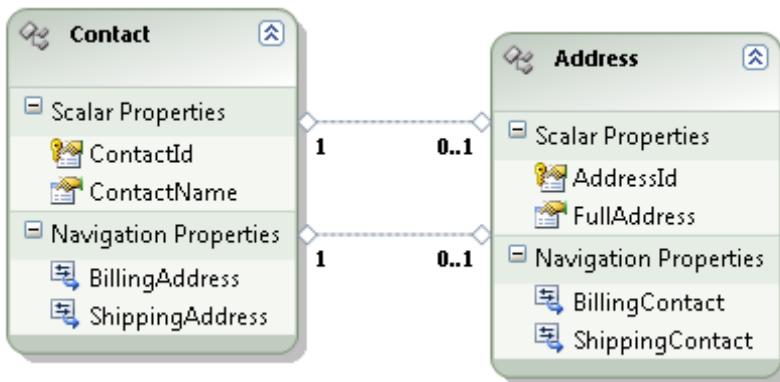
```

        INNER JOIN [dbo].[Movies] AS [Extent3] ON [Extent3].[MovieId] =
[Extent2].[MovieId] ) AS [Project2] ON [Limit1].[ActorId] =
[Project2].[ActorId]
UNION ALL
...
FROM    (SELECT TOP (1)
[Extent4].[ActorId] AS [ActorId],
[Extent4].[Name] AS [Name],
1 AS [C1]
FROM [dbo].[Actors] AS [Extent4] ) AS [Limit2]
INNER JOIN (SELECT [Extent5].[ActorId] AS [ActorId],
[Extent5].[MovieId] AS [MovieId2], [Extent6].[MovieId] AS [MovieId1],
[Extent6].[Title] AS [Title]
FROM (SELECT
[MoviesWithSupportingActor].[ActorId] AS [ActorId],
[MoviesWithSupportingActor].[MovieId] AS [MovieId]
FROM [dbo].[MoviesWithSupportingActor] AS [MoviesWithSupportingActor])
AS [Extent5]
INNER JOIN [dbo].[Movies] AS [Extent6] ON [Extent6].[MovieId] =
[Extent5].[MovieId] ) AS [Join3] ON [Limit2].[ActorId] =
[Join3].[ActorId]) AS [UnionAll1]

```

3.1.6 Using Include at entity client layer

Problem: Figure below shows the EDM model with relationship between Contact and Address entity.



A contact can have two addresses, a billing address and shipping address. You want to retrieve contact using esql query. Along with the contact you want to retrieves billing and shipping address.

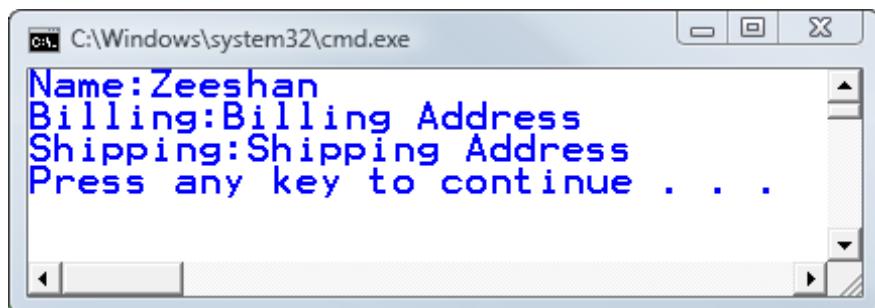
Solution: Include operator not only works with linq queries but you can also use Include operator with esql queries as well. To retrieve contact and its related address, we need to create an ObjectQuery that returns the contact

based on the criteria we request and then append the Include statements specifying additional navigation properties that we want to load with Contact entity.

Discussion: Code below shows an example of retrieving Contact Zeeshan and its related address navigation properties.

```
var db = new MultipleAssoEntities();
    string query = "select value c from Contacts as c";
    var contacts = new ObjectQuery<Contact>(query, db);
    contacts = contacts.Where("it.ContactName = @ContactName", new
ObjectParameter("ContactName", "Zeeshan"));
    var contact =
contacts.Include("BillingAddress").Include("ShippingAddress").First();
    Console.WriteLine("Name:{0}", contact.ContactName);
    Console.WriteLine("Billing:{0}",
contact.BillingAddress.FullAddress);
    Console.WriteLine("Shipping:{0}",
contact.ShippingAddress.FullAddress);
```

On the code above, I am using ObjectQuery to build a query for contacts and then using Builder methods I am filtering the query on ContactName to retrieve contact Zeeshan. To retrieve the contact's Billing and shipping address, I am using Include operator twice followed by First operator to retrieve the first contact as I know the query will only return one result. To confirm that I have Billing and shipping addresses loaded, I am printing the contact's address on the console window. Figure below shows the screen for contact information on the console window.



3.1.7 Common Pitfalls with Include operator

Problem: What are some common gotchas and incorrect usages of Include operator that EF developer must be aware of when writing eager loading queries.

Solution: Include operator offers eager loading of related entities. However if the related end is the many side of the relationship, then EF would bring redundant data to application server memory and then based on meta data would remove redundancy and build the complete object graph. So Include would offer good performance when the related end is one side of the relationship because then there would not be any duplication of data. Having too many include can complicate the query causing the query timeout. You should evaluate the scenario and decide if lazy loading related entities would offer better performance in contrast to eager loading.

When Include is used to eagerly load related enteritis on derived entities, you must use OfType operator to reach to derive entity before you can apply include operator otherwise EF will try to look for a navigation property on base entity which would not exist and cause runtime errors.

When Include is used with an entity that participate in an existing query such as join or nested from clause, EF will lose the include operation and the related entities would not be eagerly loaded. To ensure Include works correctly, ensure that Include is the last operation performed on the query.

If Include is applied on an entity that participates in a query but the final projection of the query, the select statement is an anonymous type, EF will silently drop the include statements. The reason is anonymous type results in ObjectQuery<DbDataRecord> and Include only works when the return type is entities.

Currently in version 1 release of EF, there is no way to filter entity or entities defined inside the include statement. If you have a need to filter related entities, consider changing the query to use anonymous type and inside the select portion of the query, load both the original entity and the filtered related entity. Then using the Attach method, attach the related entity back to the original entity.

Discussion: On the above solution, I mentioned common pitfalls I perceive developers may run into. For instance, you have to take caution when you include a navigation property that is many side of the relationship because

this would lead to redundant data brought from the database. The code below retrieves all the addresses for a customer.

```
var db = new OneToManyEntities();
    var customer = db.Customer
        .Include("Addresses")
        .First(c => c.ContactName == "Zeeshan");
Console.WriteLine("Customer:{0} Total Addresses:{1}",
    customer.ContactName, customer.Addresses.Count());
```

Since a customer has many addresses, EF will flatten the query to retrieve customer and its address in one single query. To identify the problem of redundancy, I have captured the sql statement that was executed for the above linq query.

```
SELECT ...
FROM (
    SELECT ...
    ...
    CASE WHEN ([Extent2].[AddressId] IS NULL) THEN CAST(NULL AS int) ELSE 1
END AS [C2],
    CASE WHEN ([Extent2].[AddressId] IS NULL) THEN CAST(NULL AS int) ELSE 1
END AS [C3]
    FROM (SELECT TOP (1)
        [Extent1].[CustomerId] AS [CustomerId],
        [Extent1].[ContactName] AS [ContactName],
        1 AS [C1]
        FROM [onetomany].[Customer] AS [Extent1]
        WHERE N'Zeeshan' = [Extent1].[ContactName] ) AS [Limit1]
    LEFT OUTER JOIN [onetomany].[Address] AS [Extent2] ON
[Limit1].[CustomerId] = [Extent2].[CustomerId]
) AS [Project2]
    ORDER BY [Project2].[CustomerId] ASC, [Project2].[C3] ASC
```

Screen shot below shows the above query executed on the sql server.

CustomerId	ContactName	C1	C2	C3	AddressId	Address1	City	State	Zip	CustomerId1
1	Zeeshan	1	1	1	1	Happy st1	Dallas	Tx	76111	1
2	Zeeshan	1	1	1	2	Happy st2	Dallas	Tx	76111	1

Although there is only one contact Zeeshan, from the above result you can see that Zeeshan is repeated twice because there are two addresses which results in duplication of data. You can imagine duplication of data could increase rapidly if we start to introduce more includes that are many side of the relationship. So beware when using includes that sometimes lazy loading a

collection may outperform eager loading a collection that could potentially have lot of redundant data.

If you are performing any grouping operation in a linq query, you must use Include as the last operation. For instance the code below shows an incorrect and correct usage of grouping.

```
var db = new NorthwindEFEntities();
    //incorrect query Include is lost
    //var prods = from p in db.Products.Include("Category")
    //            group p by p.Category.CategoryID into g
    //            select g.FirstOrDefault(p1 => p1.UnitPrice ==
g.Max(p2 => p2.UnitPrice));

    var prods = from p in db.Products
                group p by p.Category.CategoryID into g
                select g.FirstOrDefault(p1 => p1.UnitPrice ==
g.Max(p2 => p2.UnitPrice));
    prods = prods.Include("Category");
    foreach (var product in prods)
    {
        Console.WriteLine("Product:{0}
Category:{1}", product.ProductName, product.Category.CategoryName);
    }
```

The above commented linq query will not cause Category to be eagerly loaded with Product entity because when we apply grouping operation, Include is lost. To ensure that eager loading works as desired make sure that Include is the last operation performed on the query. Not all query operators cause this behavior with Include. For instance if you use orderby or where operators, Include works correctly when you use it early in the query.

When include is used with nested from clause, Include operator is also as well. Code below shows an example of that.

```
//nested from clause would also cause the include to get lost
var db = new NorthwindFullEntities();
//var orders = from o in db.Orders.Include("Customer")
//              from od in o.Order_Details
//              where od.Quantity > 120
//              select o;

var orders = from o in db.Orders
              from od in o.Order_Details
              where od.Quantity > 120
              select o;
orders = orders.Include("Customer");
foreach (var order in orders)
{
```

```

        Console.WriteLine("OrderID:{0}
Customer:{1}",order.OrderID,order.Customer.ContactName);
    }

```

The commented out query where we are eagerly loading Customer for an Order, does not work as expected and we do not get the customer for an order. However the next query applies the Include operator as the last operation which ensures that customer for the Order is loaded.

Another subtle case to be aware of is the join clause. Join clause meets the same fate where using Include early in the query results in losing the Include operation. Code below shows an example of that.

```

var db = new NorthwindFullEntities();
    var ods = from od in db.Order_Details
              join o in db.Orders on od.OrderID equals o.OrderID
              where o.ShipCity == "Caracas"
              select od;
    ods = ods.Include("Products");
    foreach (var od in ods)
    {
        Console.WriteLine("Product:{0}
Quantity:{1}",od.Products.ProductName,od.Quantity);
    }

```

Code above shows the correct usage where I am loading product information for every order detail item. As suggested the correct usage is to ensure that Include is the last operation on the query.

What if we wanted to filter the related collection being eagerly loaded? Currently Include operator does not have any option to filter on the included collection. If you have requirement which the related collection must be filtered, then you should consider return an anonymous type that contains the original entity as well the related collection. Code below shows an example of using anonymous type to filter the related collection.

```

var db = new NorthwindFullEntities();
    var cat = from c in db.Categories
              where c.CategoryName == "Beverages"
              select new
    {
        Category = c,
        Products = c.Products.Where(p =>
p.Suppliers.SupplierID == 1)
    };
    var beverage = cat.Select(c => c.Category).First();
    //attach the product to the category

```

```

var prods = cat.SelectMany(c => c.Products).AsEnumerable();
beverage.Products.Attach(prods);

Console.WriteLine("Category:{0} TotalProducts:{1}",
    beverage.CategoryName,beverage.Products.Count());

```

On the above code, I am retrieving the Category and its related products. Since I am only interested in related products where supplierid equal to one, I am apply the where to filter the Products returned for a given category. Then using the Select operator, I retrieve the first category from the anonymous types. Similarly to retrieve the Products I use SelectMany operator to fetch the product collection from the anonymous type. To build the complete graph in memory I attach the products retrieved from the anonymous type to the product collection of the category. This is one of the ways you can return a partial collection for products for a given category.

3.2 Using Load Operator to Lazy Load Collection and entity reference

Problem: You have navigation properties on your class. Some of the properties are collection and some are entity reference. You want to know how to use Load operator to lazy load navigation properties.

Solution: Unlike linq to sql, entity framework requires every database operation to be explicit. When you access a navigation property such as an entity reference or entity collection, by default relationships are not loaded. In the case of entity reference when it's not loaded, the object would be set to null. In the case of entity collection, the collection would be empty. To load either of the navigation properties you have to call Load method on entity reference or entity collection. Load method is designed to always make a database call regardless if the collection or reference is loaded or not. To prevent reloading a collection that is already loaded, you can check for IsLoaded property which would be set to true once the collection is loaded. Code below shows example of using load method to load orders for a customer and also loading entity reference customer for a given order.

Listing 1-1

```

var db = new LazyLoadingEntities();
    var alfki = db.Customers.First(c => c.CustomerID == "ALFKI");
    if (!alfki.Orders.IsLoaded)
    {
        alfki.Orders.Load();
    }

    var orders = db.Orders.Where(o => o.ShipCity == "London");
    int databasecalls = 0;
    foreach (var order in orders)
    {
        order.CustomerReference.Load();
        databasecalls++;
    }
    Console.WriteLine("Total database calls " + databasecalls);

```

Discussion: In the above code, I am checking to see if orders for ALFKI customer are loaded by calling IsLoaded property. Since this is the first time I am loading the orders, IsLoaded property returns false and a database call is made to fetch the orders for ALFKI customer. After orders entity collection is populated, IsLoaded property is set to true. This way if the method is called second time around, Load will not get called.

In Listing 1-1, I am also loading Orders with ship city of London. Order entity exposes a navigation relationship Customer which by default is not loaded. I am loading the customer only if the customer has not been loaded before. Checking for isloaded is very crucial in this case because there are 33 orders returned from the query. However those 33 orders only belong to 5 unique customers. Therefore we really do not want to make 33 database calls. By checking the IsLoaded property we only make 5 database calls that equal the unique customers for the 33 orders retrieved from the query.

Following example covers various usages of Load in different scenarios.

```

var db = new LazyLoadingEntities();
    var ALFKI = db.Customers.First(c => c.CustomerID == "ALFKI");
    var ANATR = db.Customers.First(c => c.CustomerID == "ANATR");

    //simple lazy loading of all orders for a ALFKI customer.
    ALFKI.Orders.Load();
    Console.WriteLine("Initial load " + ALFKI.Orders.Count());

    //removes all items from the collection
    ALFKI.Orders.Clear();
    Console.WriteLine("Orders cleared");

    //reloads the entire collection but count is still 0 because
    default is preserve changes

```

```

ALFKI.Orders.Load();
//confirms the count is 0
Console.WriteLine("after reloading " + ALFKI.Orders.Count());

```

In the above example, I am clearing all the orders for ALFKI customer. To discard changes and reload again from database, I use Load again. However, the count value for orders still remains 0. The reason is, if source entity customer, is being tracked, calling Load on Orders collection uses MergeOption of AppendOnly. If you have made changes to collection on the client side such as changing a property on an order, adding new orders or removing an order, those changes will be preserved and will not get overwritten by Load Call. To make sure that we take changes from the database and overwrite the changes we have made on the client side, we must call Load with OverWriteChanges. Following code use Load with MergeOption set to OverWriteChanges.

Listing 1-2

```

//we need to reload our orders and discard our client changes.
ALFKI.Orders.Load(MergeOption.OverwriteChanges);

//confirm total orders is greater than 0
Console.WriteLine("reload with overwrite " +
ALFKI.Orders.Count());

```

In Listing 1-2, after calling Load with OverWriteChanges option, I confirm the count for orders is greater than 0 and only contains orders as defined by the database.

If you makes changes to properties on the Order and call Load with AppendOnly option, your changes will stay intact. In listing 1-3 I am calling Load with AppendOnly option and therefore property changes on my Order entity is not lost.

Listsng 1-3

```

var ALFKIorder = ALFKI.Orders.Single(o => o.OrderID == 10643);
ALFKIorder.ShipCity = "London";
Console.WriteLine("existing order's city changed to London");
ALFKI.Orders.Load(MergeOption.AppendOnly);
//ship city still remains as London
Console.WriteLine("After load with appendly only existing
order city " + ALFKIorder.ShipCity);

```

In listing 1-4, I am grabbing order id 10643 for alfki customer and assigning it to ANTARA customer. This results in ALFKI customer's orders collection to reduce by 1 and ANTARA's order collection to increase by 1. When I call Load on ALFKI customer to reload its order collection using OverWriteChanges it not only fixes the orders for ALFKI but also removes the order that we assigned earlier to ANTARA customer since the order 10643 once again belongs to ALFKI as shown in the code below.

Listing 1-4

```
var ALFKIorder = ALFKI.Orders.Single(o => o.OrderID == 10643);

//assign the alfki order to anatr customer
ALFKIorder.Customer = ANATR;
//alfki orders reduces by 1
Console.WriteLine("ALFKI orders after assigning order to
ANATR " + ALFKI.Orders.Count());
ALFKI.Orders.Load(MergeOption.OverwriteChanges);
//alfki ordres is fixed and increase by 1
Console.WriteLine("ALFKI orders after with overwrite " +
ALFKI.Orders.Count());
//antara no longer owns 10643 order
Console.WriteLine("ANATR orders after with overwrite " +
ANATR.Orders.Count());
```

If you have a scenario where you have loaded part of the orders for a customer and later you decide that you need to get all the order for customer, you can call Load with AppendOnly option which will append all the orders that were previously not present in the collection. It will not overwrite any existing order in the collection. In Listing 1-5, I am retrieving part of the order by attaching orders with ship freight less than and equal to 20. Later I am loading rest of the collection from the database by calling Load.

Listing 1-5

```
//loading only pat of the orders.
ANATR.Orders.Attach(ANATR.Orders.CreateSourceQuery().Where(o
=> o.Freight <= 20));
Console.WriteLine("Antarr orders partly loaded " +
ANATR.Orders.Count());
//calling load will load the entire graph.
ANATR.Orders.Load(MergeOption.OverwriteChanges);
Console.WriteLine("ANATR orders after load with overwrite " +
+ ANATR.Orders.Count());
```

MergeOptions has another option that I have not covered which is No Tracking. When you use No Tracking, you're hinting that entity or collection you want to load should not be tracked. You can only use No Tracking option in Load when the source entity is also loaded with No Tracking option. Listing 1-6 demonstrates using Load with No Tracking.

Listing 1-6

```
var ANTON = db.Customers.First(c => c.CustomerID == "ANTON");
    //code crashes because you cant load related entity with no
tracking
    ANTON.Orders.Load(MergeOption.NoTracking);

var qry = db.Customers;
qry.MergeOption = MergeOption.NoTracking;
var AROUT = qry.First(c => c.CustomerID == "AROUT");

//default uses no tracking option to load.
AROUT.Orders.Load();

//doesn't work because customer was loaded with no tracking
//and orders are loading using overwrite changes.
//AROUT.Orders.Load(MergeOption.OverwriteChanges);

//works because customer was retrieved using no tracking as
well.
AROUT.Orders.Load(MergeOption.NoTracking);
```

In Listing 1-6, when I load orders for anton with NoTracking option, I get an exception because the source query customer was loaded with tracking option and therefore Loading orders with NoTracking is not allowed. Entity framework will only allow navigation relations to be loaded with No Tracking when source entity is also loaded with No Tracking option. When I change the query for AROUT customer to use NoTracking option and call Load, I get no errors. The reason I don't get exception is because by default Load uses the same option that was used to retrieve the source entity customer. Since I fetched customer using no Tracking option, Load uses NoTracking option to retrieve orders as well. You also have the ability to call Load with explicit Merge option of NoTracking which is same as calling Load with no options. If you have loaded an entity collection such as orders with no tracking, calling Load again will raise an exception stating that collection loaded with

NoTracking cannot be reloaded. It is defined as one of the constraints in entity framework that you cannot reload an entity collection that was initially loaded with No Tracking option.

There are certain states of entity when calling load is not allowed such as when source entity is in Added, Deleted or Detached. Calling Load in these states causes invalid operation exception. Code in listing 1-7 illustrates some of these issues.

Listing 1-7

```
var testcustomer = new Customer { CustomerID = "ALFK5", City
= "Dallas", CompanyName = "XYZ" };
//cant call load on customer that is in added state.
//testcustomer.Orders.Load();

db = new LazyLoadingEntities();
var BERGS = db.Customers.First(c => c.CustomerID == "BERGS");
db.DeleteObject(BERGS);
//discuss the error because cant call load when sourc entity
is in deleted status.
//BERGS.Orders.Load();

var COMMI = db.Customers.First(c => c.CustomerID == "COMMI");
db.Detach(COMMI);
//cant deleted cuz its in detached state.
//COMMI.Orders.Load();
```

In listing 1-7, I am adding a new customer and then calling Load on its order collection. This operation results in an exception because customer is in added state and calling load is not permitted. Similarly I am marking BERGS customer for deletion and then calling Load on its orders collection. This also results in an exception because load is also not permitted on source entities marked for deletion. Also if you detach source entity from the object context such as customer in our case, calling Load will raise an exception.

When Load is called on entity reference, it does not make use of the entity key exposed on the entity reference to load related entity. When order entity is loaded, order.CustomerReference.EntityKey contains the CustomerId for the order. However when Load call is issued against

order.CustomerReference, EF does not use that customerid to load the customer. Instead to get the customerid for the order, the query for the customer is joined against order table to retrieve the customerid from the order's table.

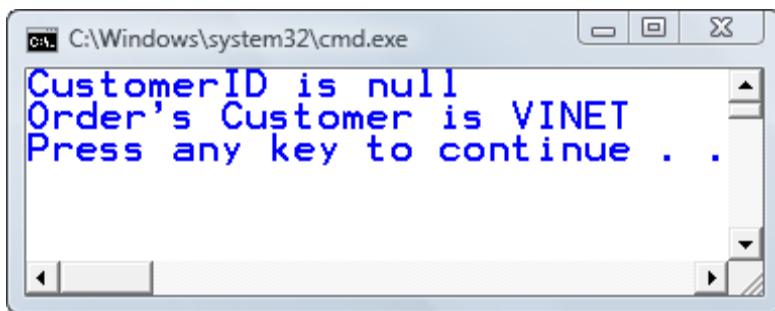
Code below shows an example that demonstrates this behavior.

```
var db = new NorthwindFullEntities();
        db.Orders.MergeOption = MergeOption.NoTracking;
        var order = db.Orders.First();
        Console.WriteLine("CustomerID is {0}",
order.CustomerReference.EntityKey == null ? "null" : "not null");
        order.CustomerReference.Load();
        Console.WriteLine("Order's Customer is
{0}", order.Customer.CustomerID);
```

The above code uses MergeOption.NoTracking to retrieve the order from the database. The side effect of notracking is there won't be any customerid for the order entity which is confirmed by printing this information on the console. Then to load the customer for the order, I am calling Load. Notice that despite that there was no customerid for the order, EF managed to load the customer reference because it queried the order table to get the customerid to load instead of reading the client side value from order.CustomerReference.EntityKey. The sql below contains the profile capture for the above load query.

```
exec sp_executesql N'SELECT
[Extent2].[Address] AS [Address],
[Extent2].[City] AS [City],
[Extent2].[CompanyName] AS [CompanyName],
[Extent2].[ContactName] AS [ContactName],
[Extent2].[ContactTitle] AS [ContactTitle],
[Extent2].[Country] AS [Country],
[Extent2].[CustomerID] AS [CustomerID],
[Extent2].[Fax] AS [Fax],
[Extent2].[Phone] AS [Phone],
[Extent2].[PostalCode] AS [PostalCode],
[Extent2].[Region] AS [Region]
FROM [dbo].[Orders] AS [Extent1]
INNER JOIN [dbo].[Customers] AS [Extent2] ON [Extent1].[CustomerID] =
[Extent2].[CustomerID]
WHERE ([Extent1].[CustomerID] IS NOT NULL) AND ([Extent1].[OrderID] =
@EntityKeyValue1)',N'@EntityKeyValue1 int',@EntityKeyValue1=10248
```

Figure below shows the result on the console window.



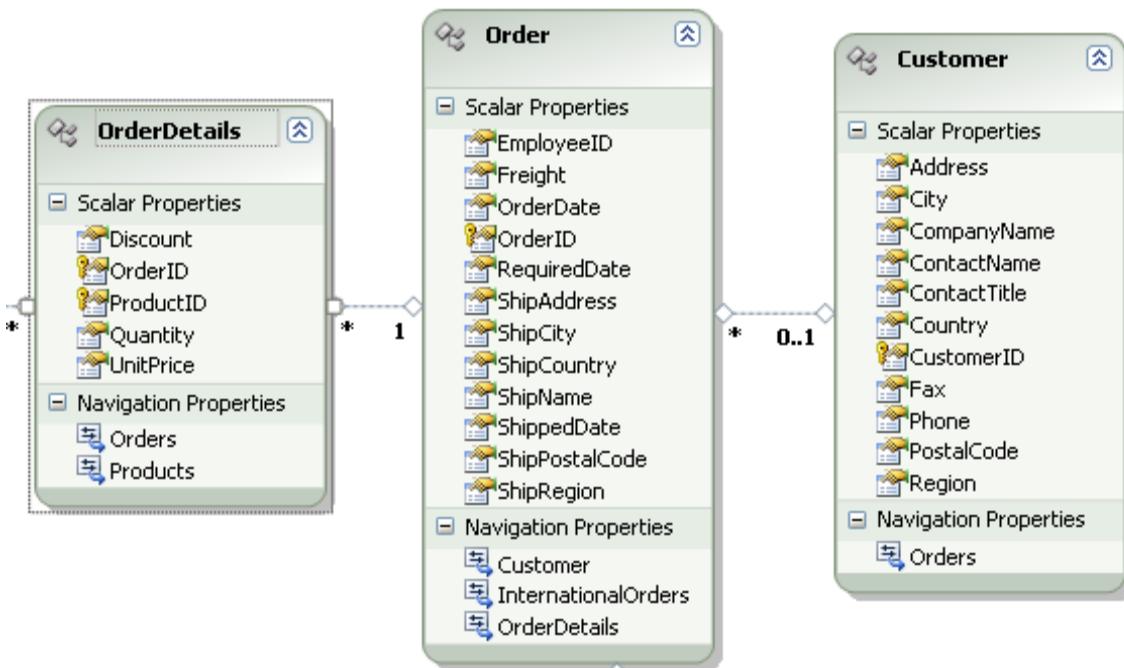
```
C:\Windows\system32\cmd.exe
CustomerID is null
Order's Customer is VINET
Press any key to continue . . .
```

Load option is not limited to many to 1 association and 1 to many associations. You can use load option to lazy load many to many navigation relations as well.

3.3 CreateSourceQuery

3.3.1 CreateSourceQuery to filter associations

Problem: Figure below shows the EDM model for Customer, Orders and OrderDetails entity.



You have retrieved a customer entity from the database and you want to retrieve only Orders for the customer which has a freight greater than thirty dollars and in addition you also want to retrieve the OrderDetails for the Orders that match that criteria.

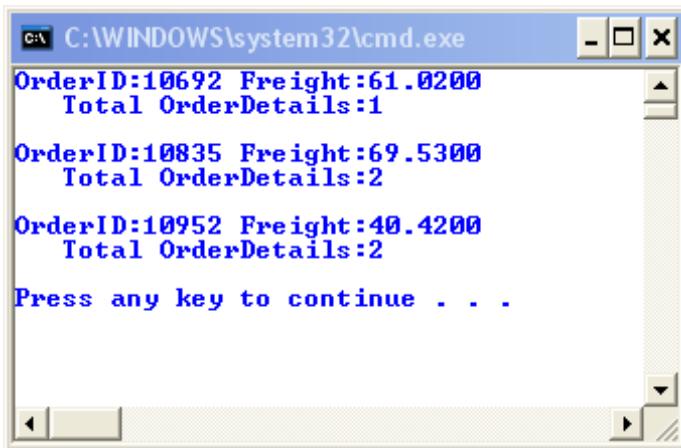
Solution: If there was no requirement to filter the Orders collection for a customer, we could use Load operator to load the Orders. Since we want to apply a filter to retrieve only Orders with freight greater than 30, we can use CreateSourceQuery method exposed on EntityCollection and EntityReference. CreateSourceQuery does not execute the query; it only creates a query that represents the collection you would get if you were to execute the query. At this moment we can pick up the query and apply further transformations such as apply Frieght filter and order Orders collection by OrderDate. In addition we can also apply include operator on the CreateSourceQuery to also return the OrderDetails for the filtered Orders. The results returned from the CreateSourceQuery call needs to be attached to the Customer entity's order collection to connect the relationship between Customer and Orders retrieved.

Discussion: To retrieve part of the Orders collection, sorted by OrderDate and also fetch the OrderDetails for those Orders, we need to use the CreateSourceQuery method to retrieve the Query that would otherwise return the Orders for the given Customer. Using that query we can apply filtering and Include operation to meet our above requirements. Once we retrieve the Orders, we must attach the Orders entity back to the Customer's Order Collection. Code below shows the required code needed.

```
var db = new CSQEntities();
var cust = db.Customers.First(c => c.CustomerID == "ALFKI");

var orders = cust.Orders.CreateSourceQuery()
    .Include("OrderDetails")
    .Where(o => o.Freight > 30.0M)
    .OrderBy(o => o.OrderDate);
cust.Orders.Attach(orders);
foreach (var order in cust.Orders)
{
    Console.WriteLine("OrderID:{0}"
Freight:{1}",order.OrderID,order.Freight);
    Console.WriteLine("    Total
OrderDetails:{0}\r\n",order.OrderDetails.Count());
}
```

On the above code, I am using Include to indicate that I want to load OrderDetails for every order that is part of the result set. In addition I am also applying a where filter and order by clause to only retrieve Orders with freight greater than 30 dollars and then sort them by OrderDate. Since the orders are now separated from the ALFKI customer, I am using Attach method to attach the Orders to Customer's Orders collection. Rest of the code iterates through the collection to confirm that we only have the Orders that meet the filter criteria and also that we have all the OrderDetails for every Order in the collection. Figure below shows the screenshot on the console window.



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text:
OrderID:10692 Freight:61.0200
Total OrderDetails:1
OrderID:10835 Freight:69.5300
Total OrderDetails:2
OrderID:10952 Freight:40.4200
Total OrderDetails:2
Press any key to continue . . .

3.3.2 CreateSourceQuery to Execute Aggregate operation on Child collections

Problem: You have Customer and Orders entity defined on EDM model. Given a customer instance you want to retrieve the total orders customer has placed without loading all the orders in memory.

Solution: We can use CreateSourceQuery method to get access to the query that would return all the Orders for a given customer. Instead of actually returning the orders we can apply Count Operator on top of the query to get total order count. The query would be translated into a count operation executed on the database ensuring that we only return the total number of orders a customer has placed rather than return all orders.

Discussion: Code below shows how to return the total orders placed by ALFKI customer without bringing all the orders for the customer.

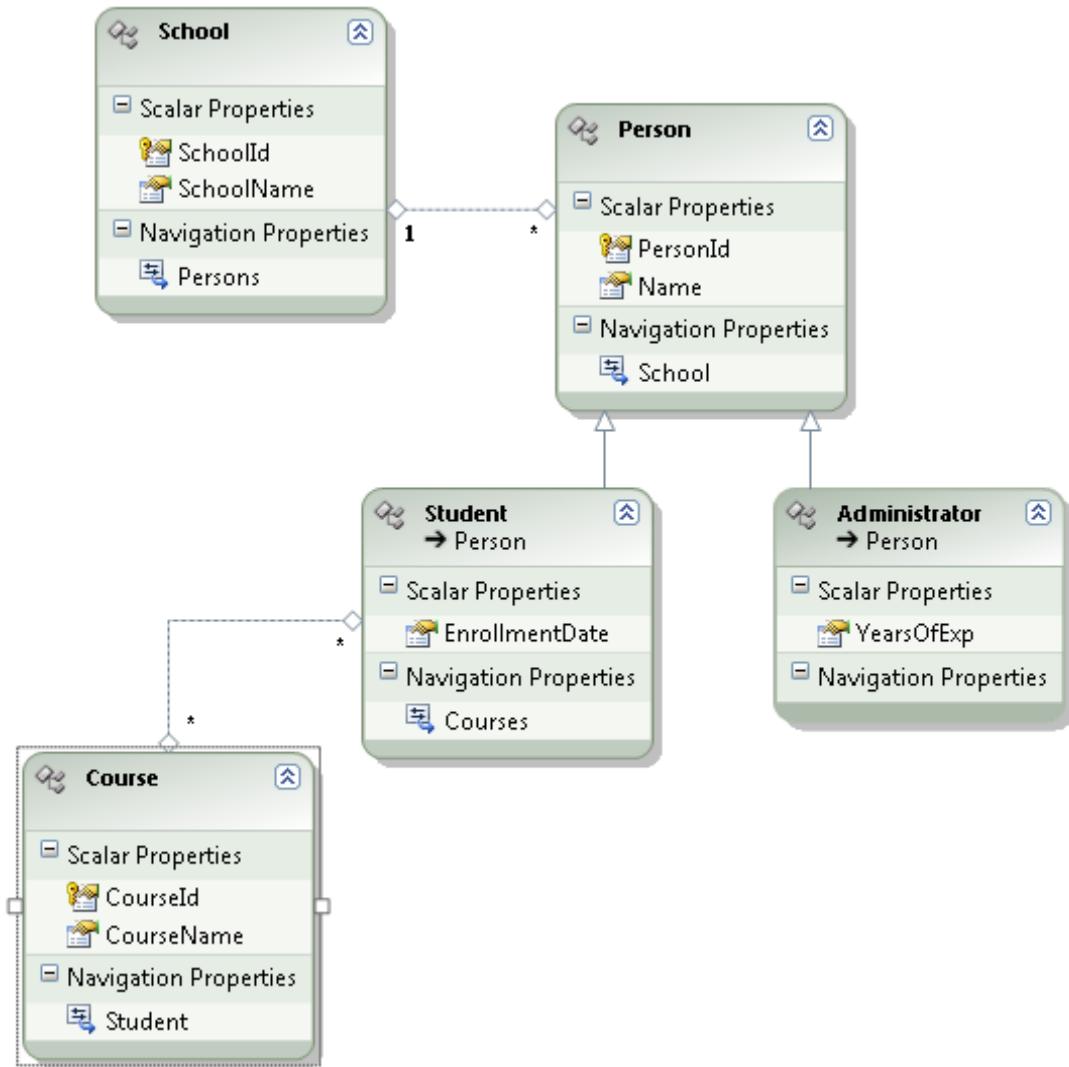
```
var db = new CSQEntities();
    var cust = db.Customers.First(c => c.CustomerID == "ALFKI");
    var totalorders = cust.Orders.CreateSourceQuery().Count();
    Console.WriteLine("Total Orders:{0}",totalorders);
```

To get the total orders for ALFKI customer, I am accessing the Orders query with CreateSourceQuery and applying count operation on it. To confirm the count operation is applied on the database I have captured the sql statement send to the database.

```
SELECT
COUNT(cast(1 as bit)) AS [A1]
FROM [dbo].[Orders] AS [Extent1]
WHERE [Extent1].[CustomerID] = @EntityKeyValue1
@EntityKeyValue1=N'ALFKI'
```

3.3.3 CreateSourceQuery to retrieve specific derived type from entity collection

Problem: Figure below shows the EDM model for School, Student, Administrator and Course entity.



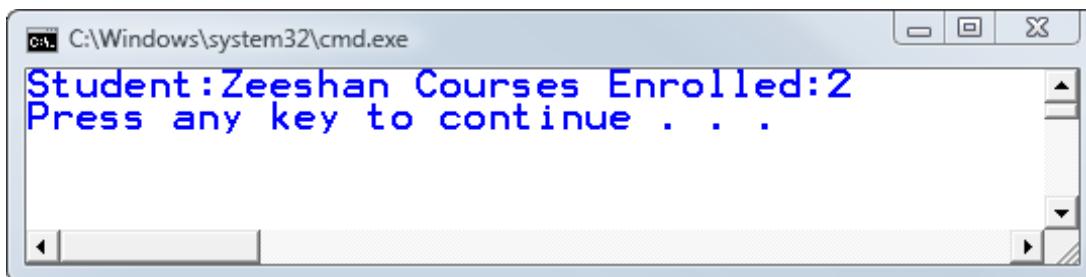
School has many persons. A person can be either a student or an Administrator. A student can be enrolled in many courses. Given an instance of school, you only want to retrieve Persons that are students. In addition for those students you also want to retrieve the courses they are enrolled in.

Solution: To retrieve only students for a school, we have to use `CreateSourceQuery` to get access to the query that would return all persons for a school. To only get Students, we can filter that query using `OfType` operator to only return students. Since we also want to return Courses for the student, we can also append an `Include` statement for courses which would ensure that for every student, we will have their courses preloaded too.

Discussion: Code below shows how to retrieve only students for a school and also eagerly load their courses.

```
var db = new CRSContainer();
    var school = db.Schools.First(s => s.SchoolName == "Habib");
    var student = school.Persons
        .CreateSourceQuery()
        .OfType<Student>().Include("Courses")
        .First();
    Console.WriteLine("Student:{0} Courses Enrolled:{1}",
student.Name, student.Courses.Count());
```

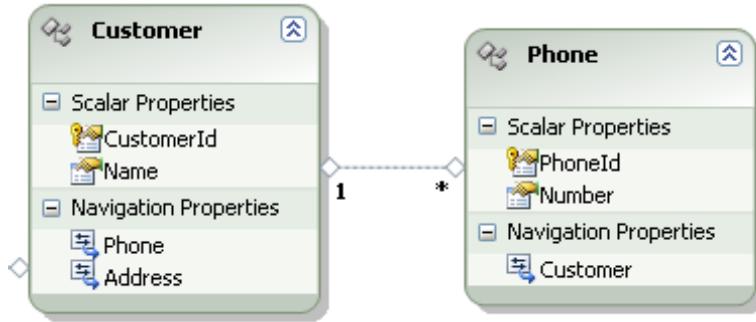
On the above query, I am applying OfType operator on CreateSourceQuery to only get Students for the school. Along with Students I am also retrieving the courses using Include followed by First operator to only return the first student. To confirm the results, I am printing the name of the Student and total courses student is registered for. Figure below shows the output on the console window.



3.4 Relationship Span

3.4.1 Taking Advantage of Relationship Span

Problem: Figure below shows the edm model for Customer and their associated Phones



You want to load all customers and their Phones in the most optimal way. You also want to ensure that when the entities are retrieved, the object graph between Customer and Phone is maintained.

Solution: There are various ways to approach the above problem. First approach is to do eager loading by using Include operator. When we load customers, we can use Include operator to eagerly load all their phone numbers. This approach would ensure that object graph is maintained when entities are returned from Object Services. Additionally it would require only one single database trip where both customer and their phones are retrieved in a single database call. However the problem with Include approach is, Phone entity is the Many side of the association which means that when EF brings Customers with Many Phones in a single query, the data returned to the application would have customer information repeated. Although with Meta data on the client, EF will remove the redundancy but bringing redundant data on the network and the cost of processing in removing the redundant data may make this option not the best approach.

Another approach is to bring Customer and Phones in an anonymous type which would be efficient but you will lose the object graph. Since Phones won't be tied to Customers, you will have to attach the Phones to their respective customers.

The last approach is to load Customers and Phones in a separate query which would lead to two separate but clean queries that simply return data from a Customer and Phone table. When entity framework brings Phone, it will also bring back the relationships for anything that is a reference and store that as stubs in ObjectStateManager. In our case those stubs would have relationship

info for Customer and Phone but Customer end of the relationship would not point to any customer. When we iterate the Phones entity either using for each or ToList operator, EF will replace the stub with a full Customer entity because it will see that the stub points to a Customer that is actually present in the state manager and will fix the relationship between Customer and Phone as if they were retrieved together. Code below shows an example of using relationship span.

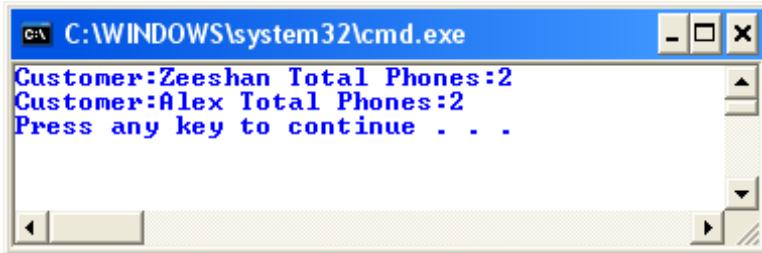
```
var db = new RsCustomerEntities();
    var customer1 = new Customer
    {
        Name = "Zeeshan",
        Phones =
        {
            new Phone{Number = "123-455-9876"} ,
            new Phone{Number = "123-455-9877"}
        }
    };

    var customer2 = new Customer
    {
        Name = "Alex",
        Phones =
        {
            new Phone{Number = "123-455-9878"} ,
            new Phone{Number = "123-455-9879"}
        }
    };
db.AddToCustomers(customer1);
db.AddToCustomers(customer2);
db.SaveChanges();
```

The code above creates two customer and assigns two phones to each customer. To retrieve all customers and their phone numbers we can separately query for customers and their phones. When we call toList on Phones EF will automatically fix the relationship between customer and Phones by escalating the customer stub to a full customer entity. Code below shows the relationship span being applied.

```
var customers = db.Customers.ToList();
    var phones = db.Phones.ToList();
    foreach (var customer in customers)
    {
        Console.WriteLine("Customer:{0} Total
Phones:{1}",customer.Name, customer.Phones.Count());
    }
```

To confirm the phones are attached to the customer, I am looping through the customer and printing the Total phones for a customer. Figure below shows the result on the console window.



```
C:\WINDOWS\system32\cmd.exe
Customer:Zeeshan Total Phones:2
Customer:Alex Total Phones:2
Press any key to continue . . .
```

Discussion: To fix relationship, EF transparently rewrites the query to bring relationship info for all relations which has multiplicity of 0..1 or 1 on the other end; in other words navigation properties that are entity reference. If an entity has relationship with multiplicity of greater than 1, EF will not bring back the relationship info because it could be performance hit and as compared to bringing a single foreign along with rest of the record. Bringing relationship info means retrieving all the foreign keys the records has. In the phone scenario, customerId foreign key resides in the Phone table so it is not expensive to bring that information along with rest of the phone table. When object services see the CustomerId foreign key, it will create relationship info record in state manager stating that Phone entity retrieved is associated to some customer that is not in the state manager. One end of the relationship would point to Phone entity and other would represent Customer entity that only contains customerId entity key with no customer entity. This partially filled state entry is called a stub. It is because of this stub that you can access the foreign key value without loading the related entity. For instance in the case of Phone, we can do this

```
Phone.CustomerReference.CustomerId
```

The above code allows us to return customerid even though the Customer instance is null because there is a stub entry created for the relationship between phone and customer entity.

So when does stub entries get upgraded to real entities. It could happen either when you are retrieve entities using query or attach an entity to the ObjectContext or load an entity using Load operator. All those options trigger

EF to check in the state manager for a stub and if found replace it with a full entity causing entire object graph to be fixed. So in our case when we loaded customers, EF found stubs for customer entity created by loading of Phone entities and replaced it with full customer. Another important fact to take away is the order in which the entities are loaded is not important. For instance in the above case we loaded phones followed by customer entity. We could have loaded customers first and then Phones. When phones are loaded with their relationship info, it would check the state manager to for the customer end of the relationship info. If customer entity is found, the graph would be hooked and relationship will be fixed.

Having the relationships fixed automatically opens interesting scenarios in disconnected environment. For instance when you use entities with web service or WCF, EF serializes the stubs along with an entity reference. This means that you can send Customer and Phones entity separately to client tier and when these entities are attached, a relationship stub entry would be created between Customer and Phone and the graph would fixed to reflect the correct relationship.

Another important use case of Relationship span comes when we try to delete a phone entity. Since Phone entity brings along the relationship info, EF will ensure that when a phone is deleted, the relationship between Customer and Phone is also deleted. When the relationship info entry is deleted from the state manager, Customer entity will reflect the correct phones in its collection as well. Currently this is not possible with linq to sql because it does not make relationship a first class citizen as EF does. Code below shows an example where marking the phones for deletion causes Phone count for customer to reach 0 indicating that customer no longer has any phones.

```
var db = new RsCustomerEntities();
    var phones = db.Phones.ToList();
    var customers = db.Customers.ToList();
    foreach (var phone in db.Phones)
    {
        db.DeleteObject(phone);
    }
    foreach (var customer in customers)
    {
```

```

        Console.WriteLine("Customer:{0} Total
Phones:{1}", customer.Name, customer.Phones.Count());
    }
}

```

```

C:\ C:\WINDOWS\system32\cmd.exe
Customer:Zeeshan Total Phones:0
Customer:Alex Total Phones:0
Press any key to continue . . .

```

Code above confirms that as soon as we marked all phones for deletion, EF removed the association between the customer and the phone causing the Customer's phone collection to reach zero.

When a collection is loaded using relationship span there is a possibility that only part of the collection may have been loaded. So in our phone example how we identify if customer's phone collection contains all the phones for the customer defined on the database or it contains only some of the phones? To better understand the problem code below shows an example.

```

var db = new RsCustomerEntities();
    var customers = db.Customers.ToList();
    var phones = db.Phones.Where("it.Number like '%455%'").ToList();
    foreach (var customer in customers)
    {
        Console.WriteLine("Customer:{0} Total Phones:{1}",
customer.Name, customer.Phones.Count());
    }
}

```

```

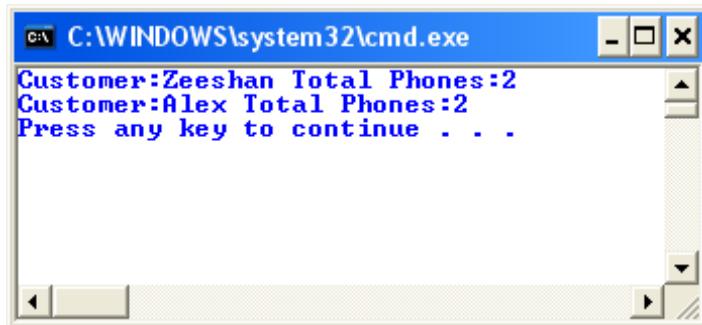
C:\ C:\WINDOWS\system32\cmd.exe
Customer:Zeeshan Total Phones:1
Customer:Alex Total Phones:1
Press any key to continue . . .

```

On the code above, I am retrieving phone based on a query. The query returns a single phone for each customer and when we iterate through the customer and print the total phones count for each customer, the result is 1. The result is certainly not true because both customers have two phones. To differentiate if

the customer's phone collection was loaded as result of fixing the object graph using relationship span or explicitly loading of the each customer's phone we can check IsLoaded property on the Phone collection. If the customer's Phone collection was not loaded explicitly then IsLoaded property would return false. If that is the case we can fetch all phones for the customer by calling Load on the Phone collection. Code below uses IsLoaded property to check if the phone collection was populated using relationship span, then explicitly call load to ensure that we have all the phones for a customer.

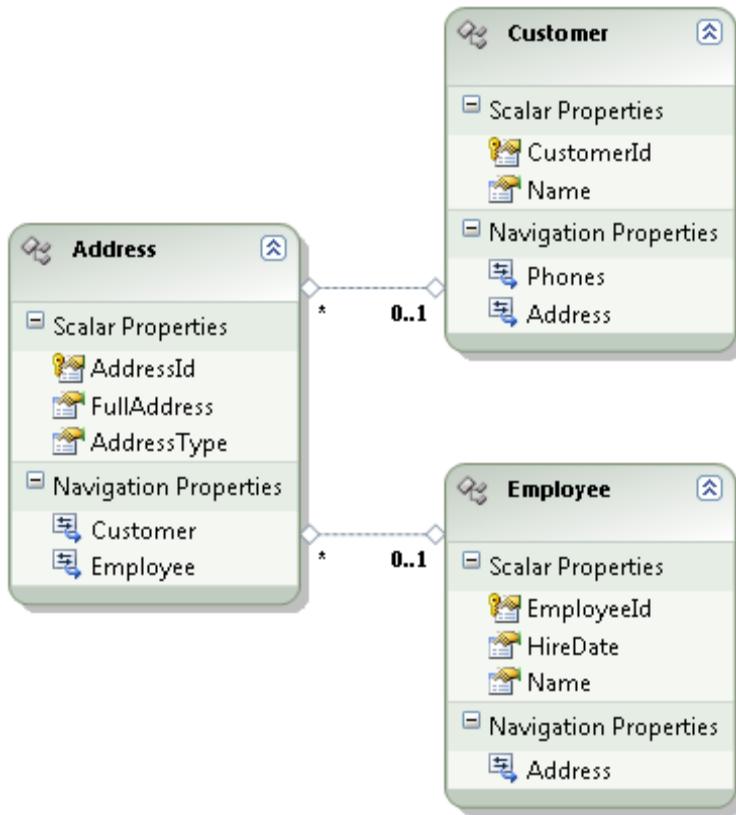
```
var db = new RsCustomerEntities();
    var customers = db.Customers.ToList();
    var phones = db.Phones.Where("it.Number like '%455%'").ToList();
    foreach (var customer in customers)
    {
        if (!customer.Phones.IsLoaded)
        {
            customer.Phones.Load();
        }
        Console.WriteLine("Customer:{0} Total Phones:{1}",
customer.Name, customer.Phones.Count());
    }
```



After calling Load, we can see in the console window that each customer has the correct phones filled.

3.4.2 Preventing Relationship span by using MergeOption.NoTracking

Problem: Figure below shows the edm model for Customer and Employee address



When a query is written to retrieve address entity, EF generates the following query.

```

SELECT
1 AS [C1],
[Extent1].[AddressId] AS [AddressId],
[Extent1].[FullAddress] AS [FullAddress],
[Extent1].[AddressType] AS [AddressType],
[Extent2].[CustomerId] AS [CustomerId],
[Extent3].[EmployeeId] AS [EmployeeId]
FROM [rs].[Address] AS [Extent1]
LEFT OUTER JOIN [rs].[CustomerAddress] AS [Extent2] ON [Extent1].[AddressId] = [Extent2].[AddressId]
LEFT OUTER JOIN [rs].[EmployeeAddress] AS [Extent3] ON [Extent1].[AddressId] = [Extent3].[AddressId]

```

Notice the query joins against CustomerAddress and EmployeeAddress table although we only requested to retrieve address information from address table. Why EF added additional joins? And how can we prevent these additional joins from happening.

Solution: As we had discussed in above examples, EF has an automatic query re-write feature where queries that retrieves entities under the covers also bring relationship information for all related entities that are entity references. On the above model Address has many Many to 0..1 relationship with both customer and Employee, so when querying for an address, EF will also bring along the foreign keys for CustomerId and EmployeeId. In a case when foreign keys are located in the same table like Address, the cost of bringing the additional stuff can be ignored. Since our tables are normalized and Employee and Address are related to each other by EmployeeAddress table, EF has to hop to another table to grab the CustomerId. Similarly to grab CustomerId EF had to join against CustomerAddress table to grab the CustomerId. So how can this behavior impact performance when it comes to simply wanting to retrieve an address entity and modifying its properties? Imagine Address entity had many to 1 association with several other entities. In that case, EF will create a left outer join with several related entities for really no reason when all you wanted to do was retrieve an entity and modify some properties. If bringing relationship info is not the desired behavior you expect, you can execute the query with MergeOption.NoTracking to indicate that Address entity does not need to be tracked and EF will not bring relationship info by joining against related tables. When you bring an address entity using NoTracking then we cannot modify address entity's properties because entity is not tracked by state manager. We additionally need to attach the entity to state manager before making a change. Code below shows an example of avoiding relationship span feature which could degrade query performance.

```
var db = new RsCustomerEntities();
    var addressquery = db.CreateQuery<Address>("select value top(1) a
from RsCustomerEntities.Addresses as a");
    var address =
addressquery.Execute(System.Data.Objects.MergeOption.NoTracking).First();
    Console.WriteLine(address.FullAddress);
    //in order to modify the address it needs to be tracked so
attached the address entity'
    db.Attach(address);
    address.FullAddress = "address changed";
```

```
db.SaveChanges();
```

On the above code, we are retrieving the top 1 address from the database using NoTracking option. This tells entity framework that since address entity is not going to be tracked there is no reason to bring additional relationship information with the query. Code below shows the sql capture for the above NoTracking query which only queries against address table to retrieve the first address.

```
SELECT TOP (1)
[c].[AddressId] AS [AddressId],
[c].[FullAddress] AS [FullAddress],
[c].[AddressType] AS [AddressType]
FROM [rs].[Address] AS [c]
```

Since we also want to modify the address, we are attaching the address entity to state manager so it can begin tracking any changes we apply to address entity properties.

Note: If you have defined an entity on a model that has Many to 1 association with several entities, then EF will rewrite the query to bring all relationship info for all related entities that are entity references. In doing so it may require joining against several tables which could degrade the query performance. If the end goal is to modify entity properties then you can retrieve the entity with NoTracking option, attach the entity to the ObjectContext and apply property changes on the entity.

4. Views

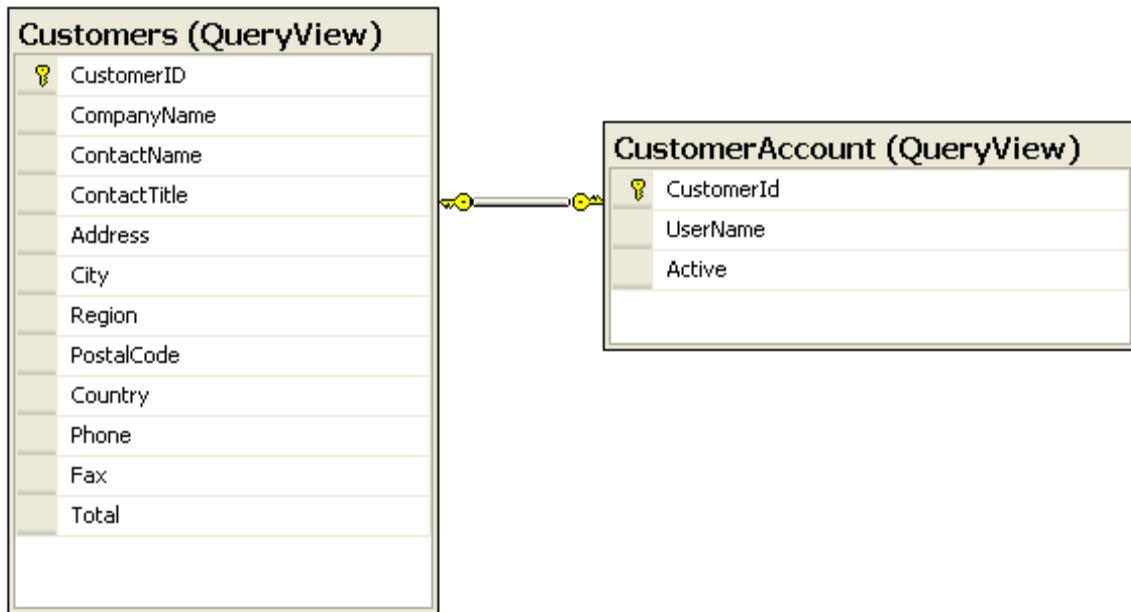
4.1 QueryView

QueryView is read-only mapping between conceptual model and storage model. You would use Query View when default mapping from the entity framework does not meet your need. For instance you want to expose an association that is not based primary and foreign key columns. If you want to

apply default filter to the table like only displaying records no older than 5 years, then using QueryView would be a good option. When applying Query View, you lose the benefits of Inserts, Update and Delete offered by the framework because entity framework is not aware of the query written inside of QueryView. Query View is written in entity sql and can only query the tables defined on the storage model. In addition any entities related to the entity using the QueryView is also required to implement QueryView. This is an enforcement set forth by the entity framework because entity framework cannot apply validation rules to parts of an entity and determine if the model created is valid or not based on the store and mapping defined. When QueryView is defined it cannot contain property mapping because the query inside the QueryView would be responsible for reading data from store model and populating the conceptual model. Query can be used in two different places. QueryView can be used to define a view for an entity set by creating a QueryView element inside of EntitySetMapping. QueryView can also be used to define a view for an association meaning how entities are related to each other. One common use of using QueryView for AssociationSetMapping is to support inheritance based on arbitrary conditions which entity framework does not provide. Currently inheritance can be implemented on condition column where the condition could be null or equal to. QueryView can be used to define greater then less or both criteria for inheritance.

4.1.1 Using QueryView To exclude columns and add computed columns

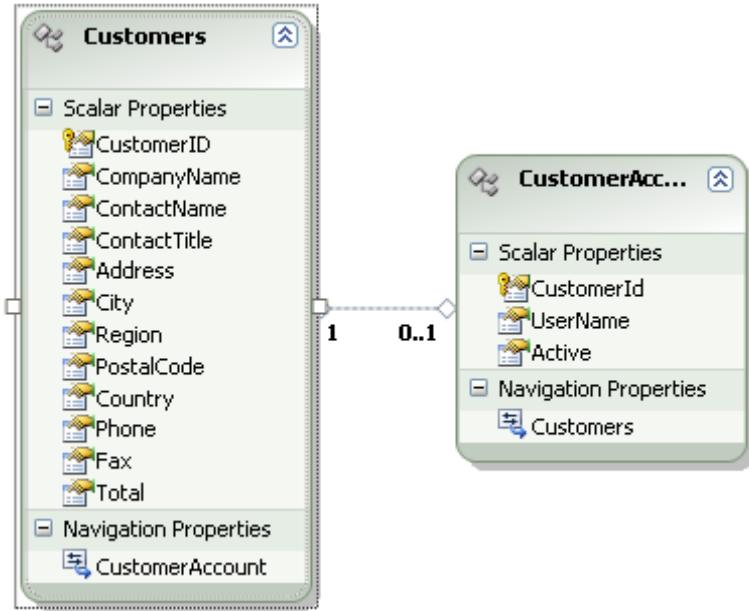
Problem: You have customer and account data in two different tables called Customer and CustomerAccount as shown below



Customer and CustomerAccount are joined by 1 to 1 relationship. You want to expose both tables as a single entity in the EDM. In addition there are additional columns on Customer table that have grown over the period of time that you are no longer interested in. You do not want these columns to surface on the entity data model. On your entity data model, you want to have an additional computed column that says if the Customer has an account or not. You want to know how to create an entity data model that validates against the given schema above.

Solution: You can use a queryview that can query the ssdl definition to join both the tables and only project columns which you have defined on the entity. To populate a computed column HasAccount, you can do a left join and check to see if CustomerAccount is not null. If CustomerAccount is null means HasAccount should be set to False.

Discussion: Customer and CustomerAccount table in the database are joined based on primary key and therefore have 1 to 1 relationship with each other. When you use the import wizard to import the database model into edm, your model looks as follows.



In the above screen shot, Customer has 0-1 association with CustomerAccount because a customer may not have an account with us. Next we need to remove fields we don't want on our Customer entity and move the fields we need from CustmoerAccount entity and then delete the CustomerAccount entity from our conceptual model as data from both tables will be expressed with a single entity. Figure below shows how the updated Customer entity looks like.



On the above entity I have made UserName property as null because UserName would not exist for a customer that does not have an account. I also removed the entity table mapping done by designer by opening the mapping window and removing the mapping.

Since we deleted the table mapping, we will use QueryView to map the Customer entity to our store definition. QueryView is not a designer supported feature in version 1 so you have to modify the msl and write the query in there. QueryView uses esql to query the ssdl model instead of database. You have to be aware that you cannot use all the esql operators inside of a query view. For instance to join CustomerAccount and Customer table, I tried to use navigate operator to navigate from one entity to another instead of doing an explicit join and entity framework complained that navigate operator is not allowed. This validation is a good thing because when you modify the msl manually, entity framework tries to parse the query view to see if it is valid against the stored model or not. So despite no designer support, you get validation for entities being referenced in correct namespaces and syntax checking. All these errors show up on the error list when you try to validate the model. To use QueryView, I have to say that data for my Customers EntitySet comes from an esql query. The esql query is shown below.

```

<EntityContainerMapping StorageEntityContainer="QueryViewModelStoreContainer"
CdmEntityContainer="Entities">
    <EntitySetName Mapping="Customers">
        <QueryView>
            select value
QueryViewModel.Customer(c.CustomerID,c.CompanyName,c.ContactName,a.UserName,
            case when a is null then False
            else True
            end
        )
        from QueryViewModelStoreContainer.Customers as c
        left join QueryViewModelStoreContainer.CustomerAccount as a on
c.CustomerID = a.CustomerID
    </QueryView>
</EntitySetName>
</EntityContainerMapping>

```

On the from clause of the query I am using `QueryViewModelStoreContainer` which is the storage entity container name as shown above and `CustomerAccount` is the entity defined on my store model. So in order to access any entity defined on the store model, you have to prefix the entity with `StorageEntityContainer`. In the query, I am doing a left join with customer account table on `CustomerId` column and then projecting the result into `Customer` Entity defined on my conceptual model. The conceptual entity must be referenced by the namespace of the conceptual model which in our case is `QueryViewModel`. I also have a case

statement which checks to see if a customer has a customer account and the result of case statement evaluation is assigned to HasAccount property on Customer entity. Code below shows how to query the model we just created.

```
var db = new QueryViewEntities();
    var custs = db.Customers.Where(c => c.CustomerID == "ALFKI" ||
c.CustomerID == "HANAR");
    Console.WriteLine("ID\t CompanyName\t\t UserName \t HasAccount");
    foreach (var cus in custs)
    {
        Console.WriteLine("{0}\t {1}\t {2}\t {3}",
            cus.CustomerID, cus.CompanyName, cus.UserName == null ?
"No UserName" : cus.UserName, cus.HasAccount);
        Console.WriteLine();
    }
```

The above code queries for two customers, one which has a customer account and other one do not. Screen shot below confirms the result of the query for HANAR which does not have an account.

ID	CompanyName	UserName	HasAccount
ALFKI	Alfreds Futterkiste	zhirani	True
HANAR	Hanari Carnes	No UserName	False

Note: Since queryviews are readonly, there is no support for insert, update and delete of an entity by entity framework. You are required to create stored procedures for each entity and associations need to be saved to the database. The mapping of the stored procedures also has to be done manually by editing edmx file because in the current version of the designer does not support mapping stored procedures to entities that use queryviews.

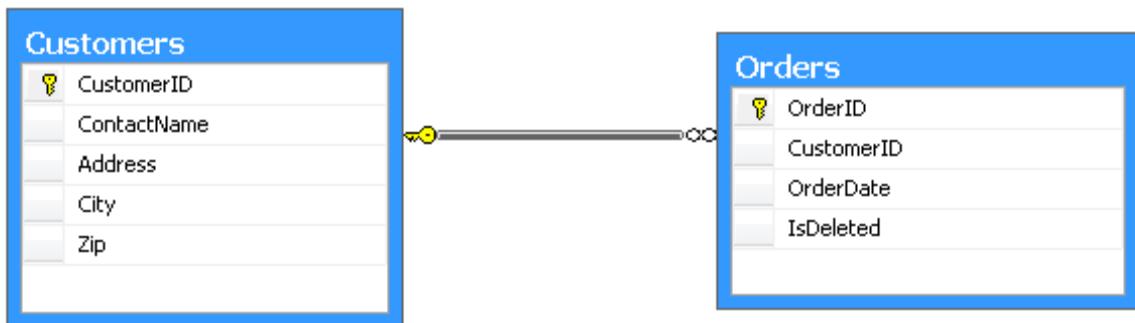
4.1.2 Using QueryView to filter collection

Problem: You have customers and Orders defined on the database. Recently the application was retrieving all the orders for a given customer but new

business rules states that application should only retrieve orders for customers that were placed after 1998. Any orders place before 1998 year should not be fetched. In addition any orders that are marked as deleted should not be retrieved. You are told to implement this logic across the entire application.

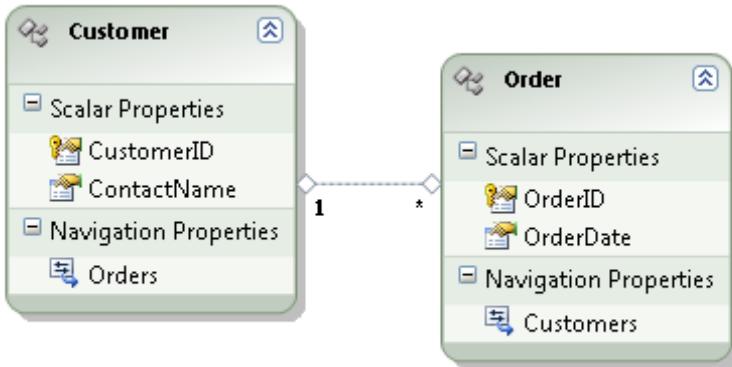
Solution: Although we could write a query that would filter the orders to the requirement stated above. However that would require fixing the orders query at numerous places and may lead to bugs. Instead we can modify the Orders ObjectQuery exposed on the Objectcontext so that anytime Orders entitycollection is accessed; the filter is already applied on it. The best way to do is to remove the entityset mapping which is mapped to a table. Instead Orders EntitySet should be mapped to a custom query that filters the orders to remove deleted orders and orders place before 1998. If there are entities associated with Orders, they will also have to rewritten to use QueryView even though their implementation does not change. This is a requirement imposed by the entity framework because it cannot confirm the graph consistency and provide validation on part of the model that uses default mapping with the rest which uses QueryView.

Discussion: The diagram below shows how customer and orders are related to each other.



To apply filter on Orders we need to define a QueryView which will filter the orders. Since Orders are related to customer, we also need to define QueryView for Customers and the association between customer and orders. When we import the tables shown above using Update Model Wizard into

EDM, we get default mappings of entities to tables. Screenshot below show how the entity data model looks like.



Since we are going to be using custom mapping we need to remove the Table mapping for both Customer and Order from mapping window. Since QueryView is not supported by the designer, we will modify the EntitySet mapping for Customers Orders and the association between customer and Orders. Example below shows the mapping for Customers entityset.

```

<EntitySetName="Customers">
    <QueryView>
        select value
    QueryWithFilterModel.Customer(c.CustomerID,c.ContactName)
        from
    QueryWithFilterModelStoreContainer.Customers as c
    </QueryView>
</EntitySetName>
  
```

Since customers are related to Orders, we are forced to provide mapping for customers. The esql query above retrieves customers from the store model and sets the value for Customer entity. QueryWithFilterModel is the name of the conceptual model namespace and QueryWithFilterModelStoreContainer is the StorageEntityContainer name.

The QueryView for Orders EntitySet looks like this

```

<EntitySetName="Orders">
    <QueryView>
        select value
    QueryWithFilterModel.Order(o.OrderID,o.OrderDate)
  
```

```

        from
QueryWithFilterModelStoreContainer.Orders as o
            where !o.IsDeleted and o.OrderDate >=
cast('1998/1/1' as Edm.DateTime)
        </QueryView>
        </EntitySetMapping>

```

On the above model, I am querying the store model for only orders that are not deleted and which have an orderdate greater than the date specified in the query. Notice to cast my string as date, I am using cast operator available on esql and specifying the type to be Edm.DateTime.

Next QueryView we need to define is the association between Customers and Orders which is define on the conceptual model like this

```

<Association Name="FK_Orders_Customers">
    <End Role="Customers"
Type="QueryWithFilterModel.Customers" Multiplicity="1" />
    <End Role="Orders" Type="QueryWithFilterModel.Orders"
Multiplicity="*" />
</Association>

```

The association requires the first parameter to be customer entity key followed by Orders entity key. Example below shows the query required create the association defined above between customer and Orders.

```

<AssociationSetMapping Name="FK_Orders_Customers">
    <QueryView>
        select value
QueryWithFilterModel.FK_Orders_Customers(
    createref(QueryViewWithFilter.Customers, row(o.CustomerID)) ,
    createref(QueryViewWithFilter.Orders, row(o.OrderID))
)
from
QueryWithFilterModelStoreContainer.Orders as o
    </QueryView>
    </AssociationSetMapping>

```

On the above associationSetMapping, I am retrieving Orders from my storage model and then using createref method exposed on esql I am retrieving Customer entity key followed by Orders entity key. Using those keys I populate the association defined on the conceptual model with

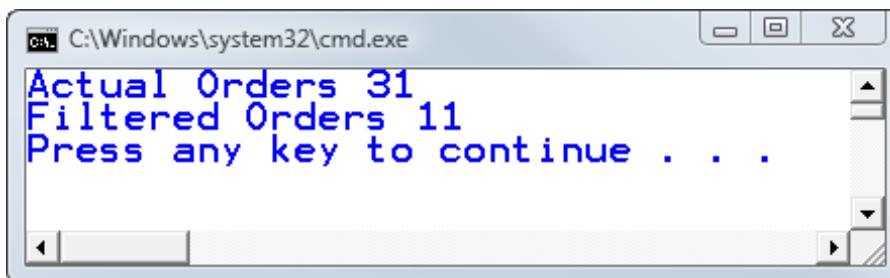
customer and order entity keys. It is important to explain some of prefixes uses with entities. Table below shows the prefixes used with the entity and how are they mapped with entity framework.

<code>StorageEntityContainer</code>	<code>QueryWithFilterModelStoreContainer</code>
<code>CdmEntityContainer</code>	<code>QueryViewWithFilter</code>
<code>conceptualModel Namespace</code>	<code>QueryWithFilterModel</code>

To confirm that on fetching orders for a customer, we do not retrieve all the orders, we can write a query that returns all the orders for a customer. In the example below I am using a regular sql query to get the count of the orders placed by SAVEA customer and then using linq, I am querying my conceptual model which applies my QueryView to only retrieve orders that meet our query criteria. Screen shot below shows our result.

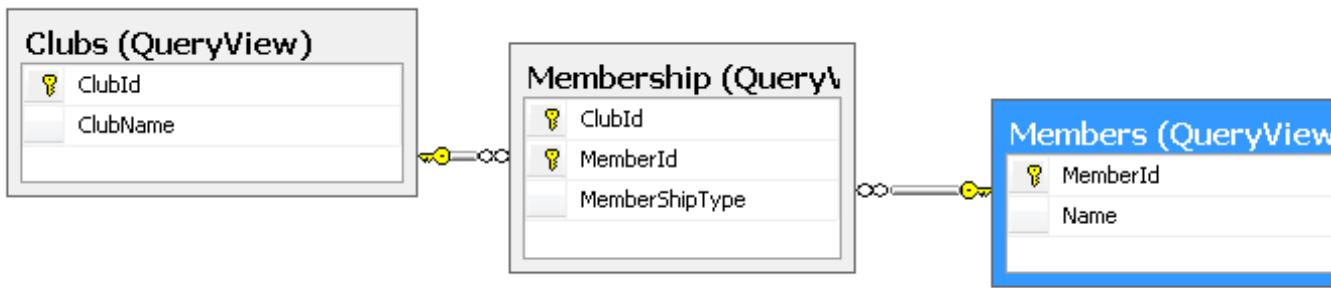
```
var db = new QueryViewWithFilter.QueryViewWithFilter();
        var cmd = db.CreateStoreCommand("SELECT COUNT(*) 
FROM ORDERS WHERE CUSTOMERID = 'SAVEA' ");
cmd.Connection.Open();
        var totalorders =
Convert.ToInt32(cmd.ExecuteScalar());
cmd.Connection.Close();

        var cus = db.Customers.Include("Orders").First(c =>
c.CustomerID == "SAVEA");
Console.WriteLine("Actual Orders " + totalorders);
        Console.WriteLine("Filtered Orders " +
cus.Orders.Count());
```



4.1.3 QueryView to map Many to Many Relationship with PlayLoad

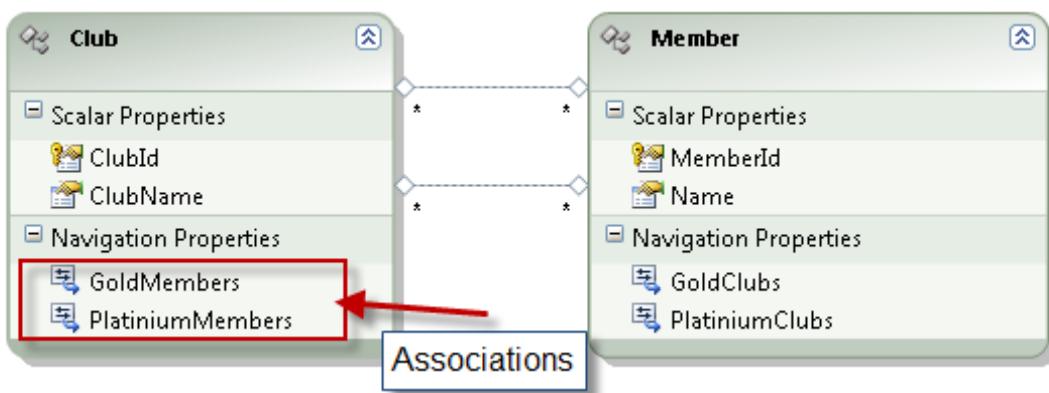
Problem: You have defined three tables in the database called Clubs, Members and Membership. A club can have many members and member can be part of many clubs. To map clubs to members, we have created a link table Membership which defines the many to Many association. The database diagram is shown below.



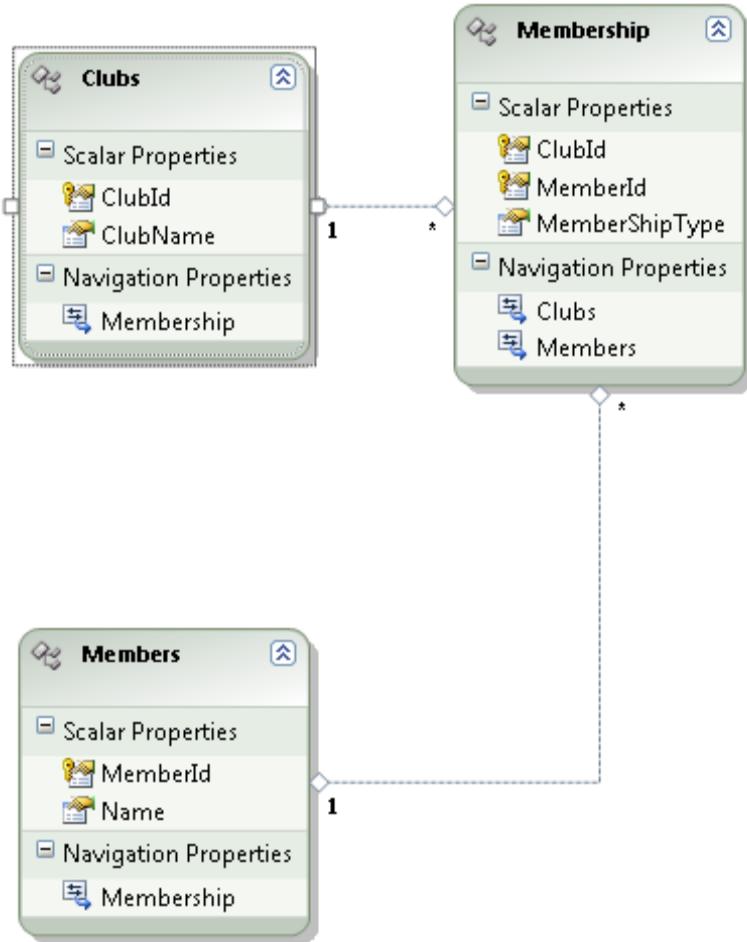
In addition to the primary key field from Clubs and Members, Membership table also contains an additional column MemberShipType that defines what kind of membership the member holds. Membership type could be of two types; Gold and Platinum. You want to expose the database relationship as Many to Many association between Club and Members. There should be two types of associations between Clubs and Members. First association should expose a navigation property GoldMembers entity collection and second association should expose a navigation property PlatiniumMembers. GoldMembers should only retrieve GoldMembers from the membership table and PlatiniumMembers should only retrieve Members who have platinum membership with the club. In addition you want to be able to insert and update and delete different types of members.

Solution: When we import the Many to Many relationships with payload column like MembershipType, entity framework does not get rid of the link table. To access the members for a given club, you have to traverse Membership table to access the members. What we need is two direct associations between Clubs and Members. First association GoldMembers will return GoldMembers of the club and PlatiniumMembers association would return PlatiniumMembers of the club. To map a single Many to Many relationship with payload as two Many to Many relationship is not directly supported by the entity framework or the designer. To achieve this

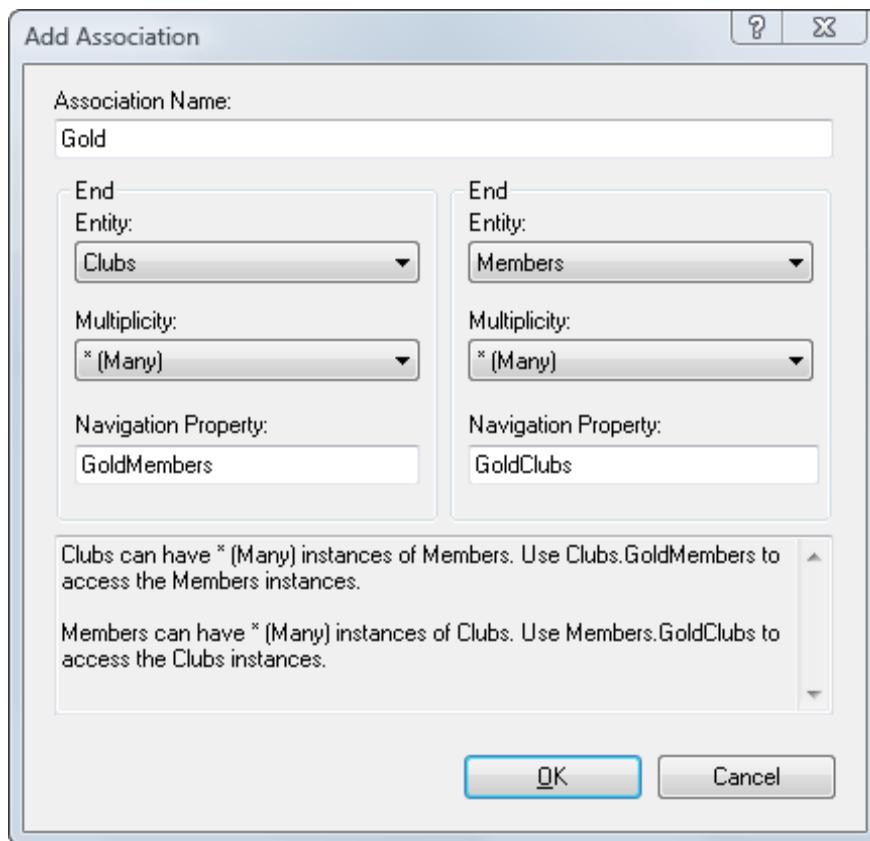
relationship we have to first remove the Membership link entity and create two associations between clubs and members using QueryView. First association would return results from link table member where membership type is Gold and second association would return rows from Membership table where membership Type is Platinum. When we use QueryView with associations, we cannot use build in support to insert, update and delete Members and clubs. We also need to specify QueryView for both Clubs and Members entity. The final EDM is shown below.



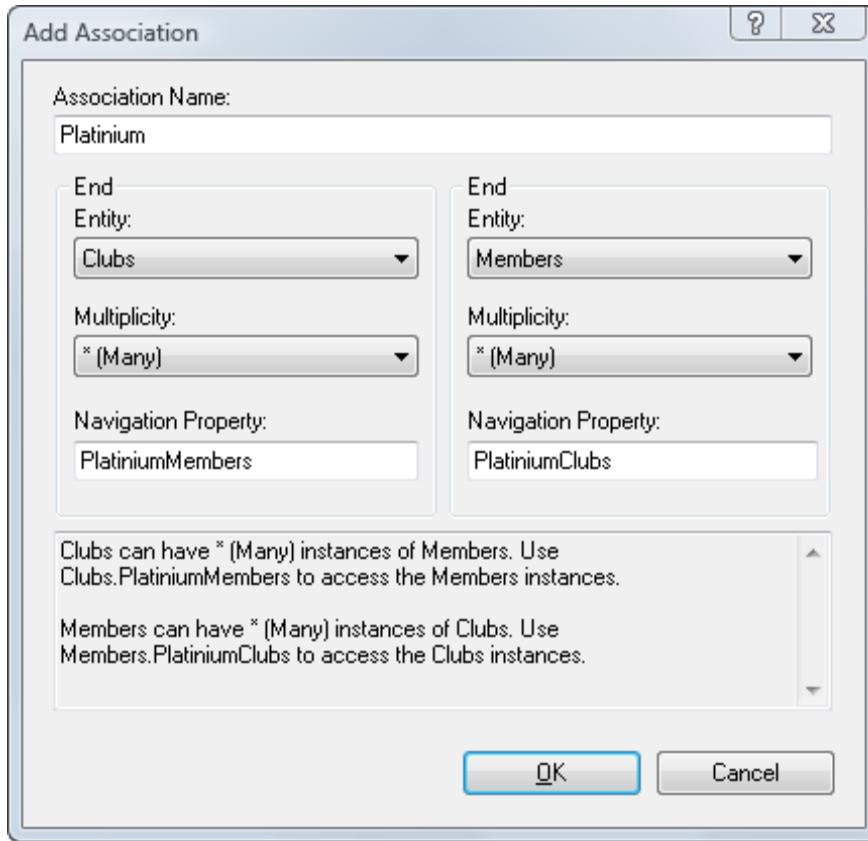
Solution: To map our relational model to entity model shown above, we will use the import wizard to get head start which will help us generate SSDL, CSDL and MSL. Screen shot below shows the model as it looks like when we use the import wizard to import Clubs, Membership and Members table.



We will first delete **Membership** entity because we want to have a direct Many to Many association between club and Member. To create association returning **GoldMembers**, right click on **Club** entity and select Association. On the screen shot below, we have selected **Members** with a multiplicity of Many and **Clubs** with multiplicity of Many. We have given the association name **Gold** and use **GoldMembers** navigation property to access **GoldMembers** for the club.



Next we need to create an association for Platinum members for the club. We use the similar step to create PlatiniumMembers association between Club and Members entity. On the screen shot below we have set Many multiplicity for Club and Members and use PlatiniumMembers navigation property to access Platinum members for the club.



To map our two associations, we will use QueryView to extract appropriate data from Membership link table. Using QueryView requires that all entities to also use queryview and cannot use the default mapping from the entity framework. Therefore we also need to remove the Club and Membership association created by the import wizard by selecting the entity, choose the mapping window and delete the mapping.

The QueryView for member queries the store model and map the results to member entity. After defining the QueryView, we do not get any support for insert, update and delete for member entity. Therefore we need to create stored procedures and map the stored procedure's parameter to properties on the entity. The msl below shows mapping required for member entity.

MSL

```
<EntitySetName="Members">
    <QueryView>
        select value
    QueryViewWithManyToMany.Member(m.MemberId, m.Name)
```

```

        from
QueryViewWithManyToManyStoreContainer.Members as m
            </QueryView>
            <EntityTypeMapping
TypeName="QueryViewWithManyToMany.Member">
                <ModificationFunctionMapping>
                    <InsertFunction
FunctionName="QueryViewWithManyToMany.Store.InsertMember">
                        <ScalarProperty Name="Name">
                            ParameterName="Name" />
                                <ResultBinding
Name="MemberId" ColumnName="MemberId" />
                            </InsertFunction>
                            <UpdateFunction
FunctionName="QueryViewWithManyToMany.Store.UpdateMember">
                                <ScalarProperty Name="Name">
                            ParameterName="Name" Version="Current"/>
                                <ScalarProperty
Name="MemberId" ParameterName="MemberId" Version="Current" />
                            </UpdateFunction>
                            <DeleteFunction
FunctionName="QueryViewWithManyToMany.Store.DeleteMember">
                                <ScalarProperty
Name="MemberId" ParameterName="MemberId" />
                            </DeleteFunction>
                        </ModificationFunctionMapping>
                    </EntityTypeMapping>
                </EntitySetMapping>

```

To define the stored procedure called inside MSL, we can either import the stored procedure from database using the designer or declare the store procedure right inside the command text property of function declaration inside the ssdl. SSDL shown below uses the second option by defining the code for the stored procedure inside the CommandText property. This is for demonstration purpose or if u do not have permissions to create database objects on the database. If you have the appropriate permissions, it would be preferable to create stored procedure, this way you would get compilation for the stored procedure and benefit the compile time check offered by the database engine. SSDL below shows function declaration need to insert, update and delete member entity.

```

<Function Name="InsertMember"      IsComposable="false"
Schema="dbo">
    <CommandText>

```

```

        insert into QueryView.Members(Name) values
(@Name)
            select SCOPE_IDENTITY() as MemberId
</CommandText>
<Parameter Name="Name" Type="varchar" Mode="In">
/>
</Function>
<Function Name="UpdateMember"      IsComposable="false"
Schema="dbo">
    <CommandText>
        update QueryView.Members set Name = @Name
    </CommandText>
    <Parameter Name="Name" Type="varchar" Mode="In">
/>
    <Parameter Name="MemberId" Type="int" Mode="In">
/>
    </Function>
    <Function Name="DeleteMember"      IsComposable="false"
Schema="dbo">
        <CommandText>
            delete QueryView.Members where MemberId =
@MemberId
        </CommandText>
        <Parameter Name="MemberId" Type="int" Mode="In">
/>
    </Function>

```

For insert stored procedure, MemberId is returned by using scope identity which returns the id of the last record inserted. The memberid returned is mapped to MemberId on member entity using ResultBinding. Similary defining QueryView for Club and mapping the stored procedure on the ssdl to properties defined on Club entity is very similar to Member entity. For completeness MSL and SSDL below shows the mapping required.

MSL

```

<EntitySetMapping Name="Clubs">
    <QueryView>
        select value
QueryViewWithManyToMany.Club(c.ClubId,c.ClubName)
        from
QueryViewWithManyToManyStoreContainer.Clubs as c
    </QueryView>
    <EntityTypeMapping
TypeName="QueryViewWithManyToMany.Club">

```

```

        <ModificationFunctionMapping>
            <InsertFunction
                FunctionName="QueryViewWithManyToMany.Store.InsertClub">
                    <ScalarProperty
                        Name="ClubName" ParameterName="ClubName" />
                    <ResultBinding Name="ClubId" ColumnName="ClubId" />
            </InsertFunction>
            <UpdateFunction
                FunctionName="QueryViewWithManyToMany.Store.UpdateClub">
                    <ScalarProperty
                        Name="ClubName" ParameterName="ClubName" Version="Current" />
                    <ScalarProperty Name="ClubId" ParameterName="ClubId" Version="Current" />
            </UpdateFunction>
            <DeleteFunction
                FunctionName="QueryViewWithManyToMany.Store.DeleteClub">
                    <ScalarProperty Name="ClubId" ParameterName="ClubId" />
            </DeleteFunction>
        </ModificationFunctionMapping>
    </EntityTypeMapping>
</EntitySetMapping>

```

SSDL

```

<Function Name="InsertClub" IsComposable="false"
Schema="dbo">
    <CommandText>
        insert into QueryView.Clubs(ClubName) values
(@ClubName)
        select SCOPE_IDENTITY() as ClubId
    </CommandText>
    <Parameter Name="ClubName" Type="varchar"
Mode="In" />
    </Function>
    <Function Name="UpdateClub" IsComposable="false"
Schema="dbo">
        <CommandText>
            update QueryView.Clubs set ClubName =
@ClubName
        </CommandText>
        <Parameter Name="ClubName" Type="varchar"
Mode="In" />
        <Parameter Name="ClubId" Type="int" Mode="In" />
    </Function>

```

```

<Function Name="DeleteClub"      IsComposable="false"
Schema="dbo " >
    <CommandText>
        delete QueryView.Clubs where ClubId =
@ClubId
    </CommandText>
    <Parameter Name="ClubId" Type="int" Mode="In" />
</Function>

```

To configure the two many to many associations defined on the conceptual model, we need to use QueryView to only retrieve data that is applicable for the association. For instance for ClubMembers we need to only retrieve relationship records where MemberShipType is G for Gold members. Similarly for Platinum Members, only records with MembershipType of P are returned. MSL below shows the queryview required for GoldMember association since Gold and Platinum are similar in behavior.

```

<AssociationSetMapping TypeName="QueryViewWithManyToMany.Gold"
Name="Gold">
    <QueryView >
        select value
    QueryViewWithManyToMany.Gold(
        createref(QueryViewMM.Clubs, row(m.ClubId)),
        createref(QueryViewMM.Members, row(m.MemberId))
    )
    from
    QueryViewWithManyToManyStoreContainer.Membership as m
    where m.MemberShipType = 'G'
    </QueryView>
    <ModificationFunctionMapping>
        <InsertFunction
            FunctionName="QueryViewWithManyToMany.Store.InsertGoldMembership"
        >
            <EndProperty Name="Clubs" >
                <ScalarProperty Name="ClubId" ParameterName="ClubId" />
            </EndProperty>
            <EndProperty Name="Members" >
                <ScalarProperty Name="MemberId" ParameterName="MemberId" />
            </EndProperty>
        </InsertFunction>
    
```

```

        <DeleteFunction
    FunctionName="QueryViewWithManyToMany.Store.DeleteMembership">
        <EndProperty Name="Clubs">
            <ScalarProperty Name="ClubId">
ParameterName="ClubId" />
            </EndProperty>
        <EndProperty Name="Members">
            <ScalarProperty
Name="MemberId" ParameterName="MemberId" />
            </EndProperty>
        </DeleteFunction>
    </ModificationFunctionMapping>
</AssociationSetMapping>

```

On the above association mapping, I am specifying the Name gold which is defined on my conceptual model along with the TypeName for the association. The query view for Gold members queries the Membership table defined on the stored model filtering the records where MembershipType is G. The Gold association on the conceptual model requires two entity keys. The first entity key is the club and second is the member entity key. To obtain the entity key, we are using createref function available on esql that uses row function to return the entity key when passed the primary key of the table. The store procedure mapping grabs the entity key from both ends of the association. From the club end of the association, I am retrieving ClubId and mapping it to ClubId parameter on the stored procedure. For Members end of the association, I am retrieving the MemberId and mapping it to MemberId parameter on the stored procedure. The stored procedure mapping defined for the Gold association is as follow.

```

<Function Name="InsertGoldMembership" IsComposable="false"
Schema="dbo">
    <CommandText>
        insert into QueryView.Membership values
(@ClubId,@MemberId,'G')
    </CommandText>
    <Parameter Name="ClubId" Type="int" Mode="In" />
    <Parameter Name="MemberId" Type="int" Mode="In" />
/>
    </Function>
    <Function Name="DeleteMembership" IsComposable="false"
Schema="dbo">
        <CommandText>

```

```

        delete QueryView.Membership where ClubId =
@ClubId and MemberId = @MemberId
    </CommandText>
    <Parameter Name="ClubId" Type="int" />
    <Parameter Name="MemberId" Type="int" />
</Function>

```

For inserting GoldMembership, I am passing G as the third parameter to indicate that Membership inserted is of type Gold. We cannot reuse the same stored procedure for PlatinumMembership because the last parameter to the insert would be P instead of G. However Deleting Gold or Platinum membership is not any different so we can map the same delete stored procedure for both mappings.

After completing the mapping, we should have two entity collections exposed on a Club, GoldMembers and Platinum members. And we can add members to both collection and it would be inserted with appropriate MemberType in the Membership table. In the code below, I am creating a football club and assigning two Gold club members and one Platinum members and saving the club to the database. To make sure that clubs and members got inserted correctly, using the second datacontext, I am retrieving the Club and eagerly loading both Platinum and Gold members collection by using Include. To confirm both collections have the appropriate members, I am printing the results to output window. Screen shot below shows the output on the console window.

```

var db = new QueryViewMM();
    var club = new Club
    {
        ClubName = "Football Club",
        GoldMembers = {
            new Member{Name = "Scott"} ,
            new Member{Name = "Allen"} ,
        },
        PlatinumMembers = { new Member { Name = "Chris" }
    } ;
    db.AddToClubs(club);
    db.SaveChanges();

    var db2 = new QueryViewMM();
    var footballclub = db2.Clubs

```

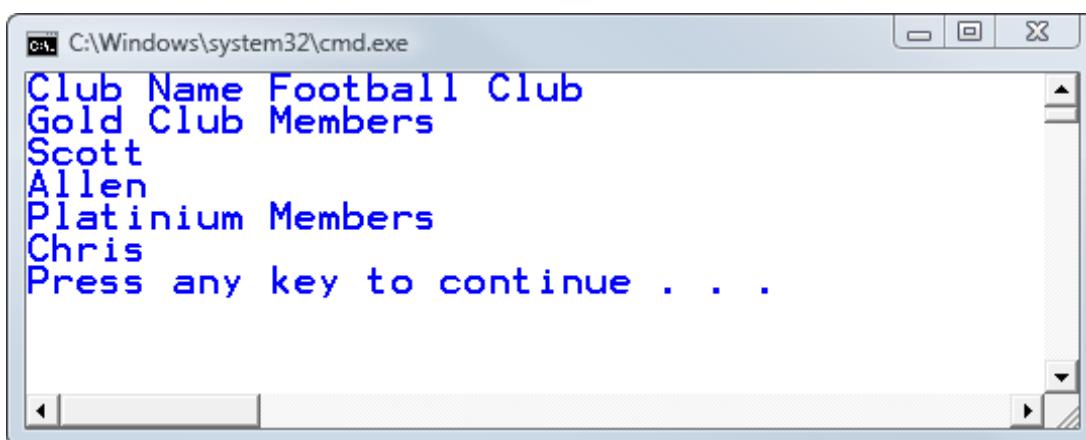
```

    .Include("PlatinumMembers").Include("GoldMembers")
        .First(c => c.ClubName ==
"Football Club");

        Console.WriteLine("Club Name " +
footballclub.ClubName);
        Console.WriteLine("Gold Club Members");
        foreach (var member in footballclub.GoldMembers)
        {
            Console.WriteLine(member.Name);
        }
        Console.WriteLine("Platinum Members");
        foreach (var member in
footballclub.PlatinumMembers)
        {
            Console.WriteLine(member.Name);

        }

```



4.2 DefiningQuery

Introduction

DefiningQuery is a query view that is defined on the store model. When a view is imported from the database using Entity model wizard, EF creates a definingQuery that does a select on the view. If you import a table that does

not have a primary key, EF will create a DefiningQuery that does a select on the table. In addition it will infer that all columns in the table participate in primary key and will mark all columns as keys on the store model. Similarly on the conceptual side, it will also make all properties as being entity keys. This behavior is very confusing because if you do not know that EF framework has mapped your table as a DefiningQuery, you will be clueless as to why your inserts, updates and deletes are failing on the conceptual entity. This brings us to a discussion of how EF handles crud operations on entities that are mapped using DefiningQuery. Since DefiningQuery is only a read-only view of data written in store specific syntax, EF does not have any understanding of how the entity is stored in the database. To save an entity that is mapped using DefiningQuery, we have to declare stored procedures on the store model and then map the stored procedures to insert, update and delete operations on the conceptual entity.

DefiningQueries allows you to use native sql syntax to create any complex projection and expose the projection as a view that an entity on the conceptual model can be mapped to. If there are modeling scenarios that you cannot accomplish using Ef because of the way the data is stored in the database, you can transform the data using DefiningQueries and project it in a way that is friendly with EF modeling scenarios. DefiningQueries does not support parameters so a sql written inside of DefiningQuery section cannot contain parameters that you specify a value for at runtime.

Entity framework also allows you to define a complex view using QueryView that uses esql. Unlike DefiningQueries, QuerViews are declared inside the mapping section and queries the store model to fetch its data. It is a preferred approach of defining complex queries. However if you cannot represent a projection using QueryView because it does not support all the esql operators, you should use DefiningQuery as the last resort. Some common use of DefiningQueries would be to use constructs that cannot be mapped directly either using esql or Linq. For instance to create recursive queries sql server provides Common table Expression which does not have any translation to either esql or Linq. To use recursive queries, we can use Common Table Expression inside of DefiningQuery and rest of conceptual model can simply use the view without knowing the underlying details of the store model. In

version 1 release of EF, there is no support for spatial data type. To overcome these limitations, we can create a DefiningQuery which brings spatial data type as image and then inside of the partial class we can transform the data into a Geographical data type which will allow us to perform domain specific activities like plotting a point on map. Basically DefiningQuery provides unlimited capabilities to exploit the data agnostic features which EF cannot leverage.

Another great use of DefiningQueries is to declare a complicated sql join inline. For maintainability perspective, it is good to define a view in the database and import it as DefiningQuery on the store model. However if you do not have permissions to create database objects, you can define your view inline inside of DefiningQuery and map it to entities on the conceptual model. In the version 1 release of EF, DefiningQueries are not supported by the designer. To create a definingQuery, you have to open the edmx file in the xml, find the entityset on the store model that you want its result to come from DefiningQuery and create a nested DefiningQuery section. Since there is no support for syntax checking, it is better to create the sql using ManagementStudio, test it and make sure it runs and then paste it inside of Defining Query section.

Entity framework does not support mapping tables that reside on different database. With DefiningQuery, you can create a view that joins tables across multiple databases and returns a view that entity framework can consume without knowing if the data is coming from multiple databases.

It is important to know that in verion1 release of EF, if you create a Defining Query in the ssdl and try to update the model from database, the DefiningQuery will be overwritten from the database and you may lose all the manual edits that you have applied like change the entity key from all columns to column which you think should be the key value for the view. Also if have not used a view to define a DefiningQuery and have written the sql inline, those edits will be lost.

4.2.1 Operators supported on QueryView

4.2.2 Mapping Foreign Key column to Multiple Associations Using DefiningQuery

Problem: Figure below shows the database diagram for a Gun and its promoters.



A gun show can only have a single promoter identified by PromoterId. However the PromoterId could be the Id column that represents ClubId in GunClubs table or ShootingRangeId in ShootingRange table. To identify the promoterId belongs to which table, there is a PromoterType field added on GunShow table that defines if the Id belongs to GunClubs or ShootingRange. You want to import the above model with GunShow having two different associations. One association would expose an entity ref GunClub and other association would expose a ShootingRange.

Solution: If we look at the above database model, we will realize that we are mapping PromoterId as a foreign key to multiple tables. But for a single GunShow it is either mapped to GunClub or ShootingRange not both. Entity Framework does not allow a single column to be mapped to multiple associations because that could corrupt the model. Although in our case it is

genuine mapping because only 1 mapping is valid at a given point. To get around this limitation we need to create a view that exposes PromoterId column on GunShow as two columns ShootingRangeId and ClubId. If the GunShow's promoter is GunClub , then ClubId will have a value but ShootingRangeId would be null. Similarly if GunShow's promoter is ShootingRange, then ShootingRangeId would have a value and ClubId would be null. After creating the view, import the GunShow view, GunClub and ShootingRange table. Create an association between GunShow and GunClub where a GunShow can have a single GunClub as a promoter. Similarly create an association between GunShow and ShootingRange where a GunShow can have a single ShootingRange as a promoter. Ensure that mapping between GunShow and GunClub uses ClubId column on GunShow view and mapping for ShootingRange and GunShow uses ShootingRangeid column on GunShow view. Since views are not updatable, we have to create stored procedures for GunShows, ShootingRange and GunClubs to insert record into the database. So stored procedures needs to be declared inside store model and then mapped to entities on the msl section.

Discussion: As we discussed EF does not allow mapping a single column to multiple associations because that would corrupt the model and would cause data loss to happen. For instance if it allowed mapping multiple associations to a single foreign key column, then you can set PromoterId to have a ClubId and ShootingRangeId both which is not possible because PromoterId column can have only one value either it be ClubId or ShootingRangeId. Although the scenario is perfectly correct because at a given point either GunClub or ShootingRange can be a promoter but not both. Both currently in Ef there is no way to represent optional association. To get around the limitation, we can create a view that exposes promoterId as two columns ClubId and ShootingRangeId on GunShow view. Code below shows the view for GunShow.

```
create view [dbo].[vwGunShow]
as
select s.ShowId,s.ShowName,s.VendorsRegistered,r.ShootingRangeId,null ClubId
from tpt.GunShows s
join tpt.ShootingRange r  on s.PromoterId = r.ShootingRangeId
where s.PromoterType = 'SR'
union
select s.ShowId,s.ShowName,s.VendorsRegistered,null ShootingRangeId,g.ClubId
```

```

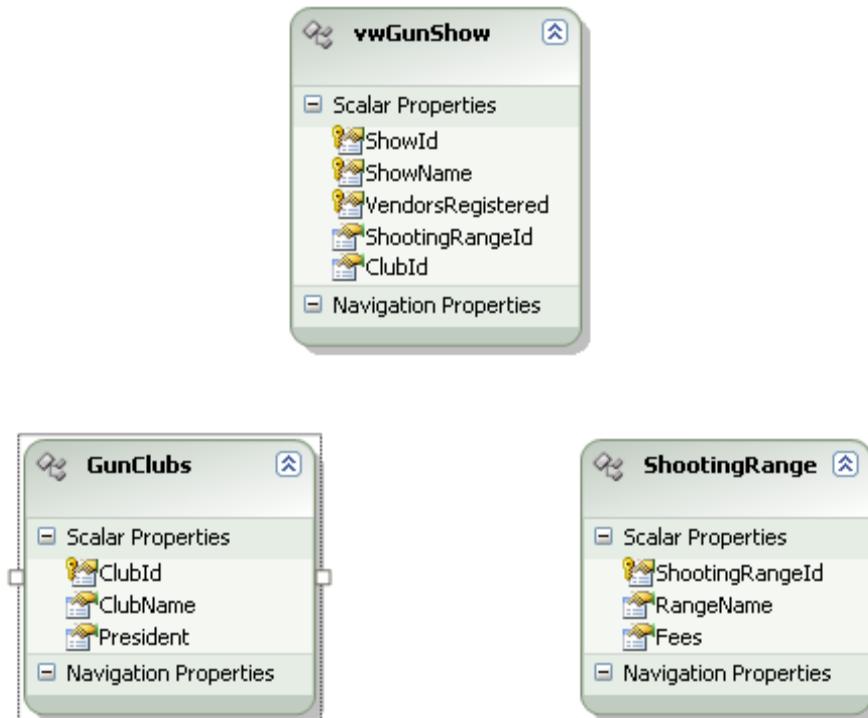
from tpt.GunShows s
join tpt.GunClubs g on s.PromoterId = g.ClubId and s.PromoterType = 'GC'
GO

```

The view above consists of two unions. The first union is a join between GunShow and ShootingRange where the PromoterType is SR. Since the join is against shootingRange table, I am setting null for ClubId column. Similarly the second join is against GunShow and GunClub where PromoterType is GC. Now that we have exposed two columns as foreign key to both GunClub and ShootingRange, we do not need to expose promotertype from our view. When we import the view into our model, EF creates a DifiningQuery which basically selects from the view created on the database.

Rest of the steps below will outline the process of importing the view, GunClub and ShootingRange into our entity data model.

1. Import vwGunShow, GunClub and ShootingRange table into EDM using entity model wizard. Figure below shows the model after the wizard has completed.



- When we imported the view, EF does not know what the primary key for the view is; therefore it marks all properties as participating in primary key. To remove primary key declaration, open the edmx in xml and go to vwGunShow entity in ssdl and make sure that only ShowId is marked as entity key. Code below shows the correct entity key for the view.

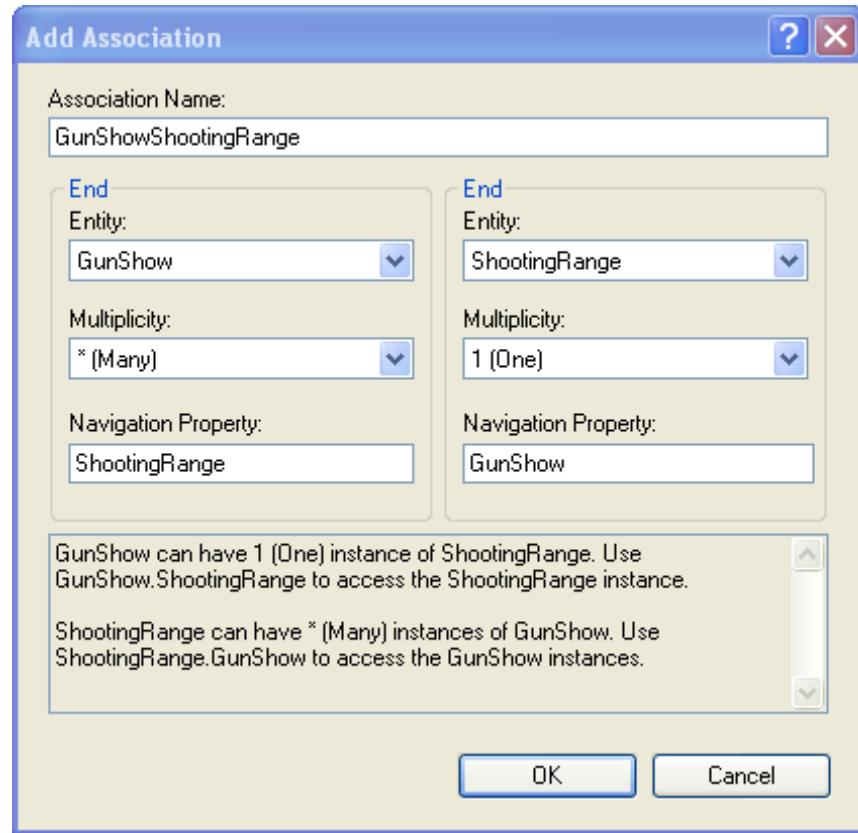
```

<EntityType Name="vwGunShow">
  <Key>
    <PropertyRef Name="ShowId" />
  </Key>
  <Property Name="ShowId" Type="int" Nullable="false" />
  <Property Name="ShowName" Type="varchar" Nullable="false"
MaxLength="100" />
  <Property Name="VendorsRegistered" Type="int" Nullable="false" />
  <Property Name="ShootingRangeId" Type="int" />
  <Property Name="ClubId" Type="int" />
</EntityType>

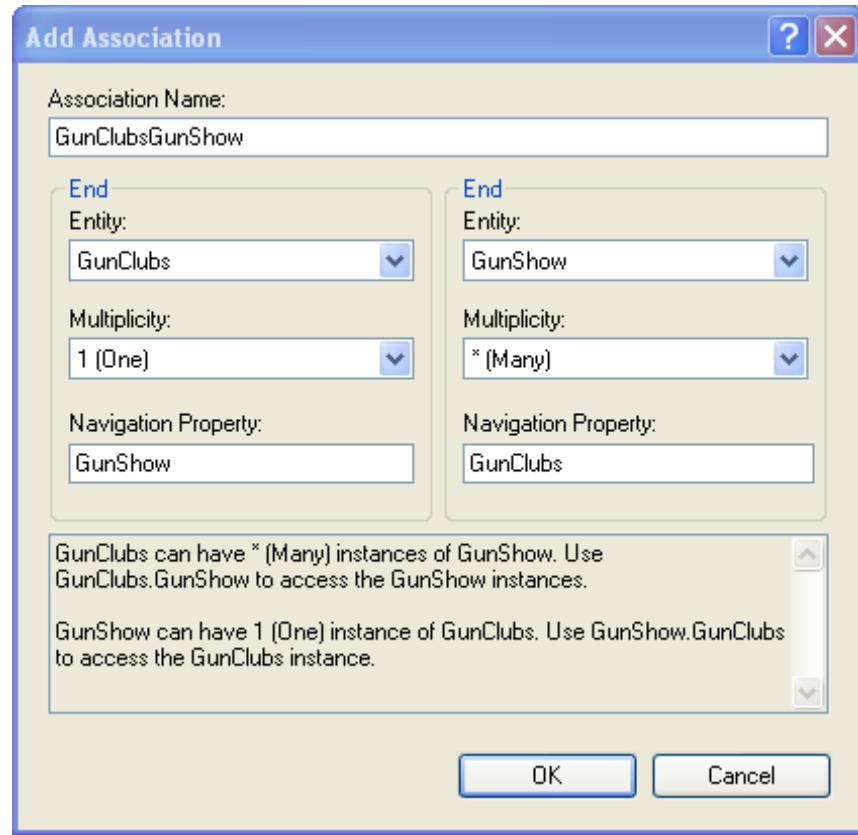
```

Also make sure that GunShow entity on the conceptual model also has ShowId as the entity key. Remove any extra columns from entity key.

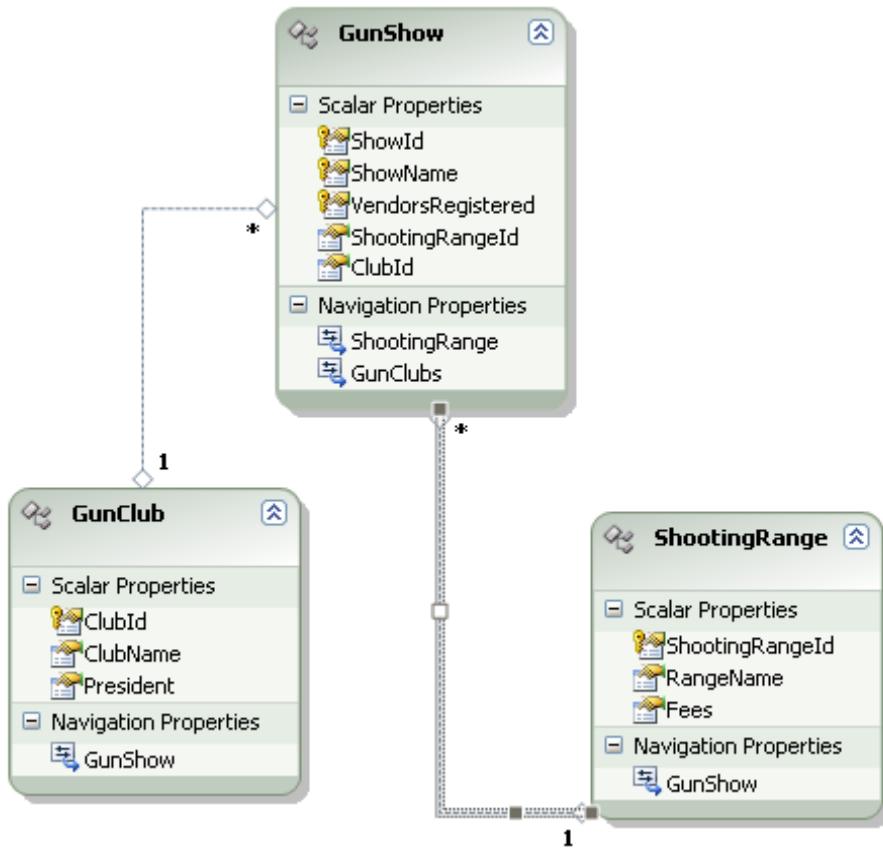
- Since ClubId and ShootingRangeId column will be used in association, it cannot be mapped to properties. So remove ShootingRangeId and ClubId property from GunShow entity.
- Create association between GunShow and ShootingRange where GunShow has a multiplicity of Many and ShootingRange has a multiplicity of 1. Figure below shows the association dialog with correct mappings.



5. Create association between GunClubs and GunShow where GunShow has a multiplicity of Many and GunClub has a multiplicity of 1. Figure below shows the association dialog to configure GunClub and GunShow.



After completing the association, the completed entity model should look like below.

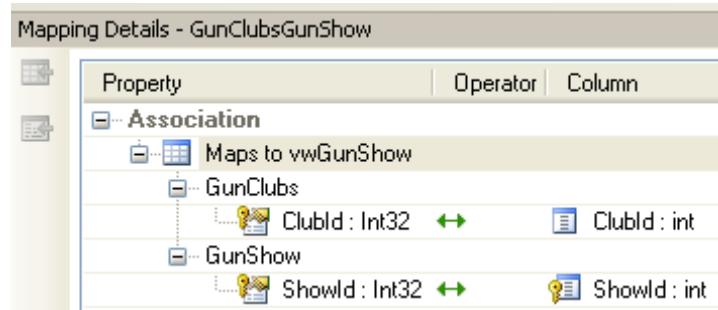


Next step is to configure the mapping for the association we created between GunShow, ShootingRange and GunClub.

6. Select the association line between GunShow and ShootingRange and open up mapping window. On the mapping window, select GunShow table. The designer would map the ShowId column ShowId entity key on GunShow entity and map ShootingRangeId column to ShootingRangeId property on ShootingRange entity. Figure below shows the mapping between GunShow and ShootingRange.



7. Select the association line between GunShow and GunClub and open up mapping window. On the mapping window select GunShow and the designer would auto map the columns to properties. Figure below shows the correct mapping for the association.



8. Since we used a view for GunShow entity, entity framework cannot insert, update and delete GunShow entity. We have to create stored procedures and map them using the designer. Another constraint EF enforces is if an entity participates in using stored procedures to perform crud, then all the related entities must also be saved using stored procedures.

SSDL below shows the stored procedure required to insert GunClub.

```

<Function Name="InsertGunClub" BuiltIn="false"
IsComposable="false">
    <CommandText>
        insert into
    tpt.GunClubs(ClubName,President) values (@ClubName,@President)
        select SCOPE_IDENTITY() as ClubId
    </CommandText>
    <Parameter Name="ClubName" Mode="In"
Type="varchar" />
    <Parameter Name="President" Mode="In"
Type="varchar" />
    </Function>
    <Function Name="UpdateGunClub" BuiltIn="false"
IsComposable="false">
        <CommandText>
            update tpt.GunClubs set ClubName =
@ClubName,President =@President where ClubId =@ClubId
        </CommandText>
        <Parameter Name="ClubId" Mode="In" Type="int"
/>

```

```

                <Parameter Name="ClubName" Mode="In"
Type="varchar" />
                <Parameter Name="President" Mode="In"
Type="varchar" />
            </Function>
            <Function Name="DeleteGunClub" BuiltIn="false"
IsComposable="false">
                <CommandText>
                    delete tpt.GunClubs where ClubId = @ClubId
                </CommandText>
                <Parameter Name="ClubId" Mode="In" Type="int"
/>
            </Function>

```

SSDL below shows the stored procedures to perform crud on ShootingRange entity.

```

<Function Name="InsertShootingRange" BuiltIn="false"
IsComposable="false">
    <CommandText>
        insert into
tpt.ShootingRange(RangeName,Fees) values (@RangeName,@Fees)
        select SCOPE_IDENTITY() as ShootingRangeId
    </CommandText>
    <Parameter Name="RangeName" Mode="In"
Type="varchar" />
    <Parameter Name="Fees" Mode="In" Type="int" />
</Function>
<Function Name="UpdateShootingRange" BuiltIn="false"
IsComposable="false">
    <CommandText>
        update tpt.ShootingRange set RangeName =
@RangeName,Fees =@Fees where ShootingRangeId =@ShootingRangeId
    </CommandText>
    <Parameter Name="ShootingRangeId" Mode="In"
Type="int" />
    <Parameter Name="RangeName" Mode="In"
Type="varchar" />
    <Parameter Name="Fees" Mode="In" Type="int" />
</Function>
<Function Name="DeleteShootingRange" BuiltIn="false"
IsComposable="false">
    <CommandText>
        delete tpt.ShootingRange where
ShootingRangeId = @ShootingRangeId
    </CommandText>

```

```

        <Parameter Name="ShootingRangeId" Mode="In"
Type="int" />
    </Function>

```

SSDL below shows the stored procedures to perform crud activity on GunShow entity. Notice that insert stored procedure checks to see if ShootingRangeId is not null then it assigns promoterType to be SR otherwise the promoterType get set to GC for GunClub. The delete stored procedure for GunShow takes both ShootingRangeId and ClubId in addition to ShowId. Although to delete a gun show from GunShow table, we only need ShowId, however entity framework requires all associationMappings be mapped to stored procedure regardless if they will be used. This restriction will be removed in the next version of EF.

```

<Function Name="InsertGunShow" BuiltIn="false"
IsComposable="false">
    <CommandText>
        declare
            @promoterid int,
            @promotertype char(2)
            if @ShootingRangeId is not null
            begin
                set @promoterid = @ShootingRangeId
                set @promotertype = 'SR'
            end
            else
            begin
                set @promoterid = @ClubId
                set @promotertype = 'GC'
            end
            insert into
        tpt.GunShows(ShowName,VendorsRegistered,promoterid,PromoterType)
            values
        (@ShowName,@VendorsRegistered,@promoterid,@promotertype)
            select SCOPE_IDENTITY() as ShowId
    </CommandText>
    <Parameter Name="ShowName" Mode="In"
Type="varchar" />
    <Parameter Name="VendorsRegistered" Mode="In"
Type="int" />
    <Parameter Name="ShootingRangeId" Mode="In"
Type="int" />
    <Parameter Name="ClubId" Mode="In" Type="int"
/>
    </Function>

```

```

<Function Name="UpdateGunShow" BuiltIn="false"
IsComposable="false">
    <CommandText>
        update tpt.GunShows set ShowName
=@ShowName,VendorsRegistered =@VendorsRegistered
            where showid =@ShowId
    </CommandText>
    <Parameter Name="ShowName" Mode="In"
Type="varchar" />
    <Parameter Name="VendorsRegistered" Mode="In"
Type="int" />
    <Parameter Name="ShootingRangeId" Mode="In"
Type="int" />
    <Parameter Name="ClubId" Mode="In" Type="int"
/>
    <Parameter Name="ShowId" Type="int" Mode="In"
/>
</Function>
<Function Name="DeleteGunShow" BuiltIn="false"
IsComposable="false">
    <CommandText>
        delete tpt.GunShows where showid =@ShowId
    </CommandText>
    <Parameter Name="ShowId" Mode="In" Type="int"
/>
    <Parameter Name="ShootingRangeId" Mode="In"
Type="int" />
    <Parameter Name="ClubId" Mode="In" Type="int"
/>
</Function>

```

Above examples are actually not stored procedures but inline sql defined on the model directly. The reason is convience for the user to see everything in one place. In real production application, it is recommended that you define actual stored procedures on the database.

9. After defining the stored procedures on the ssdl, we need to go into the designer and map the stored procedures to entities. To map GunClub entity to stored procedure, right click the entity and choose stored procedure mapping. For Insert stored procedure, choose InsertGunClub, for Update choose UpdateGunClub and for Delete choose DeletegunClub. After selecting the stored procedures the designer would auto map properties to column. Figure below shows the stored procedure mapping.

Mapping Details - GunClub

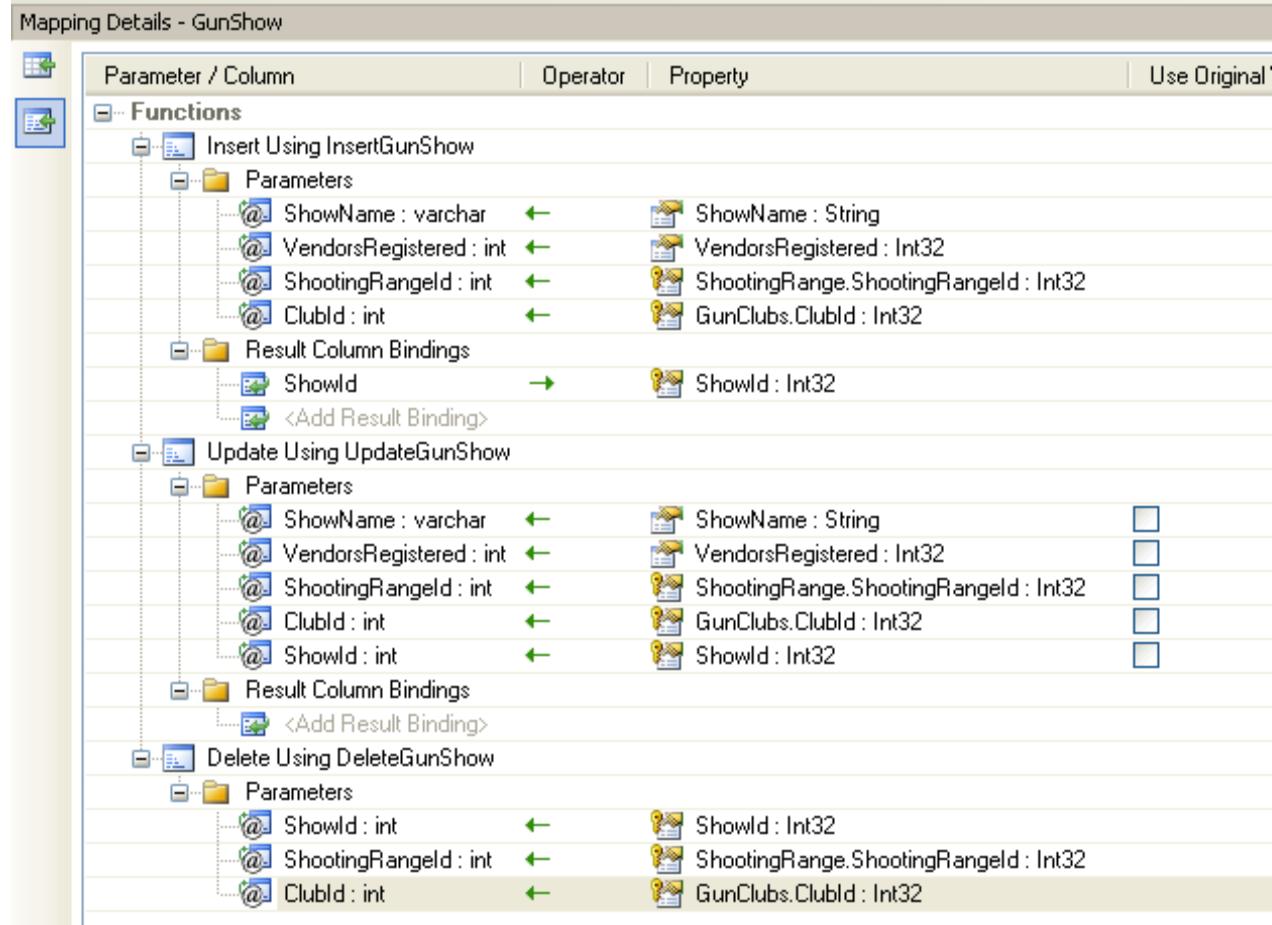
Parameter / Column	Operator	Property	Use Original Value
Functions			
Insert Using InsertGunClub			
Parameters			
@ ClubName : varchar	←	ClubName : String	
@ President : varchar	←	President : String	
Result Column Bindings			
ClubId	→	ClubId : Int32	
<Add Result Binding>			
Update Using UpdateGunClub			
Parameters			
@ ClubId : int	←	ClubId : Int32	<input checked="" type="checkbox"/>
@ ClubName : varchar	←	ClubName : String	<input type="checkbox"/>
@ President : varchar	←	President : String	<input type="checkbox"/>
Result Column Bindings			
<Add Result Binding>			
Delete Using DeleteGunClub			
Parameters			
@ ClubId : int	←	ClubId : Int32	

10. To map ShootingRange entity, open stored procedure mapping and select InsertShootingRange, UpdateShootingRange and DeleteShootingRange entity. Figure below shows the stored procedure mapping for ShootingRange entity.

Mapping Details - ShootingRange

Parameter / Column	Operator	Property	Use Original Value
Functions			
Insert Using InsertShootingRange			
Parameters			
@ RangeName : varchar	←	RangeName : String	
@ Fees : int	←	Fees : Int32	
Result Column Bindings			
ShootingRangeld	→	ShootingRangeld : Int32	
<Add Result Binding>			
Update Using UpdateShootingRange			
Parameters			
@ ShootingRangeld : int	←	ShootingRangeld : Int32	<input checked="" type="checkbox"/>
@ RangeName : varchar	←	RangeName : String	<input type="checkbox"/>
@ Fees : int	←	Fees : Int32	<input type="checkbox"/>
Result Column Bindings			
<Add Result Binding>			
Delete Using DeleteShootingRange			
Parameters			
@ ShootingRangeld : int	←	ShootingRangeld : Int32	

11. For GunShow entity, select InsertGunShow, UpdateGunShow and DeleteGun stored procedure. To Map ShootingRangeId and ClubId, we have to select the navigation property ShootingRange and GunClub to access its ShootingRangeId and ClubId entity key. For delete stored procedure we have to not only map ShowId but also ShootingRangeId and ClubId because these are the associations of GunShow entity and EF requires all associations be mapped. Figure below shows the stored procedure mapping.



To test the model, we can create an instance of GunShow entity and assign a GunGlub and save the GunShow entity to the database. Then using the second data context, we can retrieve the gun show along with the gun club and print the results to the console window. Code below accomplishes that task

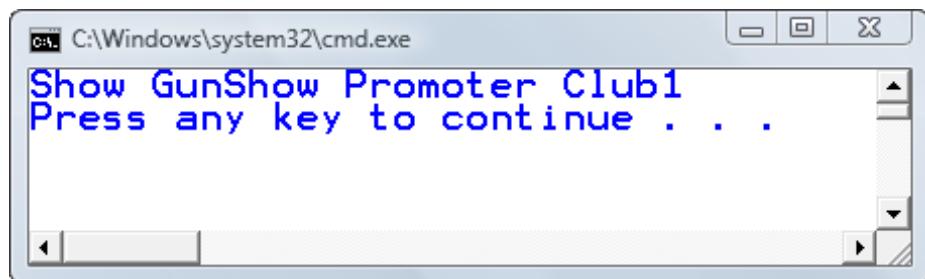
```

var db = new DQForeignEntities();
    var gunclub = new GunClub { ClubName = "Club1",
President = "Zee" };
        var shootingrange = new ShootingRange { RangeName =
"Range1" , Fees = 50 };
            db.AddToShootingRanges(shootingrange);
            db.AddToGunClubs(gunclub);
            var gunshow = new GunShow
{
    ShowName = "GunShow" ,
    VendorsRegistered = 20 ,
    GunClub = gunclub
};
db.AddToGunShows(gunshow);
db.SaveChanges();

var db2 = new DQForeignEntities();
var show = db2.GunShows.Include("GunClub").First(s => s.ShowName == "GunShow");
Console.WriteLine("Show {0} Promoter {1}", show.ShowName, show.GunClub.ClubName);

```

On the above code, I am creating an instance of GunClub and assigning it to GunClub property of the newly created GunShow entity. Using the second datacontext, I am printing the ShowName and the GunClub name to the console window.



4.2.3 Creating Dummy Defining Query to map stored procedure results

Problem: You have a stored procedure in the database that returns CustomerId and TotalPurchases customer has made. The result of the stored

procedure does not map to any table or view in the database. You want to bind the results of the store procedure to CustomerSales entity defined on the model.

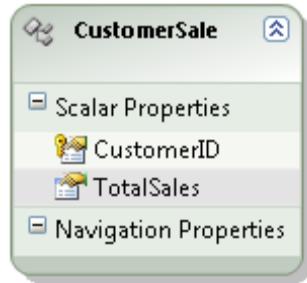
Solution: Create CustomerSales entity on the designer that matches the shape and data type result returned from the stored procedure. Import the stored procedure on the store model and using the function import map the stored procedure result to CustomerSales entity. At this point when we try to validate the model, we will get build errors that CustomerSales entity is not mapped to a table or view. In reality this is a correct case because our stored procedure returns arbitrary results that cannot be mapped to any table or view. To get around this validation error, we need to create a dummy DefiningQuery that defines a select that has the same shape as CustomerSales entity but does not return any data. Map the definingQuery to CustomerSales entity using the mapping window.

Discussion: Figure below shows the results when we execute our stored procedure GetCustSales.

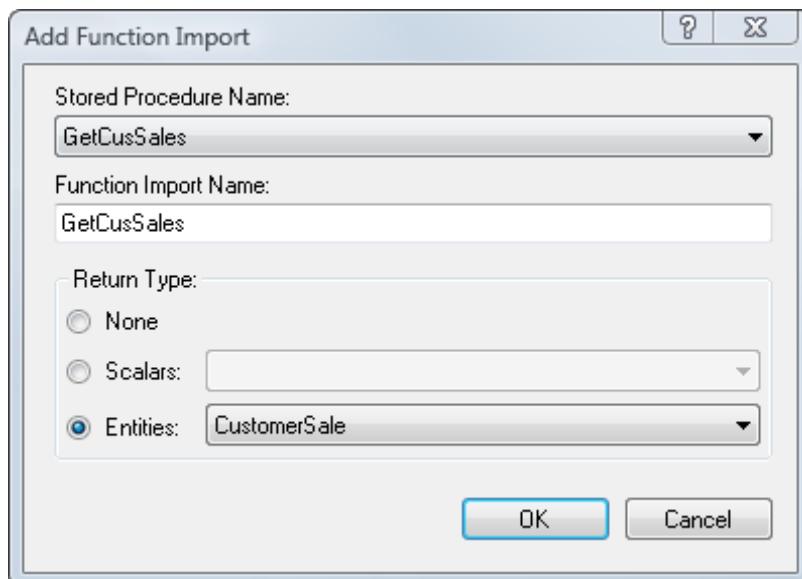
	CustomerID	TotalSales
1	QUICK	117483.39
2	SAVEA	115673.39
3	ERNSH	113236.68
4	HUNGO	57317.39
5	RATTC	52245.90

The stored procedure returns CustomerId and TotalSales. CustomerId has a string datatype and TotalSales returns a decimal value.

1. Import the above stored procedure on the store model using Entity Data Wizard.
2. Create CustomerSales entity that matches the datatype and columns returned from the stored procedure. Make sure CustomerId is set as the entitykey. Figure below shows CustomerSale entity on the model.



3. Right click the stored procedure imported in the store model from Model Browser window and select create function import. On Add Function Import dialog, set the return type to be CustomerSale entity. Figure below shows Add Function Import.



4. Since CustomerSale entity needs to be mapped to a table or view, we need to create a DefiningQuery on SSDL section of edmx file. Open the edmx file in xml format and define CustomerSales entityset as follow.

```

<EntitySet Name="GetCusSales"
EntityType="DDQModel.Store.GetCusSale">
    <DefiningQuery>
        select null CustomerID, cast(0 as
decimal) TotalSales
            where 1 = 2
    </DefiningQuery>

```

```
</EntitySet>
```

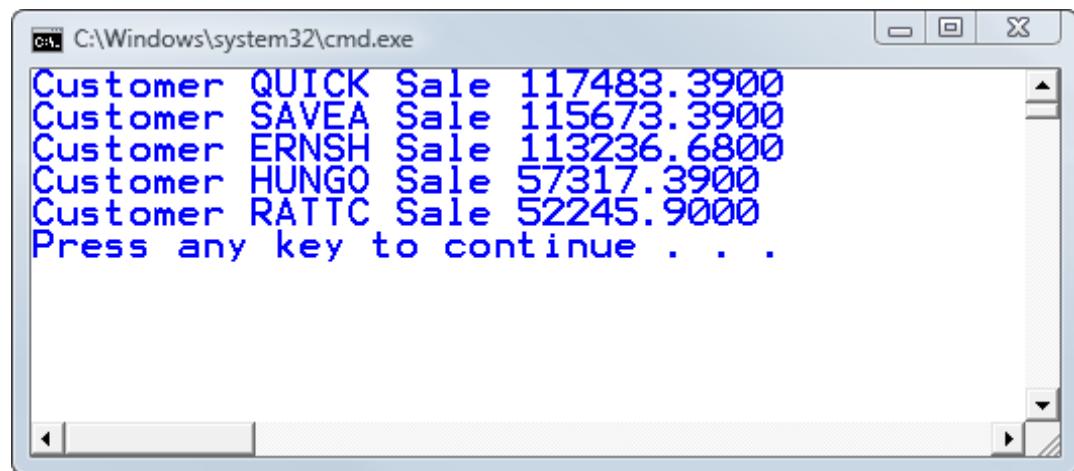
Notice the above DefiningQuery simply returns CustomerId and TotalSales column on a condition that would never be true. This is because, it is a dummy view to satisfy EF conditions that an entity defined on the model must be mapped to some table structure. In future releases of EF, a stored procedure result could be mapped to a complex type which would alleviate all these hacks we are applying to get around EF limitations. Our GetCusSales entityset is mapped to EntityType GetCusSale which we need to create. Code below created GetCusSale entity with two columns CustomerId and TotalSales as shown below.

```
<EntityType Name="GetCusSale">
    <Key>
        <PropertyRef Name="CustomerID" />
    </Key>
    <Property Name="CustomerID" Type="varchar"
Nullable="false" />
    <Property Name="TotalSales" Type="decimal"
Nullable="false" />
</EntityType>
```

5. Code below calls GetCustomerSales that calls GetCustSales stored procedure and returns a collection of CustomerSale entity. I then loop through each entity and print its output to the console window.

```
var db = new DDQEntities();
foreach (var cus in db.GetCusSales())
{
    Console.WriteLine("Customer {0} Sale
{1}", cus.CustomerID, cus.TotalSales);
}
```

Figure below shows the output printed on the console window.



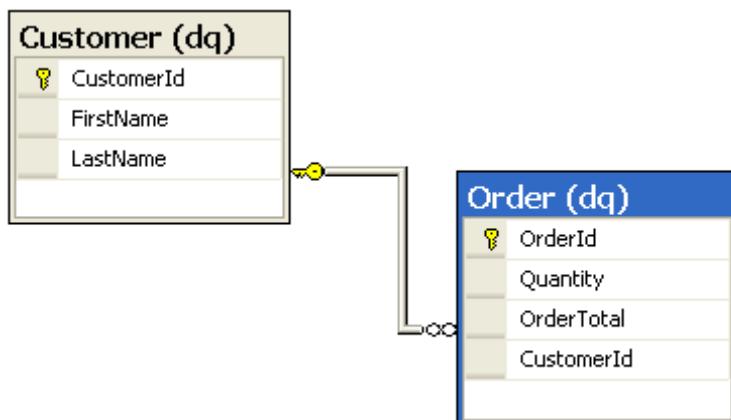
```

Customer QUICK Sale 117483.3900
Customer SAVEA Sale 115673.3900
Customer ERNSH Sale 113236.6800
Customer HUNGO Sale 57317.3900
Customer RATTC Sale 52245.9000
Press any key to continue . . .

```

4.2.4 Creating Read-only Calculated Properties using Defining Query

Problem: Figure below shows the database diagram for Customer and Orders table defined on the database.



The diagram above contains Customer and Order table. Customer has FirstName and Last Name and Orders table contains the quantity customer has ordered and the Total for the order. You want to expose the customer table as an entity on entity data model. In addition you want to expose 3 read-only properties. First read-only property Name would concatenate FirstName and LastName. Customer entity should also expose calculated property called TotalOrders which returns totalOrder the customer has placed so far,

TotalPurchases property that returns the total dollar amount purchase the customer has made so far.

Solution: Import Customer table into EDM using Import wizard. Add three new fields Name, TotalOrders and Total Purchases to Customer entity defined on the conceptual model. Since the new fields added are not defined on the Customer table structure, we need to create Defining Query that joins Customer table to Orders table and returns the calculated fields for TotalOrders and TotalPurchases. To create Defining Query, modify the ssdl section of edmx in xml and change the Customers entity set to use DefiningQuery instead of mapping to Customer table. Modify the Customer Entity on the ssdl to include the three new columns defined on the conceptual model. To map the new fields on the store model to the conceptual, open the edmx in the designer and update the mapping for Customer entity using the mapping window.

Discussion: In the version 1 release of entity framework, there is no way to expose calculated read-only properties on an entity. All properties exposed on an entity must belong or be mapped to some column in the table. To get around this limitation we can write a DefiningQuery which allows us to specify a sql statement that can return arbitrary number of columns that can either combine multiple columns or perform aggregate operation to get a column value. However once we use Defining Query, we no longer can leverage insert, update and delete behaviors using EF. We have to create and declare stored procedure on the storage model and then map those stored procedure to insert, update and delete operation on entities defined on the conceptual model. Steps below outline the process of creating Customer entity with three additional read-only calculated properties using DefiningQuery.

1. Import Customer table using Entity Model Wizard. Add three properties Name (String), Total Orders (int32) and Total Purchases (int32). Since all 3 properties are read-only change the setter access on each property to private. This would ensure that property is exposed as read-only.

2. Since these properties are not defined on the Customer table, modify the ssdl to use DefiningQuery instead of table. Open the edmx in Xml and change the Customer entity set as follows.

```

<EntitySet Name="Customer" EntityType="DQReadModel.Store.Customer">
    <DefiningQuery>
        select c.* , c.FirstName + ' ' + c.LastName
        FullName,
        (Select COUNT(*) from dq.[Order] where
        CustomerID = c.CustomerId) TotalOrders,
        (Select SUM(o.OrderTotal) from dq.[Order] o
        where Customerid = c.CustomerId) TotalPurchases
        from dq.Customer c
    </DefiningQuery>
</EntitySet>

```

On the above DefiningQuery, I am merging the FirstName and LastName to create a new field Name. To get TotalOrders, I am using Count aggregate operator and to get TotalPurchases, I am applying Sum operator to get TotalPurchases for every customer.

3. The definingQuery created contains three additional read-only fields which Customer entity on the store model does not have. So modify the Customer entity on the store model to have 3 new fields that can be mapped to the conceptual model. Code below shows the update Customer entity on the store model.

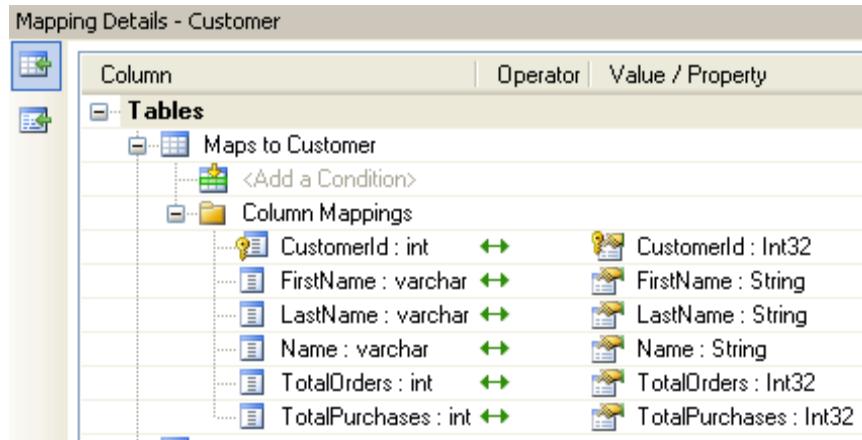
```

<EntityType Name="Customer">
    <Key>
        <PropertyRef Name="CustomerId" />
    </Key>
    <Property Name="CustomerId" Type="int" Nullable="false"
    StoreGeneratedPattern="Identity" />
    <Property Name="FirstName" Type="varchar" Nullable="false"
    MaxLength="50" />
    <Property Name="LastName" Type="varchar" Nullable="false"
    MaxLength="50" />
    <Property Name="Name" Type="varchar" />
    <Property Name="TotalOrders" Type="int" />
    <Property Name="TotalPurchases" Type="int" />
</EntityType>

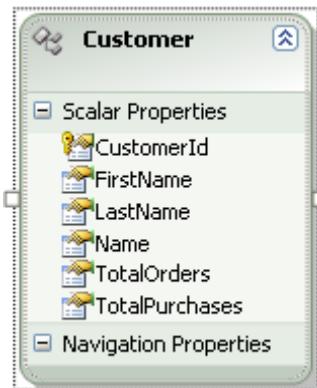
```

4. To map the three new fields defined on the store and conceptual model, open the mapping window for Customer entity and map Name property to Name column, TotalOrders property to TotalOrders column and

TotalPurchases property to TotalPurchases column. Figure below shows the updated mapping window.



The updated Customer entity should look as follows.



5. Code below loops through the customers' collection returned from the ObjectContext and prints all three readonly properties to the console window.

```
var db = new DQReadEntities();
foreach (var customer in db.Customers)
{
    Console.WriteLine("Name {0} TotalOrders {1} Purchases {2}",
customer.Name, customer.TotalOrders, customer.TotalPurchases);
}
```

Figure below shows the output from the console window.

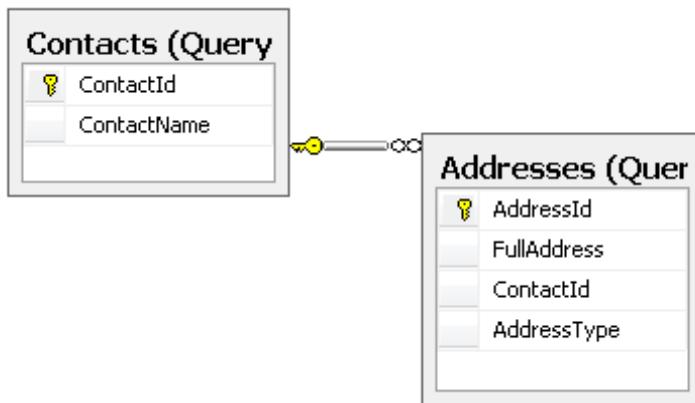
```

C:\WINDOWS\system32\cmd.exe
Name Zeeshan,Hirani TotalOrders 2 Purchases 400
Press any key to continue . . .

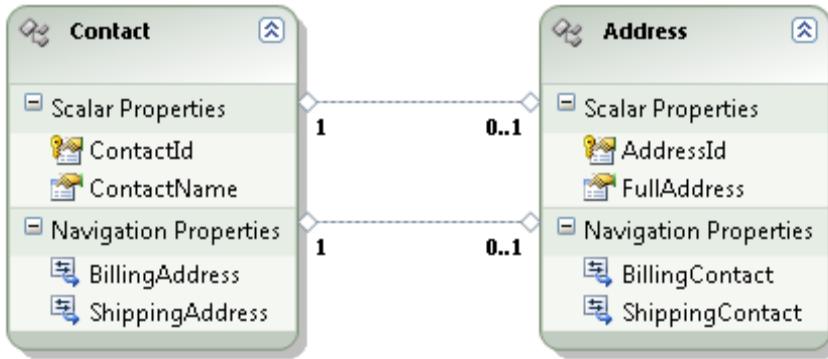
```

4.2.5 Using DefiningQuery to map multiple associations to foreign key

Problem: Figure below shows the database diagram for the relationship between Contact and their addresses.



Although in the table relationship, a contact can have many addresses and each address is either a billing or shipping identified by AddressType column. On the entity data model, you want to make sure that a Contact cannot have more than two addresses and there could be only 1 billing and 1 shipping address. To ensure the above scenario, you want to expose two entity references from Contact, a BillingAddress and ShippingAddress. The end entity data model should look as follows



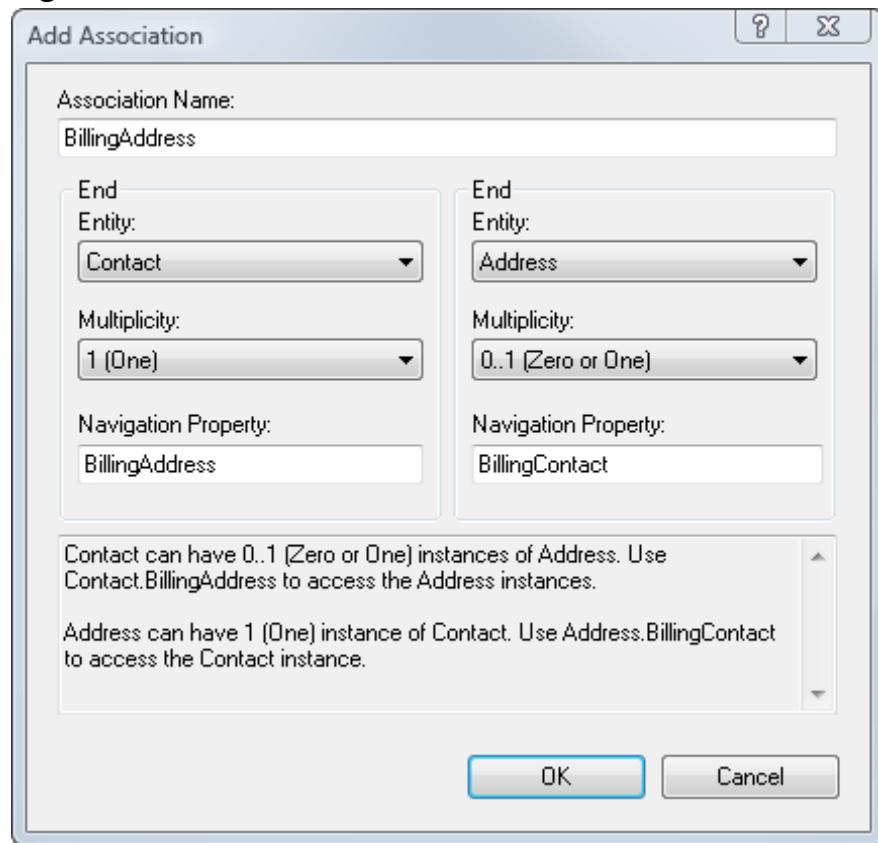
Solution: To accomplish the above solution, we have to create a DefiningQuery and instead of exposing ContactId we need to expose two columns BillingContactId and ShippingContactId where BillingContactId would not be null when AddressType is B and ShippingContactId won't be null when AddressType is S. The reason we have to expose two columns is because entity framework does not allow mapping multiple associations to a foreign key column as it would invalidate the model and cause data loss. Map two associations between Contact and Address and map billingAddress association to BillingContactId column and map the ShippingAddress association to ShippingContactId column.

Discussion: To create multiple associations between Contact and Address entity, we need to create DefiningQuery for AddressEntity. In the sql query we need to expose ContactId foreign key column multiple times, first as BillingContactId and second as ShippingContactId. When AddressType is billing, then BillingContactId should be the ContactId and when AddressType is Shipping, ShippingContactId should be the ContactId.

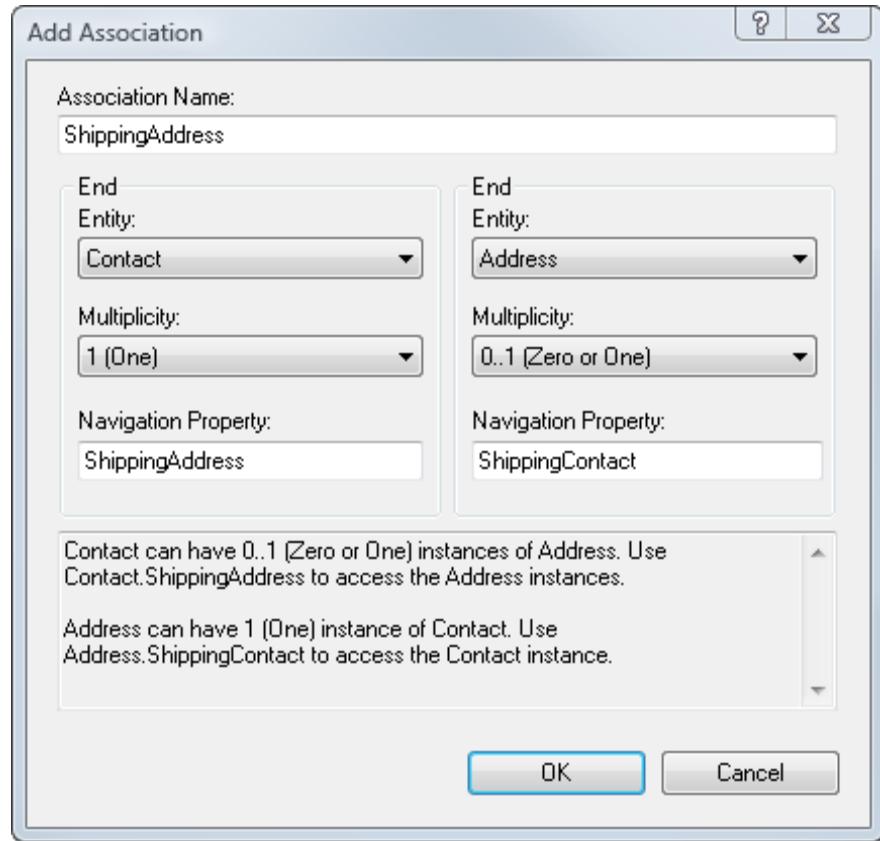
Steps below outline the process of exposing two entity reference BillingAddress and ShippingAddress for a relationship that is defined as 1 to Many in the database.

1. Import Contact table using EDM wizard.
2. Create Address entity on the designer. Add AddressId(Int32) and FullAddress(string) scalar properties to Address entity. Make AddressId as entity key.

3. Create association between Contact and Address where Contact has multiplicity of 1 and Address has a multiplicity of 0-1. The reason Address will have a multiplicity of 0-1 is because a Contact may only have zero or more addresses. Call this association as BillingAddress. Figure below shows the association values.



4. Create a similar association like in step 3 but call it ShippingAddress. Figure below shows the association values for ShippingAssociation.



5. Since we have not configured how Addresses entityset, will gets its data from database, we need to define DefiningQuery and create a similar entity Address on the store model. Since neither options are supported by the designer, open the edmx file in xml format and create the DefiningQuery as follows.

```

<EntitySet Name="Addresses"
EntityType="EcommerceModel.Store.Addresses">
    <DefiningQuery>
        select AddressId ,ContactId
BillingContactId,null ShippingContactId,FullAddress
            from QueryView.Addresses billing
            where AddressType = 'B'
            union
            select AddressId,null
BillingContactId,ContactId ShippingContactId,FullAddress
            from QueryView.Addresses billing
            where AddressType = 'S'
    </DefiningQuery>
</EntitySet>

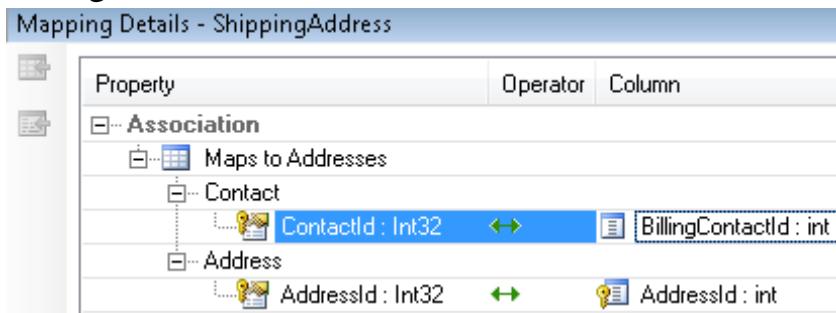
```

The above DefiningQuery does a union of two select statements where the first select gives a view where AddressType is Billing. Notice that we have assigned the ContactId to BillingContactId and ShippingContactId is passed a default value of Null. Similarly when AddressType is shipping BillingContactId is assigned a value of Null. The DefiningQuery created is mapped to Addresses entity which does not exist. So create Addresses entity on the store model as follows.

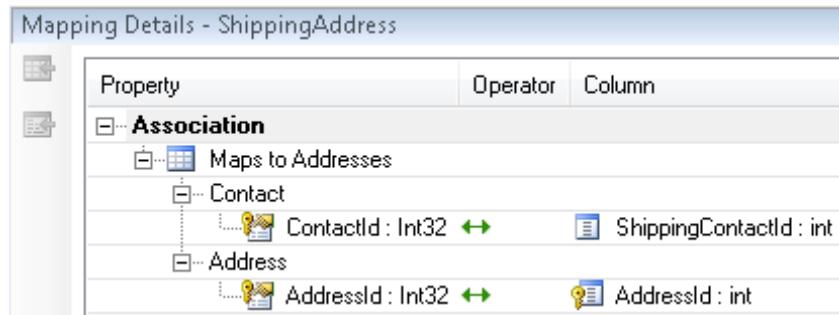
```
<EntityType Name="Addresses">
    <Key>
        <PropertyRef Name="AddressId" />
    </Key>
    <Property Name="AddressId" Type="int" Nullable="false" />
    <Property Name="FullAddress" Type="varchar" Nullable="false" MaxLength="100" />
        <Property Name="BillingContactId" Type="int" Nullable="true" />
        <Property Name="ShippingContactId" Type="int" Nullable="true" />
    </EntityType>
```

The addresses entity type simply matches the columns returned from above DefiningQuery. Now that we have created Addresses entityset, we can go back to the designer and map the Billing and Shipping associations to EntitySet defined on the store model.

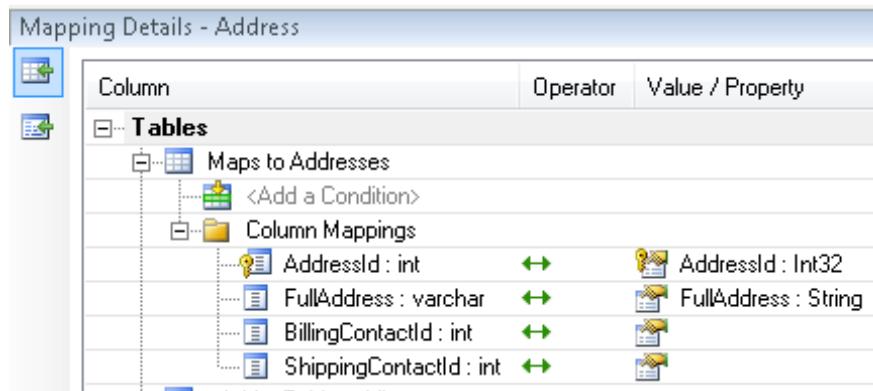
6. Map Billing association to Address table where ContactId is mapped BillingContactId as shown below.



7. Map Shipping association to Address table where ContactId is mapped to ShippingContactId as shown below.



8. Map Address entity to address table. The designer should map the AddressId to AddressId column and FullAddress to FullAddress column in Address table.



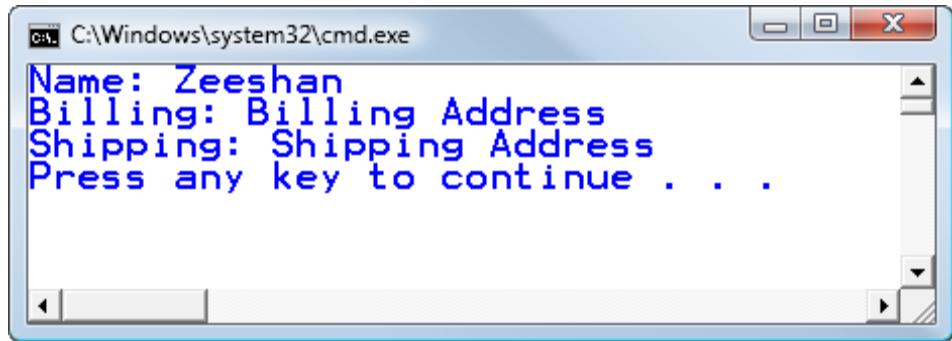
9. To test the above model, we can retrieve a Contact, its Billing and shipping Address. On the code below I am using Include to retrieve Billing and Shipping Address for contact and printing the result to the console window.

```

var db = new MultipleAssoEntities();
var contact = db.Contacts

.Include("BillingAddress").Include("ShippingAddress")
    .First(c => c.ContactName ==
"Zeeshan");
    Console.WriteLine("Name {0}", contact.ContactName);
    Console.WriteLine("Billing
{0}", contact.BillingAddress.FullAddress);
    Console.WriteLine("Shipping
{0}", contact.ShippingAddress.FullAddress);

```



5. Inheritance

Basics of Inheritance

Entity framework supports 3 different models of inheritance.

1. Table Per Hierarchy (Single Table Inheritance)
2. Table Per Type
3. Table Per Concrete Class

Of all the supported inheritance models, the most simplest and easiest to implement is Table Per Hierarchy (Single Table Inheritance). To implement this inheritance, you store all concrete types in one table. In Entity framework to identify a row as a specific concrete type, you define a discriminator column which identifies which concrete type a specific row gets mapped to. From a usability point, I have found Single table model to be very easy to get started. However from the database perspective, the model doesn't seem to favor a clean approach. The reason is you are storing all different concrete types in a single table. Some concrete types would need certain columns where as others won't. To accomplish flexibility at the table level, you have to mark all columns that are specific to their concrete implementation as allow nulls. Some database developers may find this approach not a good solution because it does not make efficient use of disk space. On the other hand table per hierarchy offers good performance because to find a concrete type, you

don't have to apply joins to another table which can be costly if the table is too big. Since all the types are stored in one table, you can apply index on the discriminator column to allow faster searches based on concrete type you are looking for. To map this structure as table per hierarchy in entity data model, you have to define the column which entity framework can use to identify each type, basically a discriminator column. Next you need to move specific field for each type from the base class to its own entity.

In Table per Type model, you define a base table that contains fields common across all types. Then you define a table for each type which contains fields that are specific to that type. In addition the primary key column defined on the derived table is also the foreign key for the base table. To map this form of table structure into table per type entity model, each individual type needs to inherit from the base type where the base type is mapped to the base table defined on the database. Each derived type needs to be mapped to its specific table in the database. Additionally, you have to delete the primary key property on the derived entity generated by the designer and map the primary key column on the derived entity to the entity key defined on the base class.

In table Per Concrete Type, each table represents the entire entity. It is not required that two tables participating in Table Per Type have same number of columns. The columns that are specific to a table that is not in another table participating in table per type, would end up as a property on the derived entity. Rest of the columns would be placed as properties on the base entity. Table Per Concrete Type is not fully supported on the designer so you start with importing the model and create your conceptual model but for modeling table per concrete type, you have to manually edit the xml file. One of the reasons you create table per concrete type is to portray data coming from multiple tables as being a single entity retrieving data from a single table. This means that primary key or entity key on the conceptual model cannot be duplicated. You cannot have primary key of 1 on table1 and primary key of 1 on table2 as well because this would cause entity framework to throw primary key violation constraint.

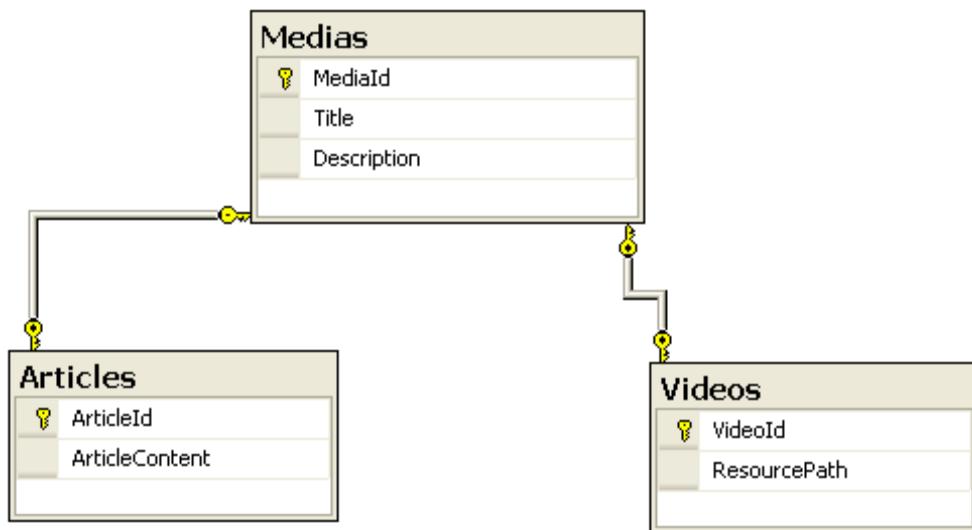
5.1.1 Table per Type Walkthrough

Problem: You have created 3 table Media, Video and Articles in the database. Media table contains common fields for both Articles and Videos. Fields specific to Video and Articles are stored in their respective table. You want to map this structure to entity data model using Table per Type inheritance.

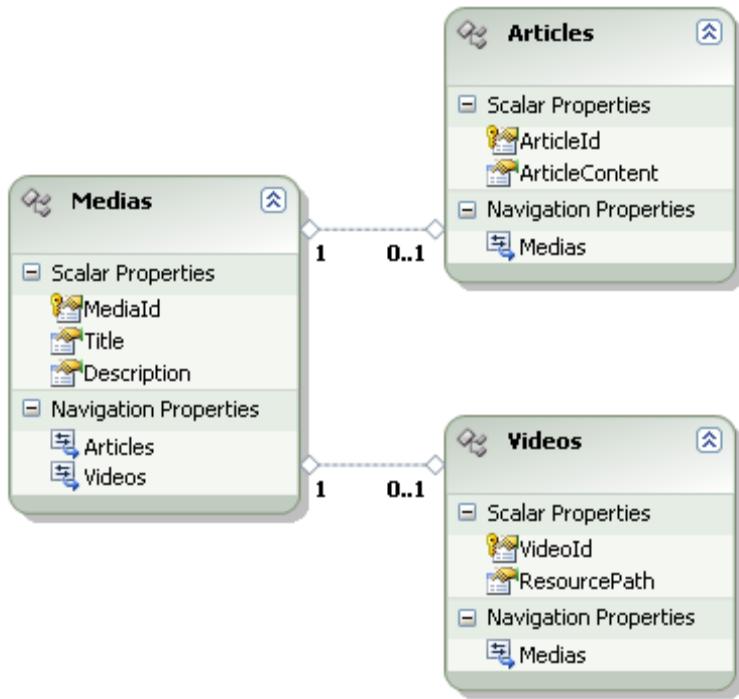
Solution:

Discussion:

Media table in the database consists of MediaId, Title and Description where MediaId represents the primary key of the table. Video table has VideoId as the primary key which is also the foreign key for Media table. Video table contains an additional column ResourcePath which is the location on the network where the media resides. Articles table has ArticleId as the primary key which is also the foreign key to Media table. ArticleContent contains the content of the article. Figure below shows the database diagram for Media.

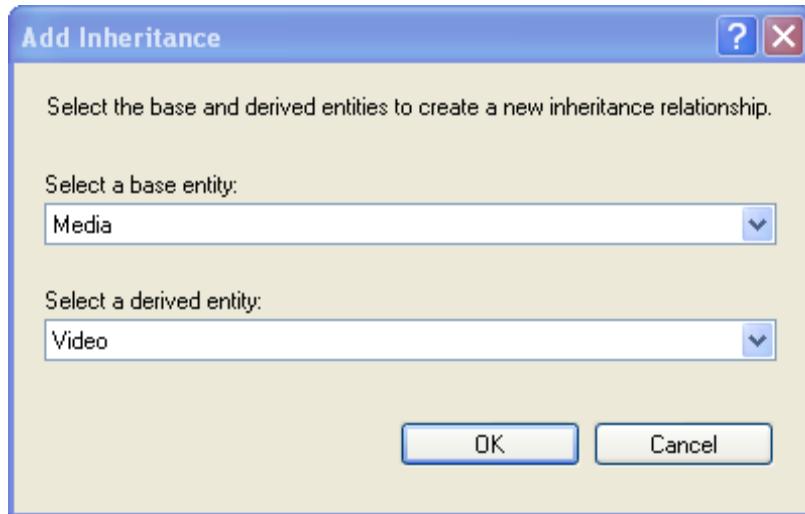


The next step is to import the database model using import database wizard. Screen shot below shows the model created by edm when we import the raw tables.

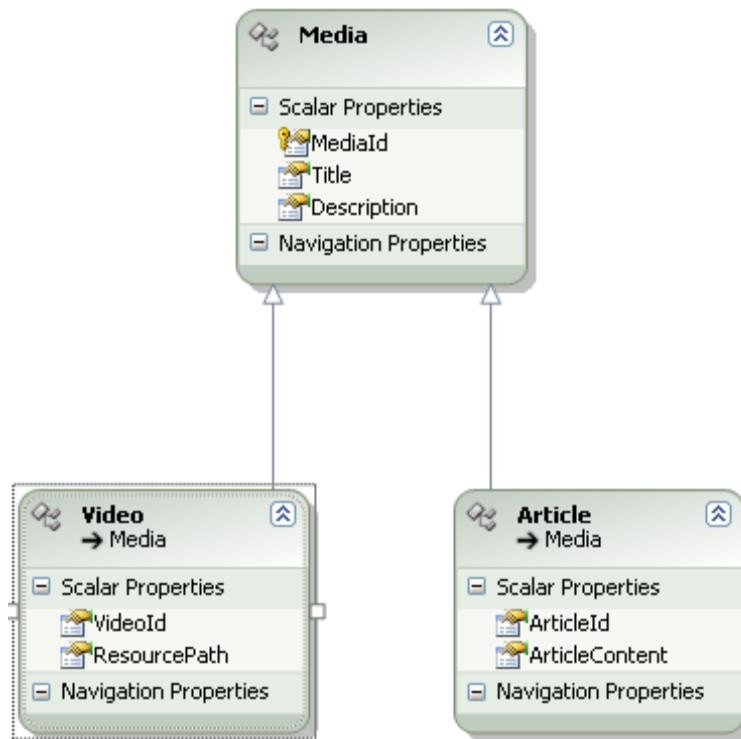


When we import the tables, entity framework maps one to one relationship between Media, Article and Video as 1 to 0-1 associations. What we want is an inheritance hierarchy. So the first step is to delete the association and add inheritance with Media as the base entity.

To delete the association, select both associations and click delete. To add inheritance right click Media entity and choose inheritance. On the inheritance window popup, select Media as the base entity and Video as the derived entity. Figure below shows the correct entity selected for inheritance between Media and Video.



To create inheritance for article, same process can be followed. After setting inheritance for both entities, Articles and Videos, final figure should look like the one below.



If you try to validate the model, you will get validation errors. To fix the validation errors remove the VideoId and ArticleId from Video and Article

entity. For video entity change the table mapping where VideoId is mapped to Mediaid as shown below.

Video Mapping

The screenshot shows the 'Mapping Details - Video' window. The 'Tables' node is expanded, revealing 'Maps to Videos' and 'Column Mappings'. Under 'Column Mappings', there are two mappings: 'VideoId : int' is mapped to 'MediaId : Int32', and 'ResourcePath : String' is mapped to 'ResourcePath : String'.

To map Article entity, map ArticleId to MediaId as shown below.

Since we are only using Media as a base and will not instantiate on its own it is important that we mark Media entity as Abstract. Therefore we can only instantiate concrete implementations of Media such as Article and Videos.

Article Mapping

The screenshot shows the 'Mapping Details - Article' window. The 'Tables' node is expanded, revealing 'Maps to Articles' and 'Column Mappings'. Under 'Column Mappings', there are two mappings: 'ArticleId : int' is mapped to 'MediaId : Int32', and 'ArticleContent : String' is mapped to 'ArticleContent : String'.

Although table per type mapping is fully supported by the designer, it is essential to understand the mapping written by the designer to persist the derived types to the database. Sample below shows the mapping for Articles and Videos with Media as the base type.

```
<EntitySetName="Medias">
    <EntityTypeMapping
        TypeName="IsTypeOf(SimpleTPTInheritance.Media)">
        <MappingFragment StoreEntitySet="Medias">
            <ScalarProperty Name="MediaId" ColumnName="MediaId" />
            <ScalarProperty Name="Title" ColumnName="Title" />
```

```

        <ScalarProperty Name="Description" ColumnName="Description"
/>
        </MappingFragment>
    </EntityTypeMapping>
<EntityTypeMapping TypeName="IsTypeOf(SimpleTPTInheritance.Video)">
    <MappingFragment StoreEntitySet="Videos">
        <ScalarProperty Name="MediaId" ColumnName="VideoId" />
        <ScalarProperty Name="ResourcePath" ColumnName="ResourcePath" />
/>
    </MappingFragment>
    </EntityTypeMapping><EntityTypeMapping
TypeName="IsTypeOf(SimpleTPTInheritance.Article)">
    <MappingFragment StoreEntitySet="Articles">
        <ScalarProperty Name="MediaId" ColumnName="ArticleId" />
        <ScalarProperty Name="ArticleContent" ColumnName="ArticleContent" />
    </MappingFragment>
    </EntityTypeMapping>
</EntityTypeMapping>

```

In the above sample, Media base entity is mapped to Medias table in the database. Notice TypeName uses IsTypeof of Media which means this mapping applies to any entity that derives from Media entity. Next two entities Articles and Videos map to their respective table with type video and Article. The IsTypeOf is also used in derived types for the case if there are other entities that derive from Articles and Videos. However in our case we can simply type in the class name for derived types without using IsTypeOf.

```

<EntitySetMapping Name="Medias">
    <EntityTypeMapping
TypeName="IsTypeOf(SimpleTPTInheritance.Media)">
        <MappingFragment StoreEntitySet="Medias">
            <ScalarProperty Name="MediaId" ColumnName="MediaId" />
            <ScalarProperty Name="Title" ColumnName="Title" />
            <ScalarProperty Name="Description" ColumnName="Description" />
/>
        </MappingFragment>
    </EntityTypeMapping>
<EntityTypeMapping TypeName="IsTypeOf(SimpleTPTInheritance.Video)">
    <MappingFragment StoreEntitySet="Videos">
        <ScalarProperty Name="MediaId" ColumnName="VideoId" />
        <ScalarProperty Name="ResourcePath" ColumnName="ResourcePath" />
/>
    </MappingFragment>
    </EntityTypeMapping><EntityTypeMapping
TypeName="IsTypeOf(SimpleTPTInheritance.Article)">
    <MappingFragment StoreEntitySet="Articles">
        <ScalarProperty Name="MediaId" ColumnName="ArticleId" />
        <ScalarProperty Name="ArticleContent" ColumnName="ArticleContent" />
    </MappingFragment>
    </EntityTypeMapping>

```

```
</EntitySetMapping>
```

To test the model, we can create an instance of the object context, add article and video to the media collection and call SaveChanges on the objectContext. In the example below, I am adding article and video entity to AddToMedias method generated by the designer. AddToMedias can take any entity derived from media class. You don't get separate methods to add Media and Articles. Similarly to read Article and Video there is so no separate property exposed on the object to directly access Articles and Videos. Instead ObjectContext exposes the base class reference Articles and you have to use OfType operator to only return Articles of a certain derived type. The entities returned are returned as derived class references which means you do not have to perform explicit cast to get a derived type instance. In the example below, to get videos, I am using a where linq operator and filtering to only return Media of Type Video. Since where operator returns a base class reference, I am additionally using Cast operator to cast the Media base class reference to Video derived type to access properties on the Video entity.

```
var db = new STPTInh();
    //inserting article and videos
    Article article = new Article
    {
        Title = "Linq Getting Started",
        ArticleContent = "article content"
    };

    Video video = new Video
    {
        Title = "Bill Wagner with More on C# 3.0",
        ResourcePath = "CsharpwithBill.wmv"
    };
    db.AddToMedias(article);
    db.AddToMedias(video);
    db.SaveChanges();

    var db2 = new STPTInh();
    //get articles
    var articles = db2.Medias.OfType<Article>();
    foreach (var art in articles)
    {
        Console.WriteLine("Title {0} Content{1}", art.Title,
art.ArticleContent);
    }

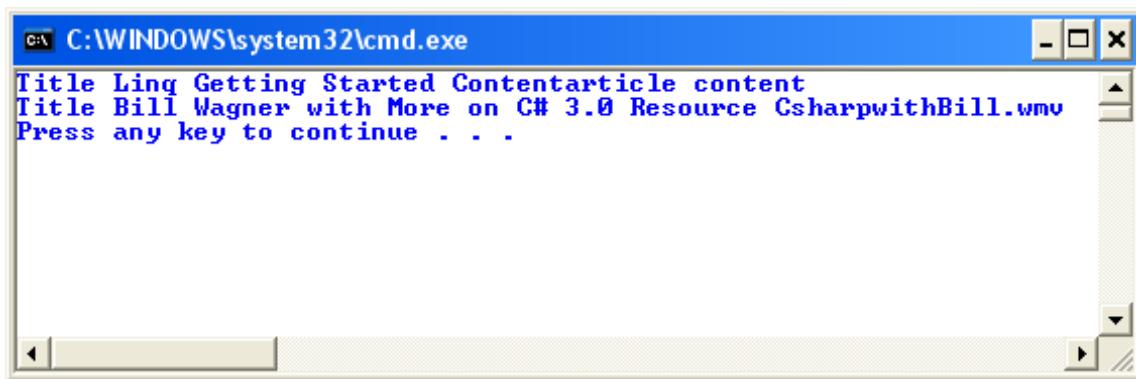
    var videos = db.Medias.Where(m => m is
Video).ToList().Cast<Video>();
    foreach (var vid in videos)
```

```

    {
        Console.WriteLine("Title {0} Resource {1}", vid.Title,
vid.ResourcePath);
    }

```

Screen shot below shows the output of the above code on Console window.



5.1.2 Table per Hierarchy (Walkthrough)

Problem: You have an employee table in the database which contains two kinds of Employee; Hourly Employee and Salaried Employee. To identify each type of Employee, the table has an additional column Employee Type which can have two values; HE for Hourly Employee and SE for Salaried Employee. You want to map this table structure to entity data model using Table Per Type.

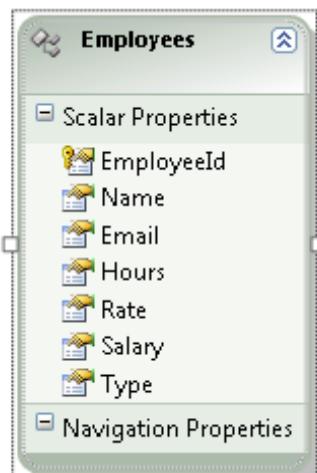
Solution: Using the import wizard will allow you to import the tables into the entity data model. After the import is completed you will see Employee entity created on the designer. Since employee will have two derived types, create two entities Hourly and Salaried Employee that inherits from Employee entity. Move the properties specific to the derived type to its own entity. Then map each derived type to the employee table and map properties on the derived type to columns on the employee table. Since employee base class does not have any definition to our concrete implementation except for the fact that it serves as our base class, we need to ensure that employee class cannot be instantiated. To satisfy the requirements make the base class abstract. If we do not make the base class abstract, Entity framework

validation will complain that Employee class needs to also contain some discriminator value that it can map to.

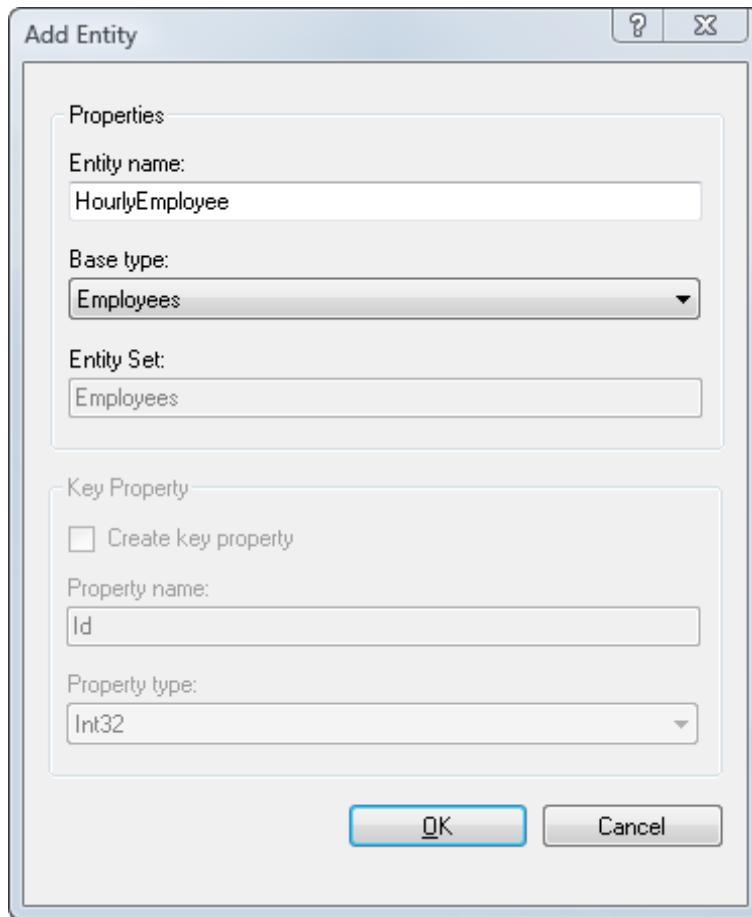
Discussion: In this walk through we will go through the steps of how to accomplish table per hierarchy using the existing employee table defined on the database. Screen shot below shows the employee table.

Employees (TBH)			
	Column Name	Data Type	Allow Nulls
key	EmployeeId	int	<input type="checkbox"/>
	Name	varchar(50)	<input type="checkbox"/>
	Email	varchar(50)	<input checked="" type="checkbox"/>
	Hours	int	<input checked="" type="checkbox"/>
	Rate	decimal(5, 2)	<input checked="" type="checkbox"/>
	Salary	decimal(5, 2)	<input checked="" type="checkbox"/>
	Type	char(2)	<input type="checkbox"/>

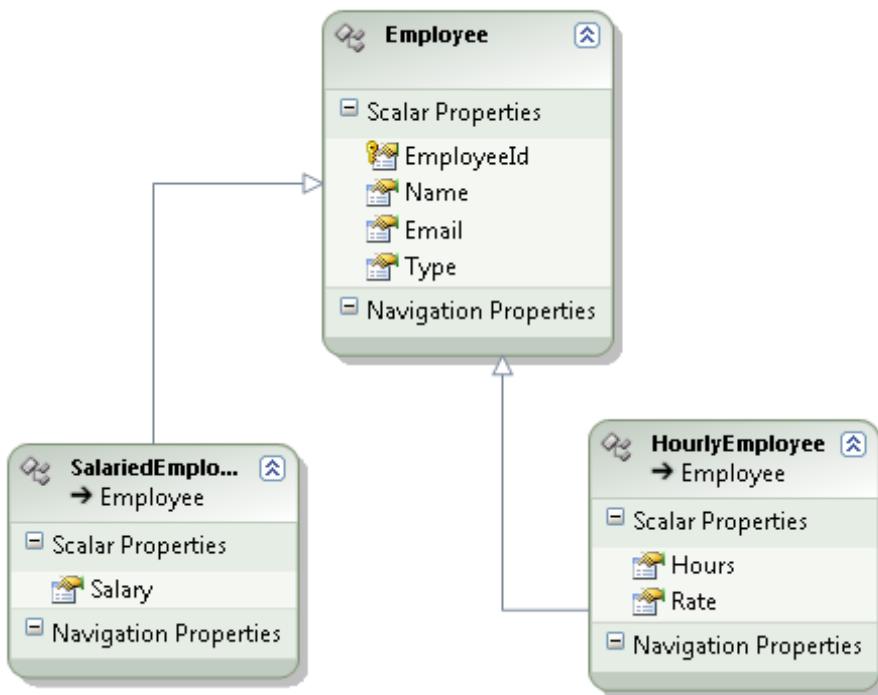
The Employee table contains all the columns required for each type of Employee. Since both types of Employee are in same table, we need to make columns specific to each Employee as allow nulls. Employee table also contains a Type column which is used to differentiate between each type of Employee. To import the table into entity data model, we will use Update Model from Database option. Screen shot below show the entity created on the design surface when we imported Employee table.



The next step is to create two entities, Hourly and Salaried Employee and move the fields' specific to each type to its own entity. One of the ways you can move fields is by cutting the fields from the base class Employee and pasting it to derived entity. To create an entity on the design surface, right click and choose Add entity. This will open up Add entity dialog which would allow us to give the entity a name and choose the base class the entity derives. For Hourly Employee, I have chosen the following settings.



The process of Salaried Employee is similar. After setting up the inheritance move the fields on the Employee class to its derived type. Figure below show the model after moving the appropriate fields to their entities.



Since we will be using Type column as a discriminator to determine what type of entity to instantiate, the column cannot be mapped to a property. Therefore we need to remove the Type column from Employee base class. To map the Hourly Employee to value on the Type column, select the table mapping for Hourly Employee, choose Employee table and add a condition where Type is equal HE for Hourly Employee. Figure below shows the mapping for Hourly Employee.

Mapping Details - HourlyEmployee

Column	Operator	Value / Property
Tables		
Maps to Employees	=	HE
When Type	=	HE
<Add a Condition>		
Column Mappings		
Hours : int	↔	
Rate : decimal	↔	
Salary : decimal	↔	Salary : Decimal
<Add a Table or View>		

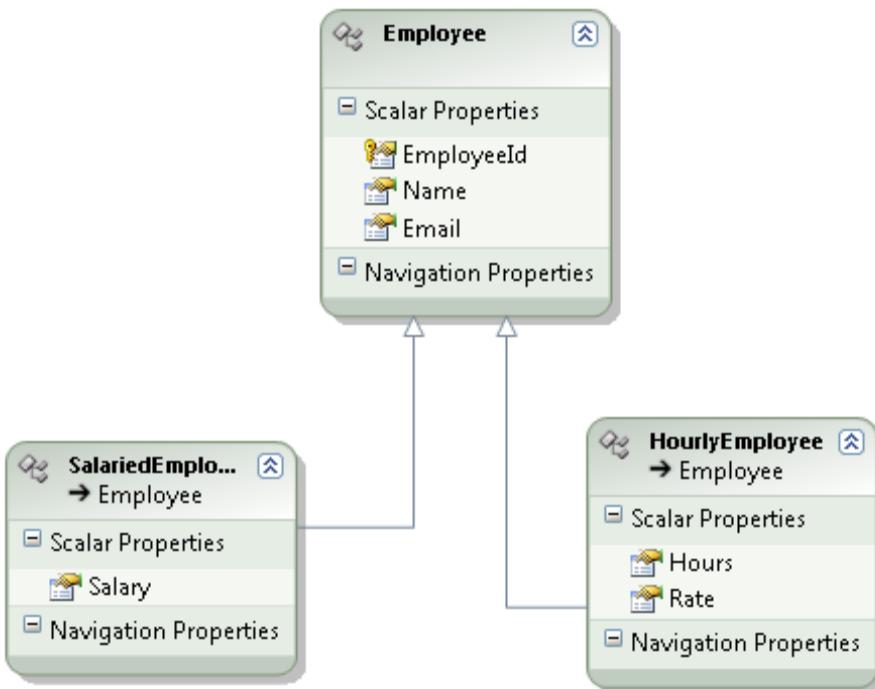
Similarly for Salaried Employee, select Employee table and set the condition for Type column with value of SE for Salaried Employee. Figure below shows the completed mapping for SalariedEmployee.

Column	Operator	Value / Property
Maps to Employees	=	SE
<Add a Condition>		
Hours : int	↔	Hours : Int32
Rate : decimal	↔	Rate : Decimal
Salary : decimal	↔	

If you try to validate the model at this time, you will get the following error.

**Error 1 Error 3023: Problem in Mapping Fragments starting at lines 139, 144, 149:
Column Employees.Type has no default value and is not nullable. A column value is required
to store entity data.**

This error is caused because we have not specified any Type value for Employee base class. If Employee entity cannot be mapped to a certain value and is only a base class, then we must set the Employee entity as an abstract class to clear the error. After making Employee entity abstract, the model should validate cleanly. The final model is shown below



To test the model, we create an object context, add different types of Employees and confirm that records got written to appropriate table. In the code below I am creating two types of Employees and adding them to AddToEmployees followed by SaveChanges. To confirm the insert happened successfully, using the second data context, I am retrieving each type of Employee and printing the properties of entity on the console window. Notice that our context does not expose any collection of derived types. The object context only exposes Employees collection which contains an entity of type Employee. To access a derived type, we need to use OfType operator passing in the derived type you needed returned.

```

var db = new STPTInh();
    var hourly = new HourlyEmployee { Name = "Alex",
Rate = 40, Hours = 40 };
    var salaried = new SalariedEmployee { Name =
"Chris", Salary = 90000 };
    db.AddToEmployees(hourly);
    db.AddToEmployees(salaried);
    db.SaveChanges();
    var db2 = new STPTInh();
    var huorly1 =
db2.Employees.OfType<HourlyEmployee>().First();

```

```

        Console.WriteLine("Name {0} Rate {1} Hours
{2}", huorly1.Name, huorly1.Rate, huorly1.Hours);

        var salary1 =
db2.Employees.OfType<SalariedEmployee>().First();
        Console.WriteLine("Name {0} Salary
{1}", salary1.Name, salary1.Salary);

```

Screen shot below shows the property values for employee printed on the console window.

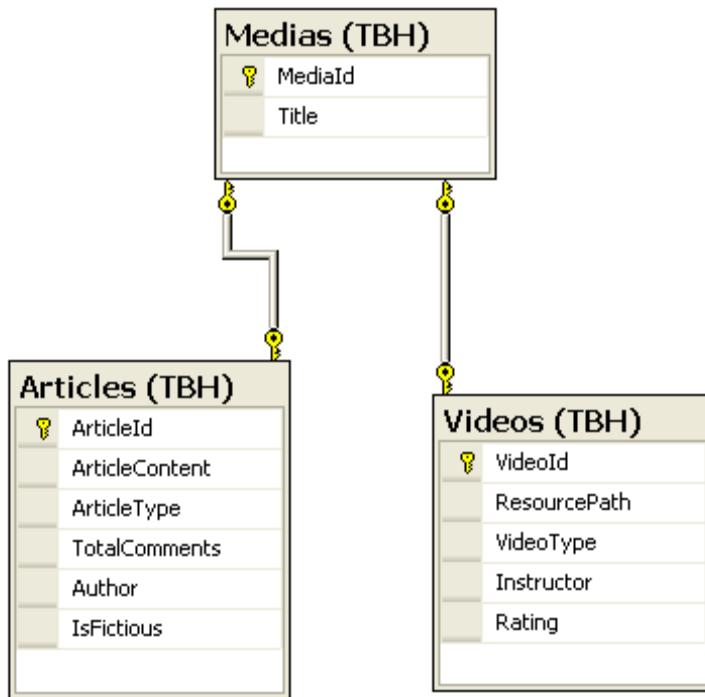
```

C:\Windows\system32\cmd.exe
Name Alex Rate 40.00 Hours 40
Name Chris Salary 90000.00
Press any key to continue . . .

```

5.1.3 Extending Table per Type with Table per Hierarchy

Problem: Figure below shows the current table structure for Media defined on the database.



On the database model above, Media is top level table that contains all different types of Media. Every Media has a title declared on Media table. Media is divided into two tables; Articles and Videos. Every Article has Article Content and an author. Article Type defines the type of the articles Article table contains. Articles can be of two types; Stories or Blog Posting. For Articles of type Story, we are also capturing IsFiction column. For Blog Posting an additional field TotalComments is captured. Similarly on Videos table, a video can be of two types; Education and Recreational Video. The type of Video is differentiated by VideoType column. Every video has Resource Path that defines the location of the video. EducationalVideo also captures Instructor and RecreationalVideo captures the rating of the video. We want to model this structure using Table per Type for Media and use Table Per Hierarchy to segregate different types of Articles and Videos.

Solution: To accomplish the above database structure, the first step is to import the model into the database using import wizard and then delete all the associations created by the designer. Extend the Media class with two derived entities; Videos and Articles. Make Media, Video and Articles as abstract class since they do not have direct mapping in the database. Then extend the

Video class with two entities EducationVideos and RecreationalVideos and move the fields specific to a derived entity to its class. Similarly for extend the Article entity with Story and Blog Posting and move specific fields from Article entity to its derived classes. The last step is to map the entities to the database tables by using the mapping window.

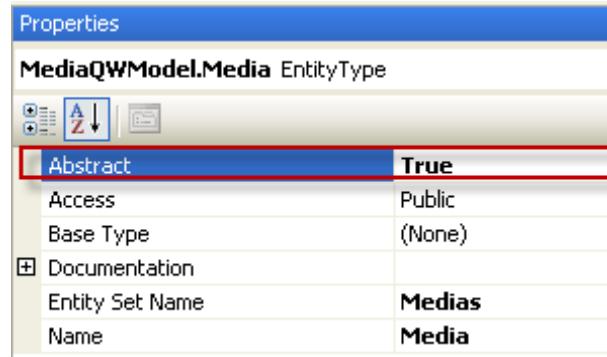
Solution: In this walk through, we will go through the steps of moving the Media tables to use Table per Type and then extending the model further to utilize Table per Hierarchy for Articles and Videos.

1. On the import wizard, import Media, Articles and Video table. Screen shot below shows the entity data model after completing the import wizard.

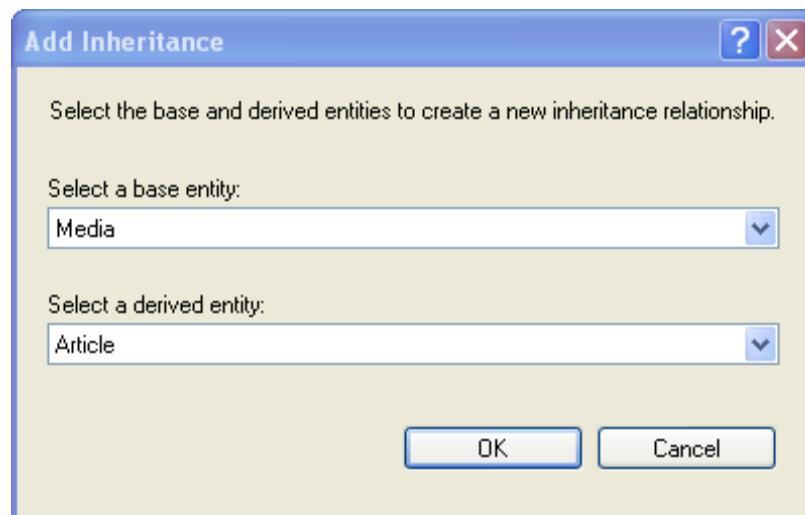


2. Delete all the associations created by the designer by selecting the association and clicking delete.

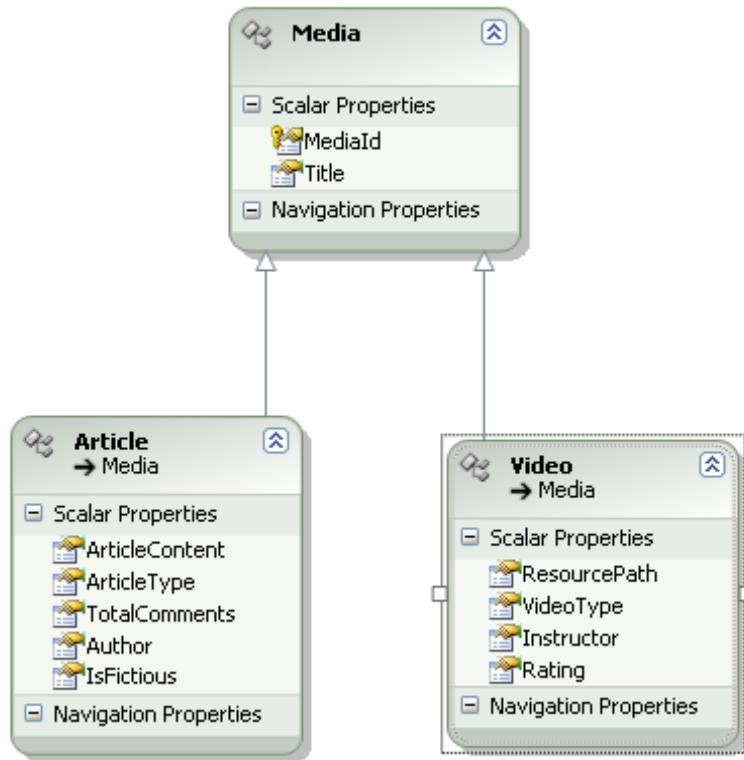
3. Make Media, Articles and Videos as abstract class because they do not have any direct translation in the database and only serves as a base class. Figure below shows a snapshot of the property window to make an entity abstract.



4. Make Article and Video entity inherit from Media entity. Remove ArticleId and VideoId because inheriting from Media entity we automatically gives MediaId as the entity which we can later used for mapping. To inherit Article from Media entity, right click Media and select Inheritance. On Add inheritance window select Media as the base entity and Article as the derived entity as shown below.



Similar steps needs to be performed to make Video entity derive from Media. Figure below shows the model after applying inheritance and removing ArticleId and VideoId



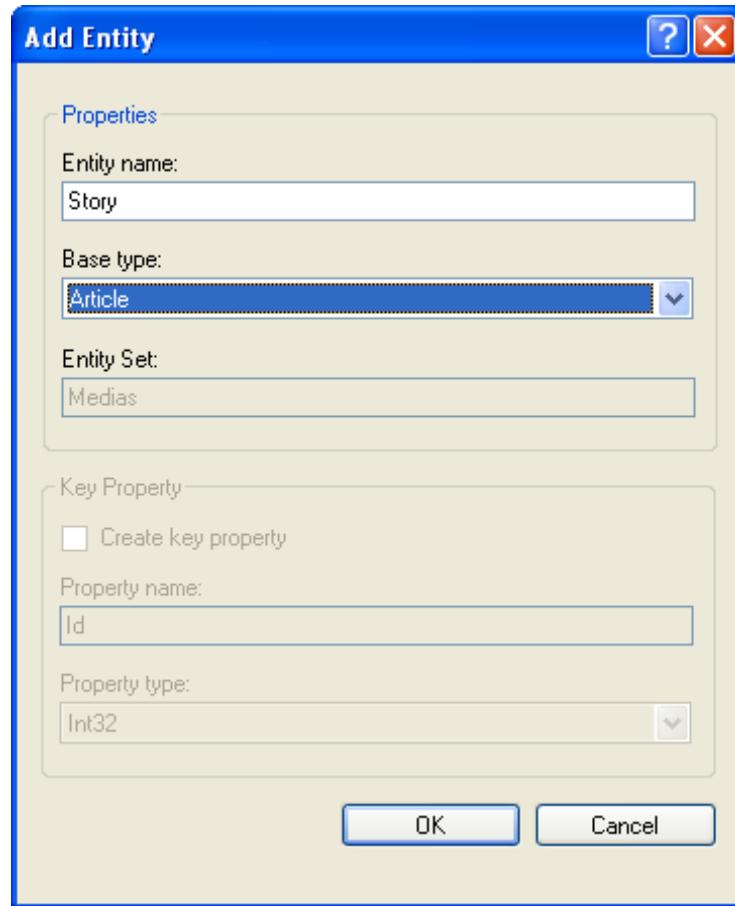
- To configure Article Mapping, select Article entity and open up mapping window. Map the articleid on Articles table to MediaId on the properties column as shown below.

Mapping Details - Article

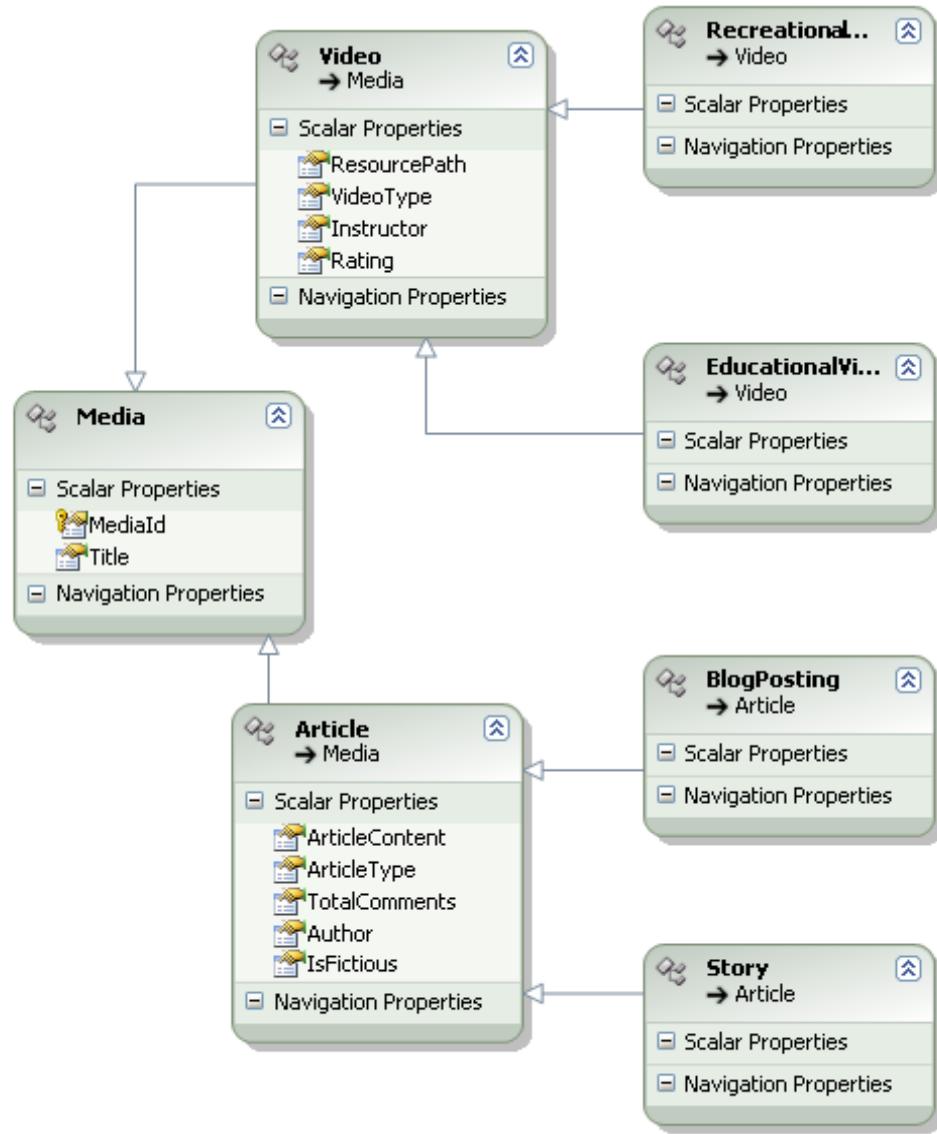
Column	Operator	Value / Property
Tables		
Maps to Articles		
<Add a Condition>		
Column Mappings		
ArticleId : int	↔	MediaId : Int32
ArticleContent : varchar	↔	ArticleContent : String
ArticleType : varchar	↔	ArticleType : String
TotalComments : int	↔	TotalComments : Int32
Author : varchar	↔	Author : String
IsFictitious : bit	↔	IsFictitious : Boolean

Perform similar steps for Video entity to map ArticleId column to MediaId property inherited from Media entity.

6. Article entity would have two derived types Story and BlogPosting. To create Story entities right click on the design surface and select entity. Give the entity name, Story and choose Article as the derived type. Screen shot below shows Add entity wizard values for Story entity deriving from Article.

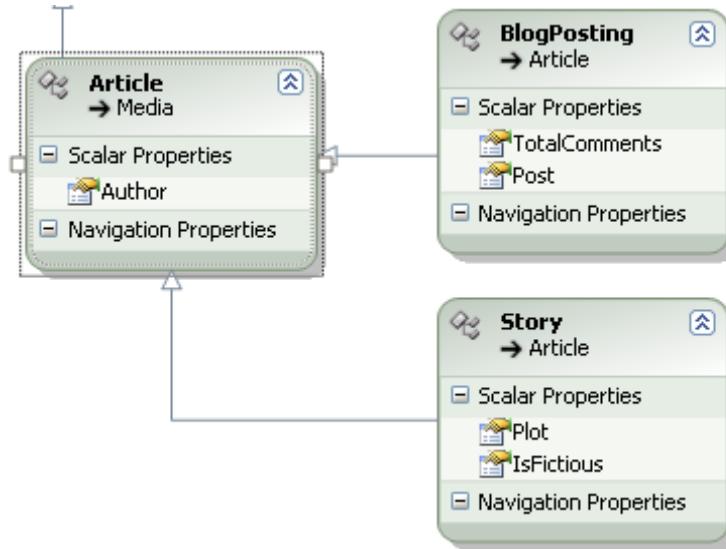


Similar steps needs to be performed to create BlogPosting entity and extend it from Article. To extend Video entity create two entities EducationalVideo and RecreationalVideo and derive it from Video entity. Figure below shows the updated entity data model.



7. We need to move the fields on the Article entity specific to the derived type to its own entity. Since every Article will have an Author, we can leave the Author property. Although both Story and BlogPosting will have ArticleContent but we want to call it differently when creating BlogPosting as compared to Story so we need to move ArticleContent and rename it on entity to be more meaningful for that entity. One of the ways you can move fields is by cutting the fields and pasting it to the other entity. We also need to remove ArticleType property from Article entity because ArticleType will be used for mapping the Article to derived types and entity framework does not allow mapping

discriminator columns to properties because that could invalidate the model. Figure below shows the updated model for Article with properties moved to Story and BlogPosting where needed.

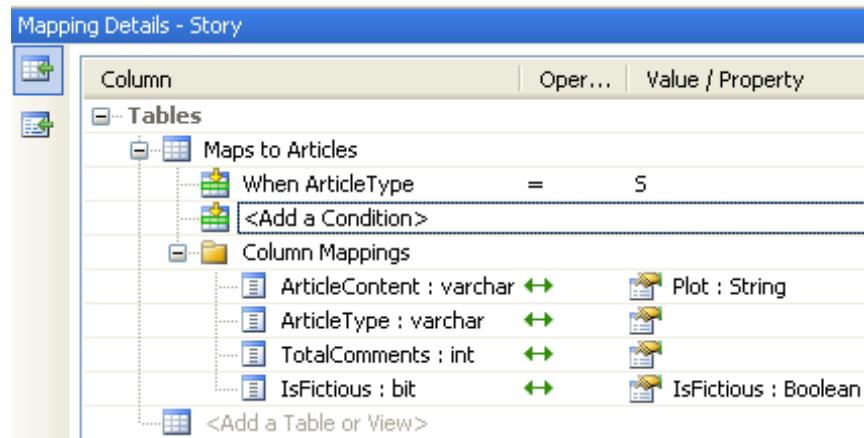


8. To map BlogPosting select Article table in the mapping window and add a condition of ArticleType of B which means that if the ArticleType is B, it is a Blog Posting. Also map TotalComments property to TotalComments column and map Post property to ArticleContent column on Article table as shown below in the mapping window.

Mapping Details - BlogPosting

Column	Operator	Value / Property
Tables		
Maps to Articles		
When ArticleType	=	B
<Add a Condition>		
Column Mappings		
ArticleContent : varchar	↔	Post : String
ArticleType : varchar	↔	
TotalComments : int	↔	TotalComments : Int32
IsFictitious : bit	↔	

9. To map Story entity set ArticleType equal to S as a condition and map rest of the columns to the Article table as shown below.



10. For Recreationalvideos and EducationalVideos, leave the ResourcePath on Video entity, delete VideoType property because we will use VideoType as a discriminator column and move rest of fields from Video entity to their derived classes. Map RecreationalVideo entity to Video table in the mapping window with a condition of VideoType equal to R and for EducationalVideo set the condition to E. Figure below shows the mapping for Recreational and EducationalVideo

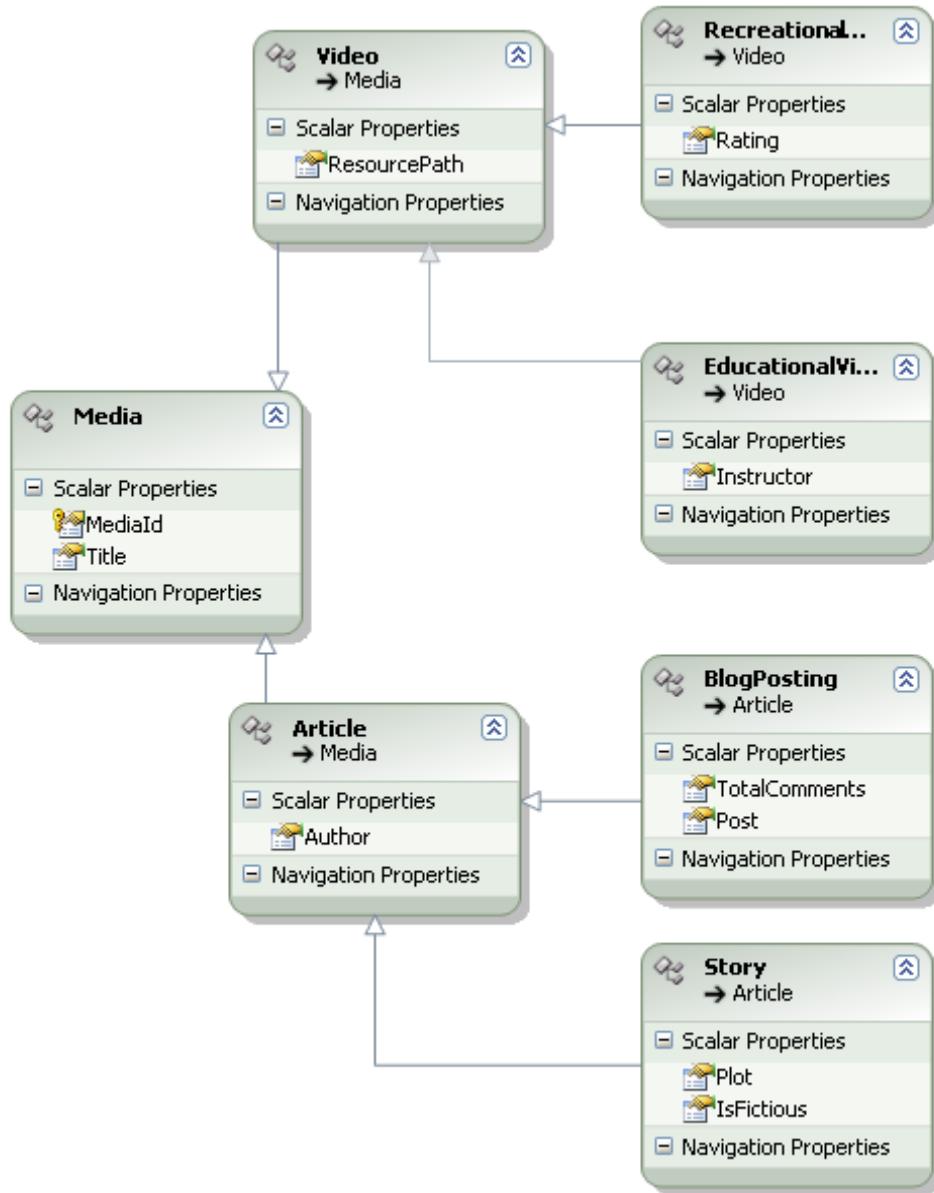
Mapping Details - RecreationalVideo

Column	Oper...	Value / Property
Tables		
Maps to Videos		
When VideoType	=	R
<Add a Condition>		
Column Mappings		
VideoType : varchar	↔	
Instructor : varchar	↔	
Rating : int	↔	Rating : Int32

Mapping Details - EducationalVideo

Column	Oper...	Value / Property
Tables		
Maps to Videos		
When VideoType	=	E
<Add a Condition>		
Column Mappings		
VideoType : varchar	↔	
Instructor : varchar	↔	Instructor : String
Rating : int	↔	

After configuring the mapping, the completed model should look like a figure below.



To test the above model created, we can create an instances of different types of Videos and Articles, save them to the database and using second datacontext, retrieve specific types and confirm if the results matches to what we have inserted on the database. On the code below, I am creating an instance of **BlogPosting**, **Story**, **Educational Video**, **RecreationalVideo** and saving it to the database by calling **SaveChanges**. Then using the second datacontext, I am retrieving the Count for **Media**, the top level entity in the **Media** class. I am also retrieving the Count for **Articles** and **Video** which should give us the

result of two for both because Articles contains both Story and BlogPosting and Videos contains both Educational and RecreationalVideos. Finally I am also retrieving the Count for Medias that are in different hierarchies for Table Per Type. Like in the example, I am retrieving the count where Media is an EducationalVideo or a Story and from our entity data model we can see that EducationalVideo belongs to Video base class and Story derives from Article base class.

```

var db = new MediaPTTBH();
    //articles
    var blogposting = new BlogPosting { Title = "Asp.net MVC", Author = "Scott", Post = "mvc content", TotalComments = 50 };
        var story = new Story { Title = "Alice In Wonderland", Author = "Charles", Plot = "story", IsFictitious = true };

    //videos
    var educvideo = new EducationalVideo { Instructor = "Zee", ResourcePath = "Asp.netintro.wmv", Title = "Asp.net Video" };
    var recreatioanlvid = new RecreationalVideo { Title = "WorldCup", Rating = 5, ResourcePath = "cricket.wmv" };

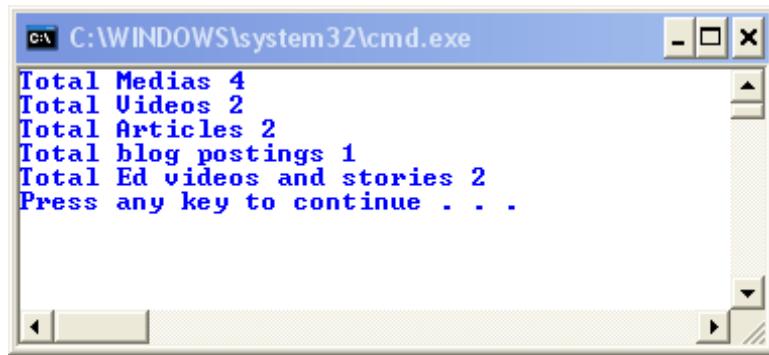
    db.AddToMedias(blogposting);
    db.AddToMedias(story);
    db.AddToMedias(educvideo);
    db.AddToMedias(recreatioanlvid);
    db.SaveChanges();

    var db2 = new MediaPTTBH();
    //getting count of total medias using esql
    Console.WriteLine("Total Medias " + db2.Medias.Count());
    Console.WriteLine("Total Videos " +
db2.Medias.OfType<Video>().Count());
    Console.WriteLine("Total Articles " +
db2.Medias.OfType<Article>().Count());
    Console.WriteLine("Total blog postings " +
db2.Medias.OfType<BlogPosting>().Count());

    var edvideoandstory = db.Medias.Where(m => m is EducationalVideo || m is Story);
        Console.WriteLine("Total Ed videos and stories " +
edvideoandstory.Count());

```

Screen shot below shows the result of the above cod in output window.



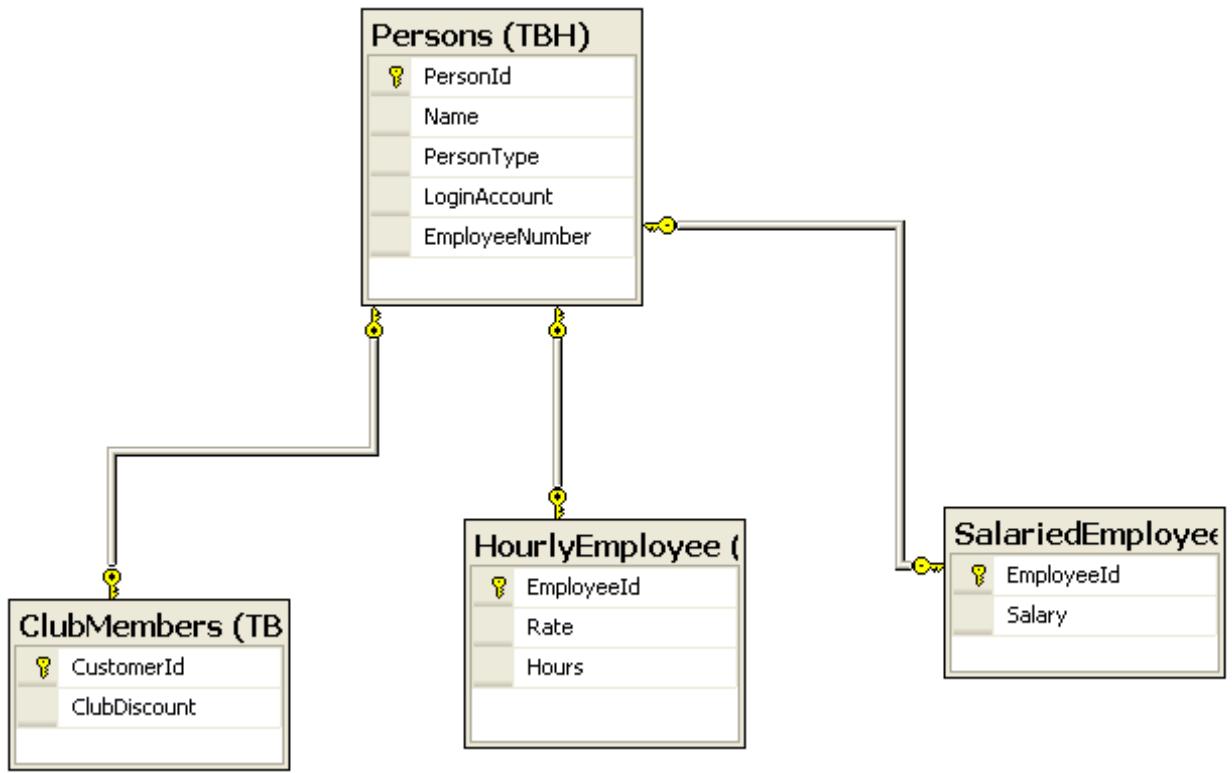
A screenshot of a Windows Command Prompt window titled "C:\WINDOWS\system32\cmd.exe". The window displays the following text in blue font:

```
Total Medias 4
Total Videos 2
Total Articles 2
Total blog postings 1
Total Ed videos and stories 2
Press any key to continue . . .
```

In this tutorial, we covered how we can use Table Per Type and Table Per Hierarchy to model our table structure. We also wrote queries against the model by using OfType and Where operator to filter and return correct entities regardless of the depth of inheritance hierarchy.

5.1.4 Extending Table per Hierarchy with Table per Type

Problem: Diagram below shows the current structure and relationship for Person, Employee and Customer table in the database.



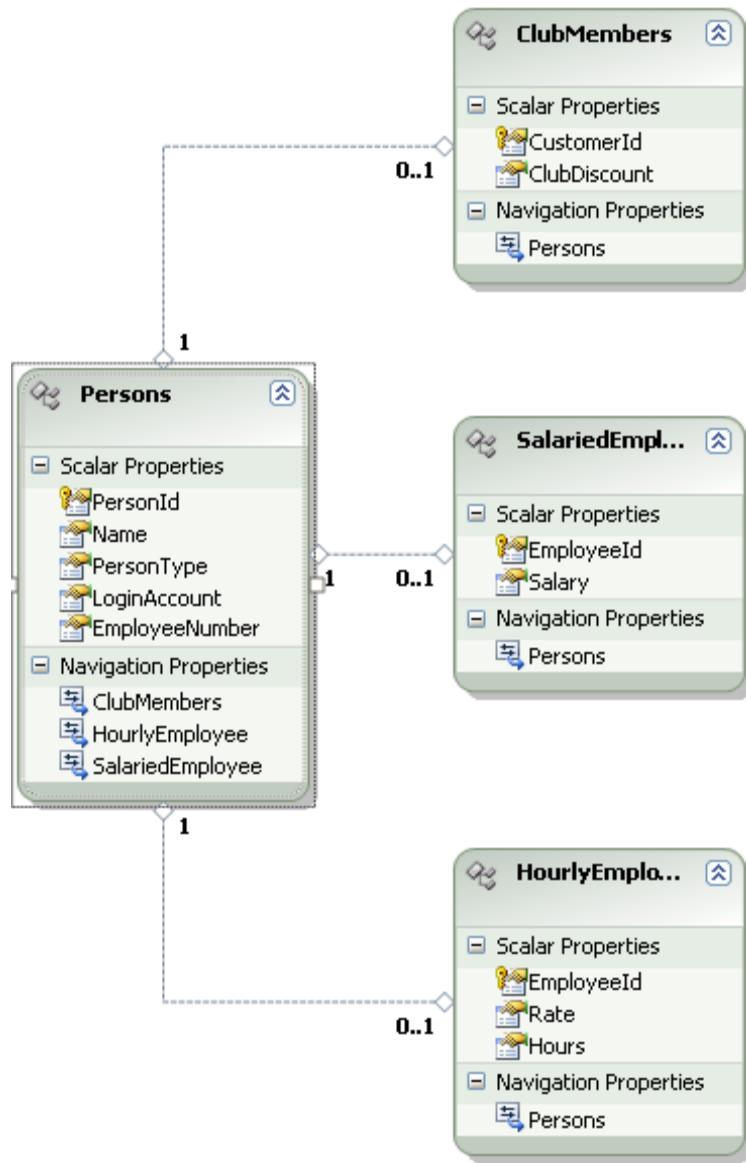
In the above table structure, we have a Person table that contains Employees and Customers identified by PersonType. If the PersonType is E, it is an Employee and if the PersonType is C, it is a Customer. Employees are subdivided into two additional types HourlyEmployee and SalariedEmployee and have separate tables as shown in the above figure. The primary key EmployeeId in Salaried and Hourly Employee is also the foreign key for Person table. ClubMembers table includes additional information about Customers who have ClubMembership with the Company. You want to implement the given table structure using Table Per Hierarchy for Customer and Employees and use table per Type for derived types of Employees and Customers.

Solution: Import the table structure using import wizard. After wizard completes, delete all the associations created by the wizard. Create two entities Customers and Employee and inherit them from Person entity. Make Person, Employee, Customers as an abstract class since this entity have no direct equivalent in our database and mainly serves as a base class. Make SalariedEmployee and HourlyEmployee inherit from Employee entity created

earlier. Move EmployeeNumber from Person entity to Employee entity because EmployeeNumber is a property specific to Employee entity. Make ClubMembers inherit from Customer entity and move LoginAccount from person entity to Customer entity. The last step is to map all the entities to the store definition imported by the wizard.

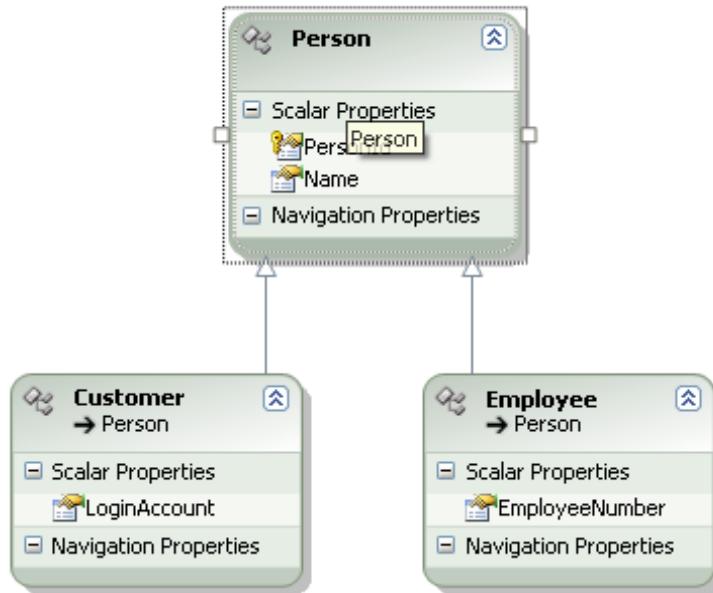
Discussion: Steps below outlines using Table Per Hierarchy with Table Per Type to model the database structure given above.

1. Import the database structure and select Person, Hourly, Salaried Employee and ClubMembers table. Figure below shows the model imported after the wizard is completed.



2. Delete all the associations created by the wizard, create two entities Customer and Employee and move employeeNumber to Employee entity and LoginAccount to Customer entity. Make Customer and Employee entity derive from Person entity and mark Person and Employee entity as abstract. We will not mark Customer entity as abstract because all persons with PersonType of C are considered Customers. If Customers have a ClubMembership, additional information would be stored in ClubMembership table. Delete PersonType property from Person entity because we will use

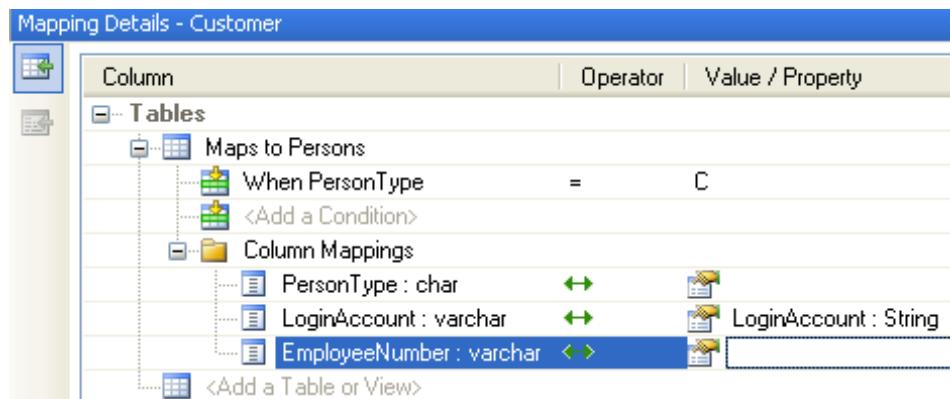
PersonType column as a discriminator column to map inheritance. Figure below shows the mapping after completing the above steps.



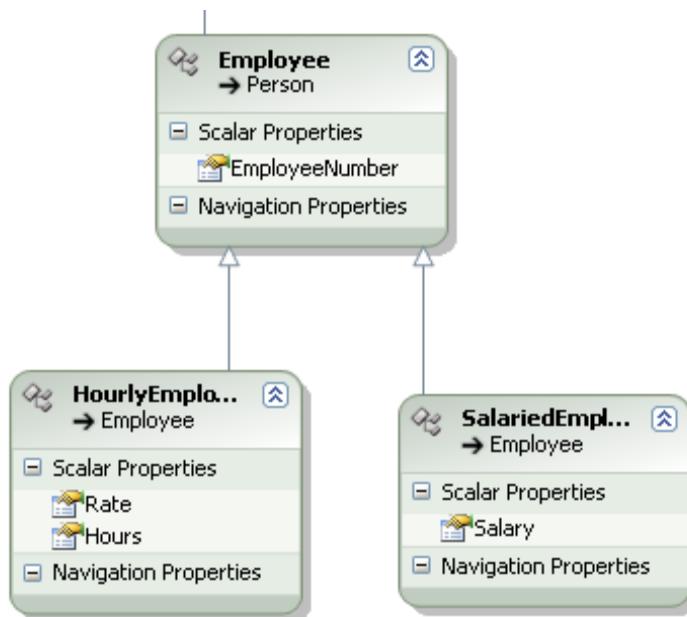
3. Map the Employee entity to Person table where PersonType is E. Map EmployeeNumber to EmployeeNumber on Person table. Figure below shows the mapping window for Employee entity.

Column	Operator	Value / Property
Maps to Persons	=	E
PersonType : char	↔	EmployeeNumber : String
LoginAccount : varchar	↔	LoginAccount : varchar
EmployeeNumber : varchar	↔	EmployeeNumber : String

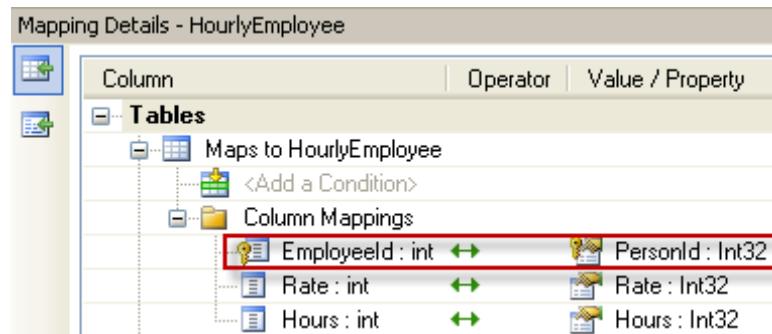
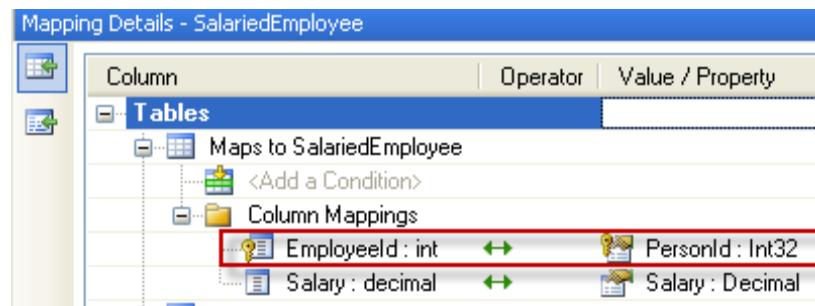
4. Map Customer entity to Person table where PersonType is C for Customers. Map LoginAccount to LoginAccount on Person table. Figure below shows the mapping window for Customer entity.



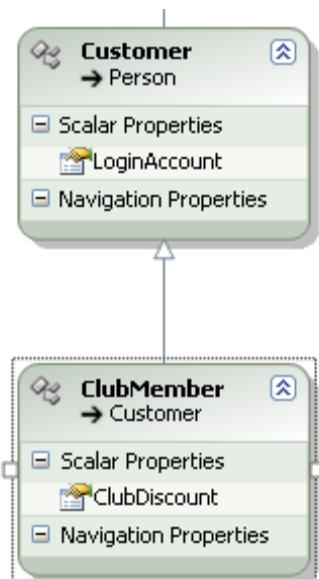
5. Make SalariedEmployee and HourlyEmployee derive from Employee entity and delete EmployeeId from both entities because we will use PersonId entity key on Person entity to map to SalariedEmployee table. Figure below shows the update model for Employee and its derived entities.



6. For SalariedEmployee mapping, map EmployeeId column to PersonId property on Person table. Perform similar operation to HourlyEmployee entity. Figure below shows the mapping for both Salaried and Hourly Employee entities.



7. Make ClubMembers derive from Customer entity and remove CustomerId entity key property because we will use PersonId entity key from Person entity. Map CustomerId column on ClubMembers table to PersonId property on the mapping window. Figure below shows the ClubMember and Customer entity inheritance structure and its mapping.

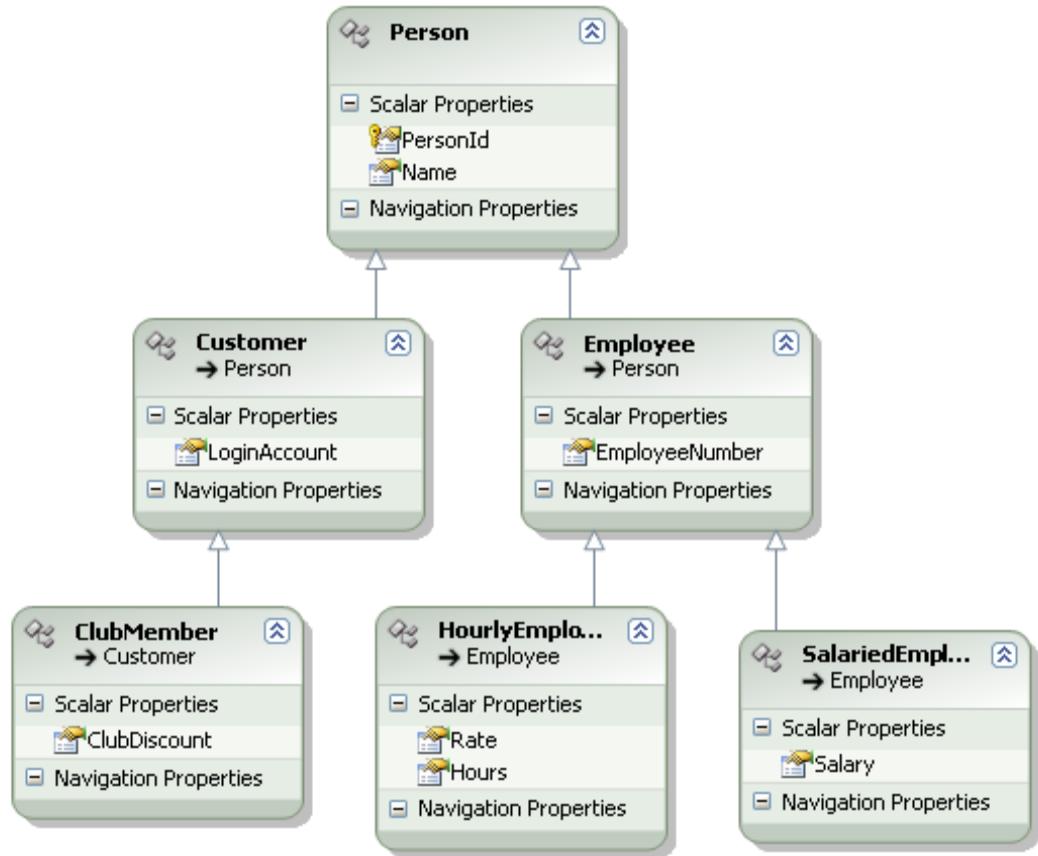


Mapping Details - ClubMember

Column	Operator	Value / Property
CustomerId : int	↔	PersonId : Int32
ClubDiscount : int	↔	ClubDiscount : Int32

The screenshot shows the EntityDataSource Configuration tool with the mapping details for the **ClubMember** entity. It displays the mapping between the database columns (**CustomerId** and **ClubDiscount**) and the corresponding EntityDataSource properties (**PersonId** and **ClubDiscount**). The mapping for **CustomerId** is highlighted with a red box.

Completed entity data model with appropriate inheritance structure is shown below.



To test our model, we can create instances of each derived type add it to Persons entity set and save them to the database. Then using another datacontext query for each hierarchy to ensure that we are returned the correct count of Persons that matches the Person entity inserted earlier. In the code below, I am creating instance of hourly and salaried employee and regular and club member customer and saving them to the database. Using the second datacontext, I use OfType operator to retrieve Count for Persons that are Employee type, Customer Type and customers who have ClubMembership. Count for Employee type is because we create 1 instance of Hourly Employee and 1 instance of Salaried Employee. Similarly we added 1 regular customer and one club membership customer, the count returned for customer is two also.

```

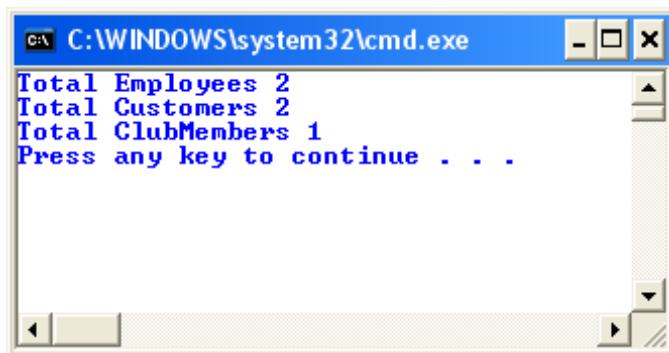
var db = new EmployeeTPHTPT();
    var hourlyemp = new HourlyEmployee { EmployeeNumber = "123", Name
= "Zee", Hours = 40, Rate = 60 };
  
```

```

        var salariedemp = new SalariedEmployee { EmployeeNumber = "111",
Name = "Alex", Salary = 90000 };
        var regularcustomer = new Customer { Name = "John", LoginAccount
= "john" };
        var clubmember = new ClubMember { Name = "Craig", LoginAccount =
"craig", ClubDiscount = 100 };
        db.AddToPersons(hourlyemp);
        db.AddToPersons(salariedemp);
        db.AddToPersons(regularcustomer);
        db.AddToPersons(clubmember);
        db.SaveChanges();
        var db2 = new EmployeeTPHTPT();
        Console.WriteLine("Total Employees " +
db2.Persons.OfType<Employee>().Count());
        Console.WriteLine("Total Customers " +
db2.Persons.OfType<Customer>().Count());
        Console.WriteLine("Total ClubMembers " +
db2.Persons.OfType<ClubMember>().Count());

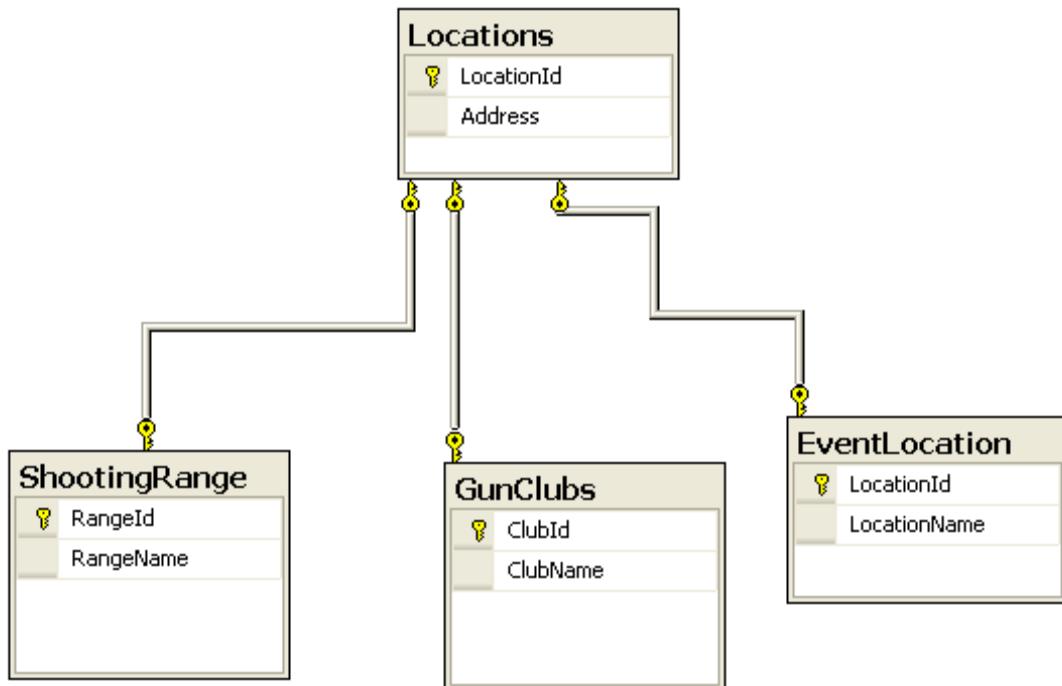
```

Output printed on console window confirms our result.

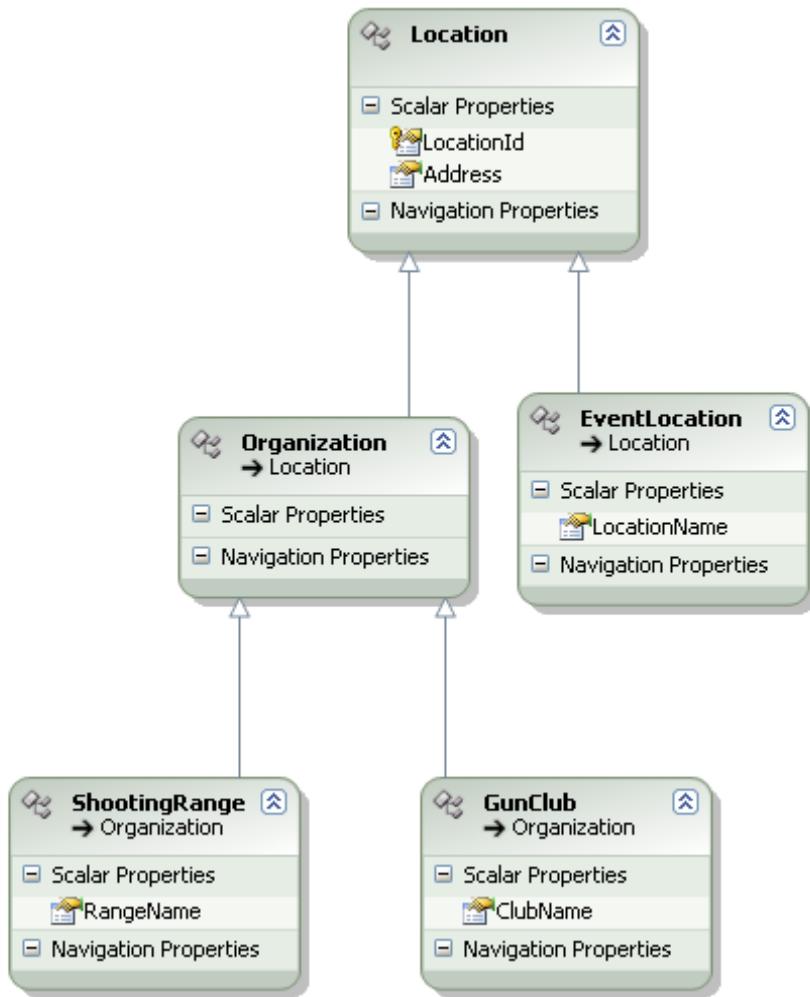


5.1.5 Creating additional hierarchy for TPT using QueryView

Problem: The database structure given to you is as follows



In the above database diagram, you have a location table which has Address information for all 3 different locations. EventLocation is a type of location where events are organized and carries an additional column `LocationName`. GunClubs is also location defined by foreign key constraint from Location table. GunClub contains an additional column `ClubName` which identifies the name of Club. ShootingRange is also a type of Location with additional column `RangeName`. You want to implement the above table structure using Table Per Type inheritance. However you want **ShootingRange** and **GunClubs** to have an additional hierarchy called **Organizations** from which they inherit from. This additional inheritance layer will allow you to program against both **ShootingRange** and **GunClubs** without duplicating code. The completed entity data model should look like the one below.

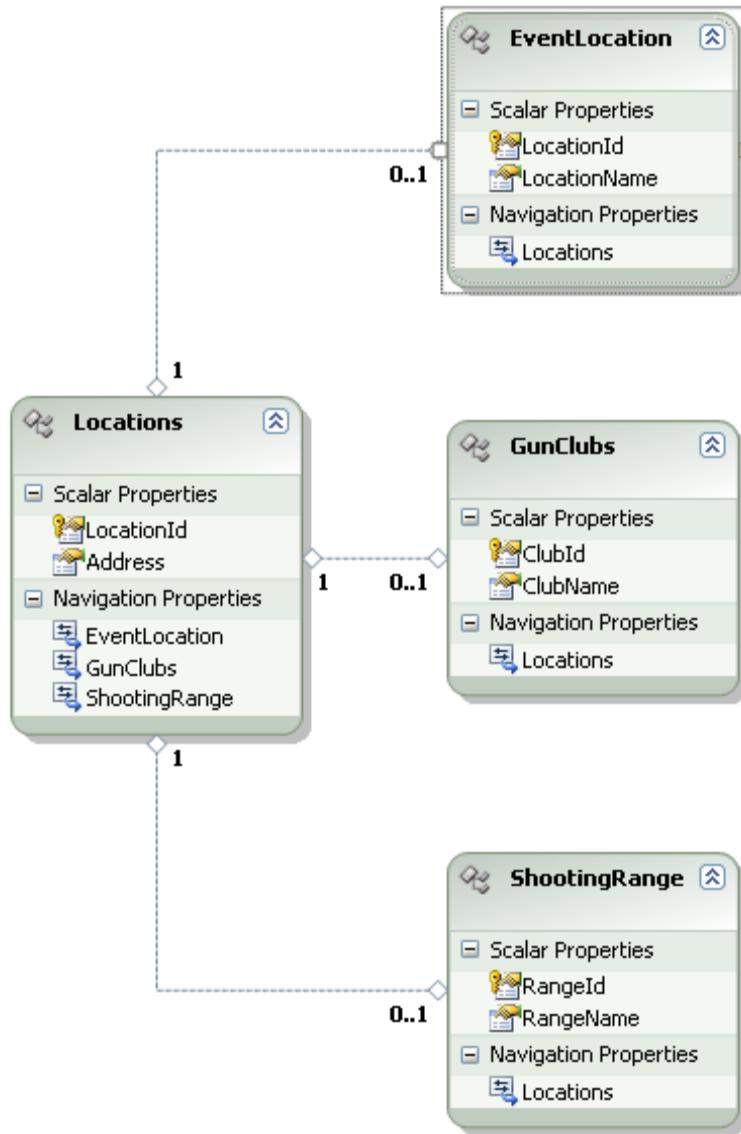


Solution: Import Locations, Gun Clubs and Shooting Ranges tables using the import wizard. Delete all the associations created by the wizard. Make EventLocation Inherit from Location entity. Create Organization entity deriving from Location entity and make ShootingRanges and Gun Clubs inherit from Organization. Since Organization entity is not available on the database and was created to facilitate applying business rules to both ShootingRange and GunClubs, it cannot be mapped. When you try to validate the model entity framework would complain that organization entity is not mapped. The only way to fix the validation error is to completely remove the table to entity mapping provided by the framework and use QueryViews. With QueryView, you write a query that extracts data from tables defined on the store model and maps the results to the entities on the conceptual model.

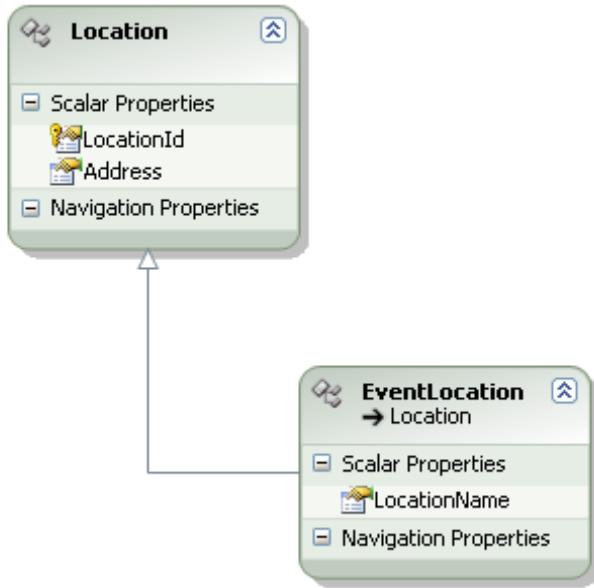
As a result entity framework does not have any knowledge of how to inserts, update and delete those entities. Saving the entities would require creation of insert, update and delete stored procedure, defining them in the store model and mapping the procedures to conceptual model in the msl.

Discussion: In the previous discussion, I have covered how to model Table per Type inheritance, so I will not cover in depth of how to model the current database structure to Table per Type. We will be discussing how to add additional layer of inheritance not given by the database and how to write Query View that maps to the current inheritance structure created on the designer. In addition we will create stored procedures for inserts, updates and deletes and map them to our entity so we can save the entities to the database.

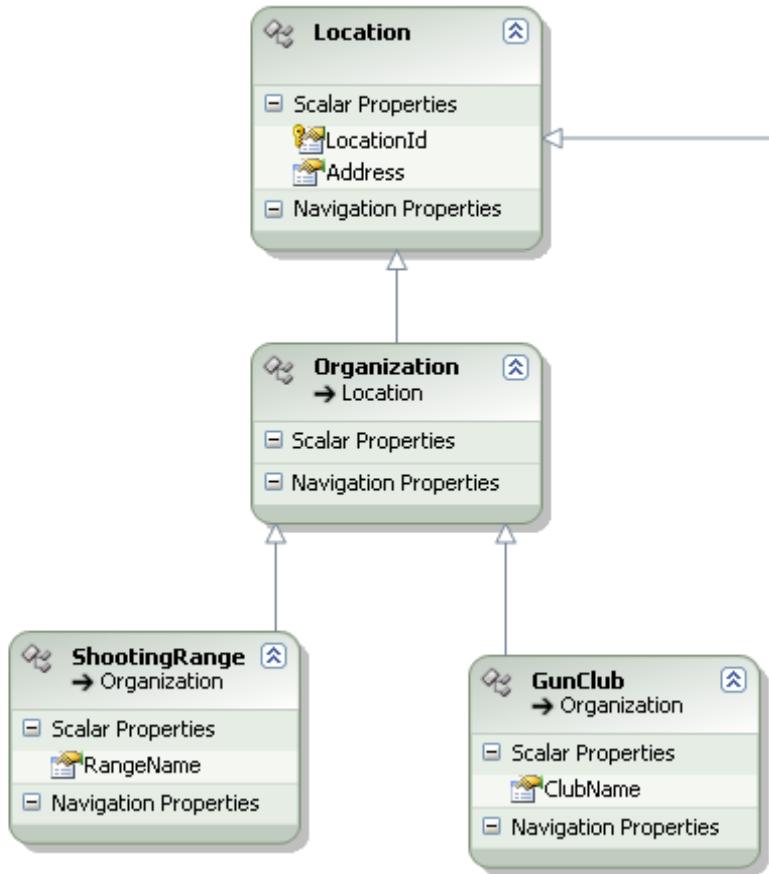
1. Import Location, EventLocation, GunClub and ShootingRange table into entity data model using the wizard. Figure below shows the model imported by the wizard.



2. Remove all the associations created by the designer. Make EventLocation inherit from Location entity and remove LocationId entity key from EventLocation because we get LocationId from base Location entity. Remove table mapping for both Location and EventLocation by opening up mapping window and removing the table. We are removing the table mapping because when we use Query View, we can no longer take advantage of the mappings offered by EF and have to map the entities using stored procedures. Figure below shows the update model for Location and EventLocation.



3. Create Organization entity inheriting from Location entity. Ensure that ShootingRange and GunClubs inherit from Organization entity. Remove all table mappings for ShootingRange and GunClubs and delete Rangid entity key from ShootingRange and ClubId entity key from GunClub entity. Figure below shows the updated model for Organization hierarchy.



- When we add additional layer of inheritance like **Organization** that cannot be mapped to any table, entity framework complains that the mapping is not specified. One way to fix it is to use **QueryView** to define how to get results from the store model and map it to the current hierarchy structure we have created. Since **QueryView** is not supported by the designer, we have to create an entityset mapping for locations inside of msl and specify our **QueryView**. Query below shows how to retrieve data from store model and map it to current hierarchy structure.

```

<EntityTypeMapping Name="Locations">
    <QueryView>
        select value
        case
        when (el.LocationId is not null) then
            EcommerceModel2.EventLocation(l.LocationId,l.Address,el.LocationName)
        when (gc.ClubId is not null) then
            EcommerceModel2.GunClub(l.LocationId,l.Address,gc.ClubName)
        when (range.RangeId is not null) then
    
```

```

EcommerceModel2.ShootingRange(l.LocationId,l.Address,range.RangeName)
    END
    from EcommerceModel2StoreContainer.Locations as l
    left join EcommerceModel2StoreContainer.EventLocation as el on
l.LocationId = el.LocationId
    left join EcommerceModel2StoreContainer.GunClubs as gc on
l.LocationId = gc.ClubId
    left join EcommerceModel2StoreContainer.ShootingRange as range
on l.LocationId = range.RangeId
</QueryView>
</EntityTypeMapping>

```

On the above code, I am joining locations table with EventLocation, GunClubs and ShootingRanges using left outer join. If there is a matching row found in eventlocation, I am creating an instance of EventLocation entity. If there is a matching row found in GunClub, I am creating a GunClub entity and finally if there is match found with ShootingRange table, I create a shootingRange entity. This means that when we access the Location EntitySet, it will contains different types of Locations but if you try to narrow your results to bring only certain types of Locations like Locations which are Organizations, entity framework has to bring all the locations from the database and then execute our QueryView to map the results to appropriate entity and filter the QueryView to return only Organizations. This could be an expensive operation because entity framework has to bring all the locations from the database apply the filter in memory. In the next example, we will discover different ways of optimizing queries that require inheritance and how we can facilitate entity framework to apply the filter operation at the database level so we don't bring any additional rows then what is needed by the application.

If you try to save the above entities to using ObjectContext, entity framework will give a runtime error because it cannot save entities that are created using QueryView. Therefore we need to map the entities to stored procedures. For further information on how to map stored procedures to entity, please look at the stored procedure section of the book.

Code below shows the mapping of entity to stored procedure.

```

<EntityTypeMapping TypeName="EcommerceModel2.EventLocation">
    <ModificationFunctionMapping>

```

```

<InsertFunction
  FunctionName="EcommerceModel2.Store.InsertEvent">
  <ScalarProperty Name="LocationName" />
  ParameterName="LocationName" />
  <ScalarProperty Name="Address" ParameterName="Address" />
  <ResultBinding Name="LocationId" ColumnName="LocationId" />
</InsertFunction>
<UpdateFunction
  FunctionName="EcommerceModel2.Store.UpdateEvent">
  <ScalarProperty Name="LocationName" />
  ParameterName="LocationName" Version="Current" />
  <ScalarProperty Name="Address" ParameterName="Address" />
  Version="Current" />
  <ScalarProperty Name="LocationId" />
  ParameterName="LocationId" Version="Current" />
</UpdateFunction>
<DeleteFunction
  FunctionName="EcommerceModel2.Store.DeleteEvent">
  <ScalarProperty Name="LocationId" />
  ParameterName="LocationId" />
</DeleteFunction>
</ModificationFunctionMapping>
</EntityTypeMapping>
<EntityTypeMapping TypeName="EcommerceModel2.GunClub">
  <ModificationFunctionMapping>
    <InsertFunction
      FunctionName="EcommerceModel2.Store.InsertClub">
        <ScalarProperty Name="ClubName" ParameterName="ClubName" />
        <ScalarProperty Name="Address" ParameterName="Address" />
        <ResultBinding Name="LocationId" ColumnName="LocationId" />
      </InsertFunction>
      <UpdateFunction
        FunctionName="EcommerceModel2.Store.UpdateClub">
          <ScalarProperty Name="ClubName" ParameterName="ClubName" />
          Version="Current" />
          <ScalarProperty Name="Address" ParameterName="Address" />
          Version="Current" />
          <ScalarProperty Name="LocationId" />
        ParameterName="LocationId" Version="Current" />
      </UpdateFunction>
      <DeleteFunction
        FunctionName="EcommerceModel2.Store.DeleteClub">
          <ScalarProperty Name="LocationId" />
        ParameterName="LocationId" />
      </DeleteFunction>
</ModificationFunctionMapping>
</EntityTypeMapping>
<EntityTypeMapping TypeName="EcommerceModel2.ShootingRange">
  <ModificationFunctionMapping>
    <InsertFunction
      FunctionName="EcommerceModel2.Store.InsertRange">
        <ScalarProperty Name="RangeName" />
        ParameterName="RangeName" />
        <ScalarProperty Name="Address" ParameterName="Address" />
        <ResultBinding Name="LocationId" ColumnName="LocationId" />
      </InsertFunction>

```

```

<UpdateFunction
  FunctionName="EcommerceModel2.Store.UpdateRange">
  <ScalarProperty Name="RangeName" ParameterName="RangeName" Version="Current"/>
  <ScalarProperty Name="Address" ParameterName="Address" Version="Current" />
  <ScalarProperty Name="LocationId" ParameterName="LocationId" Version="Current" />
</UpdateFunction>
<DeleteFunction
  FunctionName="EcommerceModel2.Store.DeleteRange">
  <ScalarProperty Name="LocationId" ParameterName="LocationId" />
</DeleteFunction>
</ModificationFunctionMapping>
</EntityTypeMapping>

```

On the above code, we are mapping GunClub, ShootingRange and EventLocation to insert, update and delete stored procedure defined on the store model. Code below shows the store model which contains the stored procedure responsible for taking the parameters mapped on msl and inserting the entry into appropriate tables and returning identity column back to msl to map it to the entity key property on the entity on the conceptual model.

```

<Function Name="InsertEvent" IsComposable="false" Schema="dbo">
  <CommandText>
    declare @locid int
    insert into Locations(Address) values (@Address)
    set @locid = SCOPE_IDENTITY()
    insert into EventLocation(LocationId, LocationName) values
    (@locid, @LocationName)
    select @locid as LocationId
  </CommandText>
  <Parameter Name="LocationName" Type="varchar" Mode="In" />
  <Parameter Name="Address" Type="varchar" Mode="In" />
</Function>
<Function Name="UpdateEvent" IsComposable="false" Schema="dbo">
  <CommandText>
    update Locations set Address = @Address where locationid =
    @LocationId
    update eventlocation set locationname = @LocationName where
    LocationId = @LocationId
  </CommandText>
  <Parameter Name="LocationName" Type="varchar" Mode="In" />
  <Parameter Name="Address" Type="varchar" Mode="In" />
  <Parameter Name="LocationId" Type="int" Mode="In" />
</Function>
<Function Name="DeleteEvent" IsComposable="false" Schema="dbo">
  <CommandText>
    delete Locations where LocationId = @LocationId
    delete eventlocation where locationid = @LocationId
  </CommandText>

```

```

</CommandText>
<Parameter Name="LocationId" Type="int" Mode="In" />
</Function>
<!-- club crud -->
<Function Name="InsertClub" IsComposable="false" Schema="dbo">
<CommandText>
    declare @locid int
    insert into Locations(Address) values (@Address)
    set @locid = SCOPE_IDENTITY()
    insert into GunClubs(ClubId,ClubName) values (@locid,@ClubName)
    select @locid as LocationId
</CommandText>
<Parameter Name="ClubName" Type="varchar" Mode="In" />
<Parameter Name="Address" Type="varchar" Mode="In" />
</Function>
<Function Name="UpdateClub" IsComposable="false" Schema="dbo">
<CommandText>
    update Locations set Address = @Address where locationid =
@LocationId
    update GunClubs set ClubName = @ClubName where LocationId
=@LocationId
</CommandText>
<Parameter Name="ClubName" Type="varchar" Mode="In" />
<Parameter Name="Address" Type="varchar" Mode="In" />
<Parameter Name="LocationId" Type="int" Mode="In" />
</Function>
<Function Name="DeleteClub" IsComposable="false" Schema="dbo">
<CommandText>
    delete GunClubs where ClubId = @LocationId
</CommandText>
<Parameter Name="LocationId" Type="int" Mode="In" />
</Function>
<!-- Shooting Range -->
<Function Name="InsertRange" IsComposable="false" Schema="dbo">
<CommandText>
    declare @locid int
    insert into Locations(Address) values (@Address)
    set @locid = SCOPE_IDENTITY()
    insert into ShootingRange(RangeId,RangeName) values
(@locid,@RangeName)
    select @locid as LocationId
</CommandText>
<Parameter Name="RangeName" Type="varchar" Mode="In" />
<Parameter Name="Address" Type="varchar" Mode="In" />
</Function>
<Function Name="UpdateRange" IsComposable="false" Schema="dbo">
<CommandText>
    update Locations set Address = @Address where locationid =
@LocationId
    update ShootingRange set RangeName = @RangeName where RangeID
=@LocationId
</CommandText>
<Parameter Name="RangeName" Type="varchar" Mode="In" />
<Parameter Name="Address" Type="varchar" Mode="In" />
<Parameter Name="LocationId" Type="int" Mode="In" />
</Function>
<Function Name="DeleteRange" IsComposable="false" Schema="dbo">

```

```

<CommandText>
    delete ShootingRange where RangeId = @LocationId
</CommandText>
<Parameter Name="LocationId" Type="int" Mode="In" />
</Function>

```

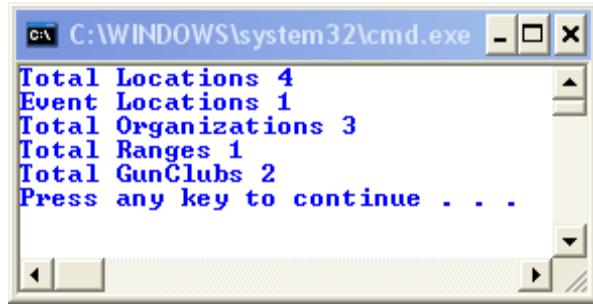
On the above functions, we first insert an entry into the location table and then Scope_identity we grab the inserted locationId and insert an another entry into the appropriate table like EventLocation, GunClub or ShootingRange. To test the model we can create instances of each type of entity, save them to the database. Using the second datacontext, we can query for each type of entity and confirm that the count returned matches the entity types inserted earlier. On the code below, I am creating an instance of EventLocation, two GunClubs and a ShootingRange and saving all the locations to the database. Using the second database, I am getting the count of all the locations, locations which are organization, ShootingRanges and GunClubs.

```

var db = new InhQVLocation();
    var evlocation = new EventLocation { LocationName = "Abba Shrine
Center", Address = "7701 Hitt Road" };
        var club1 = new GunClub { ClubName = "Cambria Hunting Club",
Address = "240 Woodstream Dr" };
            var club2 = new GunClub { ClubName = "Head Hunter Gun Club",
Address = "680 Porterville Rd" };
                var range = new ShootingRange { RangeName = "Cooper Range",
Address = "Bean Creek Road" };
                    db.AddToLocations(evlocation);
                    db.AddToLocations(club1);
                    db.AddToLocations(club2);
                    db.AddToLocations(range);
                    db.SaveChanges();
                    var db2 = new InhQVLocation();
                    Console.WriteLine("Total Locations {0}", db2.Locations.Count());
                    Console.WriteLine("Event Locations {0}",
db2.Locations.OfType<EventLocation>().Count());
                    Console.WriteLine("Total Organizations {0}",
db2.Locations.OfType<Organization>().Count());
                    Console.WriteLine("Total Ranges {0}",
db2.Locations.OfType<ShootingRange>().Count());
                    Console.WriteLine("Total GunClubs {0}",
db2.Locations.OfType<GunClub>().Count());

```

Figure below shows the screenshot of the Count for different types of location returned from the above query.



5.1.6 Optimizing QueryView for Inheritance

Problem: On the example 3.9.5, we looked at using QueryView to add another layer of inheritance hierarchy. One of the problems with the above approach is when you query for certain type of entity, entity framework, had to use execute the entire QueryView which joined against all related tables such as GunClubs, ShootingRanges and EventLocation and apply the type filter in memory. This could be expensive if records returned from all 3 tables are in large number. You want to improve the QueryView performance so that entity filter is applied on the database and only records requested by the client are returned from the database.

Solution: To understand the solution, we have to look at QueryView once again.

```
<EntitySetName="Locations">
    <QueryView>
        select value
        case
            when (el.LocationId is not null) then
                InhQVLocationModel.EventLocation(l.LocationId,l.Address,el.LocationName)
            when (gc.ClubId is not null) then
                InhQVLocationModel.GunClub(l.LocationId,l.Address,gc.ClubName)
            when (range.RangeId is not null) then
                InhQVLocationModel.ShootingRange(l.LocationId,l.Address,range.RangeName)
```

```

        END
    from InhQVLocationModelStoreContainer.Locations as
l
    left join
InhQVLocationModelStoreContainer.EventLocation as el on
l.LocationId = el.LocationId
    left join
InhQVLocationModelStoreContainer.GunClubs as gc on l.LocationId
= gc.ClubId
    left join
InhQVLocationModelStoreContainer.ShootingRange as range on
l.LocationId = range.RangeId
</QueryView>
</EntitySetMapping>

```

The queryview above is applied on an EntitySet Locations which means that if you query for any entity that is part of an entityset, EF has to execute the entire QueryView on the database, bring the results and apply the filter in memory. One of the ways to improve QueryView is to create separate QueryView for each type of entity defined for Locations entityset. If your query uses OfType operator, entity framework tries to find a queryView that matches the type specified on the OfType operator. If a match is found, the closest QueryView is used otherwise it defaults to using the QueryView with no TypeName. This means that you are required to provide a QueryView that applies to entire EntitySet and can optionally provide QueryView for each EntityType with in an EntitySet for optimization. On the code below, I am returning EventLocations by using OfType operator to filter the locations. I have captured the sql execute by the query below.

```

var db = new InhQVLocation();
string sql;
sql =
db.Locations.OfType<EventLocation>().ToTraceString();
Console.WriteLine(sql);

SELECT
.....
FROM [dbo].[Locations] AS [Extent1]
LEFT OUTER JOIN [dbo].[EventLocation] AS [Extent2] ON [Extent1].[LocationId]
= [Extent2].[LocationId]
LEFT OUTER JOIN [dbo].[GunClubs] AS [Extent3] ON [Extent1].[LocationId] =
[Extent3].[ClubId]
LEFT OUTER JOIN [dbo].[ShootingRange] AS [Extent4] ON [Extent1].[LocationId]
= [Extent4].[RangeId]

```

Notice that query applies a join to all 3 tables despite that we made an explicit intent to only return EventLocations and the query should have joined against EventLocations and Locations table only. We can improve the query by introducing all the different queryviews for each type of entity on the Location entityset. Code below shows rest of the QueryViews for Location entityset.

```

<QueryView
TypeName="IsTypeOf( InhQVLocationModel.EventLocation )">
    select value

InhQVLocationModel.EventLocation(l.LocationId,l.Address,el.LocationName)
    from
InhQVLocationModelStoreContainer.Locations as l
    left join
InhQVLocationModelStoreContainer.EventLocation as el on
l.LocationId = el.LocationId
    </QueryView>
    <QueryView
TypeName="IsTypeOf( InhQVLocationModel.GunClub )">
        select value

InhQVLocationModel.GunClub(l.LocationId,l.Address,gc.ClubName)
    from
InhQVLocationModelStoreContainer.Locations as l
    left join
InhQVLocationModelStoreContainer.GunClubs as gc on l.LocationId
= gc.ClubId
    </QueryView>
    <QueryView
TypeName="IsTypeOf( InhQVLocationModel.ShootingRange )">
        select value

InhQVLocationModel.ShootingRange(l.LocationId,l.Address,range.RangeName)
    from
InhQVLocationModelStoreContainer.Locations as l
    left join
InhQVLocationModelStoreContainer.ShootingRange as range on
l.LocationId = range.RangeId
    </QueryView>
    <QueryView
TypeName="IsTypeOf( InhQVLocationModel.Organization )">
```

```

        select value
        case
        when (gc.ClubId  is not null) then

InhQVLocationModel.GunClub(l.LocationId,l.Address,gc.ClubName)
        when (range.RangeId is not null) then

InhQVLocationModel.ShootingRange(l.LocationId,l.Address,range.Ra
ngeName)
        END
        from
InhQVLocationModelStoreContainer.Locations as l
        left join
InhQVLocationModelStoreContainer.GunClubs as gc on l.LocationId
= gc.ClubId
        left join
InhQVLocationModelStoreContainer.ShootingRange as range on
l.LocationId = range.RangeId
        </QueryView>

```

Notice that for EventLocation entity, I am doing an explicit join against EventLocation table and Location only. Similarly GunClub entity joins against GunClub table and ShootingRange table joins against ShootingRange table. When I execute my same linq query above to get Locations of type eventlocation, the query send to the database is very precise as shown below.

```

SELECT
..
FROM  [dbo].[Locations] AS [Extent1]
LEFT OUTER JOIN [dbo].[EventLocation]

```

Another important point to remember is when you are specifying the TypeName, you must use IsTypeOf operator for EF framework to select the correct queryview for the entity you are searching on. Failing to do so would cause the query to use the default QueryView with no TypeName.

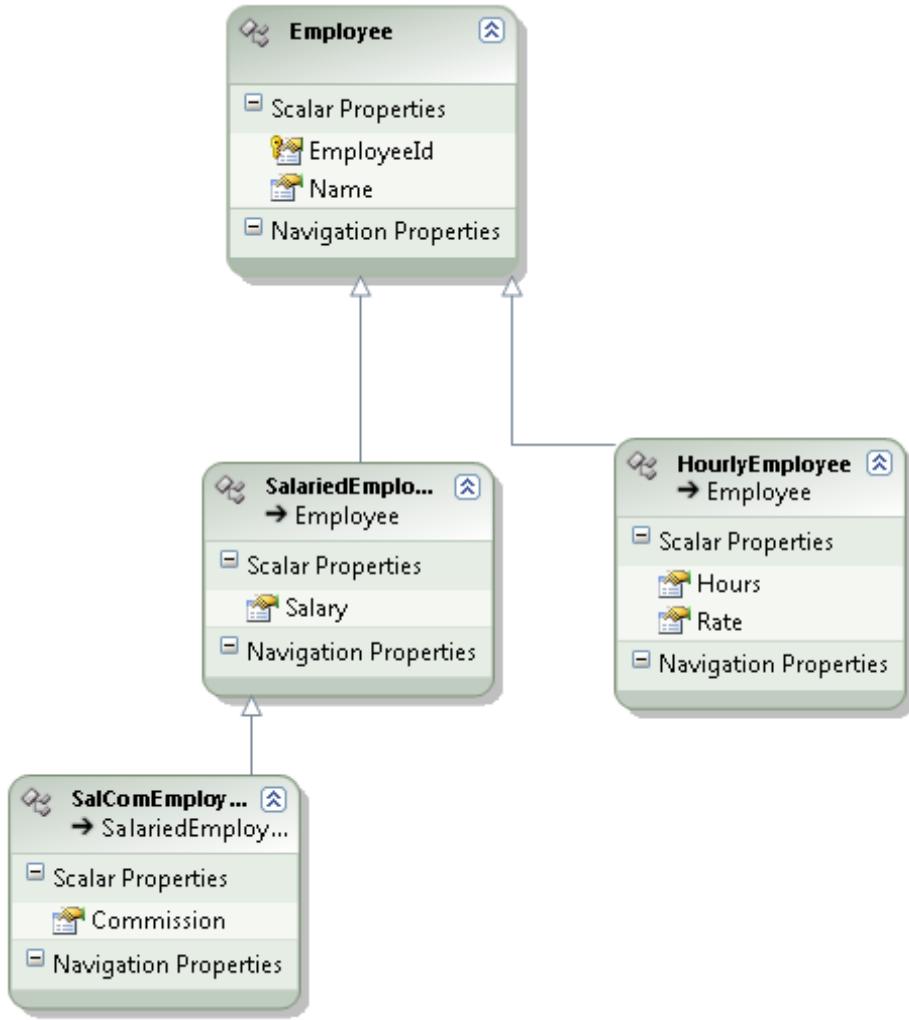
5.1.7 Overriding Conditions for nested inheritance

Problem: The figure below shows Employee table structure in our database.

Employees (tbhc)			
	Column Name	Data Type	Allow Nulls
EmployeeId	int	<input type="checkbox"/>	
Name	varchar(50)	<input type="checkbox"/>	
Hours	int	<input checked="" type="checkbox"/>	
Rate	decimal(5, 2)	<input checked="" type="checkbox"/>	
Salary	decimal(7, 2)	<input checked="" type="checkbox"/>	
GetsComm	bit	<input checked="" type="checkbox"/>	
Type	char(2)	<input type="checkbox"/>	
Commission	int	<input checked="" type="checkbox"/>	
			<input type="checkbox"/>

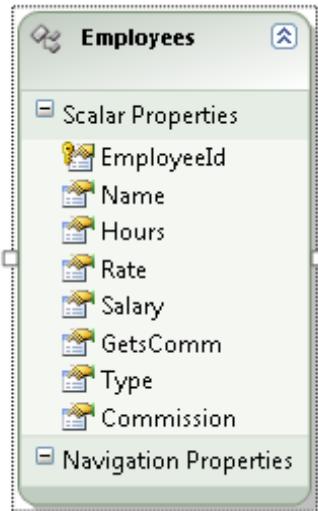
The table contains 3 types of Employees. An employee is considered hourly if the Type column has a value of H. If the value of Type is S and GetsComm is False, it is a SalariedEmployee. However if the Type has a value of S and GetComm is set to true, the Employee is considered SalariedWithComm Employee. You want to model this relationship in entity data model using 3 different employees extending from the base Employee class.

Solution: Import the model using the import wizard. Create 3 new entities, Hourly Employee, Salaried Employee and SalPlusCommEmployee. Set the mapping condition for HourlyEmployee where Type equal to H, SalariedEmployee where Type equal to S and GetsComm to 0, SalPlusCommEmployee where Type equal to S and GetComm to a value of 1 which means true. Make sure Employee entity is marked as abstract and move the fields that belong to a derived entity to its own class. Ensure Hourly and salaried Employee inherit from Employee entity and SalCommEmployee inherits from SalariedEmployee. The completed entity data model is shown below.

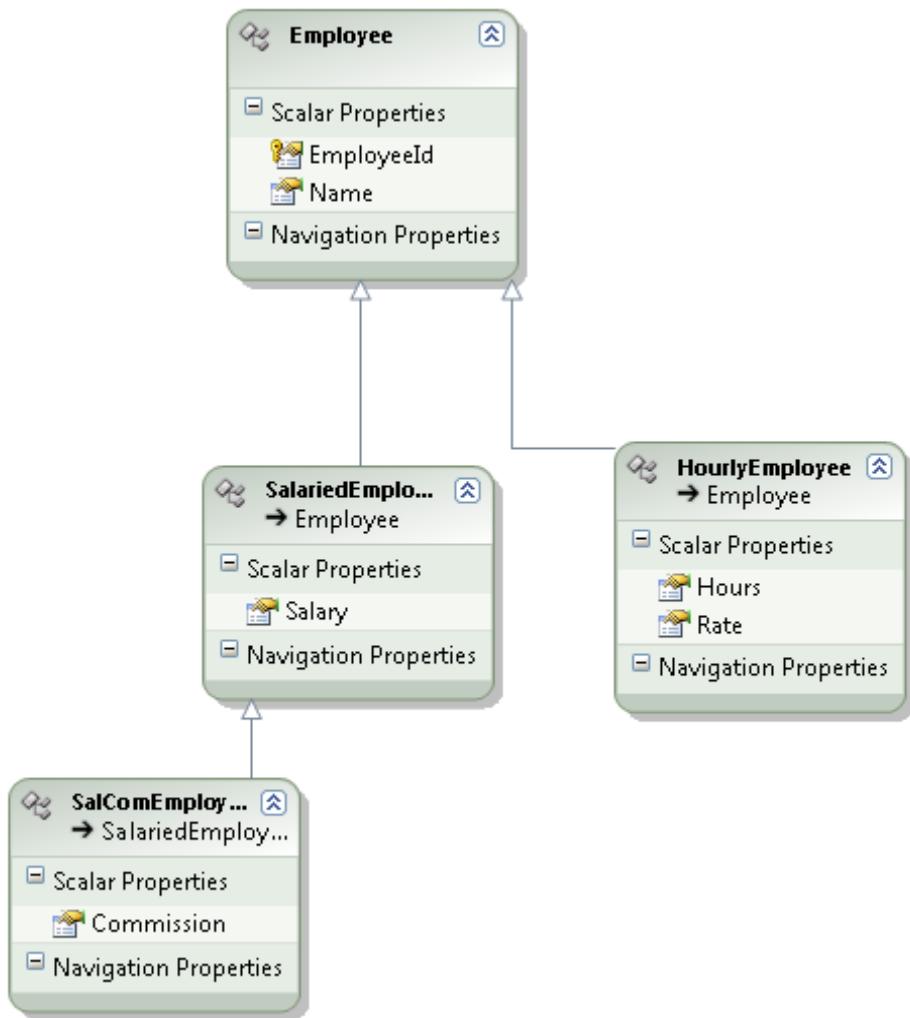


Discussion: In this example, we will cover how to override conditions to create nested inheritance hierarchy. Basically derived entity inherits all the conditions specified by the base entity. However derived entity has the option to negate or change the value for one or more of the conditions dictated by base entity. If we look at the Employee table structure given above, we will notice that GetComm is set to allow null. This is because Hourly Employee type is determined solely on the Type column where as for Salaried and SalariedPlusCommission Employee both have a Type set to S and GetsComm additional column determines if the Salaried Employee makes a commission or not. Steps below outline the procedure to move the existing Employee table to 3 derived entity types.

1. Import Employee table using the import wizard. Figure below shows the table after the wizard completes.



2. Create 3 entities Hourly Employee, Salaried Employee and SalPlusComm Employee. Make Hourly and SalariedEmployee derive from Employee entity and make SalPlusCommEmployee derive from SalariedEmployee. The reason SalPlusCommEmployee derives from SalariedEmployee is because we want to access all the employees who are paid Salary as well get access to Employee who makes Salary and Commission. Move Rate and Hours property from Employee entity to HourlyEmployee derived type. Move the Salary property to SalariedEmployee and Commissin property to SalCommEmployee. Remove GetsComm and Type property from Employee entity because we will use these columns to map to correct derived types. Entity framework does not permit mapping column to property value if the column is used as a discriminator column and becomes a determining factor on what derived entity needs to be instantiated. Only case when entity framework would permit mapping discriminator column to a property is when the column is marked as not null in the database and the values on the conceptual side comes from restricted set of values. Also make sure that Employee entity is marked as abstract class. Figure below shows the updated model after completing the steps mentioned above.



3. Configure the mapping for HourlyEmployee by selecting Employee table and the designer would auto map Hours and Rate to matching columns on the Employee table. Set the condition for HourlyEmployee to be where Type column is equal to H. Figure below shows the completed mapping for HourlyEmployee.

Column	Operator	Value / Property
Tables		
Maps to Employees		
When Type	=	H
<Add a Condition>		
Column Mappings		
Hours : int	↔	Hours : Int32
Rate : decimal	↔	Rate : Decimal
Salary : decimal	↔	
GetsComm : bit	↔	
Type : char	↔	
Commission : int	↔	

4. Configure mapping for Salaried Employee by selecting Employee table on the mapping window and designer would auto map the Salary property to the Salary column on Employee table. Set the condition for Salaried employee to be where Type column is equal to S and GetComm to 0 means false. Figure below shows the completed mapping for SalariedEmployee.

Column	Operator	Value / Property
Tables		
Maps to Employees		
When Type	=	S
And GetsComm	=	0
<Add a Condition>		
Column Mappings		
Hours : int	↔	
Rate : decimal	↔	
Salary : decimal	↔	Salary : Decimal
GetsComm : bit	↔	
Type : char	↔	
Commission : int	↔	
<Add a Table or View>		

5. Configure mapping for SalCommEmployee by selecting Employee table on the mapping window and map the commission property to commission column on Employee table. Set two conditions for the derived entity. First condition is Type equal to S and second condition

is GetsComm to a value of 1 which means the Employee makes a commission in addition to getting salary.

Column	Operator	Value / Property
Maps to Employees		
When Type	=	S
And GetsComm	=	1
<Add a Condition>		
Column Mappings		
Hours : int	↔	
Rate : decimal	↔	
Salary : decimal	↔	
GetsComm : bit	↔	
Type : char	↔	
Commission : int	↔	Commission : Int32

Notice that in step 4, we set the condition for GetsComm to false because SalariedEmployee do not make Commission but on the derived entity we overwrote that condition and said SalPlusComm employee do make a commission by setting GetsComm to 1 or True.

To test our model, we create instances of each type of entity and save them to the database. Then using second database we can query for each type of entity confirm that output returned for the entity matches what we expected. Code below shows an example of using the above model we have created.

```
var db = new MediaTPPTBH();
    var hourly = new HourlyEmployee { Name = "Zee",
Hours = 40, Rate = 60 };
    var salary = new SalariedEmployee { Name = "Garg",
Salary = 95000 };
    var salpluscomm = new SalComEmployee { Name =
"Travis", Commission = 2000, Salary = 95000 };
    db.AddToEmployees(hourly);
    db.AddToEmployees(salary);
    db.AddToEmployees(salpluscomm);
    db.SaveChanges();
var db2 = new MediaTPPTBH();
```

```

        Console.WriteLine("Total Employees " +
db2.Employees.Count());

        var hourly1 =
db.Employees.OfType<HourlyEmployee>().First();
        Console.WriteLine("Hourly Employee " +
hourly1.Name);

        Console.WriteLine("Total Salaried Employee " +
db.Employees.OfType<SalariedEmployee>().Count());
        var salary1 =
db.Employees.OfType<SalariedEmployee>().First();
        Console.WriteLine("Salaried Employee " +
salary1.Name);

        var salpluscom1 =
db.Employees.OfType<SalComEmployee>().First();
        Console.WriteLine("Sal plus Comm Employee " +
+ salpluscom1.Name);

```

On the code above, I am doing a count of employee in the database which should return 3 because we created 3 employees. Next I am printing the Name of hourly employee which should be Zee. I am then querying for all salaried employees in the database which returns a count of 2. This is correct because both SalariedEmployee and SalariedAndCommEmployee are salaried employees. On the last query I am outputting each type of Salaried employee on the console window. Screen for console window below confirms our output.

```

C:\Windows\system32\cmd.exe
Total Employees 3
Hourly Employee Zee
Total Salaried Employee 2
Salaried Employee Garg
Sal plus Comm Employee Travis
Press any key to continue . .

```

5.1.8 Applying Conditions on Base Entity

Problem: Orders table in the database contains deleted orders. The deleted Orders are marked by IsDeleted column set to true. You want to import the table structure as Orders entity that contains all orders in the database. In addition, you want an addition entity that derives from Orders entity that contains only deleted orders in the database.

Solution: The above problem requires placing a condition on the base entity with IsDeleted set to false. Create another entity and call it DeletedOrders deriving from Orders entity. Set the condition for deleted Orders where IsDeleted is true.

Discussion: Figure below shows the Orders table structure in the database.

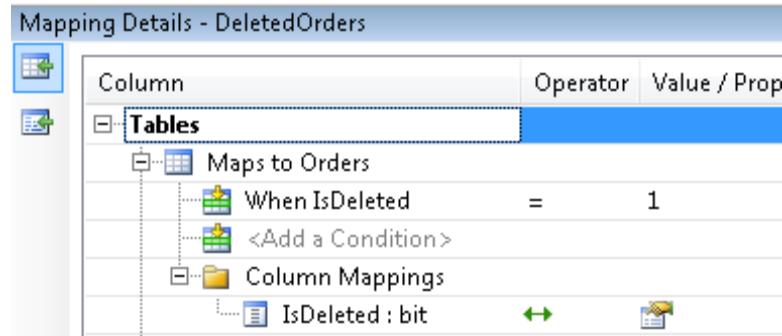
Column Name	Data Type	Allow Nulls
OrderID	int	<input type="checkbox"/>
CustomerID	nchar(5)	<input type="checkbox"/>
OrderDate	datetime	<input checked="" type="checkbox"/>
IsDeleted	bit	<input checked="" type="checkbox"/>
		<input type="checkbox"/>

The above table contains IsDeleted column that determines if the Order is deleted or not. To map the table to entity model as desired, import the table using the import wizard. Remove IsDeleted column as we will use it as discriminator column. Apply condition on the Order entity to IsDeleted = 0 means false. Figure below shows the mapping for Orders entity.

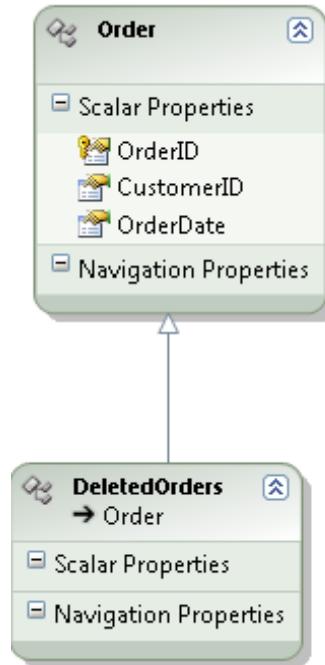
Mapping Details - Order

Column	Operator	Value / Property
Tables		
Maps to Orders		
When IsDeleted	=	0
<Add a Condition>		
Column Mappings		
OrderID : int	↔	OrderID : Int32
CustomerID : nchar	↔	CustomerID : String
OrderDate : datetime	↔	OrderDate : DateTime
IsDeleted : bit	↔	

Create a second entity Deleted Orders and derive it from Orders entity. Set the condition for Deleted Orders entity to IsDeleted = 1 means true. Figure below shows the mapping for DeletedOrders entity.



The updated entity model is shown below.

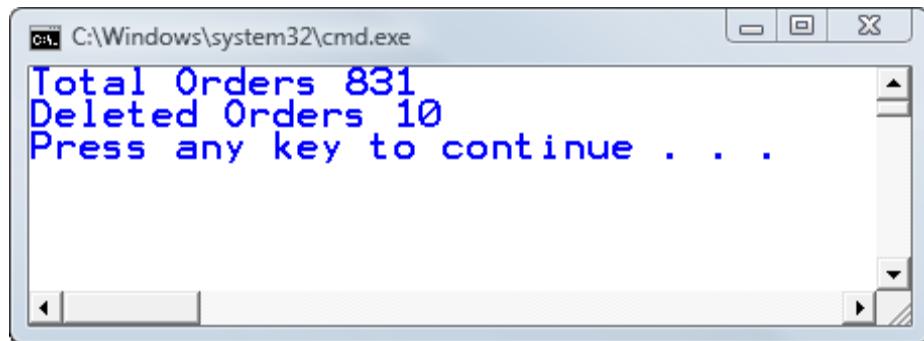


When I was working on this example, I started with putting no conditions on Orders entity and only applied IsDeleted condition to DeletedOrders entity and couldn't get the model to validate. And if you came with the same thought process that since orders entity is supposed to return all orders and shouldn't have any conditions then you won't get the model to validate. The reason is having no condition on Orders

entity makes it ambiguous for EF to determine that when an order is marked as IsDeleted, should it consider as a regular order or should it consider a deleted order. Because having no conditions on Orders entity means apply no rule to all the Orders. In short it is important to not overlap conditions. For example if you try make the top level condition as column with not null and derived to be column with some value condition. Then this case is certain ambiguous because EF does not know what to do if both conditions are met. The condition needs to be exclusive. To test our model we can query orders entity and get the count to make sure that count returned for orders matches the total orders in the database. We can also write another query which only returns deleted orders from the orders table which we can compare against the count for deleted orders in the orders table. Code below shows an example of that.

```
var db = new MediaTPTTBH();
    Console.WriteLine("Total Orders " +
db.Orders.Count());
    Console.WriteLine("Deleted Orders " +
db.Orders.OfType<DeletedOrders>().Count());
```

Screen shot below shows the output from the above code.



```
C:\Windows\system32\cmd.exe
Total Orders 831
Deleted Orders 10
Press any key to continue . . .
```

5.1.9 Using Abstract entity with no table Mapping in TPH

Problem: You have Person table in the database that contains Customer, Student and Instructors. Each type of person is identified by a Type column.

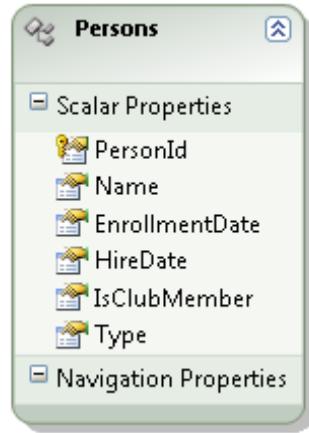
You want to import the table structure as Table Per hierarchy with Customer entity inheriting from Person entity. Instead of Student and Instructor directly inheriting from Person, you want to add another layer of inheritance hierarchy, Staff from which Student and Instructor derive from. Having another layer of inheritance would allow you to program against both Student and Instructor by using Staff entity. Figure below shows the table structure for Persons table.

Persons			
	Column Name	Data Type	Allow Nulls
PK	PersonId	int	<input type="checkbox"/>
	Name	varchar(50)	<input type="checkbox"/>
	EnrollmentDate	date	<input checked="" type="checkbox"/>
	HireDate	date	<input checked="" type="checkbox"/>
	IsClubMember	bit	<input checked="" type="checkbox"/>
	Type	varchar(20)	<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>

Solution: Import Persons table using import wizard, create four entities Customer, Student, Instructor and Staff entity. Make Customer entity inherit from Person entity. Make Student and Instructor entity derive from Staff entity. Map Customer, student and Instructor entity using table mappings. Since Staff entity is not defined on the database, make staff entity abstract. This would ensure that we do not get any validation errors for not mapping Staff entity.

Discussion: When you add additional layer of inheritance that is not defined on the database and therefore cannot be mapped, the entity must be marked as abstract. In the example below, we will go through the steps of adding Staff entity not defined on the database.

1. Import Persons table using Edm import wizard. Figure below shows Persons entity on entity designer.



2. Remove Type property as well will use the Type column to map inheritance structure. Add Customer entity deriving it from Person entity and move IsClubMember property from Person table to Customer entity. Map Customer entity to Person table where Type is Customer as the condition. Figure below shows the mapping for Customer entity.

The screenshot shows the 'Mapping Details - Customer' dialog. Under the 'Tables' section, there is a 'Maps to Persons' entry. Within this entry, there is a 'When Type' condition set to 'Customer'. Below this, there are 'Column Mappings' for two columns: 'IsClubMember : bit' and 'Type : varchar'. The 'IsClubMember' mapping is shown with a double-headed arrow and a checkmark, while the 'Type' mapping is also shown with a double-headed arrow but without a checkmark.

3. Create Staff entity and derived it from Person entity. Since Staff and Person entity do not map to any table in the database, we will make Staff and Person entity as abstract. Next create Student entity and derive it from Staff entity. Move EnrollmentDate from Person entity to Student entity and map Student entity to Person table in the database where Type is Student for the condition. Figure below shows the mapping for Student entity.

Mapping Details - Student

Column	Operator	Value / Property
Tables		
Maps to Persons		
When Type	=	Student
<Add a Condition>		
Column Mappings		
EnrollmentDate : date	↔	EnrollmentDate : DateTime
IsClubMember : bit	↔	
Type : varchar	↔	
<Add a Table or View>		

4. Create Instructor entity and derive it Staff entity. Move HireDate property from Person entity to Instructor entity and map the entity to Person table on the mapping window. Set the condition for Instructor entity to be Type equal to Instructor.

Mapping Details - Instructor

Column	Operator	Value / Property
Tables		
Maps to Persons		
When Type	=	Instructor
<Add a Condition>		
Column Mappings		
EnrollmentDate : date	↔	
HireDate : date	↔	HireDate : DateTime
IsClubMember : bit	↔	
Type : varchar	↔	

If you try to validate the model after completing the steps mentioned above, you will get following validation errors.

Error 1 Error 3024: Problem in Mapping Fragment starting at line 272: Must specify mapping for all key properties (Persons.PersonId) of the EntitySet Persons.

There is nothing wrong with the model we have created. There appears to be a designer bug where it fails to map PersonId Property for Student and Instructor entity to Person table. Msl below shows the mapping for Student and Instructor entity.

```
<EntityTypeMapping TypeName=" IsTypeOf(MediaQWModel.Student)">
    <MappingFragment StoreEntitySet="Persons">
        <ScalarProperty Name="EnrollmentDate"
ColumnName="EnrollmentDate" />
```

```

        <Condition ColumnName="Type" Value="Student"
/></MappingFragment>
        </EntityTypeMapping>
    <EntityTypeMapping
TypeName="IsTypeOf(MediaQWModel.Instructor)">
        <MappingFragment StoreEntitySet="Persons">
            <ScalarProperty Name="HireDate"
ColumnName="HireDate" />
            <Condition ColumnName="Type" Value="Instructor"
/></MappingFragment>
        </EntityTypeMapping>

```

The above mapping is missing PersonId mapping available on Person entity. When we added another inheritance structure that did not map to any tables, the designer missed mapping PersonId property to PersonId column on Person table. Msl below shows the updated mapping for Student and Instructor entity that validates cleanly.

```

<EntityTypeMapping TypeName="IsTypeOf(MediaQWModel.Student)">
    <MappingFragment StoreEntitySet="Persons">
        <ScalarProperty Name="PersonId"
ColumnName="PersonId" />
        <ScalarProperty Name="EnrollmentDate"
ColumnName="EnrollmentDate" />
        <Condition ColumnName="Type" Value="Student"
/></MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping
TypeName="IsTypeOf(MediaQWModel.Instructor)">
        <MappingFragment StoreEntitySet="Persons">
            <ScalarProperty Name="PersonId"
ColumnName="PersonId" />
            <ScalarProperty Name="HireDate"
ColumnName="HireDate" />
            <Condition ColumnName="Type" Value="Instructor"
/></MappingFragment>
        </EntityTypeMapping>

```

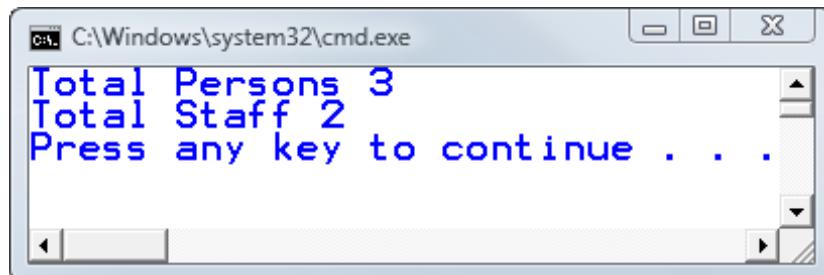
To test the above model created, we can create instance of each entity type, save them to the database and using second data context retrieve entities from the database and confirm the results match. On the code below, I am creating instance of Customer, Instructor and Student entity and saving it to the database. Using the second datacontext, I

retrieve the count of Persons in the database and count of Persons of type Staff.

```
var db = new EcommerceEntities();
        var cust = new Customer { Name = "Zee", IsClubMember
= true };
        var instructor = new Instructor { HireDate =
DateTime.Now, Name = "Alex" };
        var student = new Student { Name = "John",
EnrollmentDate = DateTime.Now };
        db.AddToPersons(cust);
        db.AddToPersons(instructor);
        db.AddToPersons(student);
        db.SaveChanges();

        var db2 = new EcommerceEntities();
        Console.WriteLine("Total Persons " +
db2.Persons.Count());
        Console.WriteLine("Total Staff " +
db2.Persons.OfType<Staff>().Count());
```

Figure below shows the results of count printed on the console window.



5.1.10 Applying IsNull condition to Table per Hierarchy

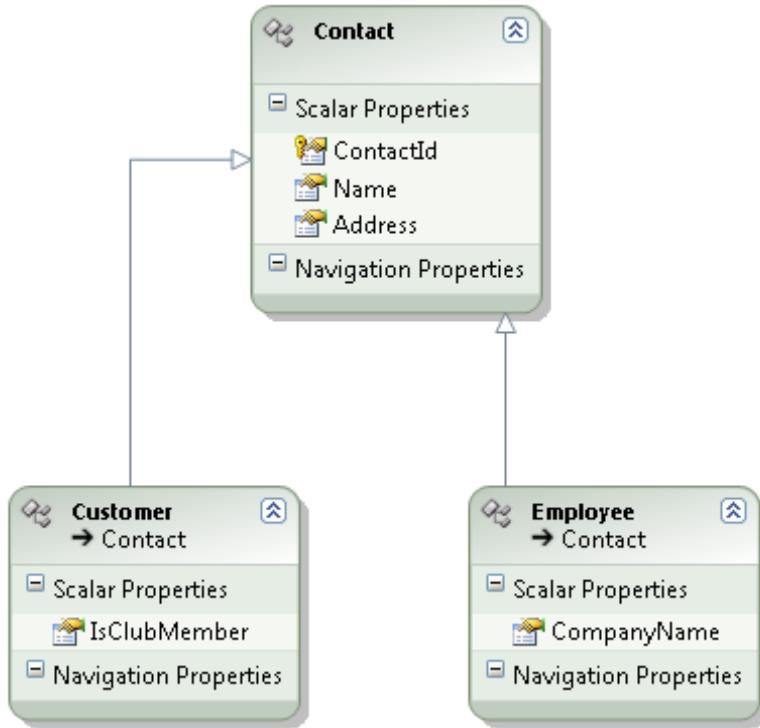
Problem: You have Contact table in the database that contains both Customer and Employee. Each type of Contact is identified by a Type column. The Type column has allow null set to true. If there is no value for Type column, it means it is a regular Contact. Otherwise it could be either Customer or an Employee depending on if the Type has a value of Customer or Employee. You want to model this relationship as base class Contact that contains the regular contacts with no type value and Customer and Employee Contact

extending Contact with their mapped to contacts with appropriate Type value. Your current database structure is shown below.

Contacts (tbhc)			
	Column Name	Data Type	Allow Nulls
1	ContactId	int	<input type="checkbox"/>
	Name	varchar(100)	<input type="checkbox"/>
	Type	varchar(50)	<input checked="" type="checkbox"/>
	CompanyName	varchar(50)	<input checked="" type="checkbox"/>
	IsClubMember	bit	<input checked="" type="checkbox"/>
	Address	varchar(100)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

Solution: Import Contact table into entity data model. Create two entities Customer and Employee deriving from Contact entity. Move fields' specific to derived entities to its own class. Apply condition on Contact entity to where Type is null. Apply condition of Type equal to Customer on Customer entity and Type equal to Employee for Employee entity.

Discussion: In this example, we will learn how to use Null condition for entity mapping. The Type column on the database is marked as null so that regular contacts will have no value for Type whereas Customer and Employee entity will have specific value for Type column on the Contact Table. Steps below illustrate how to map the above database table to model shown below.



1. Import Contacts table using the wizard. Create two entities Customer and Employee. Make Customer and Employee entity inherit from Contact entity. Move IsClubMember property to Customer entity and CompanyName to Employee entity. Set condition on Contact where Type is null. We are setting the type is null because if Type column does not have any value in the database, it is considered a regular Contact. Also remove Type property from Contact entity since we will use Type column to map inheritance. Figure below shows the mapping for Contact entity.

Column	Operator	Value / Property
Tables		
Maps to Contacts		
When Type	Is	Null
<Add a Condition>		
Column Mappings		
ContactId : int	↔	ContactId : Int32
Name : varchar	↔	Name : String
Type : varchar	↔	
CompanyName : varchar	↔	
IsClubMember : bit	↔	
Address : varchar	↔	Address : String

2. For Customer entity set Type condition to Customer and maps IsClubMember property to IsClubMember column on Contact table. Figure below shows the mapping for Customer entity.

Column	Operator	Value / Property
Tables		
Maps to Contacts		
When Type	=	Customer
<Add a Condition>		
Column Mappings		
Type : varchar	↔	
CompanyName : varchar	↔	
IsClubMember : bit	↔	IsClubMember : Boolean

3. For Employee entity set Type condition equal to Employee and maps CompanyName property to CompanyName column on Contact table. Figure below shows the mapping for Employee entity.

Mapping Details - Employee

Column	Operator	Value / Property
Tables		
Maps to Contacts		
When Type	=	Employee
<Add a Condition>		
Column Mappings		
Type : varchar	↔	CompanyName : String
CompanyName : varchar	↔	CompanyName : String
IsClubMember : bit	↔	

Code below shows how to use the current model created.

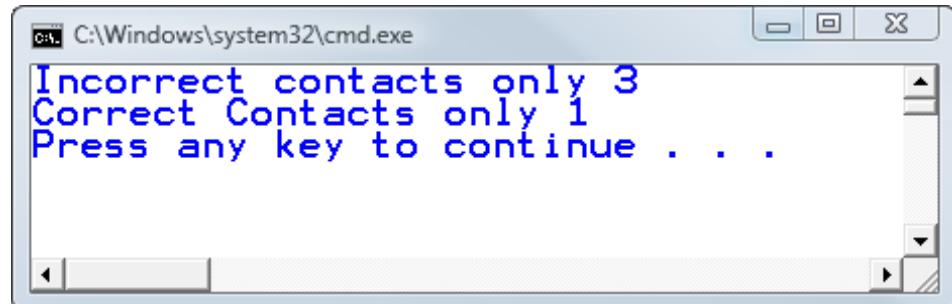
```

var contact = new Contact { Name = "Zeeshan", Address =
    "123" };
    var customer = new Customer { Name = "Alex", Address
= "123", IsClubMember = true };
        var employee = new Employee { Name = "John", Address
= "123", CompanyName = "True LTD" };
            var db = new IsNullInhEntities();
            db.AddToContacts(contact);
            db.AddToContacts(customer);
            db.AddToContacts(employee);
            db.SaveChanges();
            var db2 = new IsNullInhEntities();
            Console.WriteLine("Incorrect contacts only " +
db2.Contacts.OfType<Contact>().Count());
                var esql = @"select value c from
OfType(Contacts,only NullInheritance.Contact) as c";
                    Console.WriteLine("Correct Contacts only " +
db2.CreateQuery<Contact>(esql).Count());

```

On the code above I am creating instance of Customer, Employee and regular contact and saving it to database. Then using the second datacontext, I am trying to retrieve contacts which are of type contacts only from all contacts in the database. The count that I get is 3 which is incorrect answer but yet correct. Correct because all contacts either be regular contact, employee or Customer are Contacts. What if you only wanted the base type contact? In those cases you have to use esql operator and only demand Contact and not its derived type. Notice in

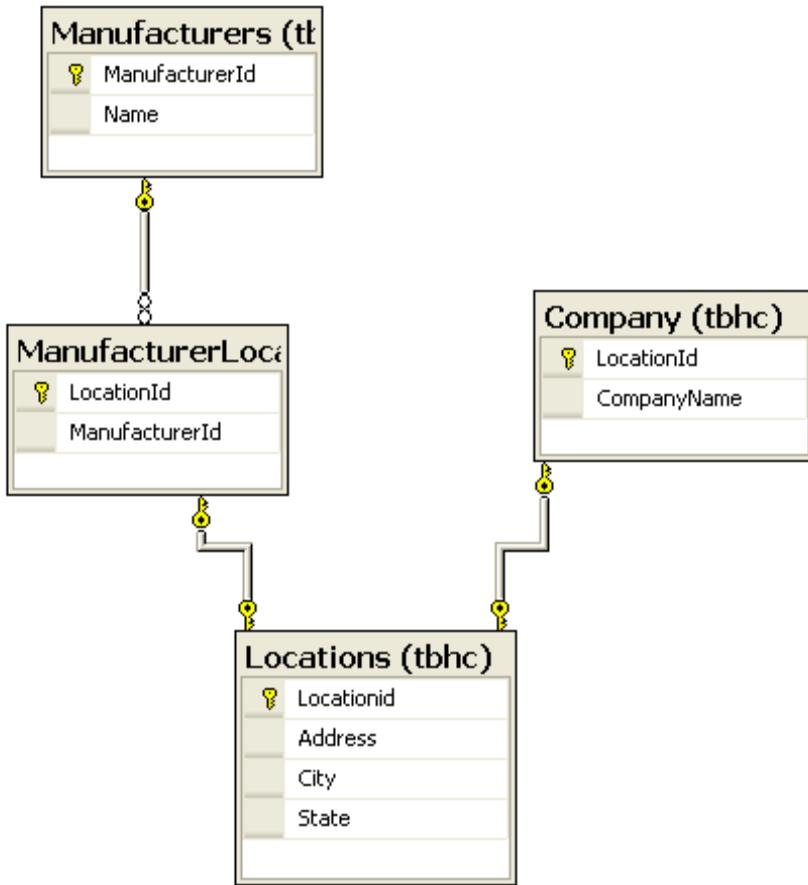
our esql query above, I am using OfType operator and explicitly stating that I want contact only by using Only keyword before contact. Figure below shows the output of the above code.



```
Incorrect contacts only 3
Correct Contacts only 1
Press any key to continue . . .
```

5.1.11 Creating Many To 1 Association on Derived Entity

Problem: Figure below shows the current database structure for Manufacturers and Companies table.



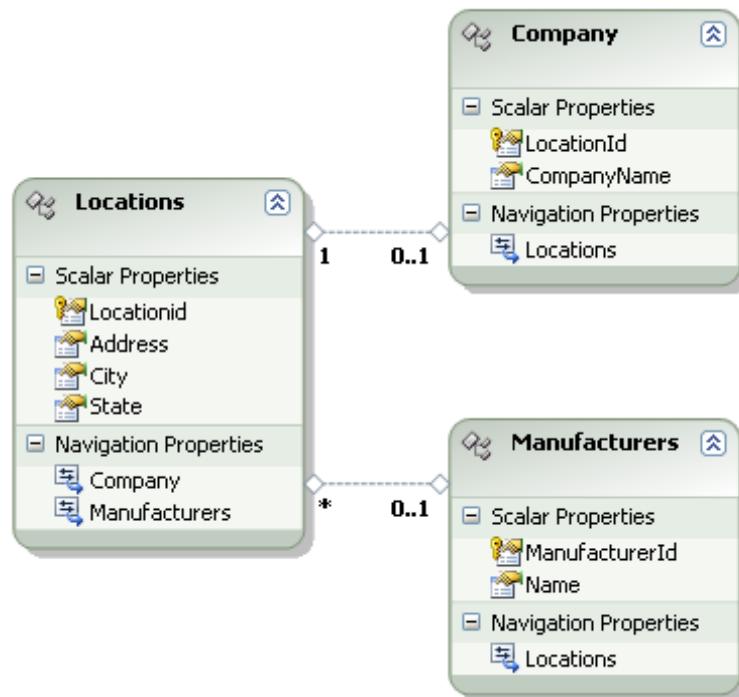
On the figure above, we have Location entity which has fields common to both Company and ManufacturerLocation. Company has additional field `CompanyName` whereas ManufacturerLocation has a relationship to Manufacturer and relationship states that a manufacturer is located in many locations. ManufacturerLocation and Company has `LocationId` as the primary key which also the foreign key for Locations table. You want to map the current table structure to table Per Type inheritance model in entity framework. ManufacturerLocation and Company should derive from Location base entity and ManufacturerLocation should have an association Many to 1 association with Manufacturer entity.

Solution: Import Locations, ManufacturerLocations, Manufacturers and Company tables on entity model designer using table import wizard. Remove all associations created by the wizard. Ensure ManufacturerLocation and Company derives from Location entity. Move properties specific to derived

entity to its own class. Make Location entity abstract since it only serves as a base class and cannot be instantiated directly. Create 1 to Many association between Manufacturer and ManufacturerLocations because a given manufacturer can be in many locations.

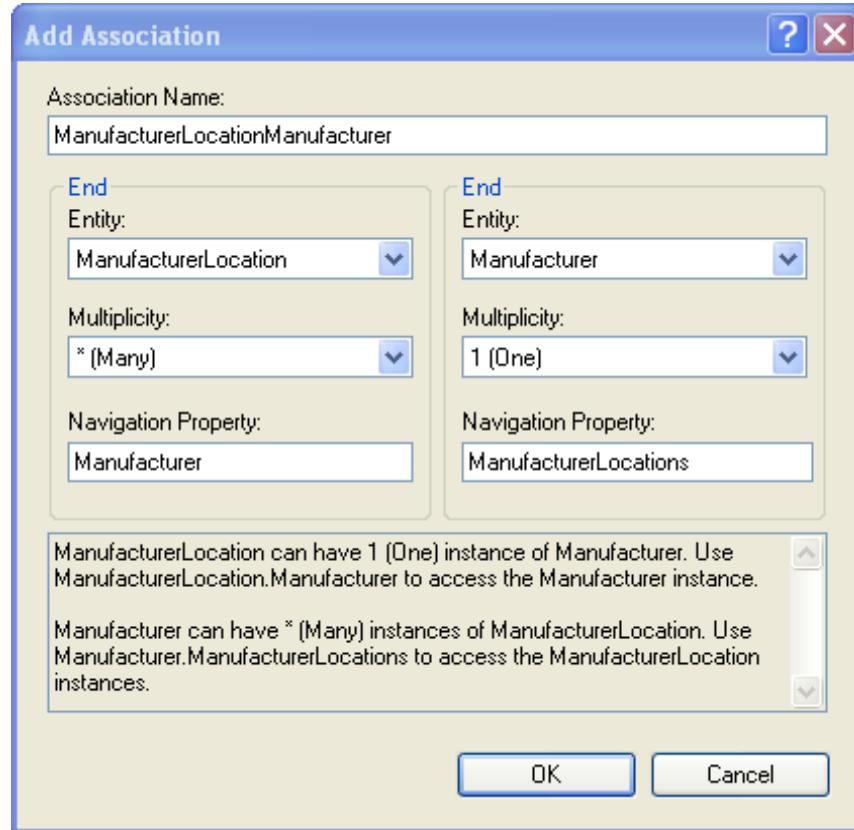
Discussion: In this example, we will learn how to create association between ManufacturerLocation a derived entity and Manufacturer. Steps below outline the process.

1. Import Manufacturers, ManufacturerLocations, Locations and Company table on EDM using import wizard. Figure below shows the model created by the designer after importing the tables from the database.

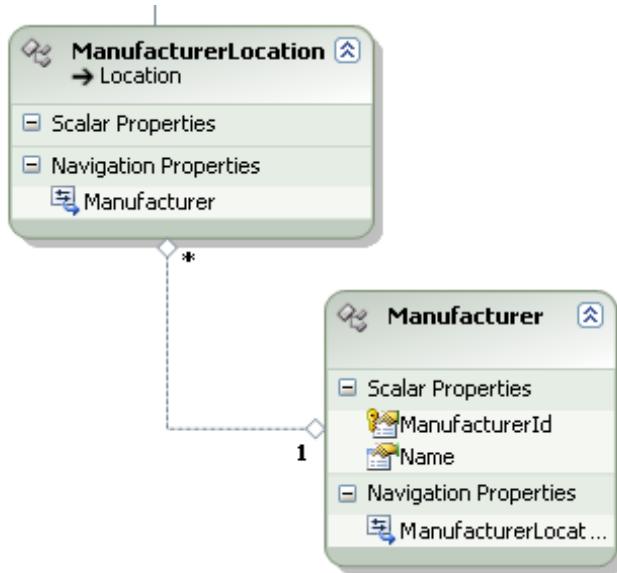


Looking at the model generated by the designer, you will notice that EF created 1 to many association between Manufacturer and Locations entity. We did not get any entity called ManufacturerLocation because entity framework turned that table as 1 to Many association. To fix the model, first remove all the associations created by the designer. This will also remove the navigation properties exposed on the Location entity to Manufacturers. Create a new entity on the designer called ManufacturerLocation that derives from Location entity. Create Many

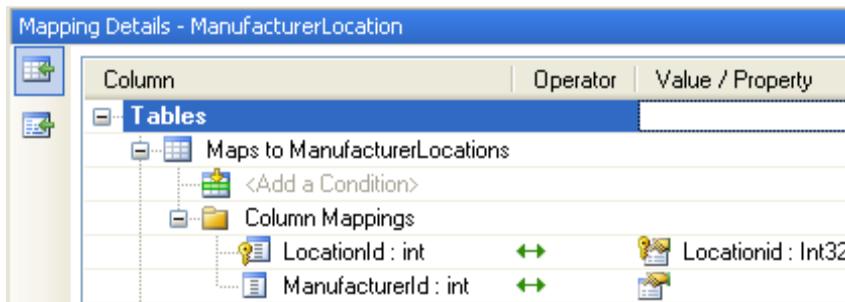
to 1 association between ManufacturerLocation and Manufacturer entity with ManufacturerLocation being the Many side and Manufacturer being the 1 side of the relationship. Figure below shows the association between ManufacturerLocation and Manufacturers.



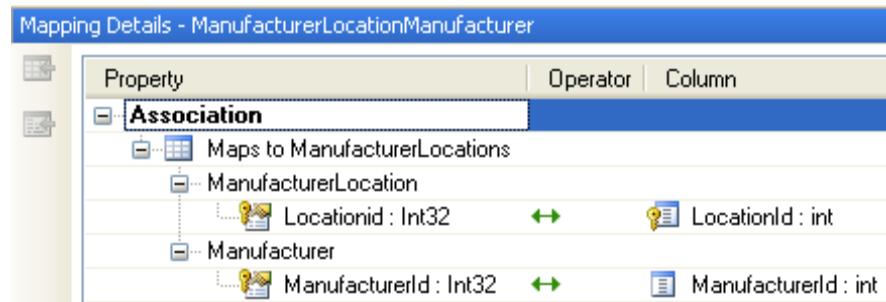
After completing the association between ManufacturerLocation and Manufacturer, EDM model should like below



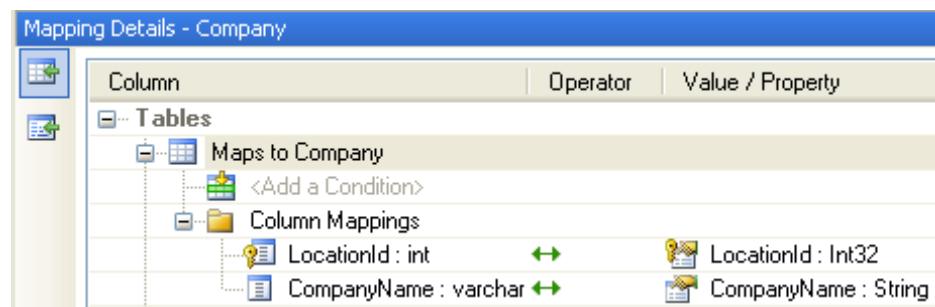
- Map **ManufacturerLocation** to **ManufacturerLocation** table using the mapping window. Set **Locationid** column to map to **LocationId** property inherited from **Location** base entity. Figure below shows the completed mapping for **ManufacturerLocation** entity.



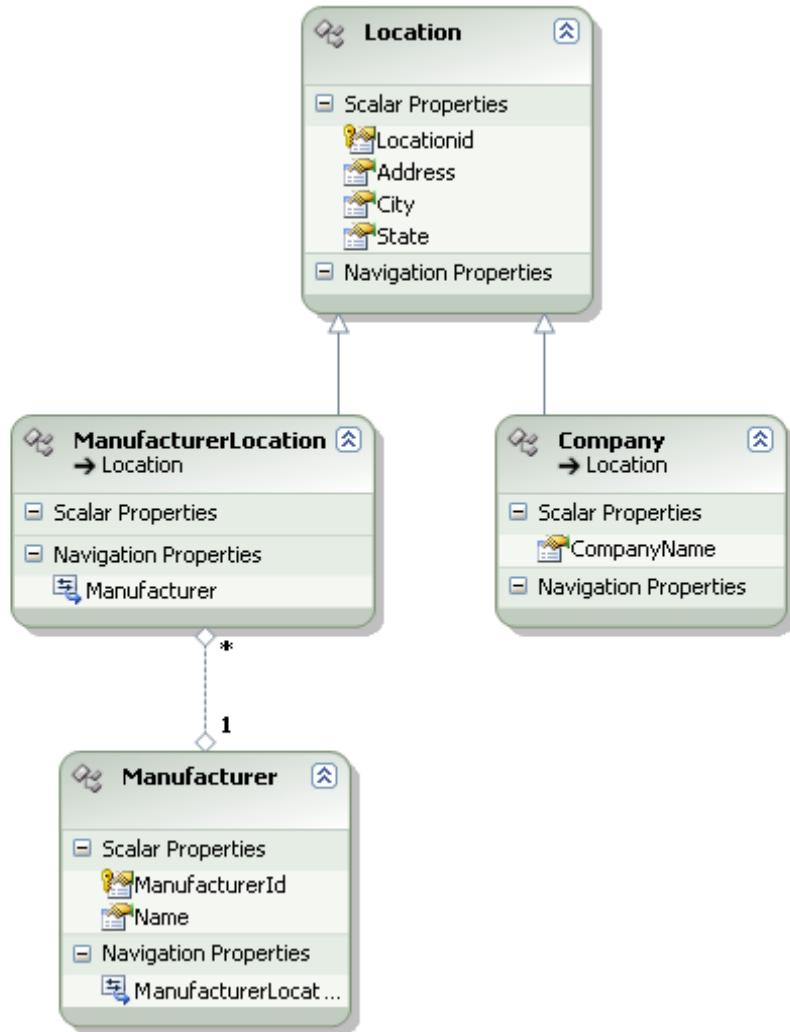
Notice on the mapping window that we did not map any property to **ManufacturerId** column. This is because we will use association between **ManufacturerLocation** and **Manufacturer** to populate **ManufacturerId**. To do this right click on the association line between **Manufacturer** and **ManufacturerLocation** and select table mapping window and choose **ManufacturerLocation** table. The designer will auto fill the columns to map to entity keys defined on both ends of the association. Figure below shows the mapping for the association.



3. Next select company entity and ensure that it derives from Location entity. Remove the Locationid property on Company entity because we inherited LocationId from Location entity. On the mapping window for Company entity, ensure that LocationId column maps to LocationId property acquired from Location base entity. Figure below shows the mapping for Company entity.



After completing the above steps, entity model for manufacturer and Company should like the one below.



Code below shows how to use create **ManufacturerLocation**, associate it with a **Manufacturer** and save it to the database. In addition we are also creating **Company** entity a derived type of **Location** and saving that to the database also.

```

var db = new MediaTPTTBH();
var manuflocation = new ManufacturerLocation
{
    Address = "1001 FullerWiser",
    City = "Euless",
    State = "Tx",
    Manufacturer = new Manufacturer { Name = "Food Ltd" }
};
var company = new Company
{
    Address="Mark blvd",
  
```

```

        City="Dallas",
        State = "Tx",
        CompanyName="EnergyDrinks"
    } ;
    db.AddToLocations(manuflocation);
    db.AddToLocations(company);
    db.SaveChanges();
    var db2 = new MediaPTTBH();
    var manloc =
db2.Locations.OfType<ManufacturerLocation>().Include("Manufacturer").First();
    var comp = db2.Locations.OfType<Company>().First();
    Console.WriteLine("Name {0} Address
{1}",manloc.Manufacturer.Name,manloc.Address);
    Console.WriteLine("Name {0} Address
{1}",comp.CompanyName,comp.Address);

```

On the above code, I am creating an instance of ManufacturerLocation a derived entity of Location and specifying its address. To assign a manufacturer to ManufacturerLocation entity, I create a new instance of Manufacturer and assign it to Manufacturer proeprty. Next I create an instance of Company entity which is also derives from Location entity and save both entities to the database. On retrieval process I use ofType operator to retrieve only ManufacturerLocation derived entity. Notice that I am using Include operator to Load Manufacturer for the location as well. It is import that you call Include after using the OfType operator. If you applied Include operator on db.Locations, you will get runtime error complaining that it cannot find Manufacturer navigation property on Location entity. The error is correct because Manufacturer navigation property is not available on base Location entity. It is available on the derived type ManufacturerLocation. When you include the hint of OfType operator, entity framework knows that you are trying to fetch ManufacturerLocations and therefore queries for manufacturer association on ManufacturerLocation entity instead of looking on Location entity. Similarly to get companies back, I use ofType operator with Company entity and output the results to console window. Figure below shows the output of the above code on the console window.

```

C:\WINDOWS\system32\cmd.exe
Name Food Ltd Address 1001 FullerWiser
Name EnergyDrinks Address Mark blvd
Press any key to continue . .

```

5.1.12 Table per Concrete Type

Problem: Figure below shows Tables Company and School defined on the database.

Company		School	
	CompanyId		SchoolId
	CompanyName		SchoolName
	President		Principal
	Address		Address

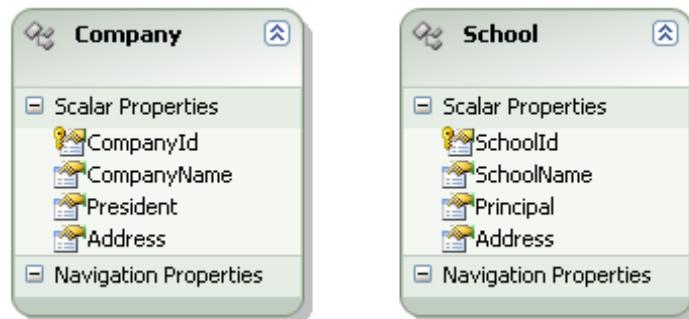
You want to use the above table in your entity data model using Table per Concrete Type implementation. The model should have base entity Location which has an address field that is common to both tables. The Location entity should have two derived entities Company and School containing properties specific to their entity. To ensure that primary key values do not collide between Company and School table, the identity column CompanyId for Company table is set to start at 200 and identity column SchoolId starts for School table starts at 1.

Solution: Import Company and School table using the import wizard. Create abstract Location entity and move address property from School and Company entity to Location entity. Make Company and School entity derive from Location entity. Since the designer does not support mapping properties

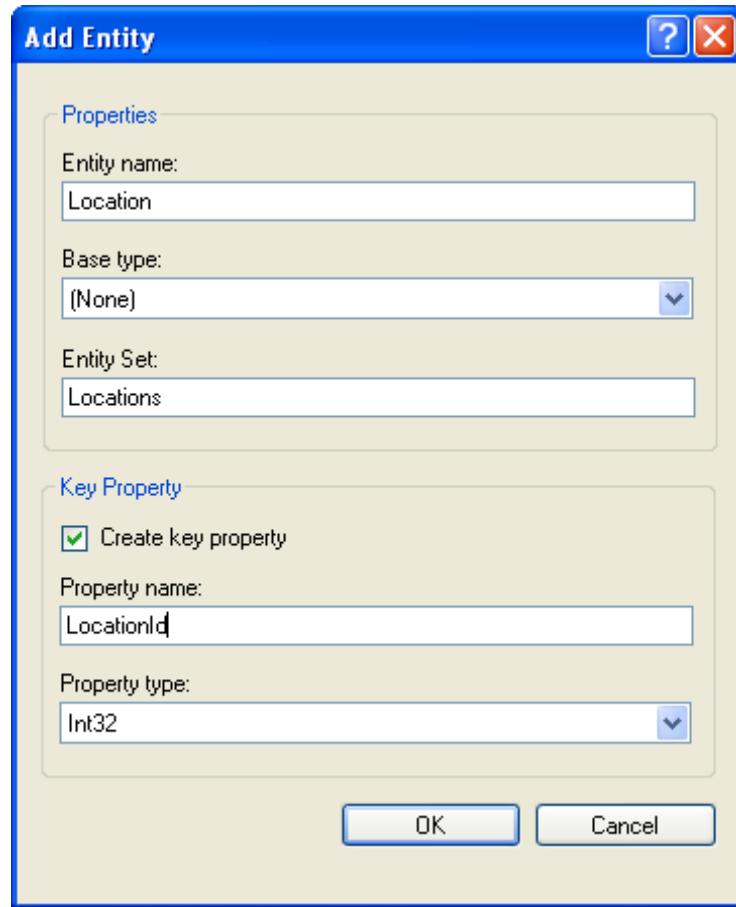
to different table on derived entity, modify the msl manually by adding LocationId and address mapping to both Company and Office entity.

Discussion: Table Per Concrete Type allows individual tables to map to same base entity. If there are column that differ in each table, those columns can appear as properties on derived entity. For Table per Concrete Type to work properly, the entity key must be unique across both tables. If entity framework finds that CompanyId and SchoolId have the same value for a primary key, it would throw primary key violation during runtime. In addition Table Per Concrete Type allow two or more tables to be under the same EntitySet umbrella making querying, inserting and updating a unified process. Steps below outline the process of importing Company and School table to use Table per Concrete Type.

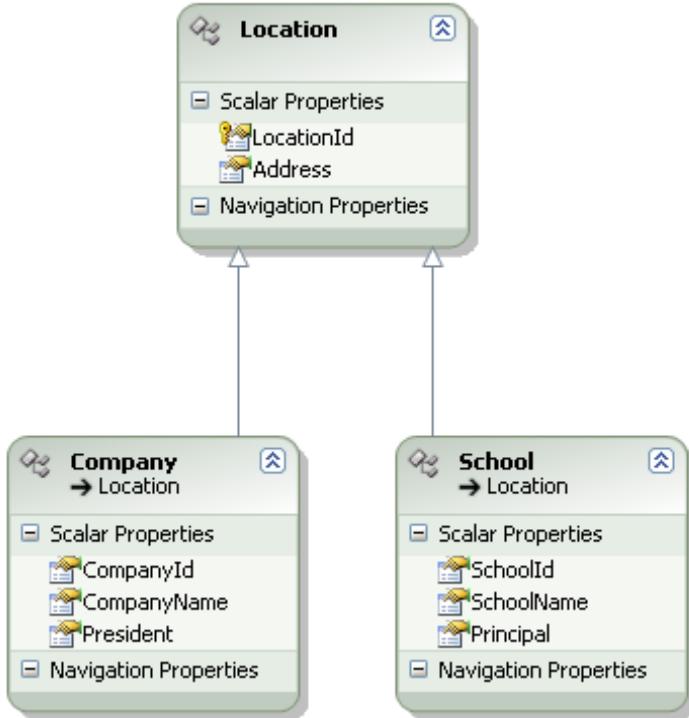
1. Import Company and School table using table import wizard. After completing the wizard, entity model show look as below.



2. Create Location entity with entity key of LocationId and make Location entity abstract. LocationId entity key will be shared between both Company and School entitiy. Figure below shows Add Entity window for Location entity.



3. Derive School and Company entity from Location entity. Remove SchoolId from School entity and CompanyId from Company entity since derived types will use LocationId entity key defined on Location entity. Remove Address property from School and Company entity and add Address property to Location entity because both derived entities have the same Address column defined on their tables. Updated entity model should look as follows.



Since we imported the table using the wizard, we have most of the mapping done by the designer for properties defined on the derived entities. For properties defined on base entity Location such as Address and LocationId, they need to be mapped to each derived entity by manually editing the msl. As said earlier, the designer does not support mapping for Table per Concrete type so rest of the mapping will require hand editing the xml defined on the msl. Figure below shows the current mapping for Company and Office.

```

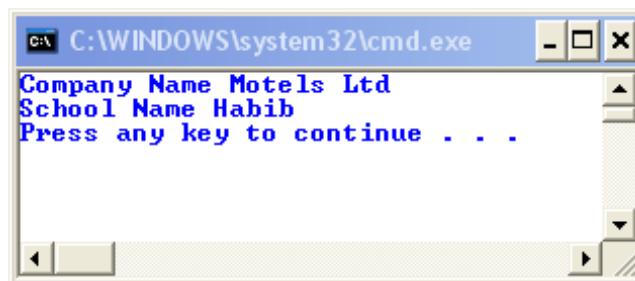
<EntityTypeMapping TypeName="IsTypeOf(INHTPCModel.Company)">
    <MappingFragment StoreEntitySet="Company">
        <ScalarProperty Name="CompanyId" ColumnName="CompanyId" />
        <ScalarProperty Name="CompanyName" ColumnName="CompanyName" />
        <ScalarProperty Name="President" ColumnName="President" />
    </MappingFragment>
</EntityTypeMapping><EntityTypeMapping
TypeName="IsTypeOf(INHTPCModel.School)">
    <MappingFragment StoreEntitySet="School">
        <ScalarProperty Name="SchoolId" ColumnName="SchoolId" />
        <ScalarProperty Name="SchoolName" ColumnName="SchoolName" />
        <ScalarProperty Name="Principal" ColumnName="Principal" />
    </MappingFragment>
</EntityTypeMapping>

```

Notice that above mapping is missing the mapping for Locationid and Address defined on the base Location entity. To complete the mapping we need to mapping for these two properties on both Company and School entity as shown below.

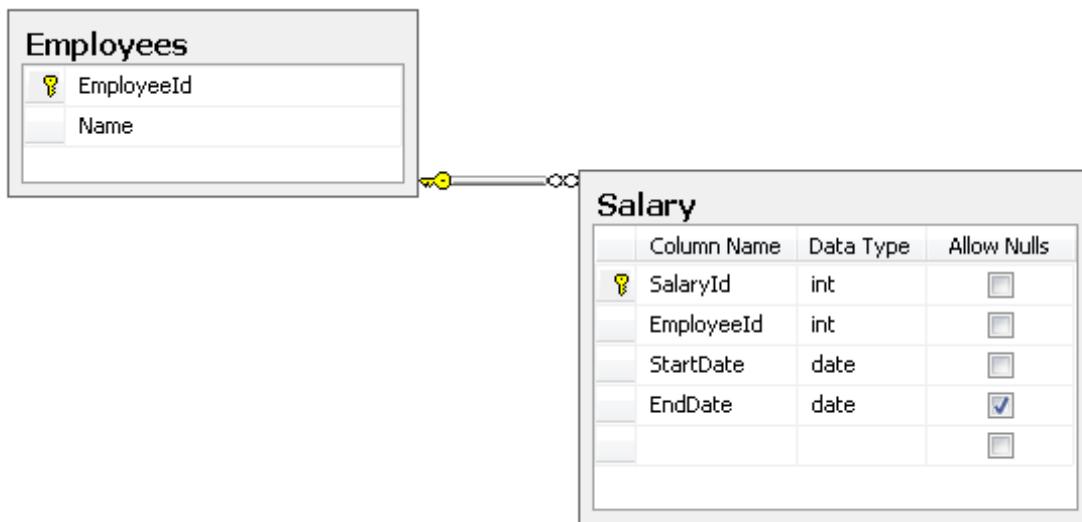
```
<EntityTypeMapping TypeName="IsTypeOf(INHTPCModel.Company)">
    <MappingFragment StoreEntitySet="Company">
        <ScalarProperty Name="LocationId"
ColumnName="CompanyId" />
        <ScalarProperty Name="Address" ColumnName="Address" />
    </MappingFragment>
</EntityTypeMapping><EntityTypeMapping
TypeName="IsTypeOf(INHTPCModel.School)">
    <MappingFragment StoreEntitySet="School">
        <ScalarProperty Name="LocationId"
ColumnName="SchoolId" />
        <ScalarProperty Name="Address" ColumnName="Address" />
    </MappingFragment>
    <ScalarProperty Name="SchoolName" ColumnName="SchoolName" />
    <ScalarProperty Name="Principal" ColumnName="Principal" />
</MappingFragment>
</EntityTypeMapping>
```

Code below creates an instance of Company and School entities and saves them to the database by calling AddToLocations. Since both Company and School are part of same entityset, we can add both entities to same entityset Locations which is what AddToLocations generated code does. Similarly to query for Company entity, I am using OfType operator on Locations entityset, followed by First since we only created a single company. Same process is applied to fetch school entity from the ObjectContext. Screenshot below shows the companynname and schoolname displayed on Console window.



5.1.13 Mapping Column Used as a Discriminator

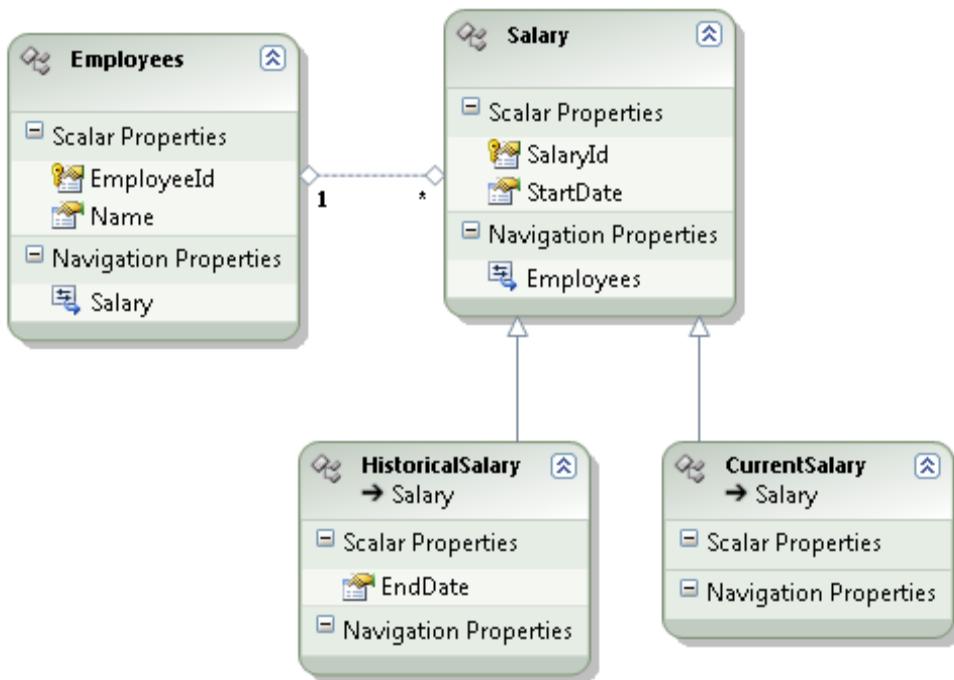
Problem: Figure below shows the current table structure for Employee and its Salary table in the database.



You want to model the above table structure using table per Hierarchy. The model should have an abstract salary entity with two derived entities; CurrentSalary and HistoricalSalary. Current and HistoricalSalary are identified by EndDate column on Salary table. If EndDate column has a null value, it is considered CurrentSalary entity otherwise it is a HistoricalSalary. The base entity Salary should have 1 to many associations with Employees because an Employee can have many salaries with only 1 salary being the current salary and rest would be HistoricalSalary.

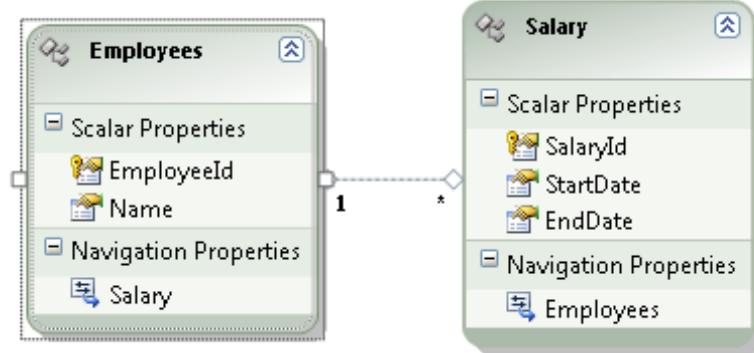
Solution: Import Employee and Salary table using the import wizard. Make Salary entity abstract. Create two entities HistoricalSalary and CurrentSalary that derive from Salary entity. For CurrentSalary apply a condition on the mapping where EndDate is null. For HistoricalSalary entity, apply the condition for the mapping where EndDate is not Null. At this point if you try to validate the model, you will receive error stating that EndDate has a condition for not null and there is no mapping specified. This makes legitimate sense because once we specify a condition for previous salary

entity stating that endDate cannot be null, then there has to be a way to map a value to that property, otherwise what should its value be? So after applying EndDate condition on HistoricalSalary, make sure to move EndDate property from base Salary entity to HistoricalSalary and map the property to EndDate column on Salary table. When we imported the model, the designer created the EndDate property as nullable since EndDate column has allow null in the database. However for HistoricalSalary, our condition states that EndDate can never be null. So the last step to validate the model requires making EndDate property as not nullable. The final entity diagram should look like below.



Discussion: In most scenarios entity framework does not allow mapping a discriminator column to a property value. Since we had applied a condition for mapping to HistoricalSalary that EndDate can never be null, the constraint mandated us to provide a value for EndDate property. On other scenarios, if you try to map a property to a column used a discriminator or applied as a condition, EF will throw validating errors stating that a discriminator column cannot be mapped because it could invalidate the model. In this walk through we will go through the steps of building the conceptual model shown above.

1. Import Employees and Salary table using the import wizard. Figure below shows the model after the wizard is completed.



2. Make Salary entity abstract. Create two entities CurrentSalary and HistoricalSalary deriving from Salary entity. Map CurrentSalary entity to Salary table and apply the condition where EndDate is null. Figure below shows the mapping for CurrentSalary entity.

Mapping Details - CurrentSalary			
Column	Operator	Value / Property	
Tables			
Maps to Salary	Is	Null	
When EndDate			
<Add a Condition>			
Column Mappings			
EmployeeId : int			

3. Map HistoricalSalary to Salary table and apply condition on EndDate column where EndDate is not null. Additionally move the EndDate column on the Salary base entity to HistoricalSalary and change the nullability of EndDate property to false. Then map the EndDate property to EndDate column on the Salary table. Figure below shows the mapping for HistoricalSalary.

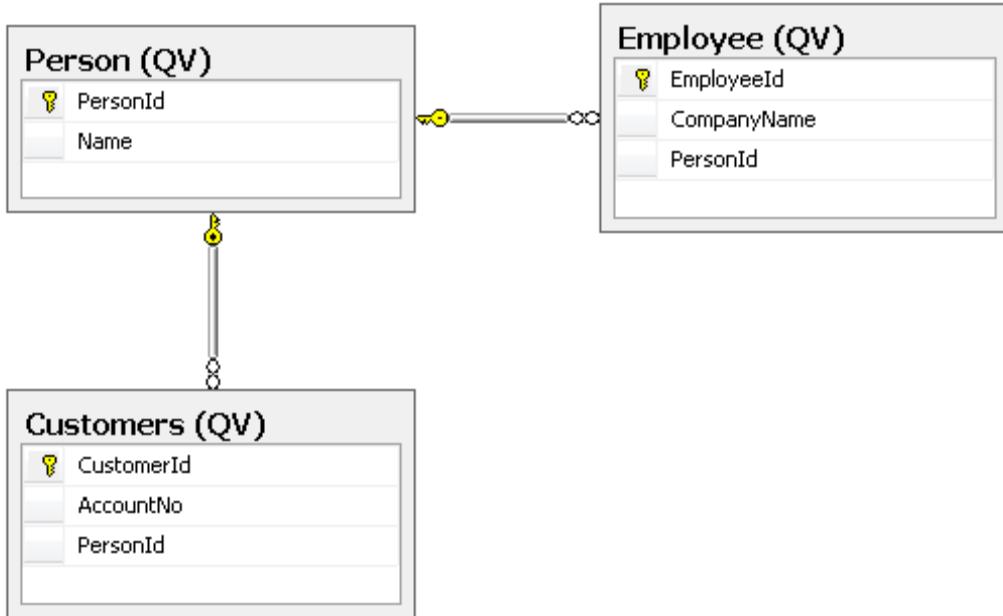
Column	Operator	Value / Property
Tables		
Maps to Salary	Is	Not Null
When EndDate		
<Add a Condition>		
Column Mappings		
EmployeeId : int	↔	
EndDate : date	↔	EndDate : DateTime

On the code below, I am using the above model, by creating two instances of Salary, Current and Historical Salary and adding both Salaries to employee's salary collection. Then using the second data context, I retrieve the employee along with its Salary collection by using Include operator. To confirm that we get each type of salary, I am printing type of salary on the console when we loop through the salary collection. Figure below shows the Console window containing our results.

```
C:\Windows\system32\cmd.exe
Sal 95000 Type HistoricalSalary
Sal 100000 Type CurrentSalary
Press any key to continue . . .
```

5.1.14 Mapping Table per Type to Foreign Key column

Problem: Figure below shows Employee and Customer structure defined on the database.



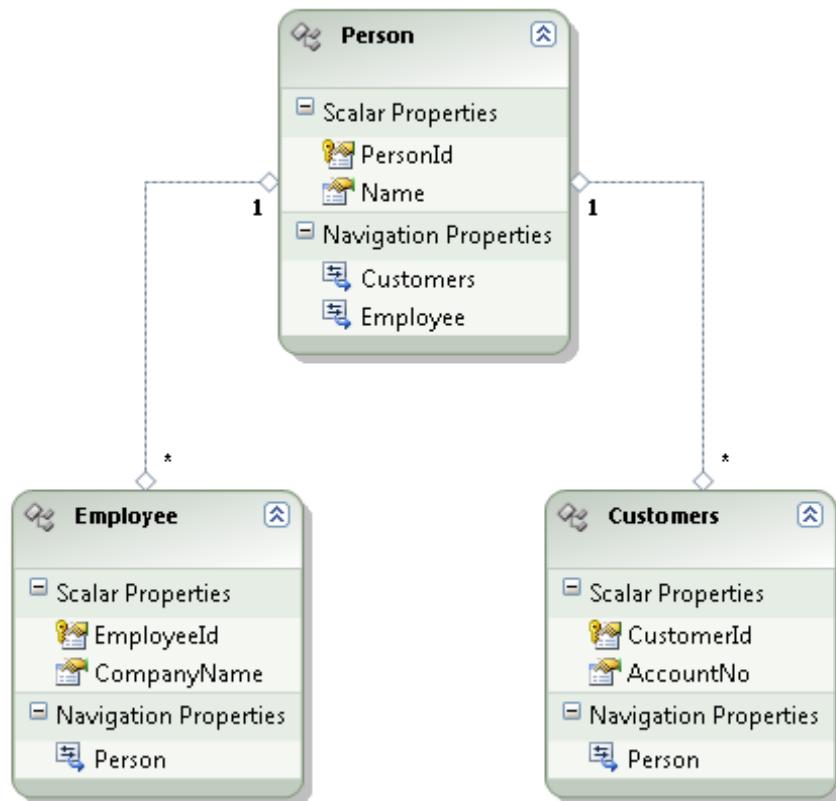
On the above table relationship, Employee and Customer inherit the Name column from Person table. Instead of regular Table per Type where derived table's primary key is the foreign key for base table, Customer and Employee table have their own primary key column like CustomerId and EmployeeId and have an additional column PersonId which is the foreign key to base table. You want to import the above table relationship as Table per Type inheritance on entity data model.

Solution: For entity framework to model table per type, the primary key from derived table also needs to be the foreign key for the base table. If this relationship is not defined on the tables, then entity framework cannot map the tables to Table per Type inheritance. What this means is you can't leverage the default mapping capabilities of the designer. However you can write QueryView in which you can write arbitrary esql to join different tables from the store model and define your own mapping. First, import Person, Employee and Customers table using Entity Model Wizard. Delete all the associations created by the designer and make Employee and Customer entity derive from Person entity. Ensure that EmployeeId and CustomerId are no longer the entity key on Customer and Employee entity because conceptual model needs to still follow EF rules which states that derived table's primary key must also be the foreign key to the base entity. Since our store model

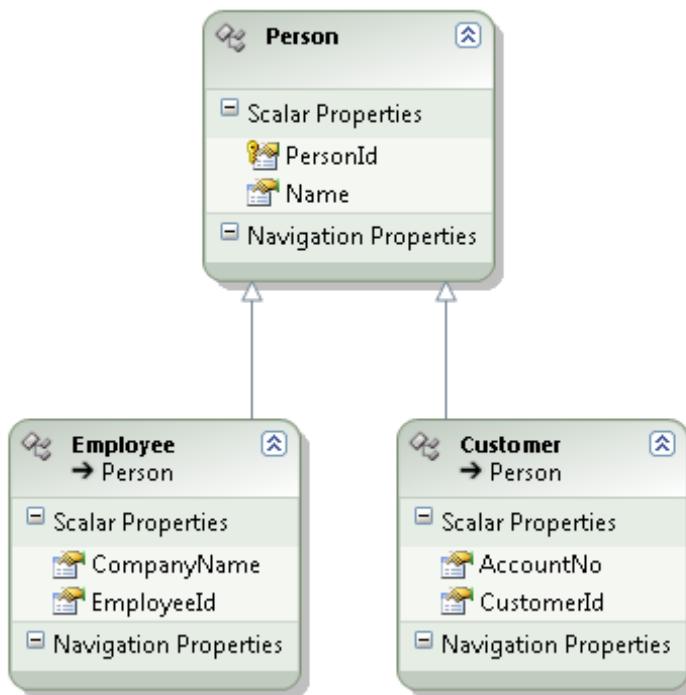
does not follow this practice, we can open the edmx file in xml format and use QueryView for Persons entityset.

Discussion: In this example, we will walk through the steps of using QueryView to map table per type inheritance to a foreign key column on a derived table that is not the primary key of the table. For instance to map the above table structure as Table Per Type, entity framework requires that PersonId primary key on Person table to be also the primary key on Customer and Employee table. However both Customer and Employee table have their own primary key and have an additional column PersonId that serves as the foreign key to Person table. To get around the problem, we will create the Table per Type model based on Ef requirements but use a custom queryview to map our tables' results to the conceptual model. Steps below outline the process.

1. Import the table using entity data model wizard. Figure below shows the model after the wizard has completed.



2. Remove all the associations created by the designer. Make Employee and Customer entity derive from Person entity. Since inheriting will give us PersonId as the entity key which also has to be the entity key for derived entity, change the CustomerId and EmployeeId to not be an entity key anymore. In addition remove all the table mappings created by the designer because we will modify the msl in xml and define a queryview to populate each of entity defined with Person entityset. Figure below shows the updated model



3. Open the edmx file in xml and modify the msl with the following QueryView for Persons entityset.

```

<EntitySetName="Persons">
    <QueryView>
        select value
        case
            when (c.PersonId is not null) then
                QvForeignModel.Customer(p.PersonId,p.Name,c.AccountNo,c.CustomerId)
            when (e.PersonId is not null) then
  
```

```

    QvForeignModel.Employee(p.PersonId,p.Name,e.CompanyName,e.EmployeeId)
        END
        from
    QvForeignModelStoreContainer.Person as p
        left join
    QvForeignModelStoreContainer.Customers as c on p.PersonId =
c.PersonId
        left join
    QvForeignModelStoreContainer.Employee as e on p.PersonId =
e.PersonId
    </QueryView>
    </EntitySetMapping>

```

On the above QueryView, I am using esql to join Person table defined on the store model to Customers and Employee table. If there is a matching row found in Customer table, I am creating an instance of Customer entity and if there is a matching row found on employee entity, I am creating an instance of Employee entity. This means that when we query for Persons collection, there would be some person of type Customer and some Employee. It is important to understand that when we write a queryview, EF has no clue how save the entities to the database. You are required to provide stored procedures that can map each entity to the database. So first we need to define mapping for insert, update and delete stored procedure for each entity that is part of Person entityset. Msl below shows the stored procedure mapping for Customer and Employee derived entities.

```

<EntityTypeMapping TypeName="QvForeignModel.Customer">
    <ModificationFunctionMapping>
        <InsertFunction
            FunctionName="QvForeignModel.Store.InsertCustomer">
                <ScalarProperty
                    Name="Name" ParameterName="Name" />
                <ScalarProperty
                    Name="AccountNo" ParameterName="AccountNo" />
                <ResultBinding
                    Name="PersonId" ColumnName="PersonId" />
                <ResultBinding
                    Name="CustomerId" ColumnName="CustomerId" />
                </InsertFunction>
                <UpdateFunction
                    FunctionName="QvForeignModel.Store.UpdateCustomer">

```

```

          <ScalarProperty
Name= "Name" ParameterName= "Name" Version= "Current" />
          <ScalarProperty
Name= "AccountNo" ParameterName= "AccountNo" Version= "Current" />
          <ScalarProperty
Name= "PersonId" ParameterName= "PersonId" Version= "Current" />
          <ScalarProperty
Name= "CustomerId" ParameterName= "CustomerId" Version= "Current" />
          </UpdateFunction>
          <DeleteFunction
FunctionName= "QvForeignModel.Store.DeleteCustomer">
          <ScalarProperty
Name= "PersonId" ParameterName= "PersonId" />
          </DeleteFunction>

      </ModificationFunctionMapping>
      </EntityTypeMapping>
      <EntityTypeMapping
TypeName= "QvForeignModel.Employee">
          <ModificationFunctionMapping>
              <InsertFunction
FunctionName= "QvForeignModel.Store.InsertEmployee">
                  <ScalarProperty
Name= "Name" ParameterName= "Name" />
                  <ScalarProperty
Name= "CompanyName" ParameterName= "CompanyName" />
                  <ResultBinding
Name= "PersonId" ColumnName= "PersonId" />
                  <ResultBinding
Name= "EmployeeId" ColumnName= "EmployeeId" />
                  </InsertFunction>
                  <UpdateFunction
FunctionName= "QvForeignModel.Store.UpdateEmployee">
                  <ScalarProperty
Name= "Name" ParameterName= "Name" Version= "Current" />
                  <ScalarProperty
Name= "CompanyName" ParameterName= "CompanyName" Version= "Current" />
                  <ScalarProperty
Name= "PersonId" ParameterName= "PersonId" Version= "Current" />
                  <ScalarProperty
Name= "EmployeeId" ParameterName= "EmployeeId" Version= "Current" />
                  </UpdateFunction>
                  <DeleteFunction
FunctionName= "QvForeignModel.Store.DeleteEmployee">
                  <ScalarProperty
Name= "PersonId" ParameterName= "PersonId" />

```

```

        </DeleteFunction>

    </ModificationFunctionMapping>
        </EntityTypeMapping>

```

The above mapping is very similar to the examples we have covered on stored procedure section. The only noticeable difference is that both InsertCustomer and InsertEmployee have two resultBindings. This is because the PersonId is the primary key for Person table generated by the database and CustomerId and EmployeeId are the primary keys for Customer and Employee table also generated by the database. After mapping the stored procedure, we need to define these stored procedures inside our storage model. We can either created procedures in the database and reference it on our storage model, or we can declare the code inline inside the commandText property of the function. Since insert, update and delete stored procedure for Employee and Customer are pretty similar, code below shows the crud proc for Customer entity.

```

<Function IsComposable="false" BuiltIn="false"
Name="InsertCustomer">
    <CommandText>
        declare @personid int
        insert into qv.person(Name) values
(@Name)
        set @personid = SCOPE_IDENTITY()
        insert into
qv.Customers(AccountNo,PersonId) values (@AccountNo,@personid)
        select @personid as
PersonId,SCOPE_IDENTITY() as CustomerId
    </CommandText>
    <Parameter Name="Name" Type="varchar"
Mode="In" />
        <Parameter Name="AccountNo"
Type="varchar" Mode="In" />
    </Function>
    <Function IsComposable="false"
BuiltIn="false" Name="UpdateCustomer">
        <CommandText>
            update qv.person set Name = @Name
where personid = @PersonId
            update qv.Customers set AccountNo
=@AccountNo where CustomerId =@CustomerId
        </CommandText>

```

```

<Parameter Name="Name" Type="varchar"
Mode="In" />
<Parameter Name="AccountNo"
Type="varchar" Mode="In" />
<Parameter Name="PersonId" Type="int"
Mode="In" />
<Parameter Name="CustomerId" Type="int"
Mode="In" />
</Function>
<Function IsComposable="false"
BuiltIn="false" Name="DeleteCustomer">
<CommandText>
    delete qv.person where personid
    =@PersonId
    delete qv.Customers where personid
    = @PersonId
</CommandText>
<Parameter Name="PersonId" Type="int"
Mode="In" />
</Function>

```

For InsertCustomer stored procedure, I am inserting the Name into Person table and then assigning the PersonId inserted to local variable personid. Then using PersonId and AccountNo parameter, I am inserting a record inside of Customer entity. To return both PersonId and CustomerId back to the conceptual layer using ResultBinding, I am executing a select statement with both personid and CustomerId. The alias that I am using for selecting PersonId and CustomerId has to match with the ResultBinding parameter on the stored procedure mapping defined on the msl. Update stored procedure updates both Person and Customer table with the appropriate values from the parameter. Similarly delete procedure only uses PersonId to delete the person first from Person table and then delete the customer that matches the personId in the Customer table.

Code below is used to test the model we created earlier and confirm that our crud procedures performs the insert, update and deletes correctly. In the code, I am creating different type of Person such as Customer and Employee entity, adding it Persons entityset and saving it to the database. Using second datacontext, I am retrieving both Customer and Employee and

printing the Name, PersonId , CustomerId and EmployeeId on the console window.

```
var db = new QvForeignEntities();
    var customer = new Customer { AccountNo = "123",
Name = "Zeeshan" };
        var employee = new Employee { CompanyName = "Chem
Ltd", Name = "John" };
            db.AddToPersons(customer);
            db.AddToPersons(employee);
            db.SaveChanges();

var db2 = new QvForeignEntities();
var cus2 = db2.Persons.OfType<Customer>().First();
Console.WriteLine("Name {0} PersonId {1} CustId
{2}", cus2.Name, cus2.PersonId, cus2.CustomerId);

var emp2 = db2.Persons.OfType<Employee>().First();
Console.WriteLine("Name {0} PersonId {1} CustId
{2}", emp2.Name, emp2.PersonId, emp2.EmployeeId);
```

Figure below shows the output from the console window.

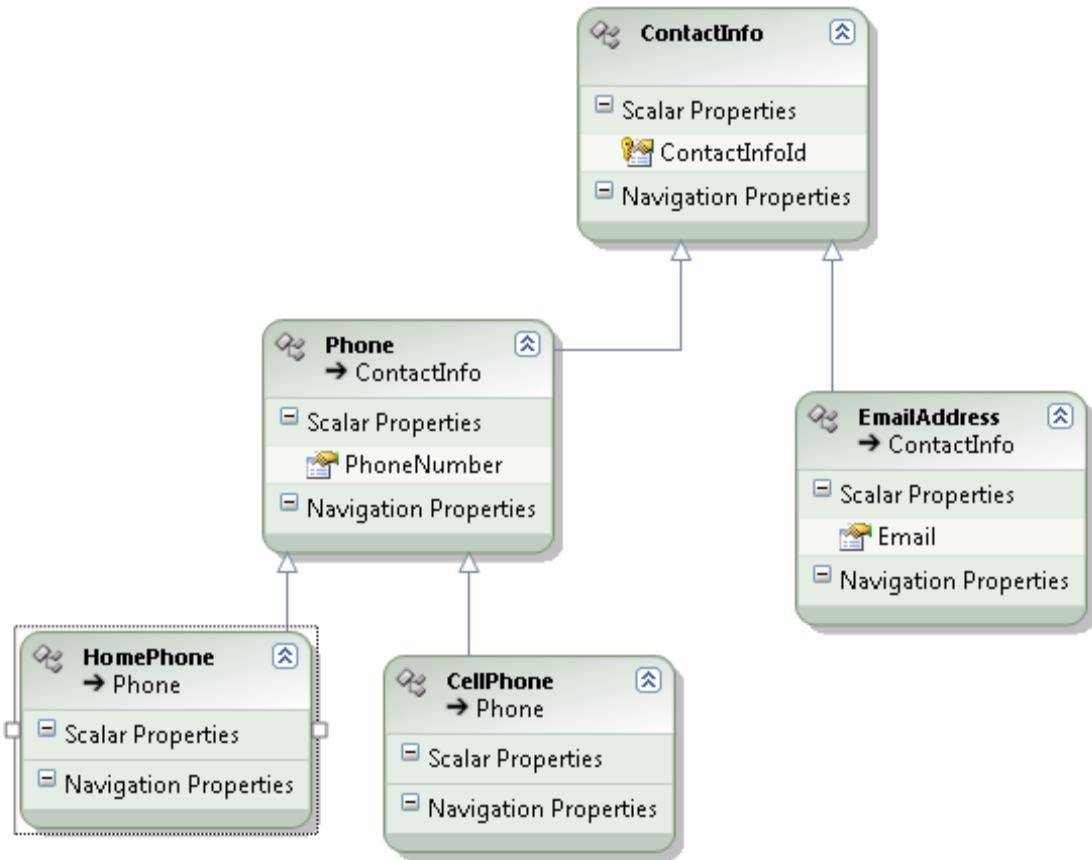
```
C:\Windows\system32\cmd.exe
Name Zeeshan PersonId 7 CustId 5
Name John PersonId 8 CustId 3
Press any key to continue . . .
```

5.1.15 Using QueryView with TPH to create additional inheritance layer

Problem: Figure below shows the ContactInfo table structure that contains different types of contacts based on ContactType column.

ContactInfo (TBH)			
	Column Name	Data Type	Allow Nulls
>ContactInfoId	int	<input type="checkbox"/>	
ContactType	char(2)	<input type="checkbox"/>	
PhoneNumber	varchar(50)	<input checked="" type="checkbox"/>	
Email	varchar(50)	<input checked="" type="checkbox"/>	

Contacts stored in the table can be of 3 different types; HomePhone, CellPhone, and EmailAddress. You want to model the above table structure using Table per Hierarchy but want to add an additional layer of inheritance on your conceptual model. Both CellPhone and HomePhone should extend from Phone entity and Phone entity should derive from ContactInfo entity. Figure below shows how the conceptual model should look like after modeling.

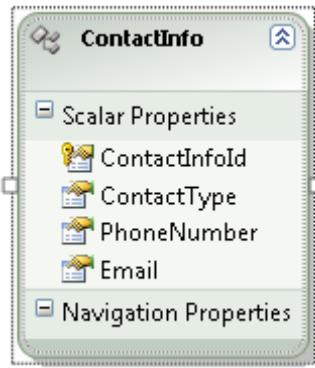


Solution: Import ContactInfo table using Entity data model wizard. Create Phone and EmailAddress entity that derive from ContactInfo entity. Move Email property from ContactInfo entity to EmailAddress entity and move PhoneNumber from ContactInfo to Phone entity. Make ContactInfo and Phone entity as abstract. Create two entities HomePhone and CellPhone deriving from Phone entity and remove all the table mappings created by the designer. Modify the entityset mapping for ContactInfo by writing a QueryView that queries the store model and create appropriate derived entities based on the value for ContactType.

Discussion: Adding an additional inheritance structure Phone requires a logical OR operation for the condition. The mapping we want for phone entity is, if ContactType is either HomePhone or CellPhone it should be considered a Phone. However entity framework can only map inheritance based on logical And conditions. There is no option to specify that ContactType could be either HomePhone or CellPhone. To get around this limitation, we can

write QueryView that queries the store model and maps the data appropriately to derived entities. Steps below outline the process.

1. Import ContactInfo table using Entity data model wizard. Figure below shows the model created by the wizard.



2. Create Phone entity deriving from ContactInfo. Move PhoneNumber field from ContactInfo entity to Phone entity. Mark Phone entity as abstract.
3. Create EmailAddress entity deriving from ContactInfo and move Email field ContactInfo to EmailAddress entity.
4. Remove ContactType property from ContactInfo entity because ContactType column will be used as a discriminator column to map inheritance and if the column is used as a discriminator column it cannot be mapped. Ensure that ContactInfo entity is marked as abstract. Since we will use QueryView to do custom mapping remove the mapping created by the wizard from ContactInfo entity. Mapping can be removed by right clicking the entity, selecting the mapping window and deleting the table from table selection.
5. Create two entities HomePhone and CellPhone deriving from Phone entity. Both derived entities will inherit PhoneNumber property from Phone entity.
6. To create a QueryView for ContactInfo entity, open the edmx file in xml and define a QueryView section inside of ContactInfo entityset. Msl below shows QueryView required for mapping.

```
<EntitySetName="ContactInfos">
  <QueryView>
```

```

        select value
        case
        when c.ContactType = 'HP' then
            TPHQVContactModel.HomePhone(c.ContactInfoId,c.PhoneNumber)
        when c.ContactType = 'CP' then
            TPHQVContactModel.CellPhone(c.ContactInfoId,c.PhoneNumber)
        when c.ContactType = 'EA' then
            TPHQVContactModel.EmailAddress(c.ContactInfoId,c.Email)
        END
        from
TPHQVContactModelStoreContainer.ContactInfo as c
    </QueryView>
    </EntitySetMapping>

```

On the above QueryView, if the ContactType has a value of HP, create an instance of HomePhone, if ContactType has a value of CP, create an instance of CellPhone and if ContactType has a value of EA, create an instance of EmailAddress entity. So an entitySet of ContactInfos will contain different types of Contacts ranging from CellPhone, HomePhone and EmailAddress and since we have derived HomePhone and CellPhone from Phone entity, we can query for just phones and the results returned would contain instances of both Home and Cell entities. When we use QueryView, we cannot leverage the out of the box support for insert, update and deletion of the entities. We have to write stored procedure, define them on the store model and map the store procedures to entities on the msl to be able to save the entities to the database. Msl below shows the mapping of stored procedures to entities.

```

<EntityTypeMapping TypeName="TPHQVContactModel.HomePhone">
    <ModificationFunctionMapping>
        <InsertFunction
            FunctionName="TPHQVContactModel.Store.InsertHomePhone">
                <ScalarProperty
                    Name="PhoneNumber" ParameterName="PhoneNumber" />
                <ResultBinding
                    Name="ContactInfoId" ColumnName="ContactInfoId"/>
            </InsertFunction>

```

```

                <UpdateFunction
        FunctionName="TPHQVContactModel.Store.UpdatePhone">
                    <ScalarProperty
        Name="PhoneNumber" ParameterName="PhoneNumber" Version="Current"
    />
                    <ScalarProperty
        Name="ContactInfoId" ParameterName="ContactInfoId"
        Version="Current" />
                </UpdateFunction>
                <DeleteFunction
        FunctionName="TPHQVContactModel.Store.DeleteContactInfo">
                    <ScalarProperty
        Name="ContactInfoId" ParameterName="ContactInfoId" />
                </DeleteFunction>
                </ModificationFunctionMapping>
            </EntityTypeMapping>
            <EntityTypeMapping
        TypeName="TPHQVContactModel.CellPhone">
                <ModificationFunctionMapping>
                    <InsertFunction
        FunctionName="TPHQVContactModel.Store.InsertCellPhone">
                        <ScalarProperty
        Name="PhoneNumber" ParameterName="PhoneNumber" />
                        <ResultBinding
        Name="ContactInfoId" ColumnName="ContactInfoId" />
                    </InsertFunction>
                    <UpdateFunction
        FunctionName="TPHQVContactModel.Store.UpdatePhone">
                        <ScalarProperty
        Name="PhoneNumber" ParameterName="PhoneNumber" Version="Current"
    />
                        <ScalarProperty
        Name="ContactInfoId" ParameterName="ContactInfoId"
        Version="Current" />
                    </UpdateFunction>
                    <DeleteFunction
        FunctionName="TPHQVContactModel.Store.DeleteContactInfo">
                        <ScalarProperty
        Name="ContactInfoId" ParameterName="ContactInfoId" />
                    </DeleteFunction>
                    </ModificationFunctionMapping>
            </EntityTypeMapping>
            <EntityTypeMapping
        TypeName="TPHQVContactModel.EmailAddress">
                <ModificationFunctionMapping>
                    <InsertFunction
        FunctionName="TPHQVContactModel.Store.InsertEmail">

```

```

                                <ScalarProperty Name="Email" 
ParameterName="Email" />
                                <ResultBinding
Name="ContactInfoId" ColumnName="ContactInfoId"/>
                            </InsertFunction>
                            <UpdateFunction
FunctionName="TPHQVContactModel.Store.UpdateEmail">
                                <ScalarProperty Name="Email" 
ParameterName="Email" Version="Current" />
                                <ScalarProperty
Name="ContactInfoId" ParameterName="ContactInfoId" 
Version="Current" />
                            </UpdateFunction>
                            <DeleteFunction
FunctionName="TPHQVContactModel.Store.DeleteContactInfo">
                                <ScalarProperty
Name="ContactInfoId" ParameterName="ContactInfoId" />
                            </DeleteFunction>
                        </ModificationFunctionMapping>
                    </EntityTypeMapping>

```

The stored procedure mapping is similar to the mapping we have discussed on the stored procedure section so I will not cover each aspect of the mapping. For each entity in the ContactInfo entityset, we are mapping Insert, Update and Delete stored procedure. Since we are targeting the same table, all three entities are using the same delete stored procedure for deleting a specific record from Contactinfo table. After completing the functionmapping, we also need to define the stored procedure in the store model. Code below shows the stored procedure declaration.

```

<Function Name="InsertHomePhone" BuiltIn="false"
IsComposable="false">
    <CommandText>
        insert into
tbh.ContactInfo(ContactType,PhoneNumber) values
('HP',@PhoneNumber)
        select SCOPE_IDENTITY() as ContactInfoId
    </CommandText>
    <Parameter Name="PhoneNumber" Type="varchar" 
Mode="In" />
    </Function>
    <Function Name="UpdatePhone" BuiltIn="false"
IsComposable="false">

```

```

<CommandText>
    update tbh.ContactInfo set PhoneNumber
=@PhoneNumber where ContactInfoId =@ContactInfoId
</CommandText>
<Parameter Name="PhoneNumber" Type="varchar"
Mode="In" />
<Parameter Name="ContactInfoId" Type="int"
Mode="In" />
</Function>
<Function Name="DeleteContactInfo" BuiltIn="false"
IsComposable="false">
<CommandText>
    delete tbh.ContactInfo where ContactInfoId
=@ContactInfoId
</CommandText>
<Parameter Name="ContactInfoId" Type="int"
Mode="In" />
</Function>
<Function Name="InsertCellPhone" BuiltIn="false"
IsComposable="false">
<CommandText>
    insert into
tbh.ContactInfo(ContactType,PhoneNumber) values
('CP',@PhoneNumber)
    select SCOPE_IDENTITY() as ContactInfoId
</CommandText>
<Parameter Name="PhoneNumber" Type="varchar"
Mode="In" />
</Function>
<Function Name="InsertEmail" BuiltIn="false"
IsComposable="false">
<CommandText>
    insert into
tbh.ContactInfo(ContactType,Email) values ('EA',@Email)
    select SCOPE_IDENTITY() as ContactInfoId
</CommandText>
<Parameter Name="Email" Type="varchar" Mode="In"
/>
</Function>
<Function Name="UpdateEmail" BuiltIn="false"
IsComposable="false">
<CommandText>
    update tbh.ContactInfo set Email =@Email
where ContactInfoId =@ContactInfoId
</CommandText>
<Parameter Name="Email" Type="varchar" Mode="In"
/>

```

```

        <Parameter Name="ContactInfoId" Type="int"
Mode="In" />
    </Function>

```

For InsertHomePhone, we are hardcoding the ContactType value to be HP for HomePhone and for InsertCellPhone, we are specifying CP for ContactType column signifying that contact entry we are adding is a CellPhone.

Code below uses the above model by creating an instance of HomePhone, CellPhone and EmailAddress entity, adding it to ContactInfo entityset and saving the entityset to the database. Then using the second datacontext, I retrieve each entity type and prints its result to the Console window to confirm the output returned is what I inserted.

```

var db = new TPHQVEntities();
    var homephone = new HomePhone { PhoneNumber = "817-
354-9989" };
    var cellphone = new CellPhone { PhoneNumber = "817-
354-9988" };
    var emailaddr = new EmailAddress { Email =
"abc@gmail.com" };

    db.AddToContactInfos(homephone);
    db.AddToContactInfos(cellphone);
    db.AddToContactInfos(emailaddr);
    db.SaveChanges();

    var db2 = new TPHQVEntities();
    var hphone =
db2.ContactInfos.OfType<HomePhone>().First();
    var cphone =
db2.ContactInfos.OfType<CellPhone>().First();
    var eaddr =
db2.ContactInfos.OfType<EmailAddress>().First();

    Console.WriteLine("Home Phone {0}",
hphone.PhoneNumber);
    Console.WriteLine("Cell Phone {0}",
cphone.PhoneNumber);
    Console.WriteLine("Email Addr {0}",
eaddr.Email);

```

Figure below shows the results from the above code in Console window.

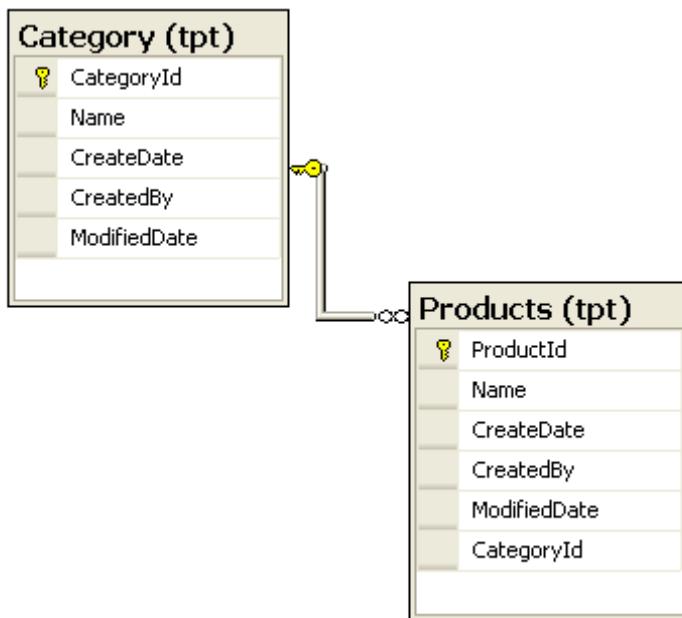
```

cmd C:\Windows\system32\cmd.exe
Home Phone 817-354-9989
Cell Phone 817-354-9988
Email Addr abc@gmail.com
Press any key to continue . .

```

5.1.16 Sharing Audit Fields across entities using TPC

Problem: Figure below shows the database diagram for relationship between Category and Product table.



On the above database diagram, both Category and Product table has audit fields such as CreateDate, CreatedBy, ModifiedDate and ModifiedBy. Instead of having these same fields on each entity on your entity data model, you want to create a base entity called Audit which contains all the audit fields including the primary key for both tables. You then want to make sure that both entities Category and Product derive from base Audit entity so they can

leverage the base properties as well as the business logic that revolves around them.

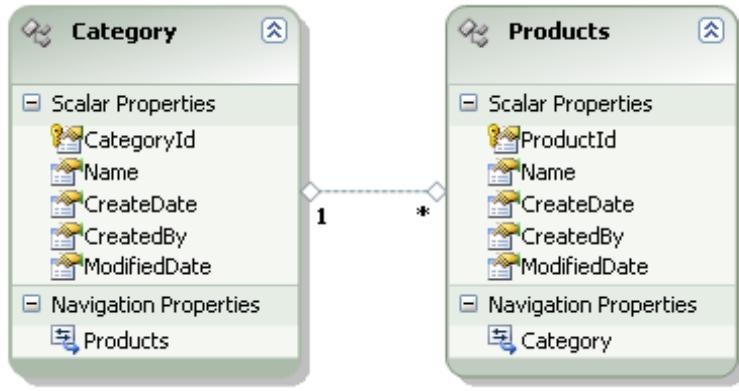
Solution: When we import Category and Products table into EDM and try to map both entities as derived entities of Audit entity base class, EF would complain because both tables do not have anything in common in terms of primary key or entity key. They are in completely different entityset.

Category entity belongs to Category entityset and Product entity belongs to Products entityset. In the current version of EF, the designer does not allow mapping tables to derived entities that belong to completely different entity set. We have to create the conceptual using the designer. Once that's accomplished, we need to modify the mapping manually in the xml and map the properties defined on the Audit entity to each table defined for derived entity. In addition we also need to ensure that base entity Audit is marked as abstract and does not belong to any entityset.

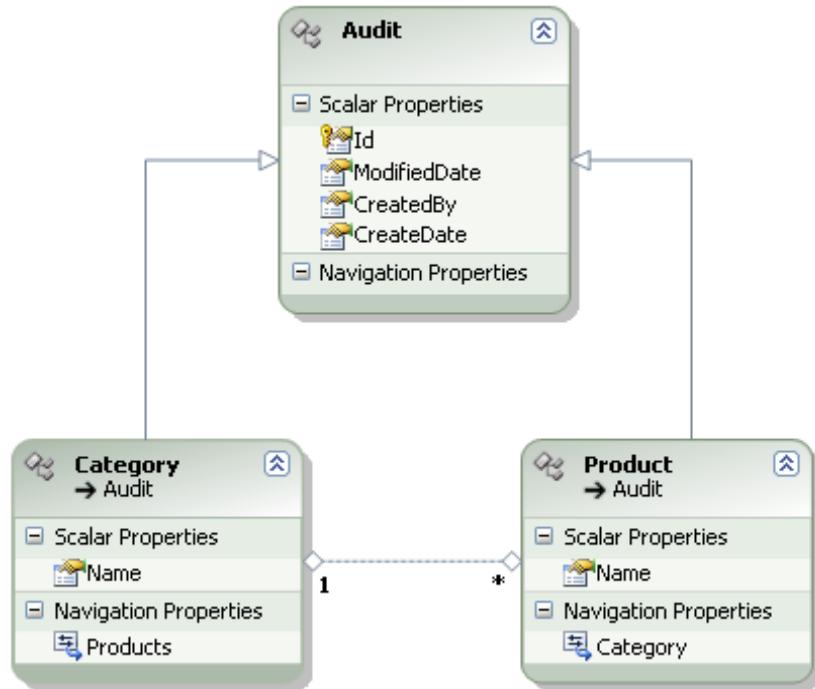
Discussion: The implementation that we will apply to share audit fields across all entities defined on our model can be mapped closely to how we would model Table per Concrete Type. In TPC each tables having a common set of columns are defined at the top level entity. Additional columns are declared on the derived entity. One unique feature of TPC is that identity columns in TPC cannot overlap each other because you define TPC under a single entityset. This enforces a requirement that tables participating in TPC must not contain entity key values that are same in both tables. Since Audit table would be shared across all the tables in the database, there is a very likely chance that two table will have the same primary key? To ensure that we do not end up with a runtime exception, we will have each entity resides in its own entityset and base entity audit will be set as abstract having no entityset.

Steps below outline the process of mapping audit fields common in all tables defined on the base entity and mapped to each table in the database.

1. Import Category and Products table using entity Data Model Wizard.
Figure below shows the model after finishing the wizard.



2. To share audit fields between Category and Products entity, create a new entity Audit and move CreateDate, CreatedBy and ModifiedDate properties to the new entity. Since Audit entity must have an entity key, create an Id property and make it the entity key. This key will be used by Product and Category entity and therefore remove ProductId and CategoryId entity key defined on Category and Products.
3. Change Product and Category entity so that they inherit from Audit entity leveraging audit fields as well as EntityKey Id. Also make Audit entity abstract because there will be no mapping for the base entity. Figure below shows the updated model after completing the above steps.



Now that we have configured our conceptual model, we need to map the model to our storage model. EF designer does not support mapping TPC inheritance that will reside in different entitysets; therefore we need to modify the MSL in xml.

4. When we created inheritance structure, EF assumed that both Category and Products will belong to same entityset which is the AuditSet. However in our case to ensure that there is no primary key overlap each derived entities, create two entitysets Categories and Products and remove AuditEntitySet created by the designer. Code below shows the two entitysets defined on the conceptual model.

```
<EntityType Name="Categories" EntityType="AuditTPCModel.Category" />
    <EntityType Name="Products"
        EntityType="AuditTPCModel.Product" />
```

Change the Associationset's role to reflect the new entitysets created above. Code below shows the updated AssociationSet for FK_Products_Category

```
<AssociationSet Name="FK_Products_Category"
Association="AuditTPCModel.FK_Products_Category">
    <End Role="Category" EntitySet="Categories" />
    <End Role="Products" EntitySet="Products" />
</AssociationSet>
```

5. The mapping created by the designer for Category and Products are incomplete because it does not include audit properties that we moved to base entity Audit. Add audit field mappings to each derived entity defined on the mapping. Also make sure that both categories and Products are mapped to a different entityset which is not the configured mapping done by the designer. Code below shows the correct mapping for Category and Product entity.

```
<EntityTypeMapping Name="Categories">
    <EntityTypeMapping
TypeName="IsTypeOf(AuditTPCModel.Category)">
        <MappingFragment StoreEntitySet="Category">
            <ScalarProperty Name="Name" ColumnName="Name" />
            <ScalarProperty Name="Id" ColumnName="CategoryId" />
            <ScalarProperty Name="CreateDate" ColumnName="CreateDate" />
        </MappingFragment>
    </EntityTypeMapping>
</EntityTypeMapping>
```

```

        <ScalarProperty Name="CreatedBy" />
        <ScalarProperty Name="ModifiedDate" />
    </MappingFragment>
</EntityTypeMapping>
</EntitySetMapping>
<EntitySetMapping Name="Products">
    <EntityTypeMapping
        TypeName="IsTypeOf(AuditTPCModel.Product)">
        <MappingFragment StoreEntitySet="Products">
            <ScalarProperty Name="Name" />
            <ScalarProperty Name="Id" />
            <ScalarProperty Name="CreateDate" />
            <ScalarProperty Name="CreatedBy" />
            <ScalarProperty Name="ModifiedBy" />
            <ScalarProperty Name="ModifiedDate" />
        </MappingFragment>
    </EntityTypeMapping>
</EntitySetMapping>

```

Since the designer does not fully support this feature, it also missed the mapping for the association between Categories and Products defined on the conceptual model. Code below shows the mapping for FK_Products_Category Association defined on the conceptual model.

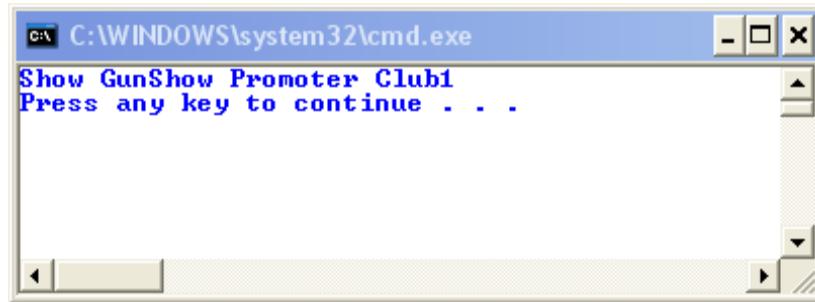
```

<AssociationSetMapping Name="FK_Products_Category"
    TypeName="AuditTPCModel.FK_Products_Category" StoreEntitySet="Products">
    <EndProperty Name="Category">
        <ScalarProperty Name="Id" ColumnName="CategoryId" />
    </EndProperty>
    <EndProperty Name="Products">
        <ScalarProperty Name="Id" ColumnName="ProductId" />
    </EndProperty>
</AssociationSetMapping>

```

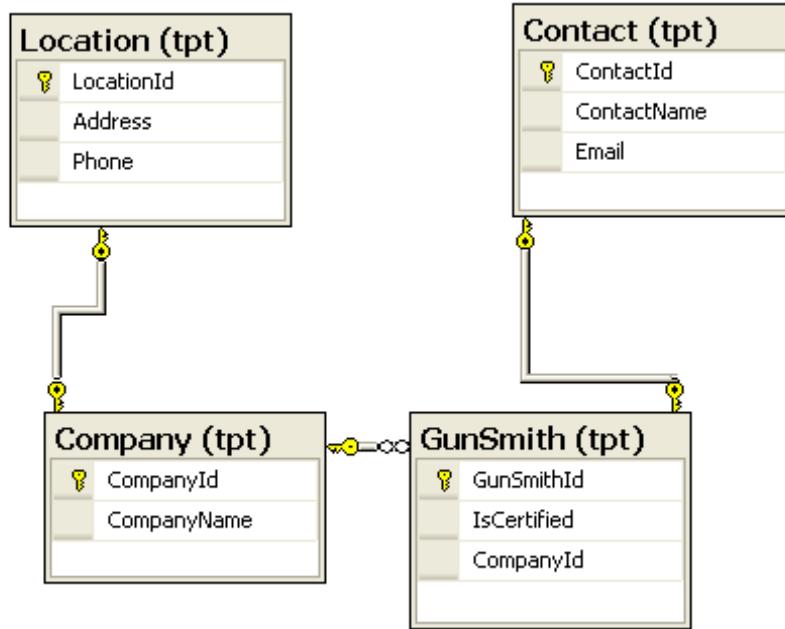
6. By defining the audit fields on the base entity, both entities can leverage the auditing behavior and do not require duplicating properties on every entity defined on the model. Hopefully future versions of EF will provide support to map this architecture using designer. Code below creates an instance of Category and Product entity and also defines values for audit fields declared on the base Audit entity. Then product entity is added to the category's product collection and both entities are saved to the database. To confirm the save went

successfully using second data context, I am retrieving the product and its category and displaying the result to output window.



5.1.17 Creating Association between Two Table Per Type entities

Problem: Figure below shows the database diagram for Gunsmith and its contact information.



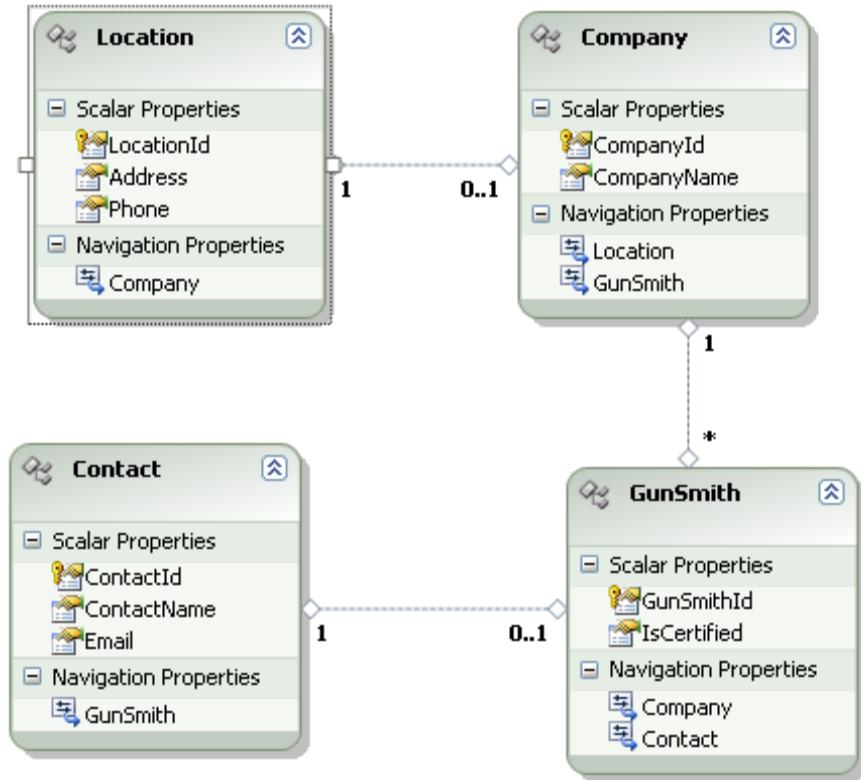
On the above table relationship, GunSmith extends Contact table with IsCertified field. This means that ContactId is generated in Contact table

using identity column and the same value for the contacted is used as the primary key for GunSmith table. A Gunsmith is associated with a company using companyid column as foreign key to Company table. Company table also extends a Location table which contains address for the company. You want to model the above table relationship using two tables per types; one with GunSmith extending contact entity and Company extending location entity. In addition you want to have a one to many relationships between Company and GunSmith.

Solution: The above table relationship can be represented using two Table per Type inheritance. Import the above tables using the import wizard. Remove all the associations created by the designer. Make GunSmith entity derive from Contact entity and make Company entity derive from Location entity. Remove GunSmithId from GunSmith entity and CompanyId from Company entity. Create 1 to many association between company and GunSmith entity where GunSmith is the many side of the association. Validate the model and ensure that there are no build errors.

Discussion: Steps below outline the process of importing the above table structure using Table Per Type inheritance.

1. Import the above table structure using Entity Data Wizard. Figure below shows the model created by the wizard.

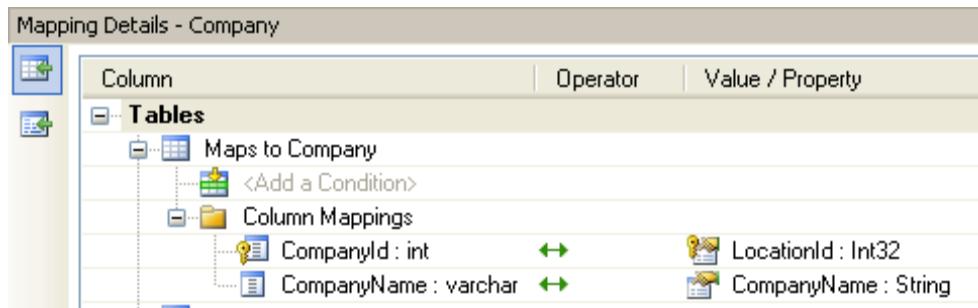


2. Remove association between Location and Company entity and remove association between GunSmith and Contact entity.
3. Make GunSmith entity derive from Contact and delete GunSmithId property from GunSmith entity as ContactId will serve as the entity key. On GunSmith mapping, set GunSmithId column to map to ContactId property available from Contact entity. Figure below shows the mapping for GunSmith entity.

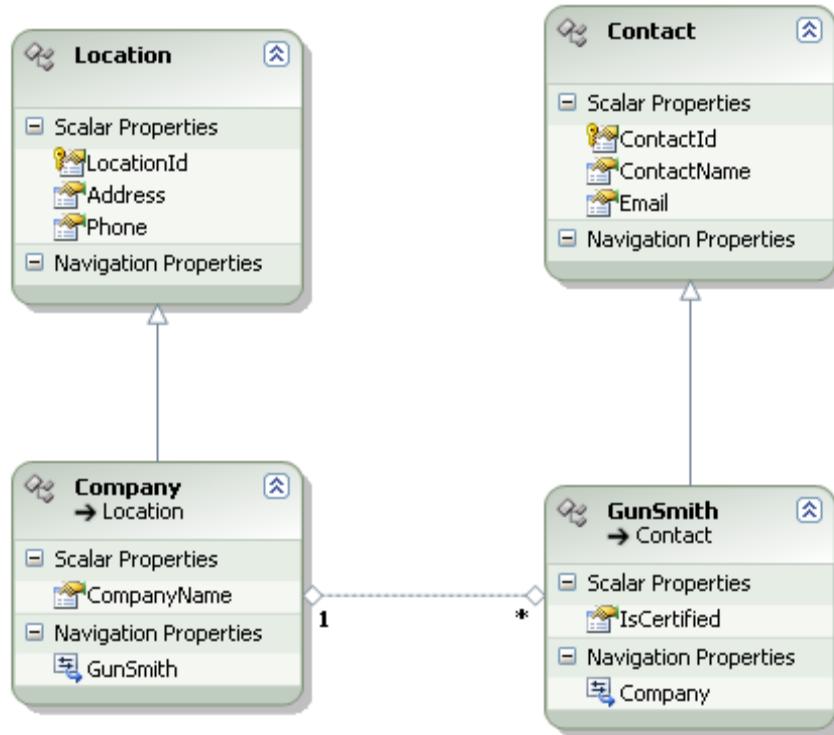
Mapping Details - GunSmith

Column	Operator	Value / Property
Tables		
Maps to GunSmith		<Add a Condition>
Column Mappings		
GunSmithId : int	↔	ContactId : Int32
IsCertified : bit	↔	IsCertified : Boolean
CompanyId : int	↔	

4. Make Company entity derive from Location and delete CompanyId from Company entity as LocationId will serve as the entity key. On Company mapping, set Companyid column to map to LocationId property inherited from Location entity. Figure below shows the mapping for Company entity.



5. Because of inheritance, the mapping information between Company and Gunsmith got lost. To recreate the mapping select the association link and configure the LocationId to map to CompanyId column and ContactId property to map to GunSmithId column. Validate the model and make sure the model can build with no errors. Figure below shows the updated entity model with correct configurations.



6. To test the model we can create an instance of GunSmith and Company entity and assign the company entity to GunSmith and save both entities to the database. Code below accomplishes the above objective.

```

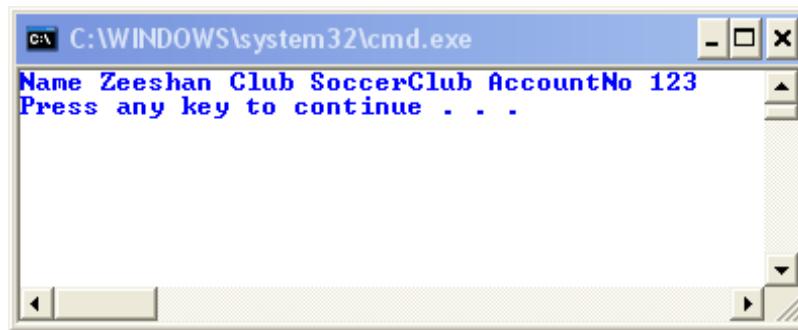
var db = new TwoTPTEntities();
    var company = new Company { Address = "123 Happy St", CompanyName
= "Widgets", Phone = "918-998-9956" };
    var gunsmith = new GunSmith
    {
        ContactName = "Zeeshan",
        Email = "abc@gmail.com",
        IsCertified = true,
        Company = company
    };
    db.AddToContact(gunsmith);
    db.SaveChanges();

    var db2 = new TwoTPTEntities();
    var gsmith =
db2.Contact.OfType<GunSmith>().Include("Company").First();
    Console.WriteLine("Contact {0} Company
{1}", gsmith.ContactName, gsmith.Company.CompanyName);

```

In the above code, to confirm that Gunsmith can be retrieved after saving it, I am using OfType operator to retrieve GunSmith. It is after using the OfType operator I am calling Include to also retrieve the

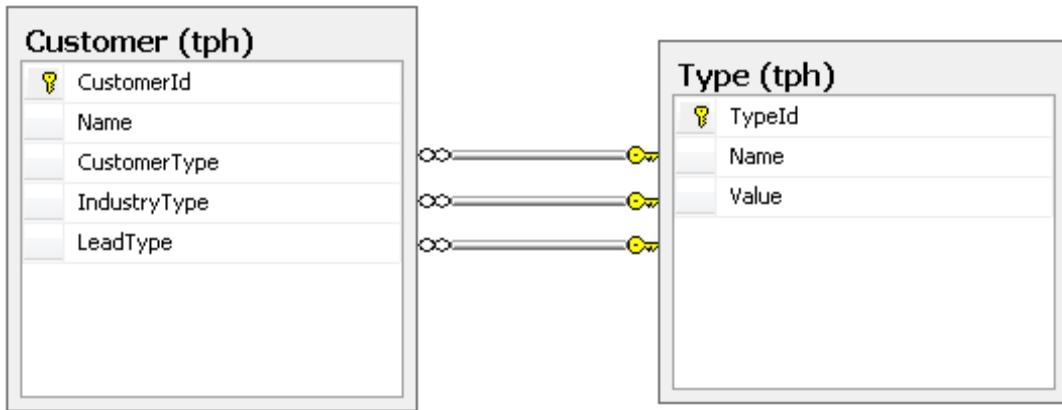
company for the gunsmith. It is essential to apply company include after the OfType operator because Company entity is only available on the derived type GunSmith not on Contact entity. Figure below shows the output written to the console window.



```
C:\WINDOWS\system32\cmd.exe
Name Zeeshan Club SoccerClub AccountNo 123
Press any key to continue . . .
```

5.1.18 Creating Associations on Derived Entities using Table per Hierarchy

Problem: Figure below shows the database diagram between Customer and a Lookup table Type.



In the above diagram, Type table contains different types such CustomerType, IndustryType and LeadType. Figure below shows the data in Type table.

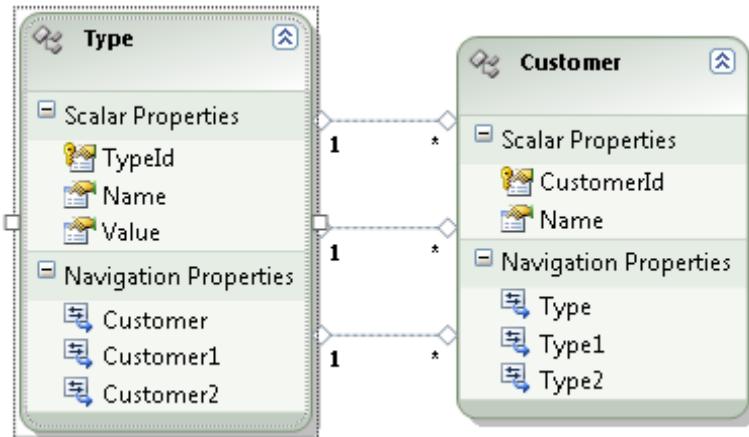
	TypeId	Name	Value
1	1	CustomerType	Advertiser
2	2	CustomerType	Agency
3	3	IndustryType	Education
4	4	IndustryType	Finance
5	5	LeadSource	Internet
6	6	LeadSource	TV

Instead of having each type stored in a different table, All Types about a Customer are stored in one Table and differentiated based on Name column. Like CustomerType could have two different values Advertiser or an Agency. For each Type defined on Type table, Customer has the foreign key associated to it. For a given Customer we are capturing what are the CustomerType, IndustryType and LeadType. We want to import the above table structure as Table Per Hierarchy for Type table with three entities deriving from Type table; CustomerType, IndustryType and LeadType. Each derived table should have association to the Customer entity.

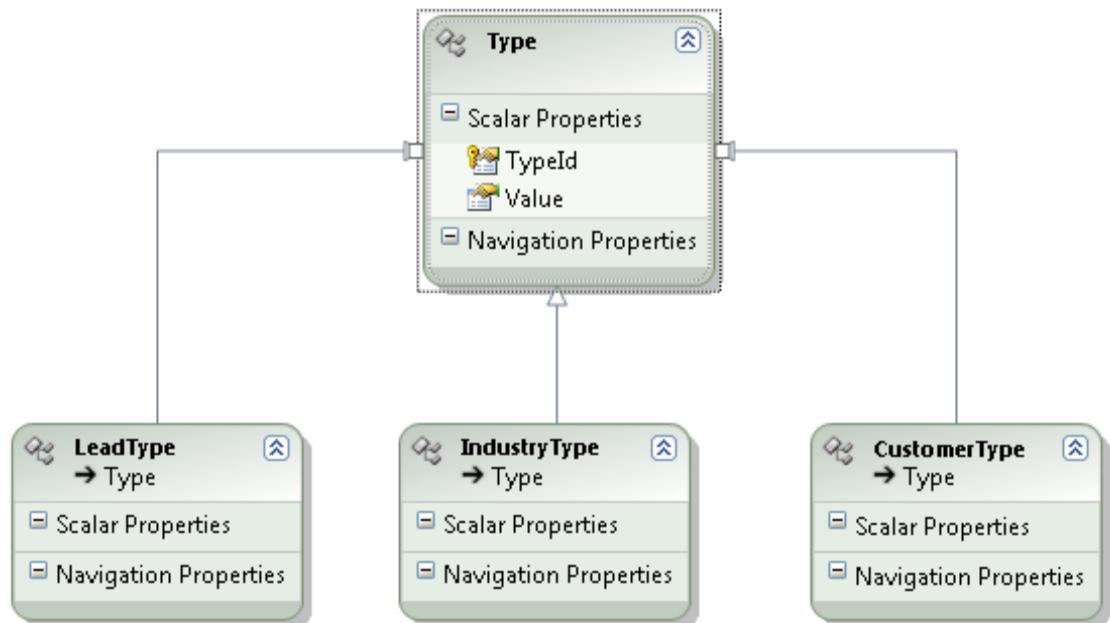
Solution: Import the above table using EDM wizard. Remove all associations created by the designer. Create three entities; CustomerType, LeadType and IndustryType and make sure they derive from Type entity. Make Type entity as abstract because we will not be instantiating it from code. Create three associations from Customer going to CustomerType, IndustryType and LeadType where Customer has a Multiplicity of 1 on each association.

Discussion: Instead of creating three derived entities we could have created three associations directly to Type table. However in that case, a developer could assign a wrong type to any of the given types we are capturing for Customer entity. For instance it would be easy to assign CustomerType to have a value from IndustryType which would be incorrect. By having three derived entities we are narrowing the values that could be assigned to each Type in Customer entity. Steps below outline the process of importing the above table structure using Table per Hierarchy.

1. Import Type and Customer table using EDM wizard. Figure below shows the model wizard has generated.



- Remove all the associations between Customer and Type entity. Create three entities CustomerType, LeadType and IndustryType. Make sure all entities derive from Type entity and make Type entity as abstract. Since we will use the Name column as a condition for each derived entity, it cannot be mapped to a property. Remove the Name column from base entity Type. Figure below shows the updated model.



- Apply condition to LeadType entity where Name equal to LeadType. Figure below shows the condition on the mapping window.

Column	Operator	Value / Property
Tables		
Maps to Type		
When Name	=	LeadSource
<Add a Condition>		
Column Mappings		
Name : varchar	↔	

4. Apply condition to IndustryType entity where Name equal to IndustryType. Figure below shows the condition on the mapping window.

Column	Operator	Value / Property
Tables		
Maps to Type		
When Name	=	IndustryType
<Add a Condition>		
Column Mappings		
Name : varchar	↔	

5. Apply condition to CustomerType entity where Name equal to CustomerType. Figure below shows the condition on the mapping window.

Column	Operator	Value / Property
Tables		
Maps to Type		
When Name	=	CustomerType
<Add a Condition>		
Column Mappings		
Name : varchar	↔	

6. Each foreign key column on Customer entity needs to be associated to a derived entity for base entity Type. Create an association between LeadType and Customer where Customer has a multiplicity of many and LeadType has a multiplicity of 1. Map the association to Customer table where CustomerId maps to CustomerId column and TypeId on

LeadType entity maps to LeadType column on Customer table. Figure below shows the mapping.

Mapping Details - CustomerLeadType		
Property	Operator	Column
Association		
Maps to Customer		
Customer		
CustomerId : Int32	↔	CustomerId : int
LeadType		
TypeId : Int32	↔	LeadType : int

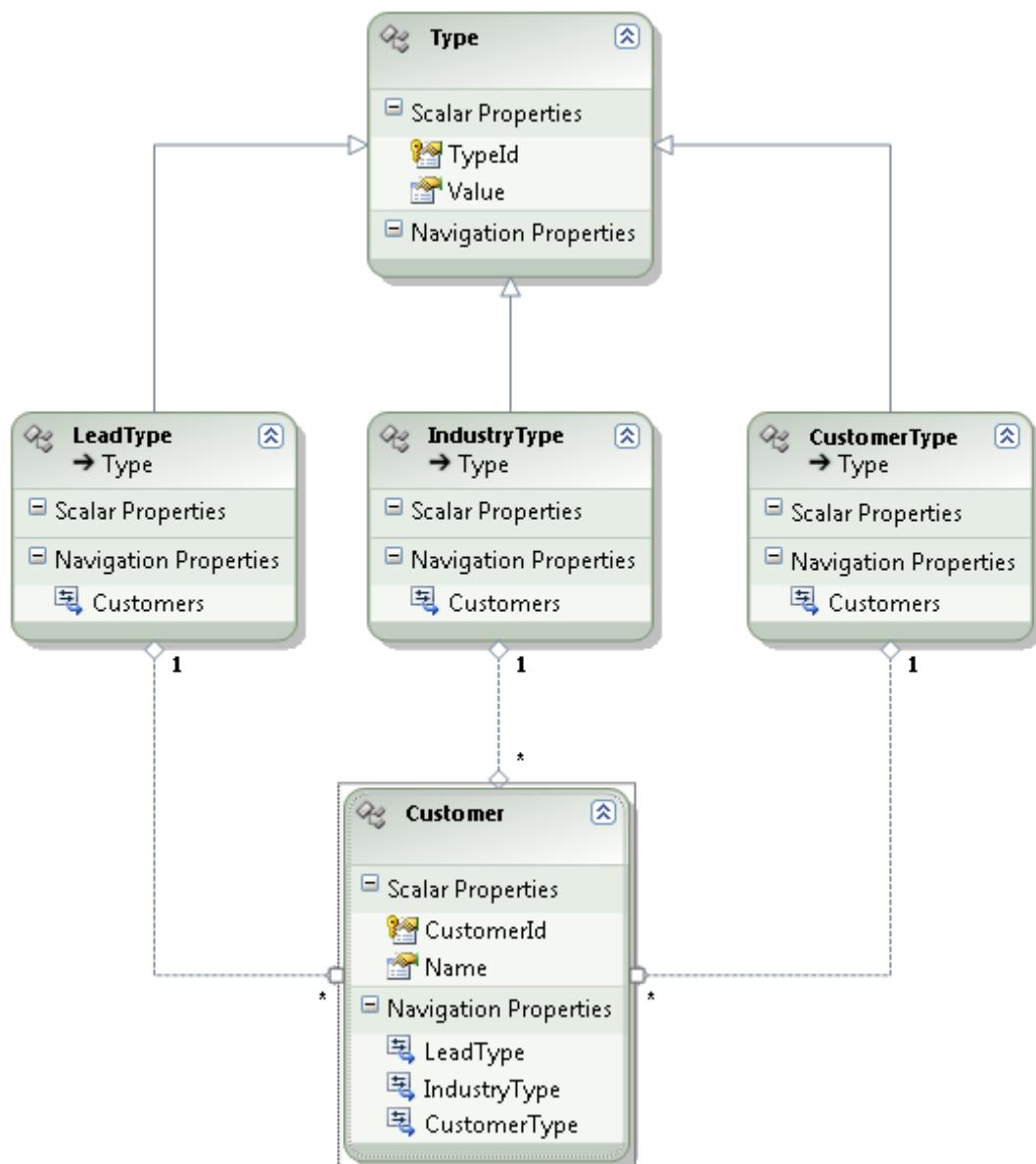
7. Create an association between IndustryType and Customer where Customer has a multiplicity of many and IndustryType has a multiplicity of 1. Map the association to Customer table where CustomerId maps to CustomerId column and TypeId on IndustryType entity maps to IndustryType column on Customer table. Figure below shows the mapping.

Mapping Details - IndustryTypeCustomer		
Property	Operator	Column
Association		
Maps to Customer		
IndustryType		
TypeId : Int32	↔	IndustryType : int
Customer		
CustomerId : Int32	↔	CustomerId : int

8. Create an association between CustomerType and Customer where Customer has a multiplicity of many and CustomerType has a multiplicity of 1. Map the association to Customer table where CustomerId maps to CustomerId column and TypeId on CustomerType entity maps to CustomerType column on Customer table. Figure below shows the mapping.

Mapping Details - CustomerTypeCustomer		
	Property	Operator
	Column	
Association		
Maps to Customer		
CustomerType		
TypeId : Int32	↔	CustomerType : int
Customer		
CustomerId : Int32	↔	CustomerId : int

Figure below shows the completed entity data model.



9. To test the above model, I have written some code below that creates an instance of Customer entity and assigns CustomerType, LeadType and IndustryType and saves the Customer entity to the database. Using second datacontext, I am retrieving the entity and printing Customer info to the console window.

```

var db = new TPHAEntities();
    var customertype =
db.Types.OfType<CustomerType>().First(t => t.Value ==
"Advertiser");
    var leadtype = db.Types.OfType<LeadType>().First(t
=> t.Value == "Internet");
    var industrytype =
db.Types.OfType<IndustryType>().First(t => t.Value ==
"Education");

    var customer = new Customer
{
    Name= "Zeeshan",
    CustomerType = customertype,
    LeadType = leadtype,
    IndustryType = industrytype
};
db.AddToCustomers(customer);
db.SaveChanges();

var db2 = new TPHAEntities();
var cust =
db2.Customers

```

```

.Include( "CustomerType" ).Include( "LeadType" ).Include( "IndustryTy
pe" )
    .First(c => c.Name == "Zeeshan");

```

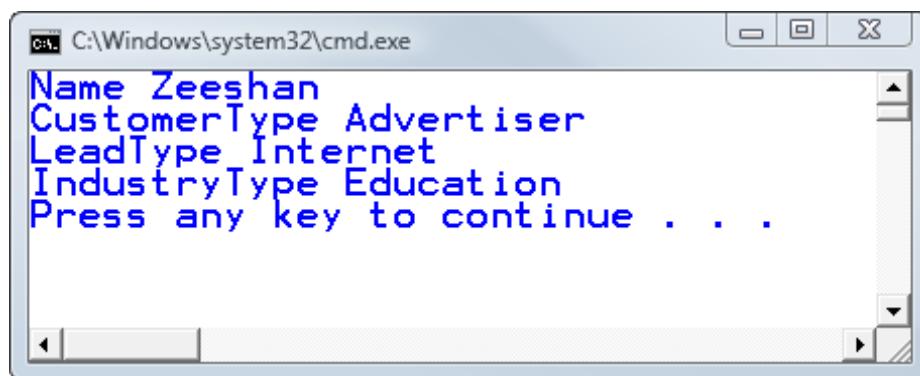
```

Console.WriteLine( "Name {0}" , cust.Name );
Console.WriteLine( "CustomerType {0}" ,
cust.CustomerType.Value );
Console.WriteLine( "LeadType {0}" ,
cust.LeadType.Value );
Console.WriteLine( "IndustryType {0}" ,
cust.IndustryType.Value );

```

Few things worth mentioning about the above code is, for retrieving CustomerType, I am using the ofType operator to get an instance of

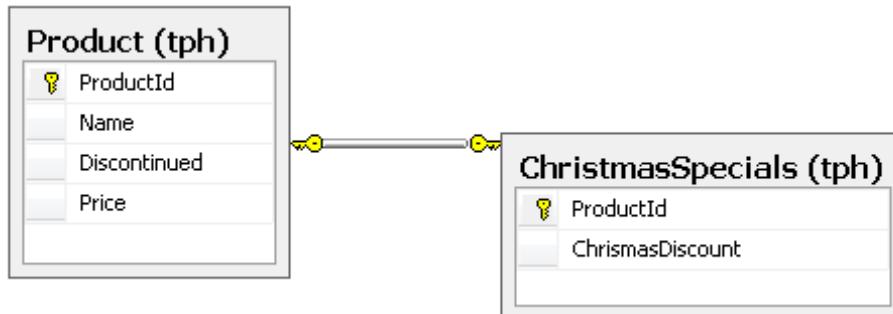
CustomerType derived entity. If I did not do that, then I would have to cast the base entity Type to CustomerType. The cast would have been required because Customer.CustomerType will only allow the derived type instance of CustomerType instead of allowing base type reference. On retrieving Customer, I am eagerly loading related references for Customer such as CustomerType, LeadType and IndustryType. Figure below shows the result on the console window.



```
C:\Windows\system32\cmd.exe
Name Zeeshan
CustomerType Advertiser
LeadType Internet
IndustryType Education
Press any key to continue . . .
```

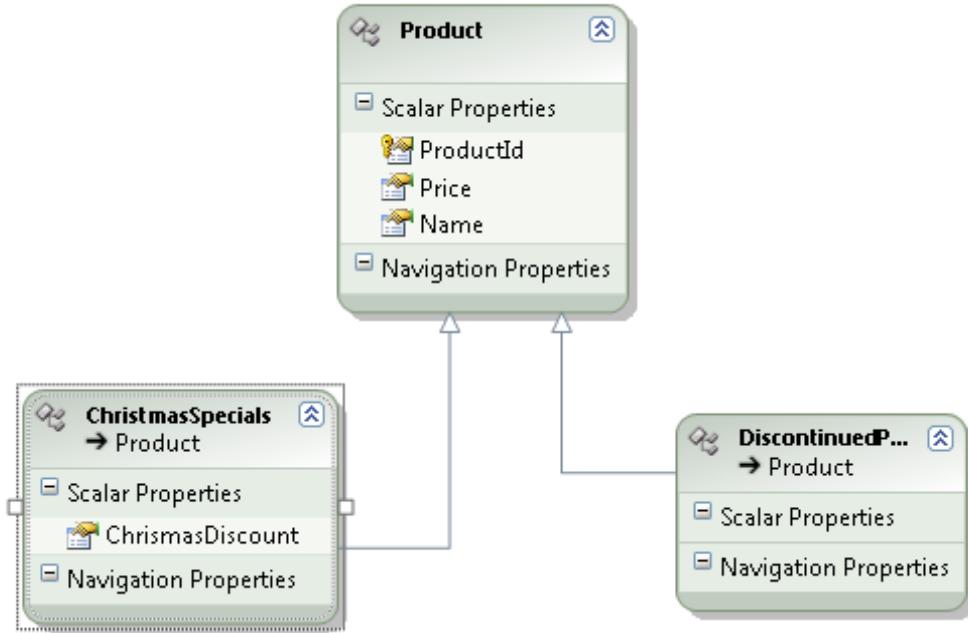
5.1.19 Table per Hierarchy and Table per Type Hybrid

Problem: Figure below shows the table structure and the relation between product and ChrismasSpecial table.



You want to import the above table as inheritance hierarchy into EDM. The base class Product should contain two derived classes; DiscontinuedProduct and ChrismasSpeical. DiscontinuedProduct is identified by discontinued column in the Product table and ChristmasSpecial product are products that have rows present in the ChrismasSpecial table as which indicates the

discount customer will receive on those products. The final EDM model should look as below.

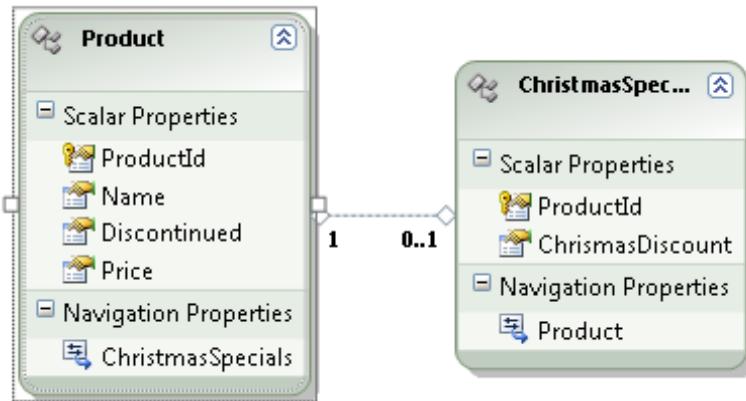


Solution: A hybrid approach where the same table **Product** is used as Table per Hierarchy and Table per Type is not directly supported by EF designer. We can model the conceptual model using the designer, however mapping between the conceptual and storage model cannot be fully accomplished using the designer. To configure the mapping, **DiscontinuedProduct** needs to be mapped to product table where discontinued column has a value of true. For product entity, the condition for discontinued column has to be set to false. **ChristmasSpeical** entity also must have discontinued value of false in addition to product entity. This is not supported in the designer and has to done manually in the msl by adding multiple types inside the product mapping. Code below shows the manual change required in the designer.

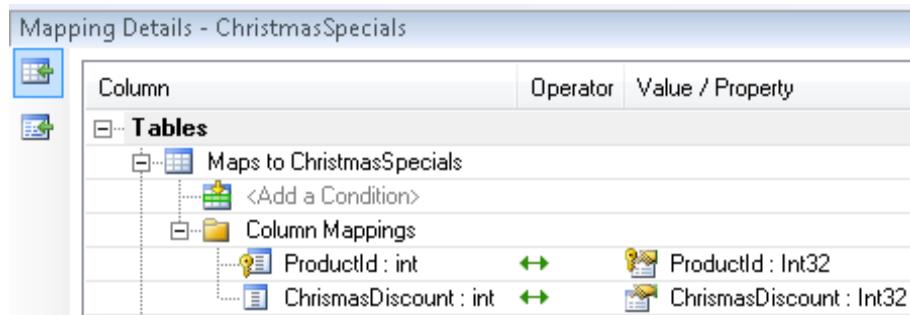
```
<EntityTypeMapping  
TypeName="ProductsHierarchyModel.Product; IsTypeOf(ProductsHierarchyModel.ChristmasSpecials)">
```

Discussion: Steps below outline the process of importing the above table structure using TPH and TPT to create Product base entity and two derived entities; ChrismasSpecial and DiscontinuedProduct.

1. Import Product and ChristmasSpecial table into edm using EDM import wizard. Figure below shows the model created by the wizard.



2. Since we will be using inheritance to model the above EDM relationship, remove the association created between Product and ChristmasSpecial entity.
3. Make ChristmasSpecial entity derive from Product entity. Remove ProductId property from ChristmasSpecial entity since ProductId from Product entity will be used as the entity key. Configure the table mapping for ChristmasSpecial where ProductId column maps to ProductId property on Product entity. Figure below shows the mapping for ChristmasSpecial entity.



4. Create DiscontinuedProduct entity deriving from Product. Configure the DiscontinuedProduct entity to map to Product table where

Discontinued equal to true. Since Product entity is the base entity and it should contain all the products, set condition on the product entity where discontinued equal to false. This would ensure that when we request for products, we not only get discontinued Product but also the active products. Also removed discontinued column because discontinued column is used as a discriminator value.

Code below shows the mapping created so far by the designer.

```

<EntitySetMapping Name="Product">
    <EntityTypeMapping TypeName="ProductsHierarchyModel.Product">
        <MappingFragment StoreEntitySet="Product">
            <ScalarProperty Name="ProductId"
ColumnNames="ProductId" />
            <ScalarProperty Name="Discontinued"
ColumnNames="Discontinued" />
            <ScalarProperty Name="Name" ColumnName="Name" />
            <ScalarProperty Name="Price" ColumnName="Price" />
            <Condition ColumnName="Discontinued" Value="false" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="IsTypeOf(ProductsHierarchyModel.ChristmasSpecials)">
        <MappingFragment StoreEntitySet="ChristmasSpecials">
            <ScalarProperty Name="ProductId" ColumnName="ProductId" />
            <ScalarProperty Name="ChrismasDiscount"
ColumnName="ChrismasDiscount" />
        </MappingFragment></EntityTypeMapping>
    <EntityTypeMapping TypeName="IsTypeOf(ProductsHierarchyModel.DiscontinuedProduct)">
        <MappingFragment StoreEntitySet="Product" >
            <ScalarProperty Name="Discontinued" ColumnName="Discontinued" />
            <ScalarProperty Name="Name" ColumnName="Name" />
            <ScalarProperty Name="Price" ColumnName="Price" />
            <ScalarProperty Name="ProductId" ColumnName="ProductId" />
            <Condition ColumnName="Discontinued" Value="true" />
        </MappingFragment>
    </EntityTypeMapping>
</EntitySetMapping>

```

So far we have not configure the mapping completely because DiscontinuedProduct entity has Discontinued column set to true, Product entity has discontinued set to false but we have not specified what should the value for discontinued column be for Christmas product? After all ChristmasSpecial entity must specify a value for discontinued column because the column is marked as not null. This is

where designer falls short because there is no way through the designer to tell that both Product and ChristmasSpecial entity should have discontinued set to false. Code below shows the correct mapping for the above model.

```

<EntityTypeMapping
TypeName="ProductsHierarchyModel.Product;IsTypeOf(ProductsHierarchyModel.ChristmasSpecials)">
    <MappingFragment StoreEntitySet="Product">
        <ScalarProperty Name="ProductId"
ColumnName="ProductId" />
        <ScalarProperty Name="Name" ColumnName="Name" />
        <ScalarProperty Name="Price" ColumnName="Price" />
        <Condition ColumnName="Discontinued" Value="false" />
    </MappingFragment>
</EntityTypeMapping>
<EntityTypeMapping
TypeName="IsTypeOf(ProductsHierarchyModel.ChristmasSpecials)">
    <MappingFragment StoreEntitySet="ChristmasSpecials">
        <ScalarProperty Name="ProductId" ColumnName="ProductId" />
        <ScalarProperty Name="ChrismasDiscount"
ColumnName="ChrismasDiscount" />
    </MappingFragment>
</EntityTypeMapping>
<EntityTypeMapping
TypeName="IsTypeOf(ProductsHierarchyModel.DiscontinuedProduct)">
    <MappingFragment StoreEntitySet="Product" >
        <ScalarProperty Name="Name" ColumnName="Name" />
        <ScalarProperty Name="Price" ColumnName="Price" />
        <ScalarProperty Name="ProductId" ColumnName="ProductId" />
        <Condition ColumnName="Discontinued" Value="true" />
    </MappingFragment>
</EntityTypeMapping>

```

On the above code the only change made to the mapping generated by the designer is the TypeName for Product entity. The mapping for Product entity is changed to both Product and ChristmasSpecial because they both have the same condition of false for discontinued column. The line below shows the only change made to the mapping.

```

<EntityTypeMapping
TypeName="ProductsHierarchyModel.Product;IsTypeOf(ProductsHierarchyModel.ChristmasSpecials)">
```

To test the model created so far, we can create instance of product, DiscontinuedProduct and ChristmasSpecial product entity and save the entities to the database. Code below shows an example.

```

var db = new ProductsHierarchyEntities();
    var discontinued = new DiscontinuedProduct { Name =
"Discontinued", Price = 50 };
    var product = new Product { Name = "T shirt", Price = 30 };
    var christmasspecial = new ChristmasSpecials { Name = "Chrimas
Tree", Price = 55, ChrismasDiscount = 20 };
    db.AddToProduct(discontinued);
    db.AddToProduct(product);
    db.AddToProduct(christmasspecial);
    db.SaveChanges();

```

To confirm the results were inserted correctly with correct value for discontinued column and also inserted into ChristmasSpecial table, I have run the select statement on Product and ChristmasSpecial table. Figure below shows the output.

	ProductId	Name	Discontinued	Price
1	1	Discontinued	1	50
2	2	T shirt	0	30
3	3	Chrimas Tree	0	55

	ProductId		ChrismasDiscount	
1	3		20	

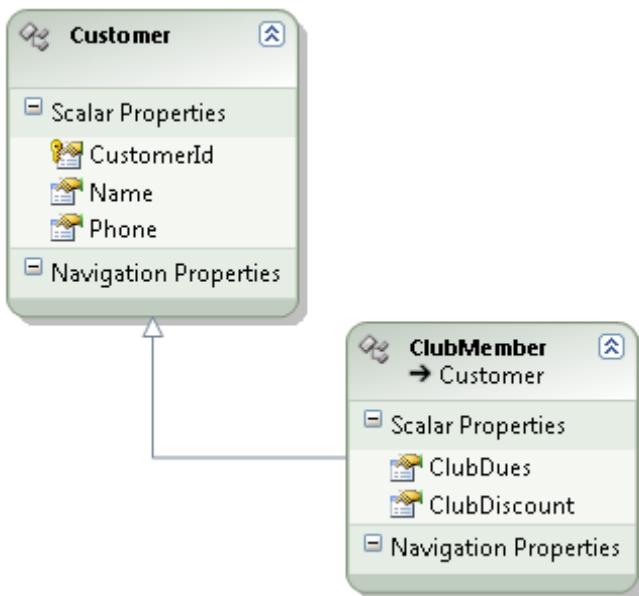
Both active and ChristmasSpecial product has a value of 0 for discontinued column where as discontinued product has a value of 1. In addition for ChristmasSpecial product, there is also an entry written inside ChristmasSpecial table containing the discount for the product.

5.1.20 Using multiple conditions for Table per Hierarchy

Problem: Figure below shows the Customer table containing customer information and additional club information if the customer has a club membership.

Customer (inh)			
	Column Name	Data Type	Allow Nulls
PK	CustomerId	int	<input type="checkbox"/>
	Name	varchar(50)	<input type="checkbox"/>
	Phone	varchar(50)	<input type="checkbox"/>
	ClubDues	int	<input checked="" type="checkbox"/>
	ClubDiscount	int	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

The above table needs to be mapped to EDM model with Customer as the base entity and ClubMember as the derived entity. If ClubDues and ClubDiscount are not null, then the customer is a club member. The completed edm model should look as below.



Solution:

Discussion: Steps below outline the process of importing the customer table into edm as customer and club member hierarchy.

1. Import customer table using EDM wizard. Add Club member entity to the model deriving from Customer entity.
2. Move ClubDues and ClubDiscount properties from customer entity to ClubMember entity. When the designer creates these two properties,

they are marked as allow null but for ClubMember, the properties cannot be null. Change both properties nullable attribute to false.

3. Set condition for customer entity mapping where ClubDues and ClubDiscount column are null. Figure below shows the mapping for customer entity.

Column	Operator	Value / Property
Tables		
Maps to Customer		
When ClubDiscount	Is	Null
And ClubDues	Is	Null
<Add a Condition>		
Column Mappings		
CustomerId : int	↔	CustomerId : Int32
Name : varchar	↔	Name : String
Phone : varchar	↔	Phone : String
ClubDues : int	↔	
ClubDiscount : int	↔	

4. Map ClubMember entity to customer table where condition on ClubDues and ClubDiscount is not null and map ClubDues and ClubDiscount properties to columns on the Customer table. Figure below show the mapping configured for the ClubMember entity.

Column	Operator	Value / Property
Tables		
Maps to Customer		
When ClubDiscount	Is	Not Null
And ClubDues	Is	Not Null
<Add a Condition>		
Column Mappings		
ClubDues : int	↔	ClubDues : Int32
ClubDiscount : int	↔	ClubDiscount : Int32

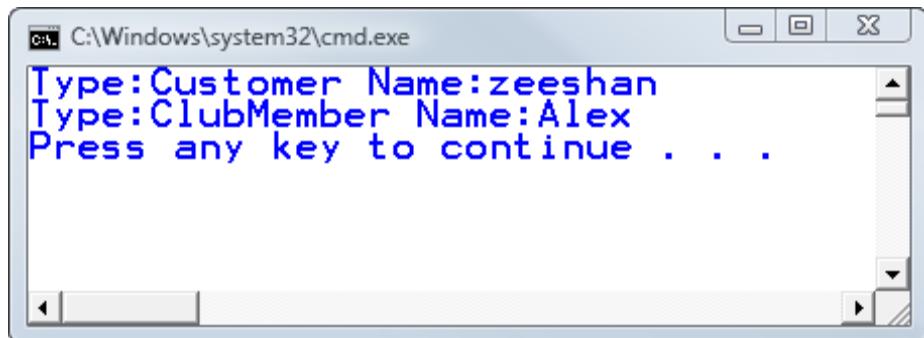
To test the model created above, we can create an instance of customer and club member and save both entities to the database. Using second

datacontext, we can query for both customers and confirm that both entities were correctly saved. Code below shows the example.

```
var db = new MultipleConditionEntities();
    var customer = new Customer { Name = "zeeshan", Phone = "717-888-9191" };
    var clubmember = new ClubMember { Name = "Alex", Phone = "712-111-4545" };
    db.AddToCustomer(customer);
    db.AddToCustomer(clubmember);
    db.SaveChanges();

    var db2 = new MultipleConditionEntities();
    foreach (var cust in db2.Customer)
    {
        Console.WriteLine("Type:{0}
Name:{1}", cust.GetType().Name, cust.Name);
    }
```

The above code creates customer and club member entity, saves them to the database. Using the second data context, I am retrieving customers and printing the type of the customer and their name. Figure below shows the result on the console window.

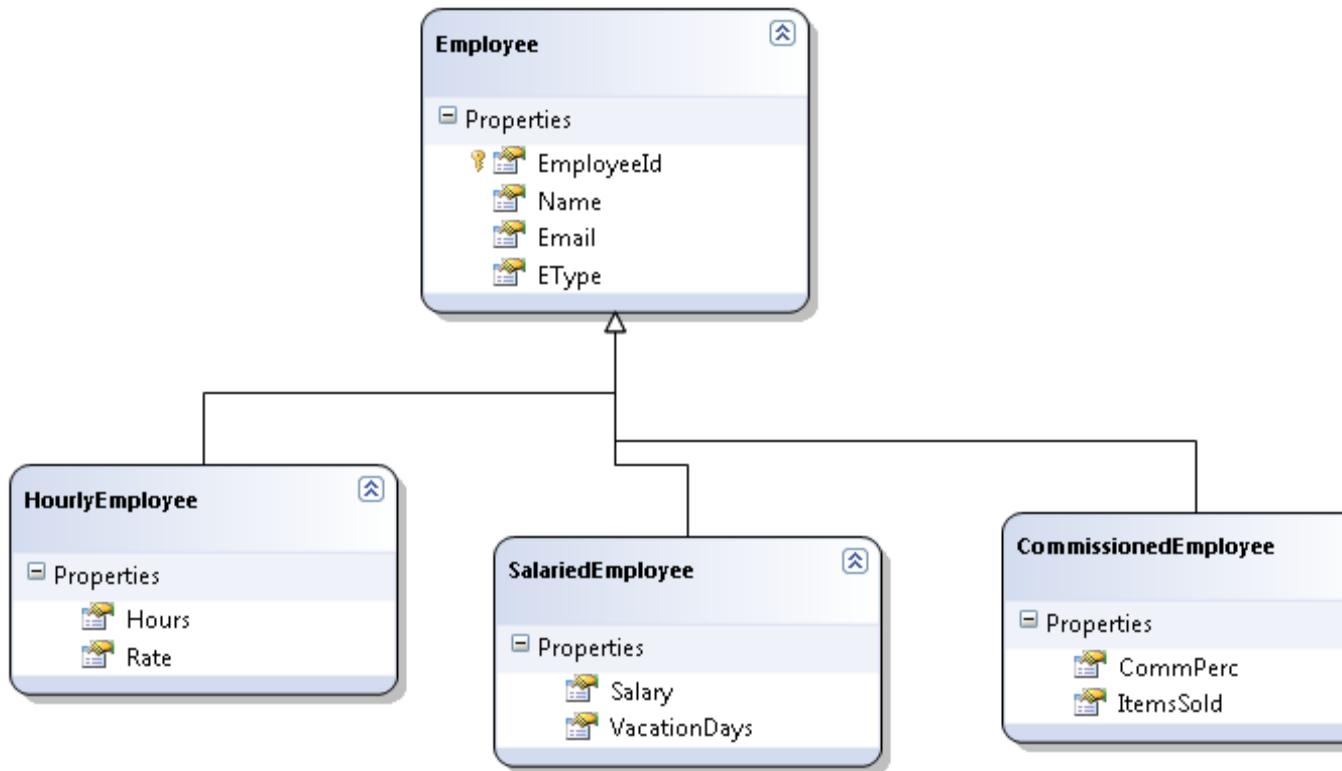


5.2 Linq To Sql

5.2.1 Table per type inheritance using Linq to Sql

Problem: You have implemented inheritance in the database using Table per Type. You want to know how many tables in your database using inheritance feature available in Linq To Sql.

Solution: Linq to Sql only support Table Per Hierarchy model where all the classes are mapped to one table in the database. Each row in the table is differentiated by Type column also called as discriminator column. Another form of inheritance that is used often in the database is Table Per Type model. In Table Per Type, each class is mapped to a separate table in the database. Each table's primary key column is also the foreign key column for a base table that contains common attributes belonging to all tables. All primary keys for the child tables are basically generated in the the base table and used as foreign key and primary key in the child tables. The advantage of using Table Per Type to Table Per Hierarchy is, you do not have to create one big gigantic table which contains fields for all concrete types and allow null on the columns because other concrete types do not have any meaning for those columns. Table Per Type approach helps reduced disk usage by not allowing null values and keeps the data integrity intact. Since Table Per Type is not supported directly by linq to sql, you will have to resort to a view to give a perception to linq to sql that you are actually using Table Per Hierarchy and for each type we will expose a Type attribute that will help linq to sql in identity which table to map to what concrete types. For read scenarios, the view options works great but what happens when you need to write the inheritance model to the database. You can't because we did not target tables, we mapped view to entities and when linq to sql applies updates and inserts to the view, sql server would complain that view is not updatable. One option you can use to get around this is use instead of Trigger that will intercept the row attempting to be inserted and map it to the appropriate table. Another option is to override the Crud process in Linq To Sql and perform your crud operations manually. To get started you need to create a view that exposes the data from all different tables and map that to linq to sql entity classes. Next you need override insert, update and delete process by registering with the partial methods exposed by DataContext for each class. Code below shows how we setup our hierarchy to map our view to entities defined. We are also customizing the insert process by overriding the insert process.



Discussion

As we discussed earlier, Linq to Sql does not support Table Per Type Inheritance, so we have to create a view that combine data from all concrete tables into a single table. Code below shows the view that combines data from all 3 different tables; **HourlyEmployee**, **SalariedEmployee** and **CommissionedEmployee**.

```

create view dbo.vwEmployees as
Select
    e1.*,
    he.Hours,
    he.Rate,
    null Salary,
    null VacationDays,
    null CommPerc,
    null ItemsSold,
    'HE' EType
from Employee e1 join HourlyEmployee he on e1.EmployeeId = he.EmployeeId
union
Select

```

```

e1.*,
null Hours,
null Rate,
se.Salary,
se.VacationDays,
null CommPerc,
null ItemsSold,
'SE' EType
from Employee e1 join SalariedEmployee se on e1.EmployeeId =
se.EmployeeId
union
Select
e1.*,
null Hours,
null Rate,
null Salary,
null VacationDays,
ce.CommPerc,
ce.ItemsSold,
'CE' EType
from Employee e1 join CommissionedEmployee ce on e1.EmployeeId =
ce.EmployeeId

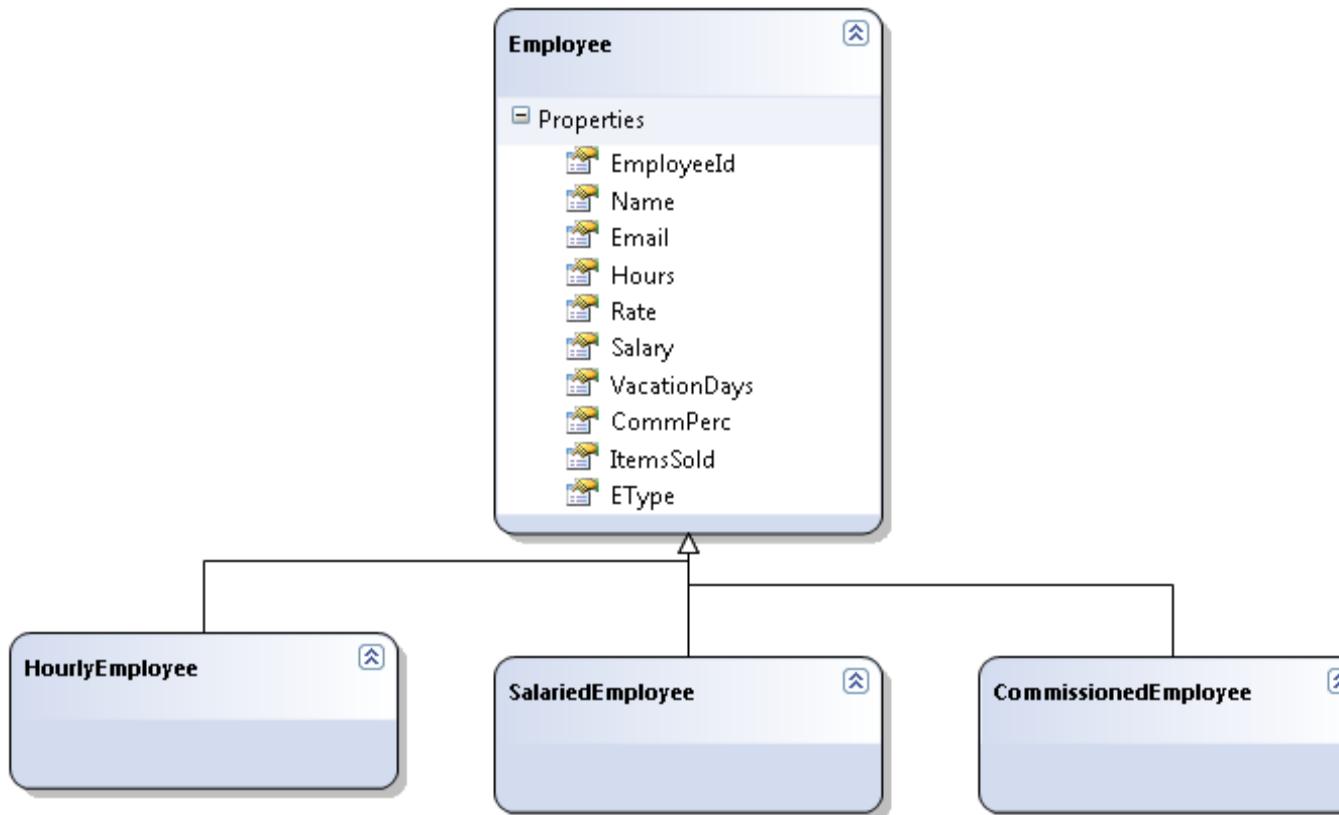
```

Above view, combines records from all 3 tables by using a union operator. Since a view requires the number of columns to match for each union operation, we have to add null columns for any extra columns to accommodate data not provided by other table. For instance the first union operation with HourlyEmployee only has Hours and Rate specific to HourlyEmployee. Since we want to match extra columns provided by other table, we added null values for Salary, VacationDays, CommPerc, ItemSold. Now that we have all 3 tables merged into one view, we need to add a Type column to differentiate each type of Employee. For this we have added EType column and for each table we have specified a different value for EType. For HourlyEmployee, I am using HE for EType, for SalariedEmployee SE and for CommissionedEmployee, I am using CE.

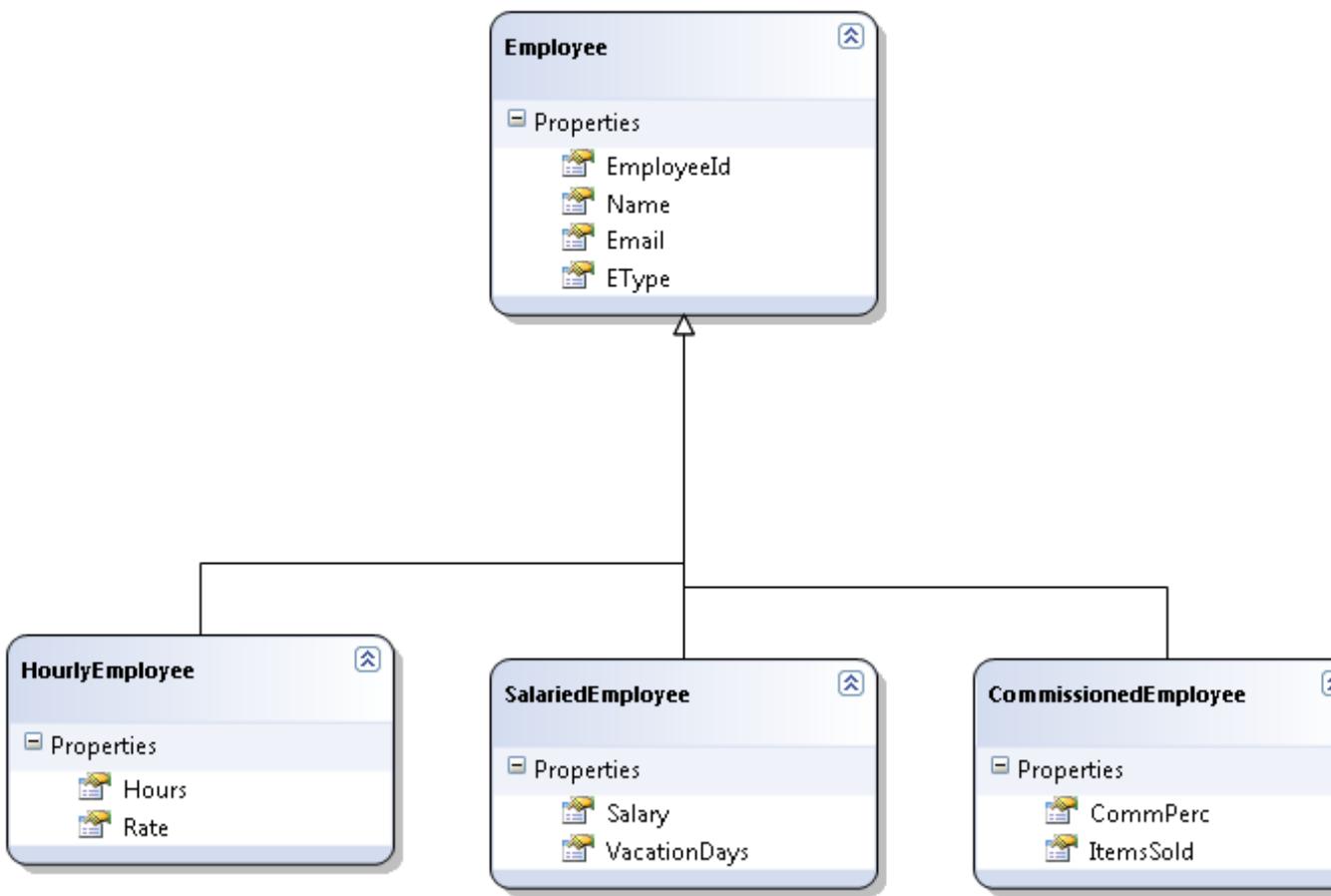
Once the view is setup, we can map the view to our specific entities. Following walk through takes you through the process of mapping our view to concrete classes defined on the linq to sql designer.

We start with dragging our view defined on the database onto the designer. We will than change our entity name from vwEmployee to just Employee. Since employee class maps to 3 distinct concrete types HourlyEmployee,

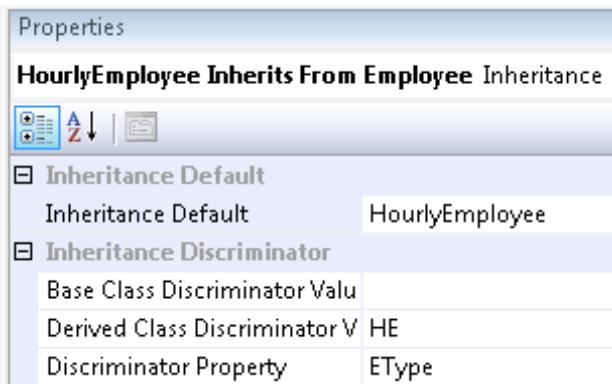
SalariedEmployee and ComissionedEmployee, we will add these classes to the designer and inherit from our base Employee class as shown below.

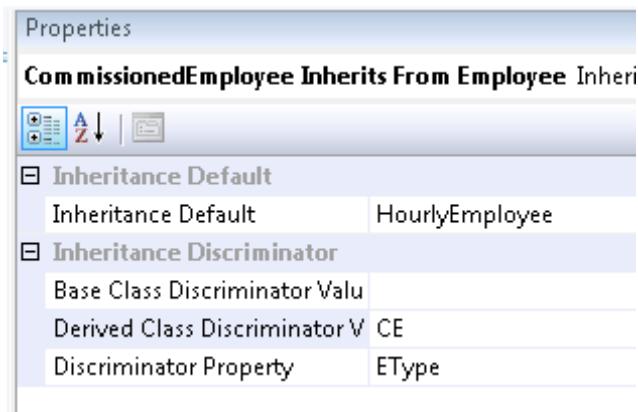
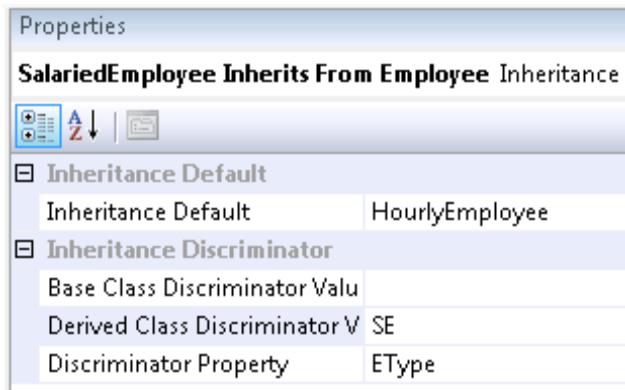


Once we have our concrete classes' setup, we can move additional fields defined in our base class that is not common to each class into their respective classes as follows.



Next we need to define how linq to sql is going to map our view data to each concrete class. For that we use the decimator column and specify discriminator value for each concrete type. We also need to specify the default class the inheritance will map to in the case a valid discriminator value is not found that matches any of the concrete type classes.





In the above screen shot, I am saying that linq to sql should use EType column defined in my view to map to each concrete class. IF EType has a value of CE, it should map to CommissionedEmployee; SE should map to SalariedEmployee and HE should map to HourlyEmployee class.

Our concrete classes are mapped to view which does not support inserts and updates and deletes. If you were trying inserting a concrete type, sql server will raise an exception saying that view is not updatable. To support the insert, updates and deletes, we can create stored procedures that can manage writing data to appropriate tables. Following code shows 3 insert stored procedures that can insert into employee base class and their respective concrete class.

```
create proc dbo.InsertHourlyEmployee
```

```

(
@EmployeeId int output,
@Name varchar(50),
@Email varchar(50),
@Hours int,
@Rate int
)
as
begin
insert into Employee(Name,Email) values (@Name,@Email)
set @EmployeeId = SCOPE_IDENTITY()
insert into HourlyEmployee(EmployeeId,Hours,Rate) values
(@EmployeeId,@Hours,@Rate)
end

create proc dbo.InsertSalariedEmployee
(
@EmployeeId int output,
@Name varchar(50),
@Email varchar(50),
@Salary int,
@VacationDays int
)
as
begin
insert into Employee(Name,Email) values (@Name,@Email)
set @EmployeeId = SCOPE_IDENTITY()
insert into SalariedEmployee(EmployeeId,Salary,VacationDays) values
(@EmployeeId,@Salary,@VacationDays)
end

create proc dbo.InsertCommissionedEmployee
(
@EmployeeId int output,
@Name varchar(50),
@Email varchar(50),
@CommPerc int,
@ItemsSold int
)
as
begin
insert into Employee(Name,Email) values (@Name,@Email)
set @EmployeeId = SCOPE_IDENTITY()
insert into CommissionedEmployee(EmployeeId,CommPerc,ItemsSold) values
(@EmployeeId,@CommPerc,@ItemsSold)
end

```

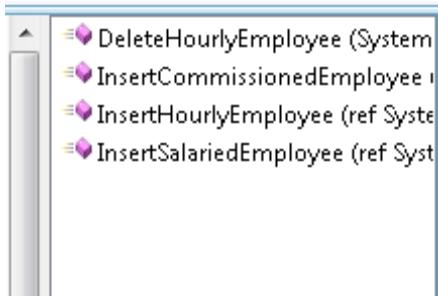
The above stored procedures are pretty similar. Like InsertHourlyEmployee, first inserts the record into Employee table and using the identity column inserts record into HourlyEmployee table. Since employee id is an autogenerated field, we have created an output parameter that we are

assigning the generated employee value to. The output parameter will later be read and assigned to EmployeeId property on Employee class.

Below is an example of Delete stored procedure that deletes an HourlyEmployee.

```
create proc dbo.DeleteHourlyEmployee
(@empid int)
as
begin
delete HourlyEmployee where EmployeeId = @empid
delete Employee where EmployeeId = @empid
end
```

In the Delete stored procedure above, I am deleting the row from first HourlyEmployee and then deleting it from Employee table. Once we have the stored procedures created for each of the classes, we can drag those stored procedures on the linq to sql designer to leverage it for customizing the insert process.



When you drag the stored procedures linq to sql, automatically creates methods exposed on the datacontext, that makes it easier to call the stored procedure. If the stored procedure returns an output parameter, you have to use ref parameter to get the value from the output parameter.

Linq To Sql allows customization of crud operations by exposing partial methods for insert, update and delete for entity when you extend the datacontext class. Registering with those partial method, means that linq to sql will no longer be responsible for the insert updates and delete for the entity. It is the responsibility of application developer to provide appropriate implementations for persisting the object to the database.

Before we can begin customizing our crud process we need to define a primary key on the Employee class. Usually when you drag on linq to sql designer, you don't have to do this because the constraint information is read by the schema of the table. Since we are mapping to Employee entity to a view, this information has to be done manually. We need to specify that EmployeeId is primary key column, it is auto generated and should be synced up after the insert of Employee as shown below.

Properties	
EmployeeId Member Property	
Code Generation	
Access	Public
Delay Loaded	False
Inheritance Modifier	(None)
Name	EmployeeId
Type	int (System.Int32)
Data	
Auto Generated Value	False
Auto-Sync	Never
Nullable	False
Primary Key	False
Read Only	False
Server Data Type	Int NOT NULL
Source	EmployeeId
Time Stamp	False
Update Check	Always

If we do not define primary column, linq to sql will not create partial methods for insert, update and delete of Employee entity and thus you cannot override the crud process.

Code below shows how we intercept the insert request for Employee entity.

```
partial void InsertEmployee(Employee instance)
{
    int? empid = instance.EmployeeId;
    if (instance is HourlyEmployee)
    {
```

```

        HourlyEmployee emp = instance as
HourlyEmployee;
        this.InsertHourlyEmployee(ref empid,
emp.Name, emp.Email, emp.Hours, emp.Rate);
        instance.EmployeeId = empid.Value;
    }
    if (instance is SalariedEmployee)
    {
        SalariedEmployee emp = instance as
SalariedEmployee;
        this.InsertSalariedEmployee(ref empid,
emp.Name, emp.Email, emp.Salary, emp.VacationDays);
        instance.EmployeeId = empid.Value;
    }
    if (instance is CommissionedEmployee)
    {
        CommissionedEmployee emp = instance as
CommissionedEmployee;
        this.InsertCommissionedEmployee(ref empid,
emp.Name, emp.Email, emp.CommPerc, emp.ItemsSold);
        instance.EmployeeId = empid.Value;
    }
}

```

In the above code, I am checking the Employee to be inserted is HourlyEmployee,SalariedEmployee or CommissionedEmployee because we not get separate partial method for insert of each concrete entity. Depending on the entity I am inserting I am calling the appropriate stored procedure that can insert that entity. Since all the stored procedures return an output parameter of Employee Id, I am passing in ref parameter for the first parameter to the stored procedure to get back the value assigned to employee id by the database. To delete a particular employee, I will similarly register the partial method for DeleteEmployee and call the appropriate stored procedure responsible for deletion operation. Following code shows how to delete HourlyEmployee from the database by calling DeleteHourlyEmployee generated method that calls our stored procedure.

```

partial void DeleteEmployee(Employee instance)
{
    if (instance is HourlyEmployee)
    {

this.DeleteHourlyEmployee(instance.EmployeeId);

```

```
        }
    }
```

Now that we have all classes ready to be inserted and queried, we can query and insert against our classes. Code below shows how we can insert various employee and query for a specific employee in the database.

```
public static void InsertEmployees()
{
    var hourlyemp = new HourlyEmployee
    {
        Email = "alis@gmail.com",
        Hours = 40,
        Rate = 50,
        Name = "Aliis"
    };
    var salaryemp = new SalariedEmployee
    {
        Email = "Sam@gmail.com",
        Name="Sam",
        Salary=90000,
        VacationDays = 15
    };
    var commemp = new CommissionedEmployee
    {
        Email = "John@gmail.com",
        Name="John",
        CommPerc=20, ItemsSold=50,
    };
    var db = new TablePerTypeDataContext();
    //inserting hourly, salaried and commission
employee
    db.Employees.InsertAllOnSubmit(new Employee[] {
hourlyemp, salaryemp, commemp });
    db.SubmitChanges();

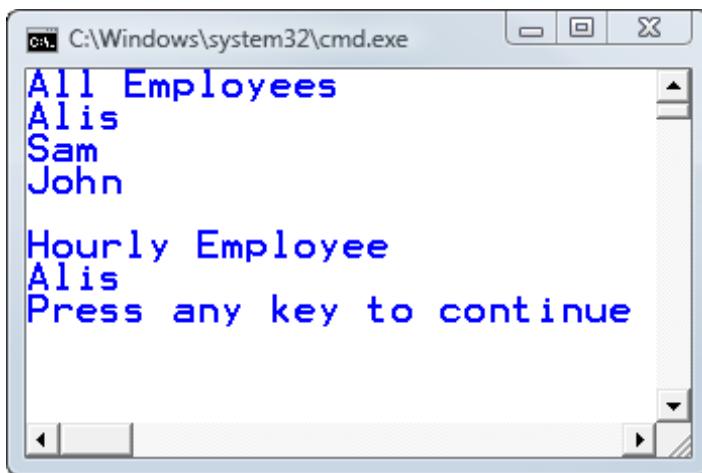
    //confirm that employee got inserted.
    var db1 = new TablePerTypeDataContext();
    Console.WriteLine("All Employees");
    foreach (var emp in db1.Employees)
    {
        Console.WriteLine(emp.Name);
    }
    Console.WriteLine();
    Console.WriteLine("Hourly Employee");
```

```

var hremp =
db.Employees.OfType<HourlyEmployee>().First();
Console.WriteLine(hremp.Name);
}

```

In the above example, I am creating an instance of 3 different employees and then adding the employees to the InsertOnSubmit method passing in the array of employees that I had created earlier. When submit changes is called, linq to sql sees that we have overridden the Insert process and therefore calls our partial method that goes ahead and inserts the employee into the database. To confirm that our records got inserted, I query for HourlyEmployee using a different data context and print the name of the Employee on the console window. Screen shot below confirms are output.



5.2.2 Table per Hierarchy With Enum Using Linq To Sql

Problem: You are using table per hierarchy inheritance in the database. You want to how map your single table to numerous derived entities defined in linq to sql designer. You also want to map your discriminator column to enum defined in your

Solution: When you are programming in .net, you can apply inheritance and polymorphism concepts on your entity classes to solve your business

problems. However when you are saving those objects to the database, there is no clean way to store different child entities having some common properties that they inherit from the base class and some properties that are specific to that child entity. One of the ways to store these objects in the database is by adding all the possible fields available for the entire child and base classes and marking them as allow null. Then you simply set appropriate values for only the fields that are applicable to that specific child classes. In order to differentiate each row and identity what child class it is, you can make use of discriminator column. Below code shows how to map single table to multiple classes using Linq To Sql

```
public enum BookType
{
    Technical=1,
    Certification = 2,
    Cooking = 3,
    Novel = 4
}
[Table]

[InheritanceMapping(Code=BookType.Novel,Type=typeof(Novel),IsDefault=true)]
[InheritanceMapping(Code = BookType.Cooking, Type =
typeof(CookingBook))]
[InheritanceMapping(Code = BookType.Technical, Type =
typeof(TecnnicalBook))]
[InheritanceMapping(Code = BookType.Certification, Type =
typeof(CertificationBook))]
public abstract class Book
{
    [Column(IsPrimaryKey=true,IsDbGenerated=true)]
    public int BookId { get; set; }

    [Column]
    public string Title { get; set; }

    [Column(IsDiscriminator=true)]
    public int Type { get; set; }
}

public class TecnnicalBook:Book
{
    [Column]
    public string Technology { get; set; }
```

```

    }

    public class CertificationBook : TechnicalBook
    {
        [Column]
        public string Exam { get; set; }
    }
    public class CookingBook:Book
    {
        [Column]
        public bool ReceipesAvailable { get; set; }
    }
    public class Novel:Book
    {
        [Column]
        public bool TrueStory { get; set; }
    }
}

```

Discussion: To begin our discussion, we will start with how a single table that maps to numerous classes looks like in the database.

	BookId	Title	Type	Technology	TrueStory	ReceipesAvailable	Exam
1	19	Programming WCF	1	Microsoft	NULL	NULL	NULL
2	20	Sql server Training	2	Microsoft	NULL	NULL	70-431
3	21	Cooks Special	3	NULL	NULL	1	NULL
4	22	Journey To Moon	4	NULL	0	NULL	NULL

In above screen shot, I have a table book, which contains different types of books. All books have Title in common which means Title column in the book table is marked as required because it applies to all books. Then every derived book has properties specific to itself. When the Type is 1, it is a technical book and we are interested in knowing about what kind of technology the book talks about. When the Type is 2, the book is a certification book and we are capturing the exam this book targets. When the Type is 3, the book is a Cooking book and we are capturing if the book contains recipes. If the Type is 4, the book is a novel and we are recording if the book contains a true story. Any column that is not specific to a specific book is set to null. To map these rows to different derived classes in C#, we will make use of Inheritance Mapping Attribute and specify which enum value maps to a specific derived class.

```
public enum BookType
```

```

{
    Technical=1,
    Certification = 2,
    Cooking = 3,
    Novel = 4
}
[Table]

[InheritanceMapping(Code=BookType.Novel, Type=typeof(Novel), IsDefault=true)]
    [InheritanceMapping(Code = BookType.Cooking, Type =
    typeof(CookingBook))]
        [InheritanceMapping(Code = BookType.Technical, Type =
        typeof(TechnicalBook))]
            [InheritanceMapping(Code = BookType.Certification, Type =
            typeof(CertificationBook))]
                public abstract class Book
{
    [Column(IsPrimaryKey=true, IsDbGenerated=true)]
    public int BookId { get; set; }

    [Column]
    public string Title { get; set; }

    [Column(IsDiscriminator=true)]
    public int Type { get; set; }
}

```

In the above code, I have marked my book class as an abstract class because the class itself does not map to any specific row in the book's table. To identify that book class should participate in linq to sql, I am using Table attribute. Next I am applying column attributes to BookId , Title, and Type to indicate that these properties map to column in the database. For BookId, I am also specifying that it is the primary key column and it is generated by the database. Failing to do so would raise exception if when I try to save any derived book class to the database because linq to sql requires primary key column to perform inserts. For Type column, I also specifying that it should be used as a discriminator column. Discriminator column determines what type of object to instantiate for a particular record in the database. Next I specify the inheritance mappings to all the derived classes based on a specific valued found in the discriminator column by making use of Code attribute. Instead of hard coding an integer value for Code, I am making use of Enum

which Linq to Sql automatically maps to the appropriate integer valued defined on the Type column. For instance when Enum is of type Novel, the record should be mapped to Novel class and so on. Linq To Sql also requires me to define a class that should be considered the default mapping in the case where a value does not match to any concrete derived class. You enable the default derived class by setting isDefault to true.

```
public class TechnicalBook : Book
{
    [Column]
    public string Technology { get; set; }
}

public class CertificationBook : TechnicalBook
{
    [Column]
    public string Exam { get; set; }
}
public class CookingBook : Book
{
    [Column]
    public bool ReceipesAvailable { get; set; }
}
public class Novel : Book
{
    [Column]
    public bool TrueStory { get; set; }
}
```

In the code above, I am creating my derived classes that I have defined mapping for on my base book class. Each derived class has specific properties that apply to its type of book and are attributed with column attribute. After configuring my classes, I can use the classes for querying, inserting and updating using linq. Code below shows various linq queries to get specific book to returning all books in the books table.

```
public static void BooksExample()
{
    var db = new TPHDataContext();
    //insert different books
    var books = new Book[]
    {
```

```

                new TechnicalBook{Title="Programming
WCF",Technology="Microsoft"} ,
                new CertificationBook{Title="Sql server
Training",Technology="Microsoft",Exam="70-431"} ,
                new CookingBook{Title="Cooks
Special",RecipesAvailable=true} ,
                new Novel{Title="Journey To
Moon",TrueStory=false}
            };
            db.Books.InsertAllOnSubmit(books);
            db.SubmitChanges();

            Console.WriteLine("All Books");
            foreach (var book in db.Books)
            {
                Console.WriteLine(book.Title);
            }

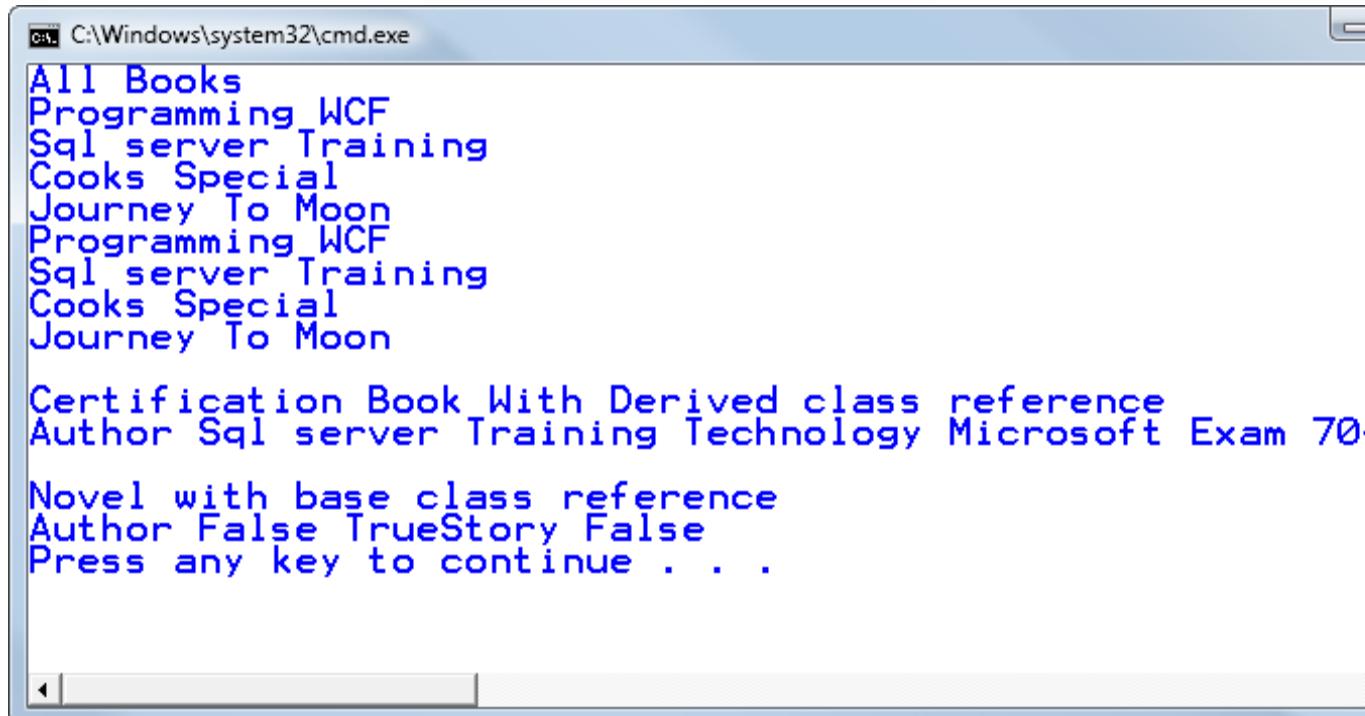
            Console.WriteLine("\r\nCertification Book With
Derived class reference");
            var certbook =
db.Books.OfType<CertificationBook>().First();
            Console.WriteLine("Author {0} Technology {1}
Exam {2}",certbook.Title,certbook.Technology,certbook.Exam);

            Console.WriteLine("\r\nNovel with base class
reference");
            var novel = db.Books.First(b => b is Novel);
            Console.WriteLine("Author {1} TrueStory
{1}",novel.Title,((Novel)novel).TrueStory);
        }
    }
}

```

In the above code, I am creating an array of various types of book by creating an instance of a specific derived book class such as Novel, CookingBook etc. To insert different types of books in the database, I am calling InsertAllOnSubmit to insert my books followed by SubmitChanges. To query for all book, I am just looping through the books property exposed on the datacontext. If I want to get specific book and also be referencing a derived instance, I can use OfType operator as shown in the example. To access a particular derived instance, with a base class reference I am using is operator in the lambda expression applied to the first operator. Is operator filter the book to the derived class you are interested in but it the reference returned is

of base class? To access property specific to derived class, you need to cast the base class to derived class as shown in the code. Code above writes following content on the output window.



```
C:\Windows\system32\cmd.exe
All Books
Programming WCF
Sql server Training
Cooks Special
Journey To Moon
Programming WCF
Sql server Training
Cooks Special
Journey To Moon

Certification Book With Derived class reference
Author Sql server Training Technology Microsoft Exam 70

Novel with base class reference
Author False TrueStory False
Press any key to continue . . .
```

The above example made use of attributes to translate columns defined in the table to properties defined in your entities. If attributes removes the clarity of the class, you can define the mapping in separate file and load the mapping in the constructor of the datacontext. Mapping file below shows how we can map Books table defined in the database to various derived entities defined in our project.

```
<Database Name="Ecommerce"
xmlns="http://schemas.microsoft.com/linqtosql/dbml/2007">
<Table Name="dbo.Book">
    <Type Name="Book">
        <Column Name="BookId" Type="System.Int32"
Member="BookId" IsPrimaryKey="true" IsDbGenerated="true"/>
        <Column Name="BookAuthor" Type="System.String"
Member="BookAuthor" />
        <Column Name="Type" Type="System.Int32"
Member="Type" IsDiscriminator="true" />
```

```

        <Type Name="TecnnicalBook" InheritanceCode="1"
IsInheritanceDefault="true">
            <Column Name="Technology"
Type="System.String" Member="Technology" />
            <Type Name="CertificationBook"
InheritanceCode="2">
                <Column Name="Exam"
Type="System.String" Member="Exam" />
            </Type>
        </Type>
        <Type Name="CookingBook" InheritanceCode="3">
            <Column Name="ReceipesAvailable"
Type="System.Boolean" Member="ReceipesAvailable" />
        </Type>
        <Type Name="Novel" InheritanceCode="4">
            <Column Name="TrueStory"
Type="System.Boolean" Member="TrueStory" />
        </Type>
    </Table>
</Database>

```

To load the mapping file, you can use `XmlMappingSource` class and call `FromUrl` specifying the name of the file which contains the mapping.

6. Working with Objects

6.1 Using auto-generated Guids as entity key

Problem: You have a customer table on the database which has `CustomerId` as the primary key column. `CustomerId` has unique identifier as the data type basically a guid. The guid is generated by the database by assigning default value to be a call to system defined function `newsequentialid` that generates a random guid. Figure below shows the `CustomerId` and its default value on the database.

General	
(Name)	CustomerId
Allow Nulls	No
Data Type	uniqueidentifier
Default Value or Binding	(newsequentialid())
Table Designer	
Collation	<database default>
Computed Column Specification	
Condensed Data Type	uniqueidentifier
Description	
Deterministic	Yes
DT5-published	No
Full-text Specification	
Has Non-SQL Server Subscriber	No

You want to import Customer table into EDM and ensure that when a new customer is created in the database, the unique identifier value generated by the database is reflected on the customer entity's CustomerId property.

Solution: Ef does not require that a primary key column be an identity column or has the data type of integer. When we import the above table structure with CustomerId as the primary key, EF will see that its data type is unqiueidentifier and will automatically set the CustomerId data type to be Guid. However EF cannot infer that CustomerId column value is generated by the database using default value. To tell EF that CustomerId value will be generated by the database, modify the CustomerId property in the SSDL to as follows.

```
<Property Name="CustomerId" Type="uniqueidentifier"
StoreGeneratedPattern="Identity" Nullable="false" />
```

The above line tells EF that we need to treat customerId property as an identity column meaning database is responsible for generating the value and EF needs to update the CustomerId property to the value generated by the database after the insert succeeds. Currently EF cannot refresh the conceptual entity with the new guid value because in sql server 2000 there is no way to return the last guid inserted in CustomerId column. With identity columns, we can obtain the last id inserted by using Scope_identity function but in our case the value generated by based on a default value and therefore is not identity value. With sqls server 2005 or above we can use output clause to get the last row inserted and from that row we can grab the guid inserted into the

CustomerId column. Since the entire process is so sql server specific, we need to use stored procedure to map insert, update and deletion of the customer entity. In the discussion below we will walk through the steps of creating stored procedure and mapping them to the customer entity.

Discussion: To indicate to EF that a value is going to be generated by the database, we have to use `StoreGeneratedPattern=Identity`. This will let the system generate the correct insert statement that will not include identity column as part of the insert. In addition EF after the insert will execute `scope_identity` function to grab the last identity inserted. This is a problem in our case because the CustomerId column is not an identity column as its value is assigned using a default value that calls system function to get a random guid.

Given this fact that EF cannot retrieve the CustomerId value after the insert, when you try to use the model as it is you get the following error.

Server generated keys are only supported for identity columns. Key column has type 'SqlServer.uniqueidentifier' which is not a valid type for an identity column

This is a clear indication of the fact that EF framework cannot handle returning the primary customerId that is based on a default value. Hopefully the next version of EF will be able to at least work with this scenario when the target databases are sql server 2005 or above by using output clause. For now we have to this work ourselves. Steps below outline the process of getting a completely working customer entity that we can insert update and delete using store procedures.

1. Import the above customer table into edm using EDM import wizard.
2. Since CustomerId value is generated on the database we need to modify the customer entity on the ssdl to indicate that. Code below shows the change required on the CustomerId column on the ssdl layer.

```
<Property Name="CustomerId" Type="uniqueidentifier"  
StoreGeneratedPattern="Identity" Nullable="false" />
```

This tells EF that database is responsible for assigning the value to the CustomerId column.

3. As discussed earlier, EF has no way of knowing how to retrieve the generated customerId column from the database. Hence we need to

create insert, update and delete stored procedure that can insert customer entity into the database. Code below shows the insert stored procedure required.

```
create proc [sr].[InsertCustomer]
(@Name varchar(50), @Email varchar(50))
as
begin
DECLARE @insertedrows TABLE (customerid uniqueidentifier)

insert into sr.Customer(Name, Email)
output inserted.CustomerId into @insertedrows
values (@Name, @Email)
select customerid from @insertedrows

end
```

On the above stored procedure to grab the customerid inserted, I am using output clause that is available on sql server 2005 or above to get the last record inserted into customer table. From the last record I simply grab the CustomerId and insert it into insertedrows customer table. Since EF requires that identity columns be returned using a select statement, I am select CustomerId column from insertedrows table. The update and delete stored procedures does not have any complications because we need to generate CustomerId value once and that happens during the insert process. Code below shows the update and delete stored procedures.

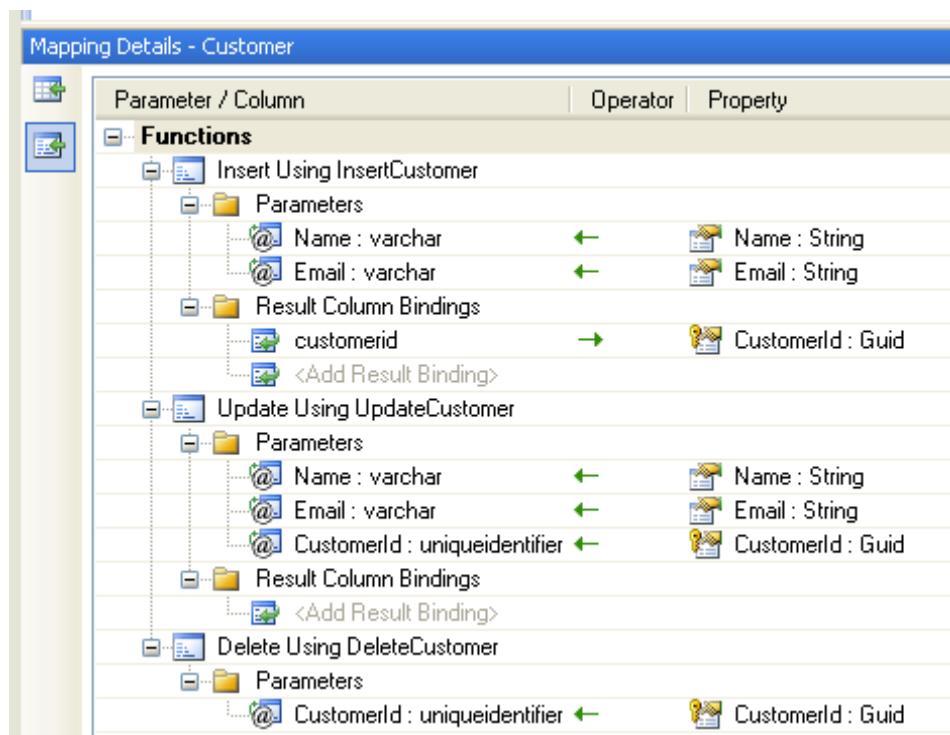
```
create proc sr.UpdateCustomer
(@Name varchar(50), @Email varchar(50), @CustomerId uniqueidentifier)
as
begin
update sr.Customer
set Name = @Name, Email = @Email where CustomerId = @CustomerId
end

create proc sr.[DeleteCustomer]
(@CustomerId uniqueidentifier)
as
begin
delete sr.Customer where customerid = @CustomerId
end
```

The update stored procedure updates Name and Email property of the Customer record based on the CustomerId guid passed in the

parameter. Similarly delete procedure takes in a customerid and deletes the customer record from the customer table.

4. Import insert, update and delete stored procedures created above into the edm using Update Model from database wizard. Since the update wizard overwrites everything on the ssdl, the changes we made to customerid column also got overwritten. Make sure CustomerId column is once again marked as identity.
5. Map the insert function to InsertCustomer stored procedure, update function to UpdateCustomer and delete function to DeleteCustomer stored procedure. Notice that for insert stored procedure there is Resultbinding where I am grabbing the CustomerId column returned from select statement in the stored procedure to CustomerId property on the customer entity.



To test the above model we can create an instance of Customer entity, insert it into the database, update the customer and delete the customer to make sure all three crud operations work correctly. Code below performs crud operation on Customer entity.

```

var db = new GuidsAsIdentityEntities();
    var customer = new Customer
    {

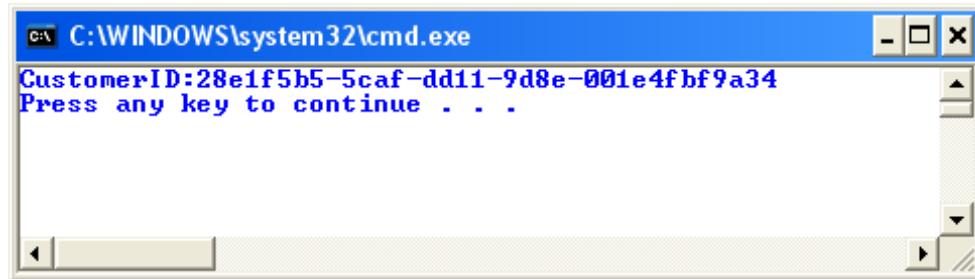
        Email = "abc@gmail.com",
        Name = "Zeeshan"
    };

db.AddToCustomers(customer);
db.SaveChanges();
Console.WriteLine(customer.CustomerId.ToString("D"));
//update teh contact
customer.Name = "Zeeshan Hirani";
db.SaveChanges();

db.DeleteObject(customer);
db.SaveChanges();

```

On the above code after inserting the customer entity I am printing the CustomerId on the console window to confirm that we have retrieve the correct CustomerId guid generated by the database. Figure below shows the results from the console window.



6.2 Reading xml data type columns using EF

Problem: Figure below shows the Candidate table in the database.

Candidate			
	Column Name	Data Type	Allow Nulls
PK	CandidateId	int	<input type="checkbox"/>
	Name	varchar(50)	<input type="checkbox"/>
	Resume	xml	<input type="checkbox"/>
			<input type="checkbox"/>

The candidate table contains the name of the candidate and their resume stored in xml format. You want to import the above table into EDM and use the resume column as a native xml type.

Solution: EF does not have any out of box support for xml data type in the version 1 release. When we import the above table structure into edm, Resume column would be imported as string data type. To use the Resume column as an xml data type, make sure that Resume property is marked private and then create partial class for Candidate entity and create a property CandidateResume that reads the private Resume property and returns the data as xml. Similarly the setter portion of the CandidateResume would write the property back to resume property created on entity model.

Discussion: Entity framework does not import xml data type as xml. When we import a table containing xml data type, entity framework will map the column as string data type on the conceptual model. It is then left to the developer to convert string back to xml for use in the application. Similarly on saving the xml data type needs to be converted back to string because entity framework does not allow saving an xml type to string. Steps below outline the process of consuming xml data from the database, updating the xml and making sure that the updated xml is saved to the database.

1. Import the candidate table using Entity Data Model Wizard. Change the getter and setter for Resume property to be private because we want to consume Resume as xml instead of simple string as created by the designer.

Figure below shows the Candidate entity on edm designer.



2. Create a partial class Candidate that has a property CandidateResume that returns Resume as xml. The setter for the property should be able to get the xml and write it to the information back to Resume property as string. Code below shows the Candidate partial class.

```

private XElement candidateresume = null;
public XElement CandidateResume
{
    get
    {
        if (candidateresume == null)
        {
            candidateresume = XElement.Parse(this.Resume);
            candidateresume.Changed += (sender, eventargs) =>
            {
                if (eventargs.ObjectChange == XObjectChange.Add)
                {
                    this.Resume = candidateresume.ToString();
                }
            };
        }
        return candidateresume;
    }
    set
    {
        candidateresume = value;
        this.Resume = value.ToString();
    }
}

```

On the above code, I have local xml variable that reads the Resume by using XElement.Parse. In addition I am also registering with the change event on the xml and every time a change happens on the xml, I am assigning the new xml back to resume property as resume property is the one that will be persisted by entity framework. For the setter I am using ToXml on the xml passed in and assigning it to Resume string property.

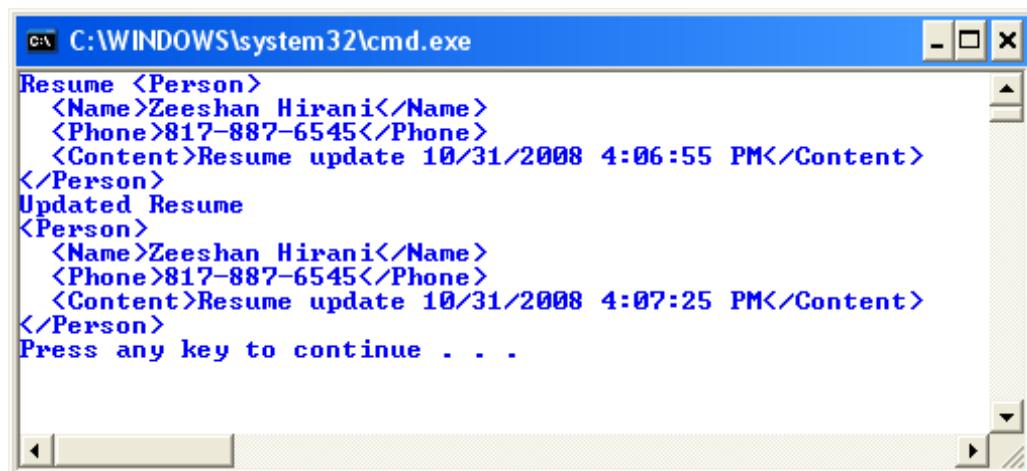
To test the model we can try reading the resume property and updating portion of the xml content and then saving the entity to the database. Then using a second datacontext, we retrieving the candidate and read the candidate's resume to confirm that our updates on the Resume column got successfully updated in the database. Code shows an example of that.

```
var db = new XmlEntities();
    var candidate = db.Candidates.First();
    Console.WriteLine("Resume " +
candidate.CandidateResume.ToString());
    candidate.CandidateResume.SetElementValue("Content", "Resume
update " + DateTime.Now);

db.SaveChanges();

var db2 = new XmlEntities();
var updatedcand = db2.Candidates.First();
    Console.WriteLine("Updated Resume " +
updatedcand.CandidateResume.ToString());
```

On the code above, I am outputting the candidate's resume using CandidateResume custom property we created earlier. Next I am changing the content section of the resume and updating the candidate entity to the database. Using the second data context, I am retrieving the same candidate and outputting the updated Candidate Resume to the console window.



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text output:

```
Resume <Person>
<Name>Zeeshan Hirani</Name>
<Phone>817-887-6545</Phone>
<Content>Resume update 10/31/2008 4:06:55 PM</Content>
</Person>
Updated Resume
<Person>
<Name>Zeeshan Hirani</Name>
<Phone>817-887-6545</Phone>
<Content>Resume update 10/31/2008 4:07:25 PM</Content>
</Person>
Press any key to continue . . .
```

6.3 How does StoreGeneratedPattern work

Problem: Figure below shows the Suppliers table in the database.

Supplier (sr)	
SupplierId	
SupplierName	
CreateDate	
modifiedDate	
TimeStamp	

The above supplier table has several fields whose value is generated by the database when a new record is inserted or updated. CreateDate column is assigned the current date by a trigger that gets fired on an insert of a new supplier in the supplier table. Once CreateDate is assigned a value, it's never updated. ModifiedDate is updated to current date using a trigger that is fired when a row is updated in supplier's table. Timestamp is assigned a new value by database engine when a new supplier is inserted or updated.

SupplierId, the primary key of the table is also generated using identity column and is only assigned when a new supplier is inserted into the supplier table.

We want to ensure that when a new supplier entity is inserted using Entity Framework, the CreateDate, modifiedDate, TimeStamp and SupplierId property should reflect the updated values from the database.

Solution: When a property value is generated by the database, we need to mark the property using StoreGeneratedPattern on the entity defined on the store model. StoreGeneratedPattern can have three values.

1. None
2. Identity
3. Computed

None is the default option which indicates that it is not a server generated value. Setting property to identity means that value is generated once on an insert and any updates on the entity should not affect this value. So in our case SupplierId and CreateDate are assigned value once on an insert and therefore needs to be marked with StoreGeneratedPatten equal to identity. When we import the Supplier table EF will see that SupplierId is an identity column and make sure StoreGeneratedPattern is set to identity. Since CreateDate is assigned a value by insert trigger, EF has no way of knowing that it is a server generated value unless we manually go in the ssdl section of the edmx and change it to identity. The last option is Computed which means that server will always generate a new value for the column and EF should refresh the column with new value on both insert and updates. In supplier's table ModifiedDate is assigned a new value on update and therefore StoreGeneratedPattern needs to set to Computed to ensure that we receive the new value after supplier entity is successfully updated. The Timstamp property on supplier entity in SSDL should also be set to computed because EF is aware of timestamp column as being server generated.

Note:

If primary key of the table is generated using other means such as default value, or insert trigger, EF cannot retrieve the inserted value even if the property is set to identity. The reason is sql server 2000 has no way of retrieving primary key value if column is not marked as identity column. Although with Sql server 2005 you can use output clause to access the inserted row and grab the primary key value from the inserted row. This feature is not supported in version 1 of EF. To see how to solve this problem in version 1, see section 8.1

Discussion: EF is aware of setting correct StoreGeneratedPattern value when a column is an identity or has a data type of TimeStamp. Since CreateDate and ModifiedDate column are assigned values using trigger which EF is not aware of, we need to manually change the CreateDate and ModifiedDate properperties on the ssdl.

Code below shows the insert and update trigger for Supplier table.

Insert trigger

```
ALTER trigger [sr].[td_SupplierInsert] on [sr].[Supplier]
```

```

for insert
as
update supplier
set createdate = getdate()
from supplier join inserted i on supplier.SupplierId = i.SupplierId

```

Update Trigger

```

ALTER trigger [sr].[td_SupplierUpdate] on [sr].[Supplier]
for update
as
update supplier
set modifiedDate = getdate()
from supplier join inserted i on supplier.SupplierId = i.SupplierId

```

On Insert trigger, createDate is assigned the current date and time using Getdate function. Similarly Update trigger, assigns the current time and date to ModifiedDate column.

Import Supplier table using EDM wizard and modify the ssdl so that CreateDate has StoreGeneratedPattern set to Identity and modifiedDate is set to Computed. Code below shows the correct settings for Supplier entity on the ssdl.

```

<EntityType Name="Supplier">
    <Key>
        <PropertyRef Name="SupplierId" />
    </Key>
    <Property Name="SupplierId" Type="int" Nullable="false"
StoreGeneratedPattern="Identity" />
    <Property Name="SupplierName" Type="varchar" Nullable="false"
MaxLength="50" />
    <Property Name="CreateDate" Type="date"
StoreGeneratedPattern="Identity" />
    <Property Name="modifiedDate" Type="datetime"
StoreGeneratedPattern="Computed" />
    <Property Name="TimeStamp" Type="timestamp" Nullable="false"
StoreGeneratedPattern="Computed" />
</EntityType>

```

Code below creates a supplier entity, saves the entity to the database and then updates the entity. To test that we have the correct createDate and modified date, at each stage we are printing the values for CreateDate and ModifiedDate on the console window.

```

var db = new StComputedEntities();
    var supplier = new Supplier
    {
        SupplierName = "Exotic Liquids"
    };
db.AddToSupplier(supplier);

```

```

        db.SaveChanges();
        Console.WriteLine();
        Console.WriteLine("supplier insert details");
        Console.WriteLine("Create Date:{0}",
supplier.CreateDate.Value.ToString());
        Console.WriteLine("Modified Date:{0}",
supplier.modifiedDate.Value.ToString("hh:mm:ss"));

        Thread.Sleep(TimeSpan.FromSeconds(3));
        supplier.SupplierName = "Exotic Liquids Ltd.";
        db.SaveChanges();
        Console.WriteLine("Update Details");
        Console.WriteLine("Modified Date:{0}",
supplier.modifiedDate.Value.ToString("hh:mm:ss"));
    
```

Screenshot below shows Create and modifiedDate printed on the console window.

```

file:///C:/Documents and Settings/z...
supplier insert details
Create Date:11/11/2008 12:00:00 AM
Modified Date:01:41:17
Update Details
Modified Date:01:41:59

```

To better understand how EF updates the properties marked as server generated values, I have capture the sql sent by Ef for insert of a product. Code below shows the sql generated.

```

exec sp_executesql N'insert [sr].[Supplier]([SupplierName])
values (@0)
select [SupplierId], [CreateDate], [modifiedDate], [TimeStamp]
from [sr].[Supplier]
where @@ROWCOUNT > 0 and [SupplierId] = scope_identity()',N'@0
varchar(14)',@0='Exotic Liquids'

```

Notice right after insert, EF is retrieving SupplierID,CreateDate, ModifiedDate and TimeStamp because four values are generated by sql server and needs to be reflected on properties on supplier entity after the insert succeeds.

6.4 Exposing EntityCollection and EntityReference properties on an entity

Problem: You have customer entity and you want to expose Orders collection so that you can navigate to the orders for a customer. You need to know what changes you need to make to conceptual and mapping model to support relationship navigation. You also need to know what changes you need to make to customer entity to expose Orders Collection property.

Solution: If an entity participates in a relationship with other entities such as Customer has Orders, than the entity must implement IEntityTypeWithRelationships interface. You implement this interface by providing read-only RelationshipManager property. Than using RelationShipManager object you access the relationship for Orders for the customer as shown in listing 1-1

Listing 1-1

```
[EdmEntityTypeAttribute  
(NamespaceName="LinqCookBook.EFUsingPOCO", Name="Customer")]  
public class Customer : IEntityWithChangeTracker,  
IEntityWithKey, IEntityWithRelationships  
{  
    #region changetracker and entity key  
    //code removed for keeping the exmaple short.  
    #endregion  
  
    #region public properties  
    //customerid, contacttitle, companyname  
    #endregion  
  
    RelationshipManager manager;  
    public RelationshipManager RelationshipManager  
    {  
        get  
        {  
            if (manager == null)  
            {  
                manager =  
                    RelationshipManager.Create(this);  
            }  
            return manager;  
        }  
    }  
}  
  
[EdmRelationshipNavigationPropertyAttribute]
```

```

( "LinqCookBook.EFUsingPOCO", "CustomerOrders", "Orders" ) ]
    public EntityCollection<Order> Orders
    {
        get
        {
            return RelationshipManager
                .GetRelatedCollection<Order>( "LinqCookBook.EFUsingPOCO.CustomerOrders", "Orders" );
        }
    }
}

```

Discussion: As discussed earlier, to support relationship between other entities, an entity must implement IEntityWithRelationships which requires a readonly property called relationship manager. Our Customer class implements IEntityWithRelationships and defines RelationshipManager property in a lazy loading fashion. If the Relationship manager does not exist the first time we call the static method Create to create the relationshipmanager. RelationshipManager keeps track of relationships between objects defined in the ObjectContext. Customer and Orders relationships is defined in our conceptual modal and mapping file defines how to map customer and Orders table to the relationship defined in the conceptual model.

To expose Orders property on a customer object, we need to mark the property as a relationship property by adding EdmRelationshipNavigationPropertyAttribute attribute. EdmRelationshipNavigationPropertyAttribute takes three parameters. First parameter is the namespace where the relationship is defined; second parameter is the name of the relationship; and third parameter is the Role that we want from the relationship. Since we are exposing Orders we are looking for the Orders role. However if we were exposing Customer object on an Order entity, our role in that case would be Customer. Both role names are explicitly defined on Association element in the conceptual modal as we will look in a second. Since we have a reference to the relationshipManager

object, all we need to do to get Orders collection is to call GetRelatedCollection on the manager passing in the generic type you want returned. GetRelatedCollection method requires two parameters to be passed in. First parameter is the fully qualified name of the relationship we want to retrieve including the namespace. Second parameter is the name of the role that you want returned from the relationship which in the case of Orders collection is the Orders Role.

Now that we are using CustomerOrders relationship name and roles defined within it, we need to define CustomerOrders in our conceptual modal. Listing 1-2 shows example of CustomerOrders relationship defined. To declare CustomerOrders relationship, we create an AssociationSet element giving the Name of CustomerOrders and Association as the fully qualified name. Inside the associationset we define two roles that map to Customer and Orders entityset declared at top. It is important to understand that associationset only defines which entitysets participates in the relationship. It is the Association element that defines how each role or entitysets relate to each other. Like in CustomerOrders association, Customer has a multiplicity of 0 to 1 and Orders has a multiplicity of many. What this means is a customer can have many Orders. So far we have only described what entities participates in a relationship by using Roles and how those roles relate to each other using Multiplicity. The next step we do is expose a navigation property on CustomerEntity by using NavigationProperty Element as shown in Listing 1-2. Inside the navigation, we are specifying from which relationship we want to go from which in Customer case, we want to go from Customers to Orders.

Listing 1-2

```
<Schema Namespace="LinqCookBook.EFUsingPOCO" Alias="Self" xmlns="http://schemas.microsoft.com/ado/2006/04/edm">
    <EntityContainer Name="NorthwindEntities">
        <EntityType Name="Customers" EntityType="LinqCookBook.EFUsingPOCO.Customer" />
        <EntityType Name="Orders" EntityType="LinqCookBook.EFUsingPOCO.Order" />
        <AssociationSet Name="CustomerOrders" Association="LinqCookBook.EFUsingPOCO.CustomerOrders">
            <End Role="Customer" EntitySet="Customers" />
            <End Role="Orders" EntitySet="Orders" />
```

```

        </AssociationSet>
    </EntityContainer>
    <Association Name="CustomerOrders">
        <End Role="Customer"
Type="LinqCookBook.EFUsingPOCO.Customer" Multiplicity="0..1"
/>
        <End Role="Orders"
Type="LinqCookBook.EFUsingPOCO.Order" Multiplicity="*" />
    </Association>
    <EntityType Name="Customer">
        <!-- code removed for clarity-->
        <NavigationProperty Name="Customer"
            <NavigationProperty Name="Orders"

Relationship="LinqCookBook.EFUsingPOCO.CustomerOrders"
                FromRole="Customer" ToRole="Orders" />
        </EntityType>
    </Schema>

```

The next step after defining the relationship is to map the relationship in the mapping file to tables. Listing 1-3 shows the mapping of CustomerOrders Association Set. In the associationSetMapping we define the association we want to map followed by mapping the roles within each association. For Customer role, we are mapping CustomerID property to CustomerId column. For Orders role, we are mapping OrderId property to OrderID column. Since every order will always have a customer, we also need to add a conditional attribute stating that Column CustomerID cannot be null for this relationship mapping.

Listing 1-3

```

<Mapping Space="C-S" xmlns="urn:schemas-microsoft-
com:windows:storage:mapping:CS">
    <EntityContainerMapping
        StorageEntityContainer="dbo"
        CdmEntityContainer="NorthwindEntities">
        <EntitySetMapping Name="Customers">
            <!-- code removed for clarity-->
        </EntitySetMapping>
        <EntitySetMapping Name="Orders">
            <!-- code removed for clarity-->
        </EntitySetMapping>
    </EntityContainerMapping>

```

```

<AssociationSetMapping Name="CustomerOrders"
    TypeName="LinqCookBook.EFUsingPOCO.CustomerOrders"
        StoreEntitySet="Orders">
        <EndProperty Name="Customer">
            <ScalarProperty Name="CustomerID"
ColumnName="CustomerID"/>
        </EndProperty>
        <EndProperty Name="Orders">
            <ScalarProperty Name="OrderID"
ColumnName="OrderID" />
        </EndProperty>
        <Condition ColumnName="CustomerID" IsNull="false"
/>
    </AssociationSetMapping>
</EntityContainerMapping>
</Mapping>

```

For relationship navigation to work properly, you have to define the same relationship that we defined in conceptual model, be defined at assembly level as follows.

```

[assembly: EdmRelationshipAttribute
    ("LinqCookBook.EFUsingPOCO", "CustomerOrders",
"Customer",
    RelationshipMultiplicity.ZeroOrOne,
    typeof(LinqCookBook.EFUsingPOCO.Customer),
    "Orders", RelationshipMultiplicity.Many,
    typeof(LinqCookBook.EFUsingPOCO.Order))]

```

The edmrelationship attribute defines how each role related to the other role using multiplicity as we defined in our conceptual model. If you do not have this attribute at the assembly level, you will get runtime error with exceptions that association cannot be found.

Now to test our code, we can write a query against our model. In the code below, we are getting customers with a contact title of Sales Representative and than for each Cutomer loading its Orders and printing the total Orders for each Customer.

```

public static void CustomerWithOrders()
{
    var db = new NorthwindEntities();

```

```

        var custs = db.Customers.Where(c =>
c.ContactTitle == "Sales Representative");
        foreach (var cus in custs)
        {
            cus.Orders.Load();
            Console.WriteLine("cust {0}
OrderCount:{1}", cus.CustomerID, cus.Orders.Count());
        }
    }

```

```

cust ALFKI OrderCount:6
cust AROUT OrderCount:13
cust BLAUS OrderCount:7
cust BSBEV OrderCount:10
cust CONSH OrderCount:3
cust FRANS OrderCount:6
cust HILAA OrderCount:18
cust HUNGC OrderCount:5
cust LACOR OrderCount:4
cust LEHMS OrderCount:15
cust OLDWO OrderCount:10
cust PERIC OrderCount:6
cust PRINI OrderCount:5
cust RANCH OrderCount:5
cust SAVEA OrderCount:31
cust TRADH OrderCount:6
cust WANDK OrderCount:10
Press any key to continue . . .

```

What if in an Order class we want to have a relationship that navigates back to the customer who placed this Order. Since we have already defined our relationship, we need to do only two things. First expose the Customer entity and second add a navigation element on Orders entity defined in the conceptual model. In Listing 1-4, I have a customer property attributed with EdmRelationshipNavigationPropertyAttribute since this is a relationship property. Using the RelationshipManager's GetRelatedReference we access Customer instance for the Order, passing in the relationship name and the role of relationship we want to retrieve which in this case is Customer because we are on Order class.

Listing 1-4

```
Customer customer;
    [EdmRelationshipNavigationPropertyAttribute
        ( "LinqCookBook.EFUsingPOCO", "CustomerOrders",
"Customer" )]
    public Customer Customer
    {
        get
        {
            var efreference =
                RelationshipManager
                    .GetRelatedReference<Customer>

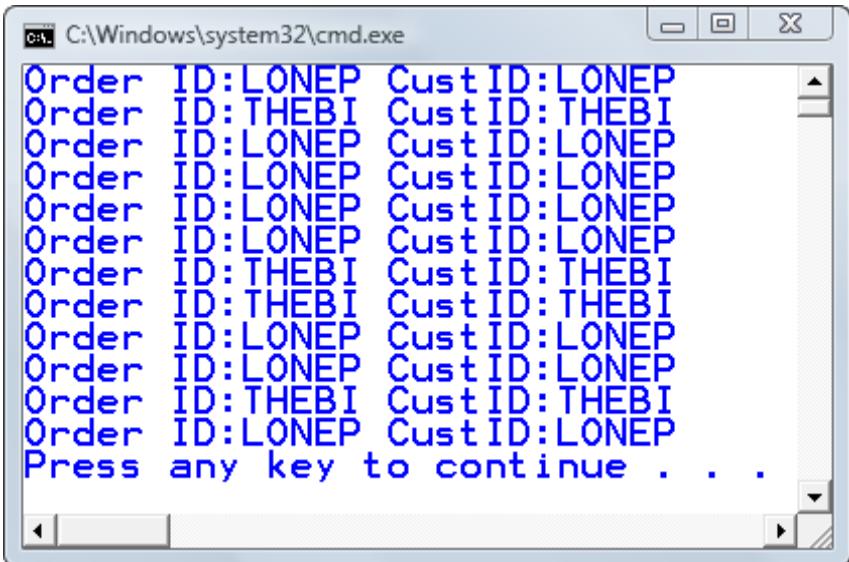
            ( "LinqCookBook.EFUsingPOCO.CustomerOrders", "Customer" );
            if ( !efreference.IsLoaded )
            {
                efreference.Load( );
            }
            return efreference.Value;
        }
    }
}
```

Next we need to add Navigation element on the Order entity defined in the conceptual modal with a role going from Order to Customer as shown below.

```
<NavigationProperty Name="Customer"
    Relationship="LinqCookBook.EFUsingPOCO.CustomerOrders"
    FromRole="Orders" ToRole="Customer" />
```

Once we have exposed our Customer property and defined navigation element inside the conceptual model, we can write a simple query navigating from Order to its customer and printing the OrderId and customerId on the print window as shown.

```
public static void OrdersWithCustomer()
{
    var db = new NorthwindEntities();
    var orders = db.Orders.Where(o => o.ShipCity ==
"Portland");
    foreach (var order in orders)
    {
        Console.WriteLine("Order ID:{1}
CustID:{1}", order.OrderID, order.Customer.CustomerID);
    }
}
```



The screenshot shows a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The window displays a list of 15 entries, each consisting of three fields separated by spaces: 'Order', 'ID:LONEP', and 'Cust ID:LONEP'. Below this list, the text 'Press any key to continue . . .' is visible. The window has standard scroll bars on the right side.

Order	ID:LONEP	Cust ID:LONEP
Order	ID:THEBI	Cust ID:THEBI
Order	ID:LONEP	Cust ID:LONEP
Order	ID:THEBI	Cust ID:THEBI
Order	ID:THEBI	Cust ID:THEBI
Order	ID:LONEP	Cust ID:LONEP
Order	ID:LONEP	Cust ID:LONEP
Order	ID:THEBI	Cust ID:THEBI
Order	ID:LONEP	Cust ID:LONEP
Press any key to continue . . .		

6.5 Monitoring collection changes (Add and Remove)

Problem: You want to listen to EntityCollection change event so that anytime an entity gets added or removed; you want to modify its parent entity.

Suppose you are organizing a show which spans several different timings. To represent the problem, you have created two tables; Show and Event Timings. For instance show spans over a 3 day period; Monday it starts at 8:00 AM to 4:00 PM, on Tuesday it starts at 9:00 AM to 4:00 PM and Wednesday it starts at 10:00AM to 3:00PM. To ensure that every time you want to search on a show, you don't have to go to EventTimings table, we have added two additional columns called StartDate and EndDate on Show table. StartDate and EndDate gets updated to reflect the minimum and maximum timings from its EventTimings relationship table. You want to know how to update the start and end date on show table every time an event is added or deleted from EventTimings collection.

Solution: To update the start and end date for a show when a new timing gets added or removed from EventTimings collection, you have to register with

AssociationChanged for EventTimings Collection. This collection exposes two objects necessary to perform your business activity. First object, Action represents the type of action performed on the collection. When you add event timing to EventTimings collection, associationChanged event gets triggered and the action is set to Add. If you remove event timing from EventTimings collection, the action is set to Remove. The second object passed is the entity that got removed or added from the collection. Using the entity passed in, you have the ability to check for business rules and determine if this insert or remove is valid operation otherwise throw exception. Below code shows to use association changed event.

```
public partial class TradeShow
{
    /// <summary>
    /// creates a trade show with name and different
    timings the show is happenning at.
    /// </summary>
    public static void CreateTestShow()
    {
        var db = new EcommerceEntities();
        TradeShow show = new TradeShow
        {
            Name = "ATLANTIC DESIGN MANUFACTURING SHOW",
            ShowInfo = "ATLANTIC DESIGN MANUFACTURING
SHOW description",
            Location = "New York",
        };
        db.AddToTradeShows(show);
        db.SaveChanges();
        //add timings would fire association changed
        event.
        show.EventTimings.Add( new EventTiming{EventDate
= DateTime.Parse("8/4/2008"),Timing="8:00AM-4:00PM"});
        show.EventTimings.Add(new EventTiming {
EventDate = DateTime.Parse("8/5/2008"), Timing = "9:00AM-
4:00PM" });
        show.EventTimings.Add(new EventTiming {
EventDate = DateTime.Parse("8/6/2008"), Timing = "10:00AM-
3:00PM" });
        db.SaveChanges();
    }

    /// <summary>
```

```

        /// update the trade show by removing one of the
timings.
        /// this would trigger Association changed event
causing start and end date
        /// for the show to update itself from its
EventTiming collection.
        /// </summary>
    public static void UpdateTradeShow()
    {
        var db = new EcommerceEntities();
        //get gunshow by name and also eventimings
collection as well.
        var show = db.TradeShows
                    .Include("EventTimings")
                    .First(s => s.Name == "ATLANTIC
DESIGN MANUFACTURING SHOW");
        //just remove the first available timing in the
collection
        var firstavailabletiming =
show.EventTimings.First();
        db.DeleteObject(firstavailabletiming);

//show.EventTimings.Remove(firstavailabletiming);
        //removing it from the collection is not enough
you have to also
        //delete eventiming as well.
        db.SaveChanges();
    }
    public TradeShow()
    {
        this.EventTimings.AssociationChanged += (sender,
e) =>
        {
            if (!this.EventTimings.IsLoaded)
            {
                this.EventTimings.Load();
            }
            if (e.Action ==
CollectionChangeAction.Add)
            {
                this.UpdateTimings();
            }
            else if (e.Action ==
CollectionChangeAction.Remove)
            {
                if (this.EventTimings.Count() > 0)
                {

```

```

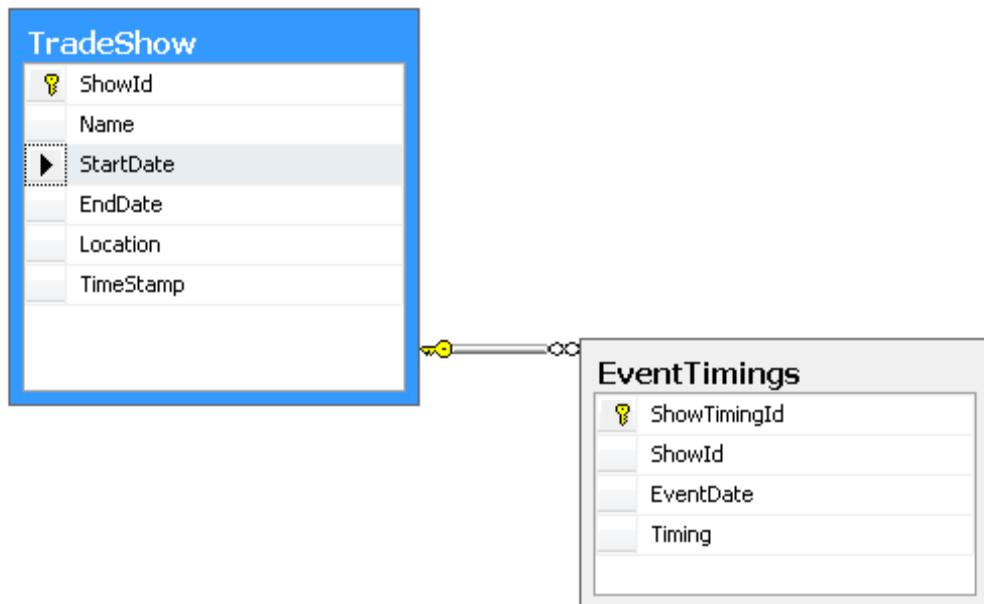
        this.UpdateTimings();
    }
    else
    {
        this.StartDate =
this.EndDate =
DateTime.MaxValue;
    }
}

};

void UpdateTimings()
{
    this.StartDate = this.EventTimings.Min(ev =>
ev.EventDate);
    this.EndDate = this.EventTimings.Max(ev =>
ev.EventDate);
}
}

```

Discussion: To start the discussion let's start with looking at how our tables look like in the database and how they are related to each other.



In the above diagram, we have a tradeshow table where we capture the name, StartDate, EndDate and Location of the tradeshow. We have another table EventTimings that captures the various days' timings for a particular show. Although we could use EventTiming table to find out the span of days for a certain show at runtime, by having StartDate and EndDate on TradeShow table it allows us to not have to join against another table and also makes search queries less complicated. Now that we have our tables in place we can use the entity framework wizard to generate entity classes and register for Association Changed event in our partial class as show below.

```

public TradeShow()
{
    this.EventTimings.AssociationChanged += (sender,
e) =>
{
    if (!this.EventTimings.IsLoaded)
    {
        this.EventTimings.Load();
    }
    if (e.Action ==
CollectionChangeAction.Add)
    {
        this.UpdateTimings();
    }
    else if (e.Action ==
CollectionChangeAction.Remove)
    {
        if (this.EventTimings.Count() > 0)
        {
            this.UpdateTimings();
        }
        else
        {
            this.StartDate =
DateTime.MaxValue;
            this.EndDate =
DateTime.MaxValue;
        }
    }
}
};
```

In the above example, I am registering with association changed event for the eventTimings collection in the constructor of the TradeShow class. I am using an inline lambda expression so that I could have my code next to the changed event. If your collection changed event is not going to be using any instance level variables than you may be better off using a static method to improve performance since you will not have a method per instance of TradeShow class. AssociationChanged event is fired after the collection has been modified which means that if you remove or add event timing, inside the AssociationChanged event, accessing the EventTimings collection will reflect the updated collection with additions and deletions of the timings

In the association Changed event, I am first checking to see if my EventTimings collection has been loaded, if not I am calling Load to load the entire collection from the database. The reason I am loading the entire collection for EventTimings is because I want to set my startdate for the show to be lowest date from the EventTimings collection and Enddate of the show to be the max date from the event timings collection. Once I have my eventtimings populated correctly, I am checking to see what kind of operation was performed on the collection by looking at the action property of the argument passed in. If the action is Add means new event timing was added to collection, I am calling Updatetimings instance method on my Tradeshow class. Below is the code for my UpdateTimings method

```
/// <summary>
    /// sets the start date for the show
    /// to be the Minimum date from eventtimings
collection
    /// for end date, take the max date in hte
eventtimings collection
    /// </summary>
    void UpdateTimings()
    {
        this.StartDate = this.EventTimings.Min(ev =>
ev.EventDate);
        this.EndDate = this.EventTimings.Max(ev =>
ev.EventDate);
    }
```

In the UpdateTimings method, I am setting my StartDate to be the minimum date of my EventTimings collection. I am also setting my EndDate to be the Max date in my eventtimings collection.

If the action passed to the collection changed event is remove, meaning event timing was removed from the collection, I am checking to see if there are any event timings available in the EventTimings collection by using Count. If Eventtimings collection is empty, I set the startdate and endDate of tradeshow to be the default value which is DateTime.Max. If EventTiming Count is greater than 0, I am calling UpdateTiming method to update the show timing with the update data available in the EventTimings collection.

CreateTestShow method shows an example where we add timings to eventTimings collection of the show.

```
/// <summary>
    /// creates a trade show with name and different
    /// timings the show is happening at.
    /// </summary>
    public static void CreateTestShow()
    {
        var db = new EcommerceEntities();

        TradeShow show = new TradeShow
        {
            Name = "ATLANTIC DESIGN MANUFACTURING SHOW",
            Location = "New York",
        };
        db.AddToTradeShows(show);
        db.SaveChanges();
        //add timings would fire association changed
        event.
            show.EventTimings.Add( new EventTiming{EventDate
= DateTime.Parse("8/4/2008"),Timing="8:00AM-4:00PM" });
            show.EventTimings.Add(new EventTiming {
EventDate = DateTime.Parse("8/5/2008"), Timing = "9:00AM-
4:00PM" });
            show.EventTimings.Add(new EventTiming {
EventDate = DateTime.Parse("8/6/2008"), Timing = "10:00AM-
3:00PM" });
        db.SaveChanges();
    }
```

In the CreateTestShow method, I am creating an instance of Tradeshow class passing in the values for the require property of the tradeshow such as Name and Location. After specifying property values for show, I am saving the trade show to the database. After saving the Tradeshow, I add 3 different timings to EventTimings collection of the trade show. Every time I add timing to the collection, AssociationChanged event gets fired and TradeShow's start and end date gets updated with the new information available in the timings collection. After adding timing to eventtimings collection, I save my changes on the datacontext, which causes my tradeshow to get updated and 3 new eventtimings are added to the database as shown in the results from database query.

	ShowId	StartDate	EndDate	Location
1	8	2008-08-04 00:00:00.000	2008-08-06 00:00:00.000	New York
	ShowTimingId	ShowId	EventDate	Timing
1	10	8	2008-08-04 00:00:00.000	8:00AM-4:00PM
2	11	8	2008-08-05 00:00:00.000	9:00AM-4:00PM
3	12	8	2008-08-06 00:00:00.000	10:00AM-3:00PM

We saw how our association changed event got fired with when adding new timings to the event Timings collection. Let's create a simple example where we remove timing from eventtimings collection. Example below shows the code where we remove event timing.

```

/// <summary>
    /// update the trade show by removing one of the
    /// timings.
    /// this would trigger Association changed event
    /// causing start and end date
    /// for the show to update itself from its
    /// EventTiming collection.
    /// </summary>
public static void UpdateTradeShow()
{
    var db = new EcommerceEntities();
    //get gunshow by name and also eventtimings
    collection as well.
    var show = db.TradeShows
        .Include("EventTimings")
        .First(s => s.Name == "ATLANTIC
DESIGN MANUFACTURING SHOW");

```

```

        //just remove the first available timing in the
collection
        var firstavailabletiming =
show.EventTimings.First();
        db.DeleteObject(firstavailabletiming);

//show.EventTimings.Remove(firstavailabletiming);
//removing it from the collection is not enough
you have to also
//delete eventtiming as well.
db.SaveChanges();
}

```

In the UpdateTradeShow method, I am retrieving a tradeshow based on name and along with the tradeshow; I am also bringing EventTimings collection by calling Include. Since I am loading EventTimings collection ahead of time, when our associationchange event fires and we call EventTimings.IsLoaded in the event, the value returned will be set to true since our collection have been already loaded. After loading the trade show, I am grabbing the first timing in the collection and deleting the timing by calling DeleteObject on the datacontext. This causes the eventtiming object to be removed from the trade show collection as well because an object that would no longer exist cannot be part of any relationship with other object such as trade show. Since eventtiming gets removed from the eventtimings collection, associationchanged event gets triggered with an action of Remove causing our Trade show's start and end date to get updated as well.

6.6 When does Association changed Event get fired.

Problem: You need to know what kind of operations on entity collection is performed that can cause Association changed event to fire.

Solution: There are various operations that you can perform which can trigger the collection changes. For instance adding item to collection, removing item from a collection, refreshing the collection, retrieving the collection from the

database, attaching item to the datacontext, attaching an item to the collection directly, detaching item from the datacontext and marking an item for deletion. Code below shows various examples when an association changed event fires and what kind of action is reported by CollectionChangedEvent arguments.

```
public static void TestForAssociationChangedEvents( )
{
    var db = new NorthwindEFEntities();
    Console.WriteLine("Following are cases when
association changed event fires");

        Console.WriteLine("Include in the query raises
an action of add");
        var THECR =
db.Customers.Include("Orders").First(c => c.CustomerID ==
"THECR");

        var TRAIH = db.Customers.First(c => c.CustomerID
== "TRAIH");
        var LAZYK = db.Customers.First(c => c.CustomerID
== "LAZYK");

        Console.WriteLine(@"Calling Load causes an
action of refresh only 1 time regardless of # of orders");
        TRAIH.Orders.Load();

        Console.WriteLine(@"Orders returned from query
belonging to a customer being tracked causes Add action");
        db.Orders.Where(o => o.ShipCity == "Walla
Walla").ToList(); //customer LAZYK

        Console.WriteLine(@"Rexecuting the same query
does not trigger any action.");
        db.Orders.Where(o => o.ShipCity == "Walla
Walla").ToList();

        Console.WriteLine("Calling Removes fires an
action of remove");
        var THECRorder = THECR.Orders.First();
        THECR.Orders.Remove(THECRorder);

        Console.WriteLine("Clearing all orders causes an
action of Refresh.");
```

```

        THECR.Orders.Clear();

        Console.WriteLine("Calling Add on Orders
Collection causes an action of Add.");
        THECR.Orders.Add(new Order { ShipCity =
"London", ShipCountry = "UK", OrderDate = DateTime.Today });

        var TRAIHorder = TRAIH.Orders.First(o =>
o.OrderID == 10574);
        Console.WriteLine(@"adding existing order causes
remove and than an add action.");
        THECR.Orders.Add(TRAIHorder);

        var LAZYKorder = db.Orders.First(o => o.OrderID
== 10482);
        Console.WriteLine("Detaching an order causes the
order to be removed from customer's order collection");
        db.Detach(LAZYKorder);
        Console.WriteLine("Calling Attach causes an
action of Add if the customer is already being tracked");
        db.Attach(LAZYKorder);

        var HANARorder = db.Orders.First(o => o.OrderID
== 10253);
        db.Detach(HANARorder);
        //calling attach does not cause an action of add
since no customer for the order was tracked.
        db.Attach(HANARorder);
        Console.WriteLine("Attaching orders to orders
collection causes an action of refresh");
        THECR.Orders.Attach(HANARorder);

        //Assigning HANARorder from customer THECR to
LAZYK
        Console.WriteLine("Assigning customer instance
on existing order causes action of remove and add");
        HANARorder.Customer = LAZYK;

        Console.WriteLine("Calling DeleteObject causes
an action of remove if customer is being tracked.");
        db.DeleteObject(HANARorder);

        Console.WriteLine("Calling delete on objectstate
entry causes an action of remove.");
db.ObjectStateManager.GetObjectStateEntry(LAZYKorder).Delete
();

```

```

    }

    void Orders_AssociationChanged(object sender,
System.ComponentModel.CollectionChangeEventArgs e)
{
    Console.WriteLine(e.Action.ToString());
}

```

Discussion: AssociationChanged event gets triggered when an action is performed on the collection. This allows the application developer to respond to the changes in the collection and also get to know what kind of change or action was performed on the collection. AssociationChanged event has one of its parameter CollectionChangedEventArgs which exposes two essential properties. One of the properties is the action property. Action property is an enum that has 3 different values.

1. Add
2. Remove
3. Refresh

When an item gets added to the collection, an action of Add is triggered. When an item is removed from the collection, the action of remove is triggered. If the collection gets refreshed from the database, an action of refresh is performed. Unlike Add and remove, Refresh is considered a batch event where Refresh is not raised for every item in the collection. In fact refreshed is called after entire collection is refreshed from the database.

Another property available on CollectionChangedEventArgs is Element that represents the object that got added or removed from the collection. It is important to realize that Association event is a changed event meaning the action has occurred and the collection has been modified whether it be Add or Delete; so if you access the collection inside the AssociationChangedEvent you will be working with a modified collection. Let's briefly walk through some of the examples in the above code and see why these actions triggers an association changed event.

Listing 1-1

```
Console.WriteLine("Include in the query raises an action of add");
```

```

var THECR =
db.Customers.Include("Orders").First(c => c.CustomerID ==
"THECR");

```

In Listing 1-1, I am loading customer and its orders at the same time. Once the query is performed the orders retrieved would be attached to the orders collection of the customers. This result in an action of Add being raised equal to the number of Orders the customer collection has. Using Include in a query is process called eager loading where you want to fetch additional entities with parent entity.

Listing 1-2

```

var LAZYK = db.Customers.First(c => c.CustomerID ==
"LAZYK");

Console.WriteLine(@"Calling Load causes an
action of refresh only 1 time regardless of # of orders");
TRAIH.Orders.Load();

```

In listing 1-2, we are calling load method to lazy load Orders for a customer. This raises collection changed event with action property set to Refresh. You would normally perform Load when you want to refresh the collection from the database. Refresh event is a batch event which gets raised only once and after the entire collection is refreshed from the store.

Listing 1-3

```

var LAZYK = db.Customers.First(c => c.CustomerID ==
"LAZYK");

Console.WriteLine(@"Orders returned from query belonging to
a customer being tracked causes Add action");
db.Orders.Where(o => o.ShipCity == "Walla
Walla").ToList(); //customer LAZYK

Console.WriteLine(@"Rexecuting the same query
does not trigger any action.");
db.Orders.Where(o => o.ShipCity == "Walla
Walla").ToList();

```

In listing 1-3, we are retrieving orders for the city of Walla Walla. The orders retrieved from the query, have orders that belong to the customer LAZYK. Since we are already tracking customer LAZYK, and when orders are found in the query that match the customer, entity framework automatically tries to add those orders to the customer collection. This causes the collection change event to fire with an action of Add for every order added to the collection. Repeating the same query again does not cause the collection to be modified because the orders returned from the query are already part of the customer collection.

Listing 1-4

```
Console.WriteLine("Calling Removes fires an action of remove");
var THECRorder = THECR.Orders.First();
THECR.Orders.Remove(THECRorder);

Console.WriteLine("Clearing all orders causes an action of Refresh.");
THECR.Orders.Clear();
```

In listing 1-4, I am getting the first order found for the customer THECR. After getting the reference to the order, I am removing the order from the orders collection of the customer by passing the reference to the order object. This action causes the orders collection to be altered causing AssociationChanged event to fire with an action of Remove. If I want to remove all the orders for a customer, you can call clear on the Orders collection. This causes an AssociationChanged event to get triggered. Since the operation performed is at the collection level, the event gets raised only once and the action property is set to refresh.

Listing 1-6

```
Console.WriteLine("Calling Add on Orders Collection causes an action of Add.");
THECR.Orders.Add(new Order { ShipCity =
"London", ShipCountry = "UK", OrderDate = DateTime.Today });

var TRAIHorder = TRAIH.Orders.First(o =>
o.OrderID == 10574);
```

```

Console.WriteLine(@"adding existing order causes
remove and than an add action.");
THECR.Orders.Add(TRAIHorder);

```

In listing 1-6, I am adding a brand new order to the orders collection of a customer. This causes collection changed event to fire because an order got added to the collection. However if you have an existing order that belongs to a customer, adding the order to new customer's order collection, would raise the collection changed event twice. First entity framework would remove the order from the existing customer causing an action of Remove and then add the order to the new customer's order collection causing action of Add.

Listing 1-7

```

var LAZYKorder = db.Orders.First(o => o.OrderID == 10482);
Console.WriteLine("Detaching an order causes the
order to be removed from customer's order collection");
db.Detach(LAZYKorder);
Console.WriteLine("Calling Attach causes an
action of Add if the customer is already being tracked");
db.Attach(LAZYKorder);

```

In Listing 1-7, I am detaching an order from the objectcontext, this removes all the tracking information and the existence of that order from ObjectStateManager, the object responsible for tracking objects in the ObjectContext. If the order being removed has an association to other objects such an order being part of customer that is also being tracked, the relationship between the customer and orders is also removed causing the collection changed event to fire with an action of Remove because order being detached is removed from the customer's collection. On attaching the same order back to the datacontext, entity framework checks the customer this order belongs to and if the customer this order belongs to is also being tracked in the ObjectContext, the order is automatically added to Orders collection of the Customer. This causes collection changed event to fire with an action of Add.

Listing 1-8

```

var HANARorder = db.Orders.First(o => o.OrderID == 10253);
db.Detach(HANARorder);

```

```

    //calling attach does not cause an action of add
    since no customer for the order was tracked.
    db.Attach(HANARorder);
    Console.WriteLine("Attaching orders to orders
collection causes an action of refresh");
    THECR.Orders.Attach(HANARorder);

```

In listing 1-8, I am detaching and attaching an order from the datacontext. This action does not cause any event to occur because the customer the order belongs to is not being tracked by the datacontext. After calling attach on an order that was earlier detached, the order is being tracked by the datacontext but does not belong to any customer. To tell the datacontext that we want the order to be attached to customer THECR, I am explicitly calling attach on the orders collation of THECR customer. This action causes association changed event to get triggered with an action of refresh.

Listing 1-9

```

//Assigning HANARorder from customer THECR to LAZYK
Console.WriteLine("Assigning customer instance
on existing order causes action of remove and add");
HANARorder.Customer = LAZYK;

```

In listing 1-9, I am setting the customer object on an existing order to point to a new customer that is already being tracked in the datacontext. This is an action where we are indicating the order belongs to a new customer instead of old customer. This causes assocaitonchanged event to fire twice first with an action of remove to remove the order from the old customer, and preceded by Add to add the order to the new customer.

Listing 1-10

```

Console.WriteLine("Calling DeleteObject causes
an action of remove if customer is being tracked.");
db.DeleteObject(HANARorder);

Console.WriteLine("Calling delete on objectstate
entry causes an action of remove.");

db.ObjectStateManager.GetObjectStateEntry(LAZYKorder).Delete
();

```

In listing 1-10, I am marking an order for deletion. Since the order belongs to a customer that is also tracked by the datacontext, this causes the order to be

removed from the customer's orders collection. On removing the order from customer's order collection, association changed event is fired with an action of Remove. You also have the ability to access the objectstateentry from objectstatemanager that is managing the Order tracking by calling GetObjectStateEntry passing in the reference to the order object. After getting the reference to the ObjectStateEntry, you can delete the entry which also causes AssociationChanged event to fire with an action of remove.

6.7 Complex Types

Problem: You have customer and employee table that contains address information. In entity data model you would like to structure the address information into its own class that can be reused with both customer and employee's address.

Solution: Entity framework supports the concept of Complex Type that can group related properties in a single composite class. For instance city, country and zip can appear as address information for multiple tables. Instead of declaring those fields as properties for every entity, you can create a single complex type Address that contains all these fields and reuse the Address object with various other entities that contains address related info. To consume a complex type in entity framework, you have to define the complex type in the conceptual model and then use the complex type as the property for the instances that contains address information. After defining the conceptual modal, you have to map each complex defined on an entity to the columns defined in the store. Example below shows how schema required to use conceptual model.

Conceptual Modal

```
<?xml version="1.0" encoding="utf-8" ?>
<Schema Namespace="NWComplexTypeModel" Alias="Self"
xmlns="http://schemas.microsoft.com/ado/2006/04/edm">
  <EntityContainer Name="NWComplexTypeEntities">
    <EntitySet Name="Employees"
EntityType="NWComplexTypeModel.Employee" />
```

```

<EntitySet Name="Customers"
EntityType="NWComplexTypeModel.Customer" />
</EntityContainer>
<EntityType Name="Employee">
<Key>
<PropertyRef Name="EmployeeID" />
</Key>
<Property Name="EmployeeID" Type="Int32"
Nullable="false" />
<Property Name="LastName" Type="String"
Nullable="false" />
<Property Name="FirstName" Type="String"
Nullable="false" />
<Property Name="Address"
Type="NWComplexTypeModel.CommonAddress" Nullable="false" />
</EntityType>
<EntityType Name="Customer">
<Key>
<PropertyRef Name="CustomerID" />
</Key>
<Property Name="CustomerID" Type="String"
Nullable="false" />
<Property Name="CompanyName" Type="String"
Nullable="false" />
<Property Name="Address"
Type="NWComplexTypeModel.CommonAddress" Nullable="false" />
</EntityType>
<ComplexType Name="CommonAddress">
<Property Name="Address" Type="String" />
<Property Name="City" Type="String" />
<Property Name="Region" Type="String" />
<Property Name="PostalCode" Type="String" />
<Property Name="Country" Type="String" />
</ComplexType>
</Schema>
```

Mapping from conceptual to store

```

<?xml version="1.0" encoding="utf-8" ?>
<Mapping Space="C-S" xmlns="urn:schemas-microsoft-
com:windows:storage:mapping:CS">
<EntityContainerMapping StorageEntityContainer="dbo" 

CdmEntityContainer="NWComplexTypeEntities">
<EntitySetMapping Name="Customers">
<EntityTypeMapping
TypeName="NWComplexTypeModel.Customer" >
```

```

        <MappingFragment StoreEntitySet="Customers">
            <ScalarProperty Name="CustomerID"
                ColumnName="CustomerID" />
            <ScalarProperty Name="CompanyName"
                ColumnName="CompanyName" />
            <ComplexProperty Name="Address"
                TypeName="NWComplexTypeModel.CommonAddress">
                <ScalarProperty Name="Address"
                    ColumnName="Address" />
                <ScalarProperty Name="City"
                    ColumnName="City" />
                <ScalarProperty Name="Region"
                    ColumnName="Region" />
                <ScalarProperty Name="PostalCode"
                    ColumnName="PostalCode" />
                <ScalarProperty Name="Country"
                    ColumnName="Country" />
            </ComplexProperty>
        </MappingFragment>
    </EntityTypeMapping>
</EntityTypeMapping>
<EntityTypeMapping Name="Employees">
    <EntityTypeMapping
        TypeName="NWComplexTypeModel.Employee">
        <MappingFragment StoreEntitySet="Employees">
            <ScalarProperty Name="EmployeeID"
                ColumnName="EmployeeID" />
            <ScalarProperty Name="LastName"
                ColumnName="LastName" />
            <ScalarProperty Name="FirstName"
                ColumnName="FirstName" />
            <ComplexProperty Name="Address"
                TypeName="NWComplexTypeModel.CommonAddress">
                <ScalarProperty Name="Address"
                    ColumnName="Address" />
                <ScalarProperty Name="City"
                    ColumnName="City" />
                <ScalarProperty Name="Region"
                    ColumnName="Region" />
                <ScalarProperty Name="PostalCode"
                    ColumnName="PostalCode" />
                <ScalarProperty Name="Country"
                    ColumnName="Country" />
            </ComplexProperty>
        </MappingFragment>
    </EntityTypeMapping>
</EntityTypeMapping>

```

```

</EntityContainerMapping>
</Mapping>

```

Discussion: To fully understand how conceptual model work, we will walk through each of the mapping file to understand how the complex types can be reused across multiple entities and mapped to different table for a given entity. The conceptual model defines an entity container more like a repository where our collection resides. Entity container contains two EntitySets or collections Customers and Orders as shown in the code

```

<EntityContainer Name="NWComplexTypeEntities">
    <EntitySet Name="Employees"
EntityType="NWComplexTypeModel.Employee" />
    <EntitySet Name="Customers"
EntityType="NWComplexTypeModel.Customer" />
</EntityContainer>

```

Each EntitySet contains an entity type which is more like table containing rows. In our example Employees entity set contains Employee entity and Customers entity set contains Customer entity as shown below.

```

<EntityType Name="Employee">
    <Key>
        <PropertyRef Name="EmployeeID" />
    </Key>
    <Property Name="EmployeeID" Type="Int32"
Nullable="false" />
    <Property Name="LastName" Type="String"
Nullable="false" />
    <Property Name="FirstName" Type="String"
Nullable="false" />
    <Property Name="Address"
Type="NWComplexTypeModel.CommonAddress" Nullable="false" />
</EntityType>
<EntityType Name="Customer">
    <Key>
        <PropertyRef Name="CustomerID" />
    </Key>
    <Property Name="CustomerID" Type="String"
Nullable="false" />
    <Property Name="CompanyName" Type="String"
Nullable="false" />
    <Property Name="Address"
Type="NWComplexTypeModel.CommonAddress" Nullable="false" />
</EntityType>

```

In the above code snippet, both Employee and Customer use a complex Type Common Address to assign to their Address property. This frees up each class from having to declare separate properties for their address information as defined on the database. On the schema you also have to option to control the visibility of the Address type property to various options such as public, private, protected or internal. Since we are using the complex property, we must declare the complex property as well. Schema below declares a complex type

```
<ComplexType Name="CommonAddress">
    <Property Name="Address" Type="String" />
    <Property Name="City" Type="String" />
    <Property Name="Region" Type="String" />
    <Property Name="PostalCode" Type="String" />
    <Property Name="Country" Type="String" />
</ComplexType>
```

In the above definition of the complex type, we are declaring scalar value properties such as Address, City, Region, Postal code and Country. A complex property can have other complex properties within it. Currently in the v1 release of the entity framework, Complex Type cannot contain navigation properties such as entity refs and entity collection. Since complex type has no notion of entity key or has an identity of its own, it cannot be tracked in the object state manager. Complex types cannot stand alone and it has to be associated to an entity to be useful. Even though the identity of complex type is tied to identity of the entity it is part of, you can still put general validation rules on the complex type that will affect all entities using the at complex type.

After creating the complex type in the conceptual model, we need to map the property on the entity that exposes complex type to columns defined on our table. Because the mapping is pretty much same for both Customer and Employee, I will just cover the mapping for Customer.

```
<EntitySetMapping Name="Customers">
    <EntityTypeMapping
TypeName="NWComplexTypeModel.Customer">
        <MappingFragment StoreEntitySet="Customers">
```

```

        <ScalarProperty Name="CustomerID"
ColumnName="CustomerID" />
        <ScalarProperty Name="CompanyName"
ColumnName="CompanyName" />
            <ComplexProperty Name="Address"
TypeName="NWComplexTypeModel.CommonAddress">
                <ScalarProperty Name="Address"
ColumnName="Address" />
                <ScalarProperty Name="City"
ColumnName="City" />
                <ScalarProperty Name="Region"
ColumnName="Region" />
                <ScalarProperty Name="PostalCode"
ColumnName="PostalCode" />
                <ScalarProperty Name="Country"
ColumnName="Country" />
            </ComplexProperty>
        </MappingFragment>
    </EntityTypeMapping>
</EntitySetMapping>

```

On the above mapping, I am mapping entity Customer to store entity set Customers. Scalars properties on Customer are pretty simple and they map 1 to 1 with columns in our database. Since address property contains complex type, we cannot directly map that to the store. Instead we map individual properties on the complex type to columns in our database. Similarly for Employee's Address property, scalar properties would map to columns defined on employee table.

Current release of entity designer does not support complex types. If you manually change the conceptual and mapping model you will lose the ability to re open the edmx file in the designer.

To re emphasize, Complex types are not same as entities. Complex types do not have independent identifiers and is associated as a property on an entity that is tracked by entity framework. Given this fact, you cannot expose any ends of an association or relation as a complex type because complex type has no key and therefore cannot be tracked separately. In future versions of entity framework, a complex type may be allowed to be part of an association.

After defining the conceptual, mapping and store model, you can run edmgene utility to generate your class files which will generate 3 classes, Customer, Employee and Object Context that exposes your entity set. Looking at generated code for Common Address Complex, you will see that it inherits from Complex Object. Complex Object is an object with no key as compared

to Entity Object from which entity derives that uses entity key to track an entity. After generating the model, we can use the complex type in our query, read values from complex type and update columns in our table by modifying values on our complex type property. Example below shows various usages of complex type with insert and read scenarios.

```
var db = new NWComplexTypeEntities();
Console.WriteLine("First 2 customer address
info");
foreach (var cus in db.Customers.Take(2))
{
    Console.WriteLine("City:{0}
Zip:{1}", cus.Address.City, cus.Address.PostalCode);
}
Console.WriteLine("\r\nFirst 2 employee address
info");
foreach (var emp in db.Employees.Take(2))
{
    Console.WriteLine("City:{0} Zip:{1}",
emp.Address.City, emp.Address.PostalCode);
}
```

In the example above, we are taking first 2 customers in our list and printing address for those customers by accessing its address property. Reading values from complex property is not any different from reading values from other objects defined on the CLR. Since we have defined in our conceptual modal that address property cannot be null, you will always have a valid reference to address object that may not have any values defined for its scalar properties.

To insert a customer and its complex type, you simply create an instance of Customer object and nested within it you can create an instance of CommonAddress object assigning it to address property of the customer. You can use object initialize syntax to assign values to Address object. Example below shows how to create a customer and Address object and then adding the object to the object context followed by submitchanges.

```
//insert customer
var customer = new NWComplexTypeModel.Customer
{
    CustomerID = "COML1",
    CompanyName = "Test Company",
    Address = new CommonAddress
    {
        Address = "address 1",
        City = "New York",
        PostalCode = "10001"
    }
}
```

```

        City = "Euless",
        Country="USA"
    }
};

db.AddToCustomers(customer);
db.SaveChanges();

```

Although in the above example, you explicitly created an instance of Common Address, you do not have to necessarily do so because accessing the address object first time will ensure the instantiation of CommonAddress object. Then all you have left to do is assign values to the scalar properties of the CommonAddress class. Code below shows assigning values to address property without creating an instance of Common Address class.

```

//insert employee
var employee = new Employee
{
    FirstName = "Zeeshan",
    LastName = "Hirani"
};
employee.Address.Address = "address 1";
employee.Address.City = "Euless";
employee.Address.Country = "USA";
db.AddToEmployees(employee);
db.SaveChanges();

```

You can also use Complex Types in queries to find a match with a specific value defined on the complex type. Code below retrieves the first customer with a city of Euless and country of USA. To specify the expression filter, I am navigating the Address object and accessing its properties.

```

//querying using complex type.
var cusquery = db.Customers.First(c =>
c.Address.City == "Euless" && c.Address.Country == "USA");
Console.WriteLine("Using Complex type in
queries");
Console.WriteLine(cusquery.CustomerID);

```

As I talked earlier, you can also nest complex types within a complex type. Example below shows CommonAddress a complex type has property AdditionalInfo that maps to Geography class that is also a complex type. Mapping the nested complex is also same where further column mapping are performed inside AdditionalInfo property which is nested inside of Address property.

CSDL

```
<ComplexType Name="CommonAddress">
    <Property Name="Address" Type="String" />
    <Property Name="Region" Type="String" />
    <Property Name="AdditonalInfo" Nullable="false" Type="NWNestedComplexTypeModel.Geography" />
</ComplexType>
<ComplexType Name="Geography">
    <Property Name="City" Type="String" />
    <Property Name="PostalCode" Type="String" />
    <Property Name="Country" Type="String" />
</ComplexType>
```

Mapping

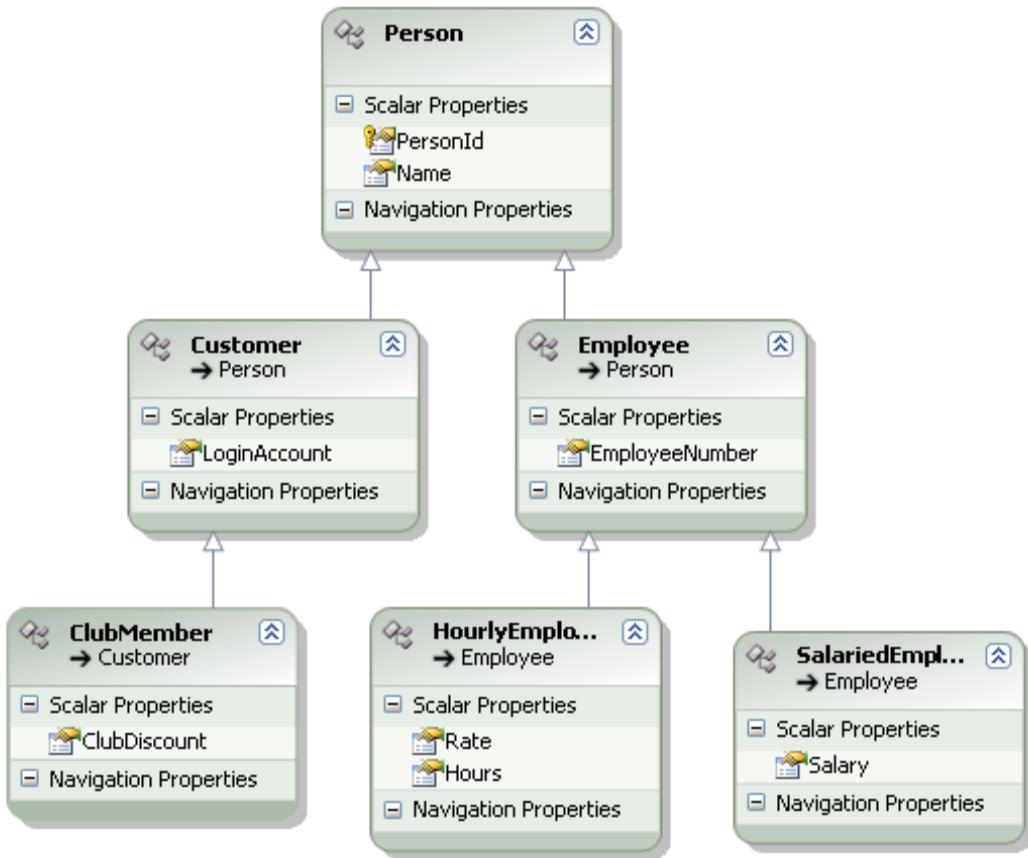
```
<ComplexProperty Name="Address"
    TypeName="NWNestedComplexTypeModel.CommonAddress">
    <ScalarProperty Name="Address"
        ColumnName="Address" />
    <ScalarProperty Name="Region"
        ColumnName="Region" />
    <ComplexProperty Name="AdditonalInfo"
        TypeName="NWNestedComplexTypeModel.Geography">
        <ScalarProperty Name="City"
            ColumnName="City" />
        <ScalarProperty Name="PostalCode"
            ColumnName="PostalCode" />
        <ScalarProperty Name="Country"
            ColumnName="Country" />
    </ComplexProperty>
</ComplexProperty>
```

Similarly you can use nested complex type inside of a query to filter the results. Code below searches for customer with address that contains Obere and country of Germany. To do this we are navigating both levels of the complex query to apply the filter.

```
var db = new NWNestedComplexTypeModel.NWComplexTypeEntities();
    var cus = db.Customers.First(c =>
c.Address.Address.Contains("Obere Str") &&
c.Address.AdditonalInfo.Country == "Germany");
    Console.WriteLine(cus.Address.AdditonalInfo.City);
```

6.8 Accessing derived types from ObjectContext

Problem: Figure below shows EDM model for Employee and customer using Table Per Hierarchy inheritance.



When you access the generated objectcontext, there is only Persons ObjectQuery exposed. You want the ability to directly access any derived type of person such customer, employee, club member etc.

Solution: When inheritance hierarchy is created using the designer, EF by default names the entity set based on the base type of the hierarchy. Then each derived type become part of the same entityset. The generated object context does not expose an entity; in fact `ObjectContext.Persons` retrieves all the entities that are within the entity set Persons that includes Customer, Employee, Club Member, Hourly Employee and SalariedEmployee. The return type although is the base type Person. To access properties specific to a derived type, the base type needs to be casted to a specific derived type in order to access its properties. To expose each derived type from the ObjectContext, we need to extend the ObjectContext with a partial class and expose properties that return each derived type. To return a specific derived

type, we can use OfType operator passing in the derived type we need from the Persons entity.

Although the designer enforces that each entity with in an inheritance hierarchy be part of the same entityset, EF does not force this restriction. You can define each entity in a hierarchy to be on a separate entityset. The Msl needs to be configured manually to map each entity in a different entity set with the table defined on the store. Once every entity is defined in a separate entityset, the generated objectcontext will expose derived entities as well.

Discussion: To use the default option where the base entity and the derived types are part of the same entityset, we need to create a partial class for the ObjectContext and add properties that return each derived type from the Person entityset. Code below shows partial class for the ObjectContext that has properties to return each derived entities.

```
public partial class EmployeeTPHTPT
{
    public ObjectQuery<Customer> Customers
    {
        get{return this.Persons.OfType<Customer>();}
    }
    public ObjectQuery<Employee> Employees
    {
        get { return this.Persons.OfType<Employee>(); }
    }
    public ObjectQuery<ClubMember> ClubMembers
    {
        get { return this.Persons.OfType<ClubMember>(); }
    }
    public ObjectQuery<HourlyEmployee> HourlyEmployees
    {
        get { return this.Persons.OfType<HourlyEmployee>(); }
    }
    public ObjectQuery<SalariedEmployee> SalariedEmployees
    {
        get { return this.Persons.OfType<SalariedEmployee>(); }
    }
}
```

To access each derived type, I am using OfType operator. OfType operator not only returns the specific derive types, but references are also of derived types which saves conversion from base type to derive type.

7. Improving Entity framework performance

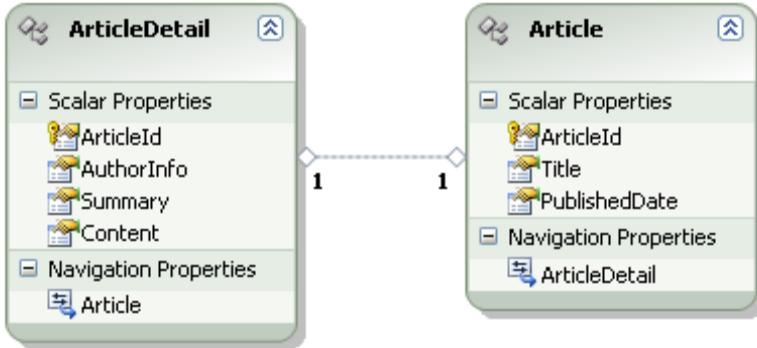
7.1 Delay Loading Expensive Fields on a Table

Problem: Figure below shows the Article table structure

Article (rs)			
	Column Name	Data Type	Allow Nulls
Y	ArticleId	int	<input type="checkbox"/>
	Title	varchar(50)	<input type="checkbox"/>
	PublishedDate	date	<input type="checkbox"/>
	AuthorInfo	varchar(MAX)	<input type="checkbox"/>
	Summary	varchar(MAX)	<input type="checkbox"/>
	[Content]	varchar(MAX)	<input type="checkbox"/>
			<input type="checkbox"/>

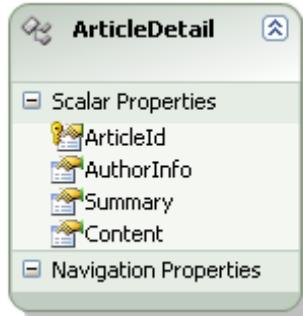
On the above article table, AuthorInfo, Summary and Content are fairly large field which are not always required when querying for article record. You want to be able to delay the load of these fields unless you specifically ask to have these fields loaded.

Solution: Unlike linq to sql, EF does not support the concept of delay loading certain properties on an entity. But Ef does support the concept of delay loading an association on an entity. To solve the above problem we can create ArticleDetail entity and move the expensive fields to ArticleDetail entity. Then create 1 to 1 association between article and Article Detail entity. Now when we retrieve Article entity, we would be only retrieving properties on the Article entity and unless we use Include operator or explicitly call load on ArticleDetail, we won't be fetching additional fields defined on ArticleDetail entity. The completed EDM model should like below

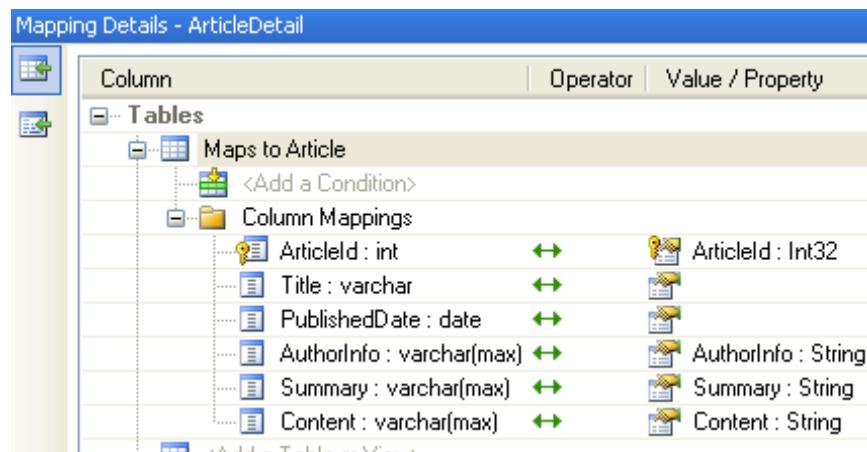


Discussion: Mapping a single table to multiple entities is not fully supported in the designer. We can start with the designer but have to add Referential constraint on the conceptual model manually by editing the xml. The next version of Ef will fully support this scenario not requiring you to edit the edmx file manually. Steps below outline the process of moving over expensive fields to ArticleDetail entity.

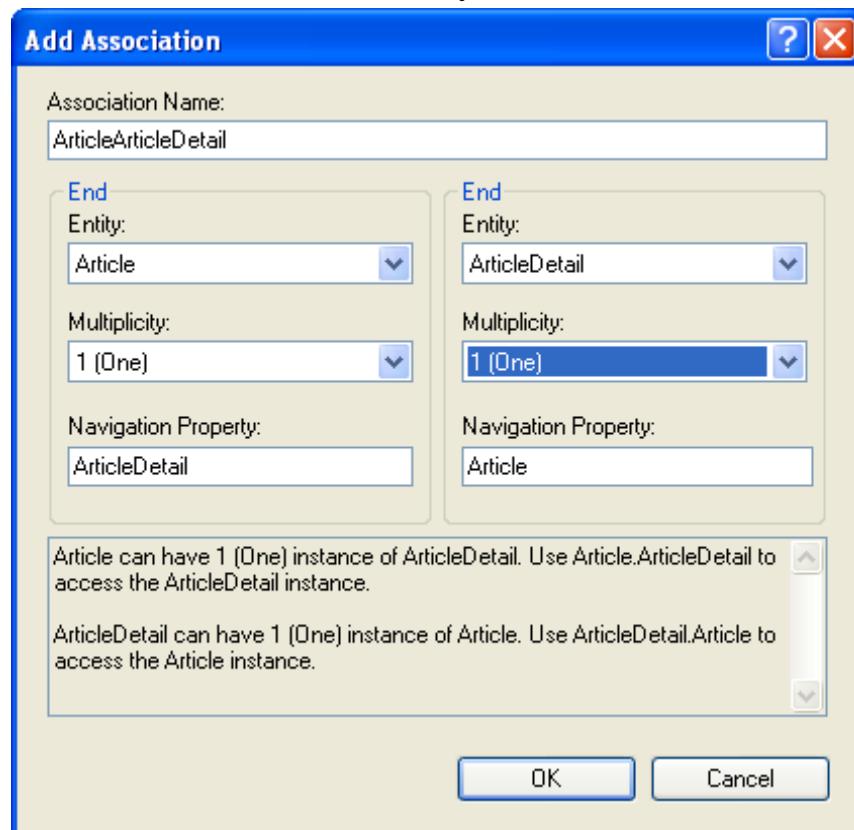
1. Import Article table using EDM wizard.
2. Create a second entity ArticleDetail and make ArticleId as the entity key. Move AuthorInfo, Summary and Content property from Article entity to Article detail. Figure below shows the ArticleDetail entity.



3. Map ArticleDetail entity to Article table and map corresponding properties on their respective columns in article table. Figure below shows the mapping window for ArticleDetail entity.



4. Create 1 to 1 association between Article and ArticleDetail entity. For splitting a table to multiple entities, association between Article and ArticleDetail must be set to 1 to 1. It cannot be 1 to 1..0 because additional fields on ArticleDetails are required fields on article table which mandates that any time an Article entity is instantiated, ArticleDetail should also be created. Failing to do so would not cause the model to validate. Figure below shows the association window for Article and ArticleDetail entity.



5. Map the association created above to Article table and entity keys to ArticleId column as shown below.



6. Open edmx file in xml and modify the association definition on the conceptual model to include referential constraint as shown below.

```
<ReferentialConstraint>
    <Principal Role="Article">
        <PropertyRef Name="ArticleId" /></Principal>
    <Dependent Role="ArticleDetail">
        <PropertyRef Name="ArticleId" />
    </Dependent></ReferentialConstraint>
```

To test the above model we can query for all articles and confirm that query send to the database only retrieved properties defined on Article entity. Code below shows an example retrieving articles from the database.

```
var db = new DelayLoadingEntities();
foreach (var article in db.Articles)
{
    Console.WriteLine("Title:{0} Publish Date:{1}",
article.Title, article.PublishedDate.ToString("d"));
}
```

The sql capture below confirms that querying for Article entity only retrieves the Title, PublishDate and ArticleId from Article table.

```
SELECT
1 AS [C1],
[Extent1].[Title] AS [Title],
[Extent1].[PublishedDate] AS [PublishedDate],
[Extent1].[ArticleId] AS [ArticleId]
FROM [dbo].[Article] AS [Extent1]
```

To bring additional fields on cases when we need it we can use Include operator to eagerly fetch ArticleDetail entity. Code below shows an example of that.

```
var db = new DelayLoadingEntities();
```

```

foreach (var article in db.Articles.Include("ArticleDetail"))
{
    Console.WriteLine("Title:{0} Publish Date:{1} Content:{2}",
        article.Title,
        article.PublishedDate.ToString("d"), article.ArticleDetail.Content);
}

```

Sql capture below confirms that having an Include in linq query causes Summary, Content and AuthorInfo fields to be also brought along with other article fields.

```

SELECT
1 AS [C1],
[Extent1].[ArticleId] AS [ArticleId],
[Extent1].[Title] AS [Title],
[Extent1].[PublishedDate] AS [PublishedDate],
1 AS [C2],
[Extent1].[AuthorInfo] AS [AuthorInfo],
[Extent1].[Summary] AS [Summary],
[Extent1].[Content] AS [Content]
FROM [rs].[Article] AS [Extent1]

```

7.1 GetObjectByKey vs First Operator

Problem: You want to know the different ways to retrieve an entity using primary key and what benefits one way offers over the other.

Solution: There are a couple of different options to retrieve an entity using primary key column. You can either query using the First operator or use GetObjectByKey. When you use First operator, there is always a database call made to retrieve the object. When using ObjectByKey, the object is first searched in ObjectStateManager service to see if it has been tracked earlier. If the object is found in the cache, there is no database hit made and entity is simply returned from cache. When object is not found, a database call is made to retrieve the object and object is stored in cache for next retrieval.

Following code shows an example of using First and GetObjectByKey methods.

```

var db = new NorthwindEFEntities();
    //using linq syntax
var ALFKI = db.Customers.First(c => c.CustomerID == "ALFKI");
Console.WriteLine("ALFKI returned using first operator " +
ALFKI.CustomerID);

```

```

    //create an entity key
    EntityKey key = new
EntityKey("NorthwindEFEntities.Customers", "CustomerID", "ALFKI");
    //if object is found in the cache use that otherwise get it
from database.
    var ALFKI2 = db.GetObjectByKey(key) as Customer;
    Console.WriteLine("Object retrieved from the cache " +
ALFKI2.CustomerID);

```

Discussion: If you had been using Linq to Sql, you may be wondering why I did not mention the use of Single operator. Unfortunately using Single operator is not supported. If you run a query using Single operator, you would get an exception stating that Single operator is not supported and consider using First operator instead. This leaves you with two concrete options First and GetObjectByKey. In the above example, I am calling First operator passing in my primary key column value to get the customer. If the query returns no data, you will get an exception. If you are not sure that your query will match any record in the database, than you should use FirstOrDefault operator. Code below shows an example of using FirstOrDefault operator.

```

private void UsingFirstOrDefault()
{
    var db = new NorthwindEFEntities();

    var cust = db.Customers.FirstOrDefault(c => c.CustomerID ==
"ABCDE");
    //no customer exists with customerid ABCDE
    Console.WriteLine("Customer found {0}", cust != null);
}

```

The above code does not raise any exception even though customerid ABCDE does not exist because we are calling FirstOrDefault. If the query results do not find a match, you get a null object instead of getting exception. You can check for null object reference to do you logic.

In Linq to Sql, if you use Single or First operator, it is optimized for caching. For instance if you retrieve an object using First or Single operator, the object gets cached in object tracking service. Next time when you query using First or Single for the same customer, Linq to Sql does not go to database if it finds the object in the cache. With entity framework, every time you use First operator, the request will go to the database without looking in the cache. After the object is retrieved from the database, it will then look in the cache to see if there is an object being tracked that has the same key that was fetched

from the database. If there is a matching key found in the cache, the object from the cache is returned instead. If match is not found in the cache, the object is than stored in the cache for tracking and also returned to the user. In the code below, I am calling the First operator twice which happens to make two database calls but when I compare the object references, they are returned same. This confirms that once an object is stored in the cache, the same object is always returned.

```
var db = new NorthwindEFEntities();
var ALFKI = db.Customers.First(c => c.CustomerID == "ALFKI");
//second database call. does not retrieve from cache.
var ALFKI2 = db.Customers.First(c => c.CustomerID ==
"ALFKI");
//object references are same.
Console.WriteLine("Is reference same {0}", ALFKI == ALFKI2);
```

If you do not want to incur a database hit on subsequent calls when querying for an entity using primary key, than you can use GetObjectByKey method available on the ObjectContext class. GetObjectByKey first checks to see if the object is available in the cache, if the object is found matching the primary key, object is returned to the user; otherwise a database call is made. Like First operator, GetObjectByKey also tracks the object after the first call to GetObjectKey so that later same object can be returned. Similar to first operator, if query does not return any match, GetObjectByKey will throw exception. If you know ahead of time that an entity may not exist in the database for a given primary key value, than you can use TryGetObjectByKey which returns a null reference if the query does not return any results. In the code, I am showing how to use GetObjectByKey and TryGetObjectByKey which returns a null reference since there is no customer found with customerid ABCDE.

```
//create an entity key
EntityKey key = new
EntityKey("NorthwindEFEntities.Customers", "CustomerID", "ALFKI");
//if object is found in the cache use that otherwise get it
from database.
var ALFKI2 = db.GetObjectByKey(key) as Customer;
Console.WriteLine("Object retrieved from the cache " +
ALFKI2.CustomerID);

EntityKey notfoundkey = new
EntityKey("NorthwindEFEntities.Customers", "CustomerID", "ABCDE");
Object notfound = null;
db.TryGetObjectByKey(notfoundkey, out notfound);
```

```
Console.WriteLine("Customer found {0}" ,notfound != null);
```

If you retrieve an object using GetObjectByKey and later detach the object from the datacontext, subsequent call to GetObjectByKey will hit the database because detaching removes the object from the cache. If you Attach an object to the context, the object is marked for tracking and therefore calling GetObjectByKey will not hit the database. Similarly objects marked for deletion can also be fetched from the cache when using GetObjectByKey since they are in deleted state but not yet deleted. If you create a new instance of an object and add the object to ObjectContext, from this point onwards the object is tracked in the cache. Calling GetObjectByKey will return the object is Added state. However if you were to query for an object in Added state using first operator, you will get an exception because the object does not exist in the database. Code below shows various examples of Using Detach and Attach, DeleteObject and new customer and how it effects GetObjectByKey in determining if the query can be fetch from the cache or a database call must be issued.

```
var db = new NorthwindEFEntities();
    var ALFKI = db.Customers.First(c => c.CustomerID == "ALFKI");
    /////detaching the object would remove the object from
tracking
    /////so query will hit the database again.
    db.Detach(ALFKI);
    var ALFKI3 = db.GetObjectByKey(ALFKI.EntityKey) as Customer;

    /////if you attach it back again then query would not hit the
database.
    db.Attach(ALFKI3);
    /////does not hit the database
    db.GetObjectByKey(ALFKI3.EntityKey);

    db.DeleteObject(ALFKI3);
    /////can retrieve objects in deleted state
    db.GetObjectByKey(ALFKI3.EntityKey);

    Customer cus = new Customer { CustomerID = "12345", City =
"London" };
    db.AddToCustomers(cus);
    //can be called on objects in added state
    db.GetObjectByKey(cus.EntityKey);

    //raises exception since object is created but not yet
inserted
    //db.Customers.First(c => c.CustomerID == "12345");
```

If you initially query for objects with No Tracking turned on, the objects are not stored in the cache and therefore calling GetObjectByKey requires a database call to fetch the object needed. However No Tracking option on the query is not honored using First operator and the object is marked for tracking and cached, subsequent call to GetObjectKey will return the object from the cache. In the code below, I am calling first operator with no tracking option but ALFKI customer still gets tracked in the context and therefore when I call GetObjectByKey, object is returned from the cache. When I search for customers in city of London using where operator with No tracking the objects do not get stored in cache and therefore calling makes database call.

```
var db = new NorthwindEFEntities();
    var custs = db.Customers;
    custs.MergeOption = MergeOption.NoTracking;
    //object is tracked despite no tracking
    var cus1 = custs.First(c => c.CustomerID == "ALFKI");
    //query does not hit the database because object is cached.
    db.GetObjectByKey(cus1.EntityKey);

    var results = db.Customers;
    results.MergeOption = MergeOption.NoTracking;
    //objects not tracked.
    results.Where(c => c.City == "London").ToList();
    //call issued to the database.
    var fromcache = db.GetObjectByKey(new
EntityKey("NorthwindEFEntities.Customers", "CustomerID", "BSBEV"));
```

7.2 Retrieving read-only entities using MergeOption.NoTracking

Problem: You need to know how to retrieve read-only entities in most efficient way.

Solution: EF allows several ways to execute a query and retrieve data as entities. By default all entities are tracked inside ObjectStateManager. If you want to retrieve an entity only for read only purposes, the best approach is to use MergeOption.NoTracking query. This will reduce the time to execute the query because EF would not have to perform identity resolution and determine if the entity is already tracked by the statemanager then return the old entity instead of new one. When entities are returned using default query option the original state of the entity is maintained inside of state manager and

this increases the memory foot print of the state manager. When queries are returned using NoTracking, there is no overhead for the memory as entities are directly returned by the objectcontext and no original values are stored. Entities returned using NoTracking query option have a state of Detached in the object state manager. Any entity in a detached state cannot leverage modification and delete features provided by the object context. To make changes to entity in detached state, the entity must be attached to the object context so it can begin tracking on the modifications made to the entity. If there are changes made to entity before it is attached, those changes will not be seen in the database.

When an entity is retrieved using NoTracking option, EF does not retrieve its related foreign key reference entity key. For instance to access the customerid for an order, we can write Order.CustomerReference.EntityKey.Values[0] to access the customerid. However in the case of NoTracking query, CustomerReference.EntityKey would be set to null.

Since NoTracking query does not hold the reference to the entity returned from the object context, identity resolution cannot be performed. This could have serious side effects in the application. For instance if you retrieved a customer with id of 1 using NoTracking and then later down made a request to retrieve the same customer again, you will be returned a completely new instance of the customer with the same customerId. If you would be doing object comparisons in your application code to check if two customers are same, the result would be false although both customers would have the same entity key.

Discussion: There are several ways to execute a query using NoTracking option. Code below shows some of the ways.

```
var db = new NorthwindEFEntities();
    //object query
    db.Customers.MergeOption = MergeOption.NoTracking;
    var custquery = db.Customers.Where(c => c.City ==
"London").Take(2);
    var custs1 =
((ObjectQuery<Customer>)custquery).Execute(MergeOption.NoTracking);

    //using esql
    string esql = "select value c from customers as c where
c.CustomerID = 'ALFKI'";
```

```

db.CreateQuery<Customer>(esql).Execute(MergeOption.NoTracking).First();

    //entity reference
    order.CustomerReference.Load(MergeOption.NoTracking);

    //entity collection
    category.Products.Load(MergeOption.NoTracking);

```

On the above query where I am returning customers in the city of London, I am applying MergeOption.NoTracking on the Customers ObjectQuery exposed on the object context. This tells EF that any query forward will use NoTracking Option.

To retrieve ALFKI customer, I am passing esql query to CreateQuery method on the object context. CreateQuery method returns an Object Query on which we are calling execute passing in NoTracking option to return ALFKI customer.

MergeOption can also be called in cases where you have to either lazy load collection or an entity reference. For instance in the above example, to retrieve customer for an order, I am calling Load method. If you do not pass any MergeOption parameter, the default option is MergeOption.AppendOnly which essentially means every entity retrieved will be tracked by the ObjectStateManager. To indicate that we need NoTracking on the query, I am using overloaded version of the Load method passing in MergeOption.NoTracking. The same behavior is also available when retrieving collections in a lazy loaded fashion like in the above case of Products for a category.

MergeOption can be only applied at the query level. There is no option to specify that all queries in the object context will use merge option. When MergeOption is set on the ObjectQuery exposed on the Objectcontext, any consecutive queries will also use the same MergeOption. Code below shows an example of that.

```

var db = new NorthwindEFEntities();
    db.Customers.MergeOption = MergeOption.NoTracking;
    var custs = db.Customers.Where(c => c.City == "London");

        var salesrep = db.Customers.Where(c => c.ContactTitle ==
"Sales Representative");

```

On the queries above, we are setting the MergeOption once on the Customers ObjectQuery. Both queries deriving from ObjectQuery of Customers end up using

the same MergeOption. This is because Customers query reference is cached in private variable. Code below shows the generated code for Customers query in the object context

```
private global::System.Data.Objects.ObjectQuery<Category> _Categories;
    /// <summary>
    /// There are no comments for Customers in the schema.
    /// </summary>
    public global::System.Data.Objects.ObjectQuery<Customer> Customers
    {
        get
        {
            if ((this._Customers == null))
            {
                this._Customers =
base.CreateQuery<Customer>("[Customers]");
            }
            return this._Customers;
        }
    }
```

Since new queries are built using the Customers Object Query, all queries inherit the same mergeOption. If this is not the desired effect, you can create a new instance of ObjectQuery using CreateQuery method. Code below shows an example.

```
var customers = db.CreateQuery<Customer>("Customers");
customers.MergeOption = MergeOption.NoTracking;
customers.Where(c => c.City == "London");
```

EF framework requires that related entities be retrieve using the same MergeOption. For instance if we load customer using default options which is AppendOnly, customer entity will be tracked with ObjectStateManager. Later if you want to lazy load Orders for a customer, Orders entity cannot be loaded using NoTracking Option. The query would compile fine but you will get a runtime exception that related entities cannot be loaded with Notracking option unless the original entity does not use the same option. Example below is an incorrect usage or MergeOption that raises error!

```
var db = new NorthwindEFEntities();
var cust = db.Customers.First();
//will raise runtime error
cust.Orders.Load(MergeOption.NoTracking);
```

In the above query Customer entity is retrieve using the default MergeOption, MergeOption.AppendOnly. For retrieving Orders collection, NoTracking option is

used. Since related entities are loaded using different mergeOption, EF throws a runtime exception.

When a query retrieves an entity with related ends which are entity reference, EF will rewrite the query to bring all foreign key values in the query. Based on the foreign key values, EF will create relationship stub for related ends of the entity. At some later point when the related ends are loaded inside the state manager, EF will replace the stub entry with a full blown entity. An example of this scenario is when we retrieve an order without retrieving the customer for the order; a stub entry would be created in the state manager. It is because of the stub, we can access customerid foreign key value for the order by Order.CustomerRefrence.EntityKey. However when order entity is retrieved using NoTracking MergeOption, Ef will make query as lean as possible and would not load any additional data then requested by the client. Therefore Order.CustomerReference.EntityKey would be null reference. If you want the related entity, you can either use Load but the Load operator also needs to be used with NoTracking Option as both MergeOptions must be same. Another option is to eagerly load both Order and Customer with NoTracking option. Code below shows an example.

```
var db = new NorthwindEFEntities();
db.Customers.MergeOption = MergeOption.NoTracking;
var cust = db.Customers.Include("Orders").First();
Console.WriteLine(cust.Orders.Count());
```

On the above code, both Customer and Orders are retrieved using NoTracking MergeOption. Although NoTracking was only set on Customer entity, since Orders entity was also part of Customer query retrieval, EF applied the same query rules to Orders collection.

Entity framework support updates and deletes of an entity retrieved using ObjectContext. To perform update to an entity, it must be tracked by the object state manager. If entity was not tracked EF cannot determine what fields have changed because it would have nothing to compare against the original values of entity. If the query is retrieved using NoTracking, it must be attached to the context before an update can be performed. Example below

shows the code required to perform update on an entity retrieved using NoTracking option.

```
var db = new NorthwindEFEntities();
    db.Customers.MergeOption = MergeOption.NoTracking;
    var customer = db.Customers.First();
    Console.WriteLine(customer.CustomerID);
    //attach teh customer
    db.Attach(customer);
    customer.Phone = "817-355-9899";
    db.SaveChanges();
```

On the above, the first customer is retrieved using NoTracking. Before I can perform an update on the customer, I need to notify the object state manager about the entity so it tracks the original values of the entity. After attaching the entity, I am modifying the phone number and save the changes to the database.

NoTracking queries are great for asp.net scenario where majority of the data displayed inside of listview and gridview is read-only. Additionally if an entity is return using NoTracking, it is already in detached state and you do not have to explicitly detach an entity before you can attach the entity to another context. A place where we do this quite often is when retrieving an entity using Object context for update scenario on a gridview and on post back the entity needs to be attached to a new context because the context that originally retrieved the entity is already disposed. If an entity is not detached from the older context is disposed it cannot be attached to a new context. But if the entity is retrieved using NoTracking, we do not have to worry about detaching the entity.

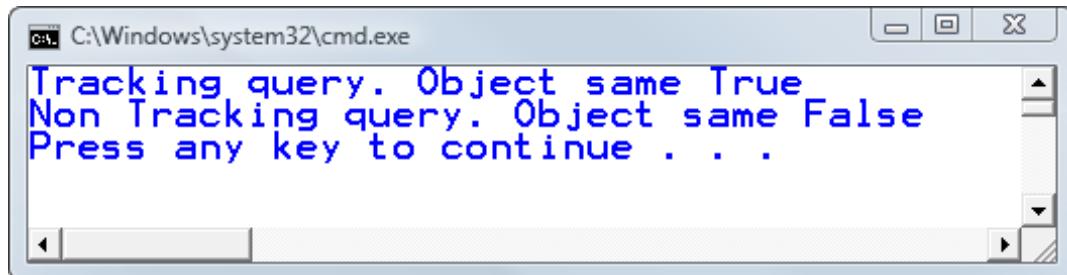
Entities returned using NoTracking do not perform identity resolution because they are not tracked. With no tracking information inside state manager, you could end up with two different entities that have the same entity key. If the queries use default option, then EF will make a database call to find the result set that match the query. After getting result set it will search for each item in the result set inside the state manager. If an entity matches the key retrieved from the database, the entity in the state manager would be returned. Having the same reference returned makes object comparison possible in the application. For instance to check if the customer matches the customer

returned from the query, object comparison would equal to comparing entity keys for both customers. When no tracking option is used, you can have two entities having the same entity key and comparing objects would return incorrect results.

```
var db = new NorthwindEFEntities();
    var cust1 = db.Customers.First();
    var cust2 = db.Customers.First();
    Console.WriteLine("Tracking query. Object same {0}", cust1 ==
cust2);

    //non tracking
    db.Customers.MergeOption = MergeOption.NoTracking;
    var cust3 = db.Customers.First();
    var cust4 = db.Customers.First();
    Console.WriteLine("Non Tracking query. Object same {0}",
cust3 == cust4);
```

On the code above, entities that are tracked have the same object references. Whereas cust3 and cust4 does not have same references because it uses MergeOption.NoTracking. Screen shot below shows the result on the console window.



```
C:\Windows\system32\cmd.exe
Tracking query. Object same True
Non Tracking query. Object same False
Press any key to continue . . .
```

MergeOption provides several other options that determine how objects are loaded into the state manager. For instance if a customer is tracked inside the state manager and some of properties of the customer is modified, if you want to overwrite those changes with the recent data from the data source, you can use OverWriteChanges option to discard all the changes the made to the customer. Although the entity returned will reflect the data from the store but you will receive the same instance that you had previously been working with in the application. Code below shows an example of using MergeOption.OverWriteChanges

```
var db = new NorthwindEFEntities();
    string esql = "select value c from customers as c where
c.CustomerID = 'ALFKI'" ;
```

```

var cust = db.CreateQuery<Customer>(esql).First();

//Berlin
Console.WriteLine(cust.City);

//change it to london
cust.City = "London";

db.CreateQuery<Customer>(esql).Execute(MergeOption.OverwriteChanges).First();

Console.WriteLine(cust.City);

```

On the code above, using esql query I am retrieving ALFKI customer, printing the city for the customer, changing the city to London and then loading the customer again from the database. When I load the customer, I use MergeOption.OverWriteChanges option to indicate that ALFKI customer data needs to be replaced with the new values from the store.

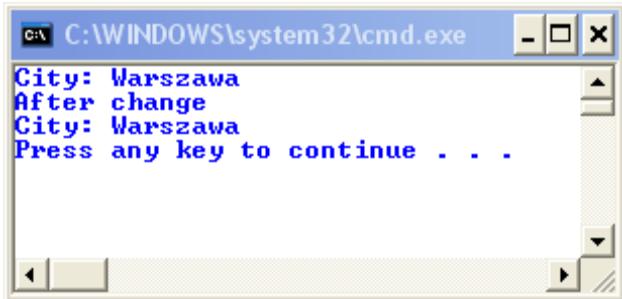
AppendOnly is another option for Merge changes which can be used when materializing objects inside of object State manager. When AppendOnly option is used, EF will first check if the entity key retrieved from the store is found inside the state manager. If object already exists, it will not be overwritten and the old object is returned to the application. Code below shows an example of Using AppendOnly

```

var db = new NorthwindEFEntities();
var customer = db.Customers.First(c => c.CustomerID == "WOLZA");
Console.WriteLine("City: " + customer.City);
//change Wolza customer's city to paris
ChangeCitytoParis();
Console.WriteLine("After change ");
db.Customers.MergeOption = MergeOption.AppendOnly;
db.Customers.First(c => c.CustomerID == "WOLZA");

```

The code above retrieves WOLZA customer and prints its city to console window. Then using straight ado.net, I am changing the city of the customer to Paris by calling changeCityToParis method. After refreshing the customer by AppendOnly option, I am printing the Wolza customer's city again on the console window to check if the city has changed. The results confirm that calling AppendOnly returns the old object from state manager without taking changes from the database. Figure below shows the screen shot for the city results.

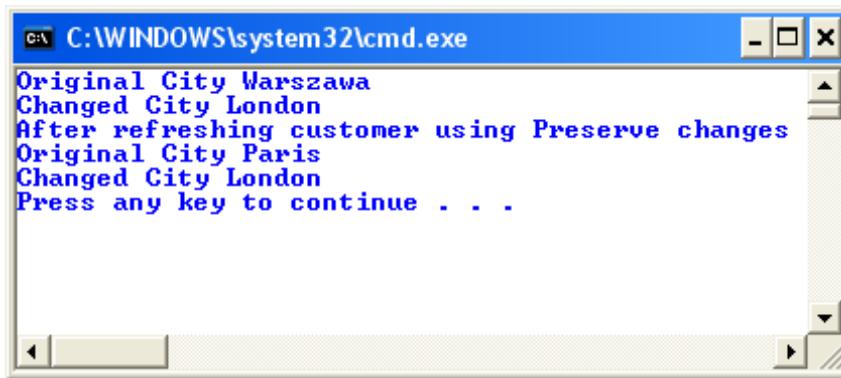


Another option for MergeOption is PreserveChanges. When you call preserve changes to refresh an entity, the original values of the entity are refreshed to what's retrieved from the data source. A good use case for this option is when you get an OptimisticConcurrency exception when saving an entity to the database. This exception could be raised because the original value of entity does not match with what's defined in the store because the values had changed during the time the entity was inside the state manager. To fix this exception, we can update the original values of entity inside the state manager by refreshing the entity using MergeOption.PreserveChanges. Code below shows an example of using PreserveChanges.

```
var db = new NorthwindEFEntities();
    var customer = db.Customers.First(c => c.CustomerID == "WOLZA");
    //change city to london.
    customer.City = "London";
    //original city is Warszawa
    string orignalcity =
(string)db.ObjectStateManager.GetObjectStateEntry(customer).OriginalValues["City"];
    Console.WriteLine("Original City " + orignalcity);
    Console.WriteLine("Changed City " + customer.City);

    //change city in the database without notifying objectcontext.
    ChangeCitytoParis();
    //refresh customer with preserve changes.
    db.Customers.MergeOption = MergeOption.PreserveChanges;
    db.Customers.First(c => c.CustomerID == "WOLZA");
    string _orignalcity =
(string)db.ObjectStateManager.GetObjectStateEntry(customer).OriginalValues["City"];
    Console.WriteLine("After refreshing customer using Preserve
changes");
    Console.WriteLine("Original City " + _orignalcity);
    Console.WriteLine("Changed City " + customer.City);
```

On the code above, we retrieve the wolza customer, change its city from Warszawa to London and print both the original and changed values on the console window. To retrieve the original values, I am accessing the state entry for the customer from the state manager and then accessing the city value. After refreshing the customer in the store with PreserveChanges, I am printing the city for wolza customer again on the console window. Based on the result in the figure below we can see that only original values got changes and current values were not affected.



```
C:\WINDOWS\system32\cmd.exe
Original City Warszawa
Changed City London
After refreshing customer using Preserve changes
Original City Paris
Changed City London
Press any key to continue . . .
```

7.3 Compiled Queries

Problem: You want to know how compiled query can improve query performance and what are different ways you can execute compiled queries.

Solution: One of expensive cost of executing a linq query includes creating a query cache. If there are queries in your application that are being executed over and over again, it is beneficial to create compiled queries. With compiled queries you don't have to re-process the same linq expression on every execution. The linq expression is compiled into a command tree once and executing it subsequently leverages the same command tree. With linq to entities to reuse the same command tree, we need to use Compiled Query class and call Compile method to compile linq query. However when executing entity sql first time, the query plan is cached automatically and subsequent execution of the same entity sql query will yield better performance. Compiled query syntax looks like this

```
CompiledQuery.Compile<ObjectContext, arguments, returntype>(delegate)
```

There are three overloaded versions of Compile method. Compile method must at least take in the object context on which queries execute. The return type is optional if the compiled query returns an anonymous type. In other cases you can have return type that could be IQueryble of T, an ObjectQuery of T, an entity or a complex type. Compile method supports up to three arguments. Those arguments can be used to apply dynamic filter to a query. If you are executing a compile queries several times, you can pass in different parameters and still reuse the command tree that was generated the first time. If you need to pass more than 3 parameters to provider dynamic filtering and sorting to a compile query, you can pass in class or struct as an argument to the compile method. Within the compiled query, you can reuse the properties declared inside struct or a class.

If entities used in a compile query is part of complex mapping scenario that includes inheritance and associations, compiling the linq expressions into command trees may be an expensive process. Those queries would ideal uses cases for compiled queries as you can compile once and re use it again. Compile method takes in an object context as a parameter so that compiled queries are done outside of the scope of an object context. This allows compiled queries to be used against multiple threads and yet leverage the command trees cached globally. Additionally in asp.net scenarios where the recommended practice is to discard the object context on every request, could also take the benefit even when you are working with new objectcontext on every request.

A compiled query only works with linq queries. If linq queries are mixed with builder methods that allow passing esql statements to linq query operators, you will get runtime exception. Builder methods are not support inside of a compiled query. However you can perform eager loading operation by calling include inside of a compiled query.

A common use case for a compiled query would be to improve performance when creating search query. For instance a web form would allow the user to search for a customer based on city. Since the city parameter is determined at runtime, it would be beneficial to use compiled query to leverage the compiled expressions.

Discussion: When a linq query is compiled, it get translated into a command tree on the initial run and consecutive runs offer better performance in the execution of the query. On the code below, I have ran some test to identity the kind of performance improvements you would get if you executed a same linq query with and without using compiled versions.

```
static readonly Func<IncludeTPTEntities, GunSmith> gunsmithquery =
    CompiledQuery.Compile<IncludeTPTEntities, GunSmith>(
        (ctx) => ctx.Contacts.OfType<GunSmith>()
            .Include("Company.Phone")
            .Include("Company.Departments").First());
public static void CompiledQueryPerformanceTest()
{
    IncludeExample1.Cleaanup();
    IncludeExample1.InsertGunsmith();
    Console.WriteLine("None compiled query");
    var db = new IncludeTPTEntities();

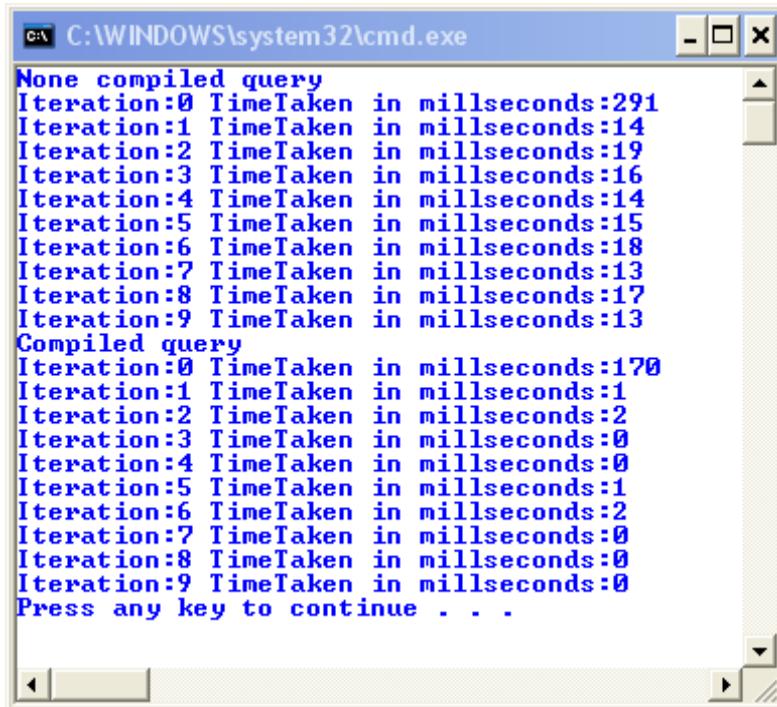
    for (int i = 0; i < 10; i++)
    {
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();
        var gunsmith =
db.Contacts.OfType<GunSmith>().Include("Company.Phone").Include("Company.Depa
rtments").First();
        stopwatch.Stop();
        Console.WriteLine("Iteration:{0} TimeTaken in
milliseconds:{1}", i, stopwatch.ElapsedMilliseconds);
    }

    Console.WriteLine("Compiled query");
    var db2 = new IncludeTPTEntities();

    for (int i = 0; i < 10; i++)
    {
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();
        var gunsmith = gunsmithquery(db2);
        stopwatch.Stop();
        Console.WriteLine("Iteration:{0} TimeTaken in
milliseconds:{1}", i, stopwatch.ElapsedMilliseconds);
    }
}
```

The code above retrieves contact of type gunsmith. In addition, we are also eagerly loading Company, Phone and department entity associated to it. Since a compiled query is created once, it is declared as a static variable and to execute it, we are passing in an instance of data context. To create a test results, I am first executing a non compile version of the gunsmith query ten

times inside a loop and capturing the time taken followed by executing a compiled version of the query ten times and capturing its result. Figure below shows the output on the console window.



The screenshot shows a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The window displays two sets of performance data. The first set, labeled 'None compiled query', shows 10 iterations of a query execution taking between 13 and 291 milliseconds. The second set, labeled 'Compiled query', shows 10 iterations where the execution time drops significantly to mostly zero milliseconds, indicating the cost of compilation. The text at the bottom of the window reads 'Press any key to continue . . .'. The code block below corresponds to the logic executed in the window.

```
None compiled query
Iteration:0 TimeTaken in milliseconds:291
Iteration:1 TimeTaken in milliseconds:14
Iteration:2 TimeTaken in milliseconds:19
Iteration:3 TimeTaken in milliseconds:16
Iteration:4 TimeTaken in milliseconds:14
Iteration:5 TimeTaken in milliseconds:15
Iteration:6 TimeTaken in milliseconds:18
Iteration:7 TimeTaken in milliseconds:13
Iteration:8 TimeTaken in milliseconds:17
Iteration:9 TimeTaken in milliseconds:13
Compiled query
Iteration:0 TimeTaken in milliseconds:170
Iteration:1 TimeTaken in milliseconds:1
Iteration:2 TimeTaken in milliseconds:2
Iteration:3 TimeTaken in milliseconds:0
Iteration:4 TimeTaken in milliseconds:0
Iteration:5 TimeTaken in milliseconds:1
Iteration:6 TimeTaken in milliseconds:2
Iteration:7 TimeTaken in milliseconds:0
Iteration:8 TimeTaken in milliseconds:0
Iteration:9 TimeTaken in milliseconds:0
Press any key to continue . . .
```

The above screen shot shows that compiled query actually took less initially and consecutive runs were extremely fast to a point some queries had a value of zero for milliseconds because majority of the cost in executing the query was compiling the linq expression to a command tree. The results above are not an accurate measure of the performance improvement between a compiled and non compiled version of the query. The results would vary depending on the data you are retrieving in the application, complexity of the model and the load on the machine. It gives you a base line to consider of how improvements you would receive when compiling linq queries.

One of the requirements of a compiled query is, you must pass it a datacontext on which to execute a query. If the query returns entities, then you have to also specify the return type of the query. Code below shows an example of returning gunsmith whose companynam starts with Food.

```
var entities = new IncludeTPTEntities();
    var gunsmiths = CompiledQuery.Compile<IncludeTPTEntities,
IQueryable<GunSmith>>(
```

```

(db) => db.Contacts.OfType<GunSmith>()
    .Where(g => g.Company.CompanyName.StartsWith("Food"))
    .Select(g => g));

Console.WriteLine(gunsmiths(entities).First().ContactName);

```

Within a compiled query, you also pass in additional parameters to filter a linq query. The example below returns trade shows that happened within certain start and end date. The start and end date filter are passed in as parameters to a query.

```

var entities = new EcommerceEntities();
    var tradeshows = CompiledQuery.Compile<EcommerceEntities,
DateTime, DateTime, IQueryable<TradeShow>>(
    (db, startdate, enddate) => db.TradeShows
        .Where(s => s.StartDate >=
startdate && s.EndDate <= enddate));
    var start = DateTime.Parse("1/1/05");
    var end = DateTime.Parse("1/1/08");
    foreach (var show in tradeshows(entities,start,end))
    {
        Console.WriteLine(show.Name);
    }

```

To execute the above linq query, I am passing the data context and start and end date parameters required by the compiled query.

As mentioned earlier, compiled queries does not allow builder methods to be used. The query would compile fine but you will get a runtime exception that method cannot be translated into a store expression. Code below shows an example where using a builder method raises an exception.

```

var entities = new NorthwindEFEntities();
    var customers = CompiledQuery.Compile<NorthwindEFEntities,
ObjectQuery<Customer>>(
    (db) => db.Customers.Where("it.City = 'London'"));
    foreach (var cust in customers(entities))
    {
        Console.WriteLine(cust.CustomerID);
    }

```

Compiled queries do not restrict you to use the query as it is. For instance if you feel that only a certain portion of the query needs to be compiled and the rest of the query should remain dynamic you can further apply transformation on top of a compiled query and leverage the benefits of some portion of the command tree being pre compiled. In the code below I am retrieving gunsmiths that belong to Widget Company using a compiled query. On top of

the compiled query, I am adding additional filter to only retrieve those gunsmiths that are certified.

```
var entities = new TwoTPTEntities();

    var gunsmiths = CompiledQuery.Compile<TwoTPTEntities,
IQueryable<GunSmith>>(
        (db) => db.Contact.OfType<GunSmith>()
            .Where(g => g.Company.CompanyName == "Widgets"));
    var smiths = gunsmiths(entities).Where(g => g.IsCertified ==
true);
    foreach (var smith in smiths)
    {
        Console.WriteLine(smith.ContactName);
    }
```

The return type for a compiled query does not need to be a full blow entity. If you have complex type defined on your model that could also be used as the return type for the compiled query. In the example below customer's address is represented as a complex type CommonAddress. Since the query is only interested in grabbing the address information for a customer, I have set the return type for the compiled query to be CommonAddress class.

```
var entities = new NWComplexTypeEntities();
    var addresses = CompiledQuery.Compile<NWComplexTypeEntities,
IQueryable<CommonAddress>>(
        (db) => db.Customers.Where(c => c.Address.City ==
"London").Select(c => c.Address));
    foreach (var addr in addresses(entities))
    {
        Console.WriteLine(addr.Address);
    }
```

The above code not only returns a complex type but also filters on a complex type to only retrieve addresses where the city is London.

A compiled query does not have to return a collection of entities such as IQueryable of Customer or ObjectQuery of Customer. It can also return a single entity. In the example below, the compiled query returns the most recent order in the system.

```
var entities = new NorthwindEFEntities();
    var orderquery = CompiledQuery.Compile<NorthwindEFEntities,
Order>(
        (db) => db.Orders.First(o => o.OrderDate ==
db.Orders.Max(od => od.OrderDate)));
    Console.WriteLine(orderquery(entities).OrderDate);
```

A compiled query also allows eager loading related entities using Include. In the example below, the compiled query retrieves MediaCategories and the medias associated to those categories in a single query.

```
var entities = new MediaSelfRefEntities();
    var categories = CompiledQuery.Compile<MediaSelfRefEntities,
ObjectQuery<MediaCategory>>(
    (db) => db.MediaCategories.Include("Medias"));
    foreach (var category in categories(entities))
    {
        Console.WriteLine("Category:{0} Total
Medias:{1}", category.Name, category.Medias.Count());
    }
```

Compiled query also allows returning anonymous type if the query does not return a full entity. To use anonymous type in a compiled query, you do not specify the return type for the compiled query and simply assign the compiled query results to var which automatically infers the type from the query declaration. In the example below, I am retrieving the CustomerID and the total purchases the customer has made up to date.

```
var entities = new NorthwindEFEntities();
    var summary = CompiledQuery.Compile((NorthwindEFEntities db) =>
        from c in db.Customers
        where c.City == "London"
        select new
        {
            CustID = c.CustomerID,
            TotalPurchases = c.Orders.SelectMany(o =>
o.OrderDetails).Sum(od => od.UnitPrice * od.Quantity)
        });
    foreach (var cust in summary(entities))
    {
        Console.WriteLine("CustID:{0}
Purchases:{1:C}", cust.CustID, cust.TotalPurchases);
    }
```

A compiled query can also return scalar value as the return type. In the example below, the compiled query returns the total units on order for all the products in the database that have a unitinStock equal to zero.

```
//total units ordered for all products
var entities = new NorthwindEFEntities();
var totalordered = CompiledQuery.Compile<NorthwindEFEntities,
int>(
    (db) => db.Products.Where(p => p.UnitsInStock == 0)
        .Sum(p => p.UnitsOnOrder).Value);
    Console.WriteLine("Total units ordered: " +
totalordered(entities));
```

The overloaded versions of Compile method only supports up to three parameters. What if the queries need to apply a dynamic filter that would require more than three parameters? To use more than three parameters, you can either declare a class or struct that has all the properties that you need to filter and pass the instance of the class or struct as a parameter to the compiled query.

```
var criteria = new Criteria{ Quantity = 50, Discount = 0, City = "London" };
    var ods =
        CompiledQuery.Compile<NorthwindEFEntities, Criteria,
IQueryable<OrderDetails>>(
            (db, search) => from od in db.OrderDetails
                            where od.Quantity > criteria.Quantity &&
od.Discount == criteria.Discount
                            && od.Orders.Customer.City == criteria.City
                            select od
                );
    var entities = new NorthwindEFEntities();
foreach (var od in ods(entities, criteria))
{
    Console.WriteLine(od.OrderID);
}
```

In the example above, the compiled query returns OrderDetails where Quantity is greater than zero, discount is zero and orders placed by customers belong to city of London. To pass all these parameters, I have created a criteria class and declared properties that contain the criteria I want to filter OrderDetails entity on. The compiled query uses the instance of the Criteria class and filters the OrderDetails based on the values defined for the properties on the Criteria class.

7.4 Detaching entities returned from stored procedure

Problem: You have created an ObjectQuery that returns all customers from customer table. Customer query is followed by a call to stored procedure that returns all the phones for the customers in the database. In the EDM model a customer has 1 to many relationship with Phones. According to relationship span technique, you expect that as you iterate over the customers, EF will

automatically fix the relationship between customer and phone entity and graph be fixed. When you iterate over the graph, you are noticing that Customer's Phone collection does not have any phones although all the phones are loaded in state manager. You need to understand why relationship is not working and why the object graph is still broken.

Solution: In version 1 release, relationship span does not work when entities are returned using stored procedure. Even if the stored procedure results includes foreign key columns for related entities which is what object state manager use to build relationship stub, EF cannot break the result into entity and relationship info entry. It is possible that this feature will make it to next release where framework would have the smartness to create stub entries if stored procedure result contains foreign key columns.

When we load phone entities using stored procedure, they are tracked in the state manager but there is no relationship stub created for the graph to be fixed when customers are loaded. To ensure the phones are attached to the appropriate customers, we have to manually attach the phones that belong to customer.

Discussion: since relationship info is brought to create stub entries inside state manager when results are returned using stored procedure, we need to manually attach each customer's phones to their phone collection. Steps below outline the process.

1. Import GetPhones stored procedure into edm using import wizard.
2. Using Add Function Import the stored procedure from storage layer to the conceptual layer. On the function import set the return type for the entity to be Phone.

On the code below, I am returning phones from stored procedure and customers from ObjectQuery. To fix the relationship, I am using refresh method on the object context to refresh phone collection retrieved from stored procedure. Refresh causes query to be executed on the database that

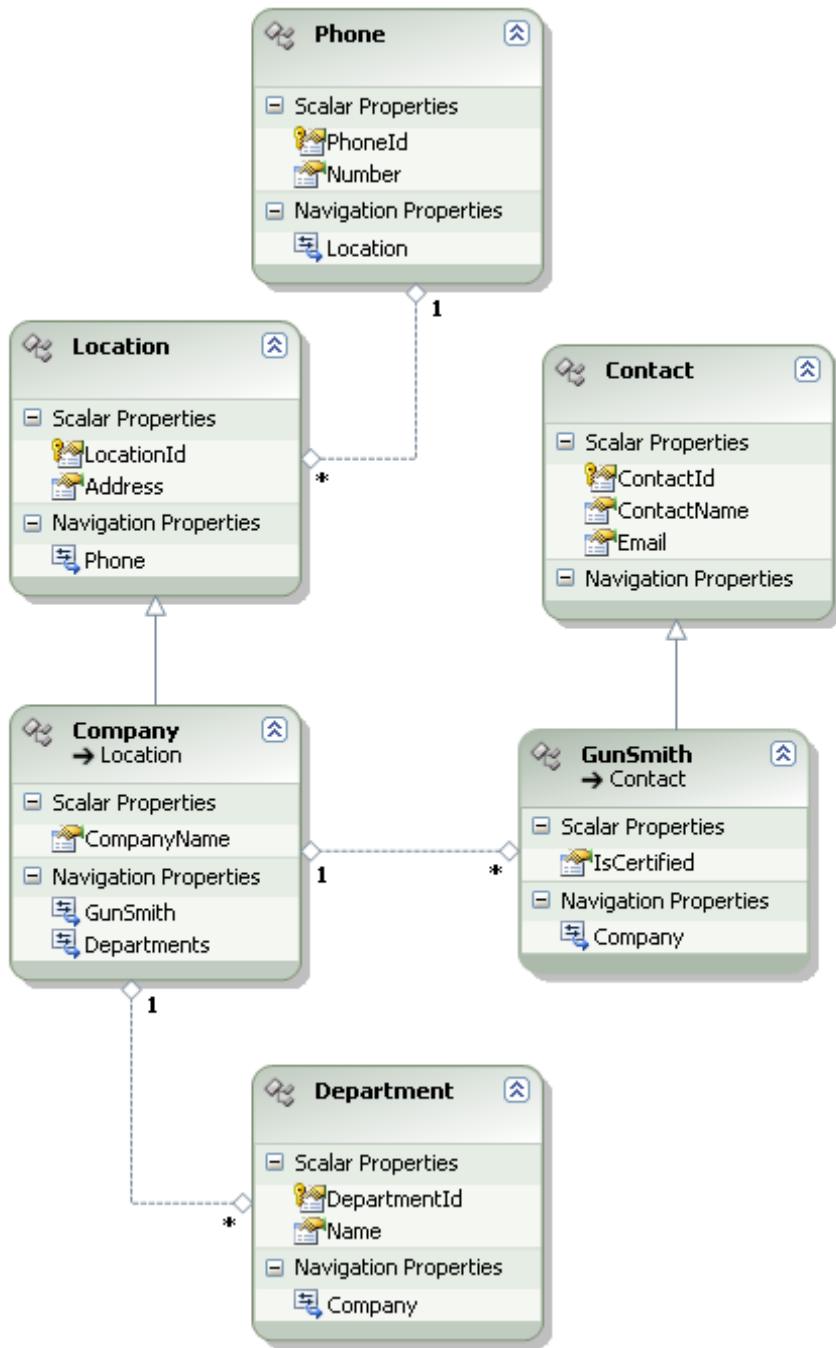
along brings along the relationship info for customer causing the graph to be fixed.

```
var db = new RsCustomerEntities();
    var customers = db.Customers.ToList();
    var phones = db.GetPhones().ToList();
    db.Refresh(System.Data.Objects.RefreshMode.ClientWins, phones);

    //customers.ForEach(c => c.Phones.Attach(
    //    phones.Where(p =>
Convert.ToInt32(p.CustomerReference.EntityKey.EntityKeyValue[0]) ==
c.CustomerId));
    foreach (var customer in customers)
    {
        Console.WriteLine("Customer:{0} Total Phones:{1}",
customer.Name, customer.Phones.Count());
    }
```

7.5 Improving loading time by generating store views

Problem: Figure below shows the model created by a developer that returns gunsmith, the company he belongs to, and its phone and department information.



When the application starts up, it takes some delay in running a simple query. You want to improve the startup cost of loading the model?

Solution: When a query is executed against an entity model, the runtime will first convert the model into esql views. Esql views are compiled versions of the mapping schema file in code. It basically contains esql queries that

represent the model being queried. The store views are created once and then cached at the app-domain level for later use. For asp.net scenarios, the view generation occurs as a startup cost when the model is queried for the first time. On subsequent request the model is cached and reused for later execution. Unless there is an IIS reset or recycling of app domain, the query view generation would not occur. There are several different ways to reduce the cold startup cost. Bullets points below describe some possible options.

1. Generate views ahead of time to avoid runtime cost using edmgen.exe utility.
2. Programmatically generate views
3. Run a dummy query to force view generation ahead of time.

Discussion: Since view generation occupies a large cost in executing the initial query, we can generate the store views ahead of time thus saving the cost of dynamically generating views. To generate the views, we can use edmgen.exe utility passing in the mode equal to ViewGeneration. Example below shows the command line parameter for generating views.

```
Edmgen.exe /mode:ViewGeneration /incsdl:IncludeTPT.csdl  
/language:CSharp /outviews:views.cs /inmsl:IncludeTPT.msl  
/inssdl:IncludeTPT.ssdl
```

The above command line execution would generate views.cs class that can be added to the class library project and will be utilized by the runtime instead of generating views. Another option of generating view would be to use edmgen2.exe utility that can be downloaded from

<http://code.msdn.microsoft.com/EdmGen2>

It is a command line tool written by one of the members of EF team and is not directly supported as a product. The benefit of using this utility is it allows passing in the edmx file that is generated by the visual studio instead of separately passing csdl, msl and ssdl files.

You can also use FullGeneration mode option with edmgen.exe to generate ssdl, msl and csdl along with the store views that contains all the mapping definition in code.

The code below queries the gunsmith model causing the view generation to happen. The query is executed under a stopwatch to capture the time taken.

```
Stopwatch watch = new Stopwatch();
watch.Start();
var db = new IncludeTPTEntities();
var gunsmiths = db.Contacts.OfType<GunSmith>()
    .Include("Company.Phone")
    .Include("Company.Departments").ToList();
watch.Stop();
Console.WriteLine("Gunsmith Time
taken:{0}",watch.ElapsedMilliseconds);
```

The above query when executed ten times takes 452 milliseconds with store views generated on demand by the runtime. When the store views code file is generated ahead of time using edmgen.exe and added to the console application, ten iterations of the query only takes 404 milliseconds. The 48 milliseconds that took extra was the cost of converting the model to esql views. For a model that only contains 6 entities the cost was fairly high, so if the model is very complex and contains large number of entities, the startup cost which is executing the first query on the model would be very high.

One of the problems with using edmgen.exe or edmgen2.exe is, you have to remember to regenerate the views when you make changes to the model otherwise the views would be out of sync with the model causing runtime errors. To avoid that you can programmatically generate the views and place the code file inside the bin directory for runtime to find it. Code below shows an example of programmatically generating the code file for a console application.

```
static string csdlNamespace = "http://schemas.microsoft.com/ado/2006/04/edm";
static string ssdlNamespace =
"http://schemas.microsoft.com/ado/2006/04/edm/ssdl";
static string mslNamespace = "urn:schemas-microsoft-
com:windows:storage:mapping:CS";

var edmx = XElement.Load(@"..\..\IncludeTPT.edmx");
var csdl = GetCsdl(edmx);
var ssdl = GetSsdl(edmx);
var msl = GetMsl(edmx);
IList<EdmSchemaError> cerrors, serrors, merrors = null;
EdmItemCollection edmitems =
MetadataItemCollectionFactory.CreateEdmItemCollection(new[] {csdl.CreateReader()},
out cerrors);
```

```

        StoreItemCollection sitems =
MetadataItemCollectionFactory.CreateStoreItemCollection(new[] {
ssdl.CreateReader() }, out serrors);

        StorageMappingItemCollection mapping =
MetadataItemCollectionFactory.CreateStorageMappingItemCollection
(edmitems,sitems,new[]{msl.CreateReader()}, out merrors);

        EntityViewGenerator evg = new
EntityViewGenerator(LanguageOption.GenerateCSharpCode);
        evg.GenerateViews(mapping, "Views.cs");

private static XElement GetMsl(XElement edmx)
{
    return (from item in edmx.Descendants(
        XName.Get("Mapping", mslNamespace))
        select item).First();

}

private static XElement GetSsdl(XElement edmx)
{
    return (from item in edmx.Descendants(
        XName.Get("Schema", ssdlNamespace))
        select item).First();

}

private static XElement GetCsdl(XElement edmx)
{
    return (from item in edmx.Descendants(XName.Get("Schema",
csdlNamespace))
        select item).First();
}

```

The above code generates Views.cs code file inside the bin directory containing all the store views.

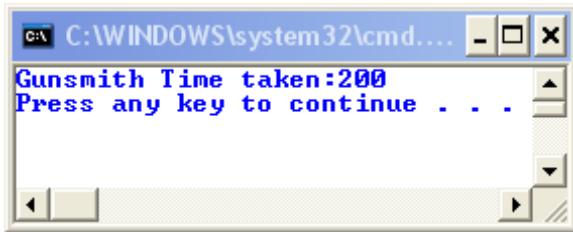
One of the other ways to force the generation of views is executing a dummy query ahead of time causing the runtime to generate the views. Code below confirms that behavior.

```

var db = new IncludeTPTEntities();
db.Contacts.First();
Stopwatch watch = new Stopwatch();
watch.Start();
//original query
var gunsmiths = db.Contacts.OfType<GunSmith>()
    .Include("Company.Phone")
    .Include("Company.Departments").ToList();
watch.Stop();
Console.WriteLine("Gunsmith Time taken:{0}",
watch.ElapsedMilliseconds);

```

On the above code, I am executing a query that returns the first contact in the database. The query causes view generation process to trigger and therefore when the actual query executes, there is no cost associated with view generation. Figure below shows the time takes for the original query to execute.



8. Inserting, Updating and Deleting entities and associations

8.1 Assigning foreign key value without loading entity reference

Problem: You want to create an address entity and assign it to a customer without loading customer entity. On the web page that allows inserting address, the customerid is passed in the query string. You want to read that customerid and assign it to the address.

Solution: Since Address has many to 1 association with Customer entity, EF will not expose the foreign key customerid on Address entity. There will be two navigation property on address entity; Customer and CustomerReference. Customer property is a pointer to a CLR reference that represents customer entity. By default the Customer navigation property would be null unless you call Load or Include on an object query. CustomerReference is a relationship entry inside of Object State Manager that relates a customer to an address. CustomerReference is of type EntityReference<Customer> which has a property EntityKey. To retrieve or assign a customerid to address entity, we need to set the entitykeyvalues for the entity key on CustomerReference.

Code below shows an example of assigning customerid 2 to the new address being created.

```
var db = new OneToManyEntities();
    var address = new Address { Address1 = "Oakumber st", City =
"Dallas", State = "Tx", Zip = "76111" };
    address.CustomerReference.EntityKey = new
EntityKey("OneToManyEntities.Customer", "CustomerId", 2);
    db.AddToAddresses(address);
    db.SaveChanges();
```

Discussion: On the example above to assign customerid of two to an address, we are creating an instance of EntityKey. EntityKey class has several overloads and one of the overloads supports composite keys if the table in the database has two or more columns representing the primary key. In the above example only CustomerId is the entity key for customer entity. First parameter to Entity key is the fully qualified entity set name which is EntityContainer.Entityset. The second parameter represents the entity key property on the customer entity and the third parameter is the value you want to assign to the customerid which is 2.

If the above code is performed repeatedly in various parts of the application, it would be better to create CustomerId property on a partial class for Address entity. The CustomerId property should be both read and write which would allow you to read the customerid for an address and also assign a new customerid. Code below shows an example of Address partial class which supports this notion.

```
public partial class Product
{
    public int ModelId
    {
        get
        {
            if (ProductModelReference.EntityKey != null)
            {
                return 0;
            }
            return
(int)ProductModelReference.EntityKey.EntityKeyValue[0].Value;
        }
        set
        {
            this.ProductModelReference.EntityKey = new
EntityKey("AdventureWorksLTEntities.ProductModel", "ProductModelID", value);
        }
    }
}
```

The above code reads the first value from the EntityKeyValues array to return the CustomerId. An entityKeyValues can contain more than one value if the entity key is a composite key. To set the CustomerId, I am creating an instance of Entity Key and assigning it to ProductModelReference.EntityKey.

If an Entity Key is already assigned to an entity reference and you want to change it to a different value you will still have to create a new instance of entity key because it is immutable and its properties cannot be changed once it is assigned. If you do try to change it you will get an exception that key cannot be changed once they are set. The code below shows an incorrect usage of assigning customerid. Instead of creating an entity key, customerid value is assigned to an existing an entity key value that throws an exception.

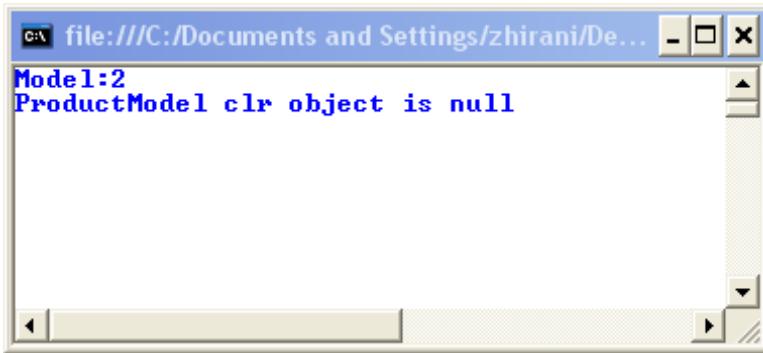
```
var db = new AdventureWorksLTEntities();
    var prod = db.Product.First(p => p.ProductNumber == "FR-R92B-
58");
    //Error:entity key value cannot be changed once they are
set.Incorrect usage.
    prod.ProductModelReference.EntityKey.EntityKeyValues[0].Value =
2;
    db.SaveChanges();
```

In a situation where you want to remove an entity reference, you can simply set the entity key to a null value. The code below retrieves the product and prints its productModelId on the console window. To remove the association of the product to Productmodel, EntityKey on ProductModelReference is set to null. When SaveChanges is called EF sees that ProductModelId entity key has a null value and updates the product table where ProductModelId column value is set to null.

EF is smart in synchronizing references when the entity key value is set. For instance if Product entity is pointing to ProductModel and if entity key is set to null, productModel would no longer be pointing to valid reference. When entity key is set to null, EF removes the relationship entry inside of Object State Manager between Product and ProductModel.

```
var db = new AdventureWorksLTEntities();
    var prod = db.Product.Include("ProductModel").First(p =>
p.ProductNumber == "FR-R92B-59");
    Console.WriteLine("Model:{0}",prod.ProductModel.ProductModelID);
    //set entitykey for model to null also removes the reference.
    prod.ProductModelReference.EntityKey = null;
    Console.WriteLine("ProductModel clr object is {0}",
prod.ProductModel == null ? "null" : "not null");
```

On the code above, I am retrieving Product and its ProductModel by using Include. To confirm that Product entity has a valid product model, I am printing the ProductModelId on the console window. When EntityKey is set to null on ProductModelReference, ProductModel navigation property is also set to null as indicated by the results shown on the console window.



```
file:///C:/Documents and Settings/zhirani/De...
Model:2
ProductModel clr object is null
```

All the above cases discussed are options that should be considered if the related entity key is not loaded inside the state manager. If related entities are available as objects and tracked by object state manager, it is cleaner to assign an object reference to the navigation property. The code below shows an example where ProductModel entity is retrieved from the store and assigned to the Product entity.

```
var db = new AdventureWorksLTEntities();
        var prod = db.Product.First(p => p.ProductNumber == "FR-R92B-
58");
        var productmodel = db.ProductModel.First(m => m.ProductModelID ==
2);
        prod.ProductModel = productmodel;
        db.SaveChanges();
```

The above example is a cleaner approach to assigning productmodelid but it requires an additional database call to retrieve the ProductModel entity as compared to setting the EntityKey on ProductModelReference property directly.

So far we looked at examples of setting entity keys for EntityReference, what if we have Many to Many relationship. For instance a student can be enrolled in many courses and a course can have many students. If you have a student entity and you know the courseId the student needs to be enrolled in, how can we add that course without loading the course entity? Code below shows an example of adding a course to student's course collection.

```

InsertStudent();
    var db = new MMStudentCoursesEntities();
    var student = db.Students.First(s => s.Name == "Harold");
    var course = new Course { CourseId = 1 };
    db.AttachTo("Courses", course);
    student.Courses.Add(course);
    db.SaveChanges();

```

In the above example, a dummy course entity is created with the courseid present in the database. To notify the Object State Manager about the course, we are attaching the course to the state manager and adding the course to the student's course collection. Since the relationship requires the courseid to be present for course entity, Ef correctly inserts the relationship into the link table to associate the student with the course.

9. Querying with Linq to entities

9.1 How to do in Clause Query

Problem: You have select items in an array and would like to right a query that filters the results based on the items in the array. The query requires an In clause. You need to know what are the various options available in entity framework that would be translated to In clause on the database.

Solution: In the version 1 of the release, the contains clause cannot be translated to in clause on the database because the operator is not supported. You have to resort to esql language support in entity framework. Code below shows how you can write in clause queries using esql language available in entity framework.

```

public static void InClause()
{
    //1st way to do it.
    var cities = new [] { "London", "Berlin" };
    var cityparams = string.Join(",",

```

```

        cities.Select(c => " " + c +
    " ")
    .ToArray() );

var db = new NorthwindEFEntities();
string sql = @"
    Select value c from Customers as c
    where c.City in { " + cityparams + " } ";
var custs =
db.CreateQuery<Customer>(sql);

Console.WriteLine("Customers for London " +
    custs.Count(c => c.City == "London "));

Console.WriteLine("Customers for Berlin " +
    custs.Count(c => c.City == "Berlin"));

//2nd way to do it.
var custs2 =
    db.Customers.Where("it.City in {@cities}",new
ObjectParameter("cities",cityparams));

Console.WriteLine("Customers for London " +
    custs2.Count(c => c.City == "London "));

Console.WriteLine("Customers for Berlin " +
    custs2.Count(c => c.City == "Berlin"));

var all = from ct in cities
          join c in db.Customers on ct equals
    c.City
          select c;
Console.WriteLine("Customers from London and
Berlin " + all.Count());
}

```

Discussion: In the example above, I have an array of cities that I would like to get customers for from NorthWind database. To pass this array to esql statement, I have to convert the array to a string delimited by comma and if the column you are searching on is string column, than you also need to make sure that each item in your list is enclosed in single quotes. You would think that you write an in clause in esql as follows.

```
var cities = new [] { "London", "Berlin" };
```

```

string sql1 = @"
    Select value c from Customers as c
    where c.City in { " + cities + "}";
db.CreateQuery<Customer>(sql1).ToList();

```

In the above code, I am passing in the array directly to esql statement as you would normally do when you are writing sql statement. However when I run this code I get an exception stating that query syntax is not valid. Since I cannot pass array directly, I first use select operator to surround each item in my arrary with single quote and then flatten my array into a string by using string.join operator. I am than building esql query as follows

```

var db = new NorthwindEFEntities();
string sql = @"
    Select value c from Customers as c
    where c.City in { " + cityparams + "}";
var custs =
db.CreateQuery<Customer>(sql);

```

In the above code, I use value operator in my select clause to return all the customers which has city that matches the cities that I am passing in using cityparams variable. To test the query returned correct results, I am doing a count operator in memory to checks how many customers in my result belong to city of London and Berlin. Following code shows how to do this.

```

Console.WriteLine("Customers for London " +
    custs.Count(c => c.City == "London"));
Console.WriteLine("Customers for Berlin " +
    custs.Count(c => c.City == "Berlin"));

```

You are not obligated to build an entire esql statement to get your results back. You can additionally use linq to entities and pass in a dynamic where clause in the form of esql statement. The benefit of this approach is you don't have to write the entire esql statements just the where clause of the query is required. Secondly using linq to entities, you also get a chance to chain the query by joins, where and sorts clause. This is an important concept in the sense you can mix and match esql with linq expressions. For cases that are supported in terms of linq query operator, you can use esql and for the rest of

query, you can enjoy the benefits of linq query operators. Following code shows an example of using esql statement as part of the where extension method.

```
//2nd way to do it.  
    var custs2 =  
        db.Customers.Where("it.City in {@cities}",new  
ObjectParameter("cities",cityparams));  
  
    Console.WriteLine("Customers for London " +  
        custs2.Count(c => c.City == "London "));  
  
    Console.WriteLine("Customers for Berlin " +  
        custs2.Count(c => c.City == "Berlin "));
```

In the above code, I am using the it syntax to access the current item passed in to the where clause. Since it operator in the current context represents a particular customer, I can access its City property and filter the list to only customers that belong to the cities passed in using cityparams.

Example below shows a linq query that performs the same operator of filtering the customers to cities that match London and Berlin.

```
var all = from ct in cities  
          join c in db.Customers on ct equals  
c.City  
          select c;  
Console.WriteLine("Customers from London and  
Berlin " + all.Count());
```

Looking at the above query you must be wondering that if linq syntax works fine than why is there a need to write esql statement to get results back.

Although the above query compiles and runs fine, it is really not an optimal query. Since join against an in memory collection is not possible and cannot be understood by sql server, linq to entities brings the entire customer records in memory applies the join in memory. Although the results will be same and the query is also much readable, query is not performed on the database which is not a good solution unless the records in the customer table are not too many to affect the performance. The most appropriate way to write a linq query that gets executed on the database, is readable and supports compile time check is by using contains operator. Currently v1 version of entity

framework does not support contains operator in contrast to linq to sql which fully supports contains operator that is translated to in clause on the database. Following example shows how to write the same query using linq to sql syntax and uses contains clause.

```
var db = new NorthwindEFEntities();
db.Customers.Where(c => cities.Contains(c.City))
```

Above code uses contains operator to tell that customer's city must be in one of the city in the array.

9.2 Returning subset of collection using Paging

Problem: You have Customers table that you are filtering based on certain criteria defined in your linq query. You want to display the results of the query in a listview control. Since list view's page size is set to 20 rows per page, you want to ensure that the data returned from the query uses paging on the server side to only bring rows enough that can be fit on one page.

Solution: Use the Skip operator to skip the number of records that you have paged through in listview control. When you use objectdatasource control with listview, objectdatasource will pass in the number of records to skip by inquiring the current page from the listview. After applying Skip operator you will use the Take operator to take the number of records the listview can display on a page. Number of records the page can display is obtained by the pagesize property of the listview control. If you use objectdatasource control, you will automatically be passed the page size of the listview control. Code below applies the contact title filter to query for customers which has a matching contact title. Instead of returning all the records we use the Skip and Take operator to only bring records equal to the page size of the listview control.

```
namespace NorthWind.Business.EF
{
    public partial class Customer
    {
```

```

        public static IQueryable<Customer>
GetCustomersByContactTitle(string contacttitle, int start,
int max)
{
    var db = new NorthwindEFEntities();

    var custs = db.Customers.AsQueryable();
    if (contacttitle != null)
    {
        custs = custs.Where(c => c.ContactTitle ==
contacttitle);
    }
    //ordering is required by entity framework if
you are going to use
    //skip operator in the query. Linq to sql does
not impose any constraints
    //for paging through the results.
    custs = custs.OrderBy(c => c.CompanyName)
                  .Skip(start).Take(max);
    var stuff = custs.ToString();
    return custs;
}

public static int
GetCustomersByContactTitleCount(string contacttitle)
{
    var db = new NorthwindEFEntities();
    var custs = db.Customers.AsQueryable();
    if (contacttitle != null)
    {
        custs = custs.Where(c => c.ContactTitle ==
contacttitle);
    }
    return custs.Count();
}
}

```

Discussion: In the above example, I have a partial class Customer which has a method GetCustomersByContactTitle. Since we do not need an instance of Customer class, I have made the static. GetCustomersByContactTitle method takes 3 parameters. First parameter is the contact title which will limit our customer results by returning customers that only match the contact title passed in. The second parameter passed in represents the number of rows we need to skip. Start parameter is populated by objectdatasource control on our

page as we will see shortly when we go through our Customers page. Objectdatasource control inquires the listview control the current page being requested and than using the page size it determines how many rows it needs to skip and passes that as the value for start parameter of our method. The third parameter max represents how many records to retrieve from the database. Max parameter is also assigned by ObjectDatasource which reads the pagesize from the data pager's Page Size property. Inside the method, we are assigning the generic Customers collection exposed by Objectcontext to custs variable. Based on if the contact title has something other than null value, we filter the results based on contact title. Than using the start and max row parameter we skip and take the number of rows required for one page. Another interesting lambda expression that we have added before the skip and Take operation is ordering the rows by CompanyName. Linq to entities requires that you specify what column you want to sorts the results by before you applying paging on the results. Failing to order the results before paging will result in the following error.

The method 'Skip' is only supported for sorted input in LINQ to Entities. The method 'OrderBy' must be called before the method 'Skip'.

Although Linq to entities enforces constraints for sorting before paging, Linq to sql does not require any sort operation for paging to work. Code below uses linq to sql datacontext to run the same query with no OrderBy operator. The results returned are exactly the same except no order and you do not get any exception either.

```
var db = new NorthWindDataContext();
        db.Log = new DebuggerWriter();
        var custs = db.Customers.AsQueryable();
        if (contacttitle != null)
        {
            custs = custs.Where(c => c.ContactTitle ==
contacttitle);
        }
        //linq to sql does not require ordering.
        custs = custs.Skip(start).Take(max);

        var stuff = custs.ToString();
        return custs;
```

Since we are only returning rows equal to the size of the page, we need to also expose another method that can help data pager identify how many pages users can browse. For that we need to return the count of total rows that meet the search criteria. To support the count operation, we have declared another method called GetCustomersByContactTitleCount which takes in contact title to filter the customer results and gets the count of total customers found matching the criteria. Since our customer query supports IQueryables, the entire query is executed on the database instead of bringing the customer results in the application and applying count operation in memory.

To display our data on a webpage, we will make use of 4 controls dropdownlist, listview, datapager and objectdatasource control. Code below shows the entire page and how it is wired up to talk to entities defined in our business layer.

```
<div>
    <asp:DropDownList AutoPostBack="true" ID="titles" runat="server">
        <asp:ListItem Text="All" Value="" />
        <asp:ListItem Text="Owner" Value="Owner" />
        <asp:ListItem Text="Sales Agent" Value="Sales Agent" />
    </asp:DropDownList>
    <br />
    <asp:ListView ID="custlistview" DataSourceID="custsource" runat="server">
        <LayoutTemplate>
            <table>
                <tr>
                    <td colspan="3">
                        <asp:DataPager runat="server" PageSize="5">
                            <Fields>
                                <asp:NumericPagerField ButtonCount="5" />
                            </Fields>
                        </asp:DataPager>
                    </td>
                </tr>
                <tr>
                    <td>Company</td>
```

```

        <td>Contact Name</td>
        <td>Contact Title</td>
    </tr>
    <tr id="itemPlaceHolder" runat="server"
/>
    </table>
</LayoutTemplate>
<ItemTemplate>
    <tr>
        <td><%# Eval( "CompanyName" ) %></td>
        <td><%# Eval( "ContactName" ) %></td>
        <td><%# Eval( "ContactTitle" ) %></td>
    </tr>
</ItemTemplate>
</asp:ListView>
<asp:ObjectDataSource ID="custsource" runat="server"
    EnablePaging="true"
StartRowIndexParameterName="start"
MaximumRowsParameterName="max"

SelectCountMethod="GetCustomersByContactTitleCount"
    SelectMethod="GetCustomersByContactTitle"
TypeName="NorthWind.Business.EF.Customer">
    <SelectParameters>
        <asp:ControlParameter Name="contacttitle"
ControlID="titles" />
    </SelectParameters>
</asp:ObjectDataSource>
</div>

```

In the example above, I have a dropdown which contains few selections for Contact Title that we will use to filter our customer query. By default when the page loads up, the selected value in the dropdown is set to empty which objectdatasource converts it to null before send the filter parameter to customer entity. Since we are checking for null value for filter, we will retrieve all records in customer table with no filter criteria set.

Next, we are displaying few properties on our customer entity such as CompanyName, ContactName and ContactTitle inside the ItemTemplate of the ListView control. To support paging on the listview, I have also added a pager control inside my Layout Template and set the pagesize of the datapager to 5 rows. For the listview to fetch its data, I am binding the

listview to objectdatasource control using DataSourceId property set on the listview.

On the objectdatasource control, I am setting the EnablePaging to true to indicate that I do not want to perform paging in memory want to send my parameter for paging to my business layer which takes care of building the appropriate linq to query be executed on the database. After enabling paging, I am setting the StartRowIndexParameterName and MaximumRowsParameterName to the parameter names that I have define on my customer entity which happens to be start and max. I am also specifying the TypeName which represents the entity that will be called to request customer data. We also have to specify which method to call on the entity that is responsible for giving us customer data which we do by specifying the method name for SelectMethod property on the objectdatasource. Since GetCustomersByContactTitle only retrieves data that can fit on one page, DataPager needs to know how many pages of data is actually available to build its navigation pager. We do that by specifying method name for SelectCountMethod that returns the total number of rows that meet our customer search criteria.

When your run the above customer query, the sql generated is slightly different in both linq to sql and linq to entities. Below code shows the different query generated by their respective providers. Both queries have been slightly modified to clean up the noise from explicit column names specified in the query.

Linq To entities

```
SELECT TOP (5) [Project1].*
FROM
( SELECT [Project1].*, row_number() OVER (ORDER BY
[Project1].[CompanyName] ASC) AS [row_number]
FROM
( SELECT [Extent1].*
  FROM [dbo].[Customers] AS [Extent1]
  WHERE [Extent1].[ContactTitle] = 'Owner'
) AS [Project1] ) AS [Project1]
WHERE [Project1].[row_number] > 5
ORDER BY [Project1].[CompanyName] ASC
```

Linq to Sql

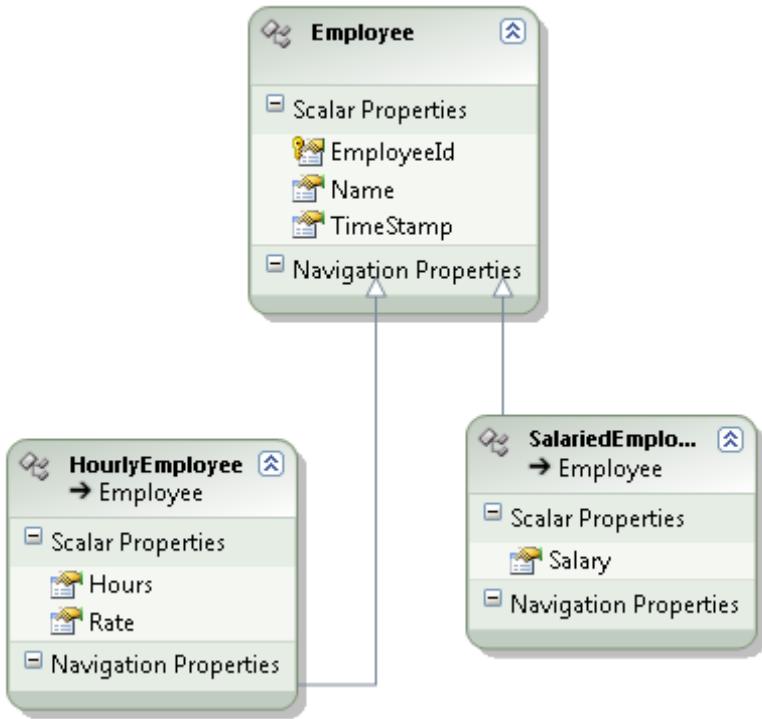
```
SELECT [t1].*
FROM (
    SELECT ROW_NUMBER() OVER (ORDER BY [t0].*) AS [ROW_NUMBER], [t0].*
    FROM [dbo].[Customers] AS [t0]
    WHERE [t0].[ContactTitle] = 'Owner'
) AS [t1]
WHERE [t1].[ROW_NUMBER] BETWEEN 5 + 1 AND 5 + 5
ORDER BY [t1].[ROW_NUMBER]
```

Linq to entities uses where clause to skip the number of rows and uses top clause to fetch the number of records to take. Whereas Linq to sql uses between operator to apply the skip and take operation specified in our linq query.

10. Concurrency and Transactions

10.1.1 Concurrency with Table per Type

Problem: Figure below shows the Table Per Type inheritance defined on entity data model.



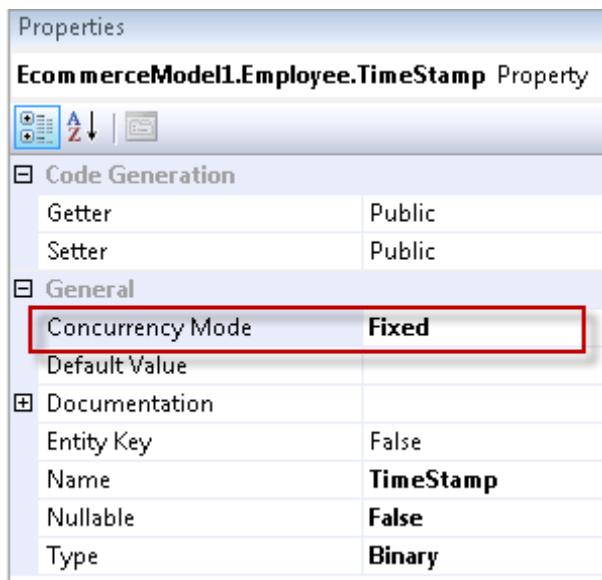
Since the above model uses table per type structure, Entity framework will write inserts to two different tables. For instance if we create an instance **SalariedEmployee**, entity framework will insert record into **employee** table followed by **SalariedEmployee** table. You want to know how you can enforce optimistic concurrency at entity level instead at table level.

Solution: To ensure optimistic concurrency, we need to add timestamp column to **Employee** table because it is the base entity defined on the model. Adding timestamp column to derive tables serve no purpose because when an update happens and regardless if only derived tables get affected, EF will still apply a dummy update to base **Employee** table causing **TimeStamp** column on base entity **Employee** to also get updated.

Discussion: To apply optimistic concurrency to Table per Type, we need to add timestamp column to **Employee** table. When entity framework sends the update for a derived entity to database, it will update the derived entity with the changes user has requested. In

addition it will apply a dummy update to Employee table regardless if there is value changed for any columns defined on Employee table. A dummy update would cause the TimeStamp column to also get updated and new TimeStamp value to be sent to the application. This process ensures that entire entity as a whole participates in optimistic concurrency.

When we add a timestamp column to Employee table and import the table on entity model designer, you do not get concurrency support out of the box. To apply concurrency, TimeStamp column must be set with concurrency model of Fixed. Figure below shows the concurrency mode for TimeStamp property.



The code below will help us dissect how EF ensures optimistic concurrency for Table Per Type. On the code below, I am creating an instance of SalariedEmployee and saving the entity to the database. Then immediately I change the Salary property of SalariedEmployee entity and send the update to the database. Notice that Salary column is defined on SalariedEmployee table and in terms of update no other column from Employee table is changed.

```
var db = new EcommerceEntities3();
    var salaryemployee = new SalariedEmployee { Name =
"Alex", Salary = 75000 };
```

```

        db.AddToEmployees(salaryemployee);
        db.SaveChanges();

        System.Threading.Thread.Sleep(5);
        salaryemployee.Salary = 85000;
        //run update statement and check for time.
        db.SaveChanges();
    
```

After executing the above code, we have also captured sql statements send by entity framework to the database engine.

```

exec sp_executesql N'insert [tpt].[Employee]([Name])
values (@0)
select [EmployeeId], [TimeStamp]
from [tpt].[Employee]
      where @@ROWCOUNT > 0 and [EmployeeId] = scope_identity()' ,N'@0
      varchar(4)',@0='Alex'

exec sp_executesql N'insert [tpt].[SalariedEmployee]([EmployeeId], [Salary])
values (@0, @1)
',N'@0 int,@1 int',@0=1,@1=7500
    
```

The above sql capture is the result of inserting a SalariedEmployee to the database. Notice that entity framework send an insert to employee table first followed by an insert to SalariedEmployee table. In addition to the insert, we are also retrieving the generated values from the database which includes EmployeeId the primary key and Timestamp column that gets created by sql server when an insert or update happens.

When we update the Salary property for SalariedEmployee, following sql capture was recorded by the profiler.

```

exec sp_executesql N'declare @p int
update [tpt].[Employee]
set @p = 0
where ([EmployeeId] = @0) and ([TimeStamp] = @1)
select [TimeStamp]
from [tpt].[Employee]
      where @@ROWCOUNT > 0 and [EmployeeId] = @0',N'@0 int,@1
      binary(8)',@0=1,@1=0x0000000000002EEE

exec sp_executesql N'update [tpt].[SalariedEmployee]
set [Salary] = @0
where ([EmployeeId] = @1)
    
```

```
' ,N'@0 int,@1 int' ,@0=85000,@1=1
```

On the above sql statements, EF first sends a dummy update to Employee table causing the TimeStamp column to get updated. It then queries for the new TimeStamp value to be sent to the client. Finally EF issues an update statement to apply the salary changes to SalariedEmployee table.

This clearly identifies that when tables are participating on Table Per Type inheritance, you have to think about concurrency at an entity level then at a table level. Because update to an entity could affect many tables that participate on Table Per Type depending on how deep the Table Per Type is configured.

11. Consuming Stored Procedures

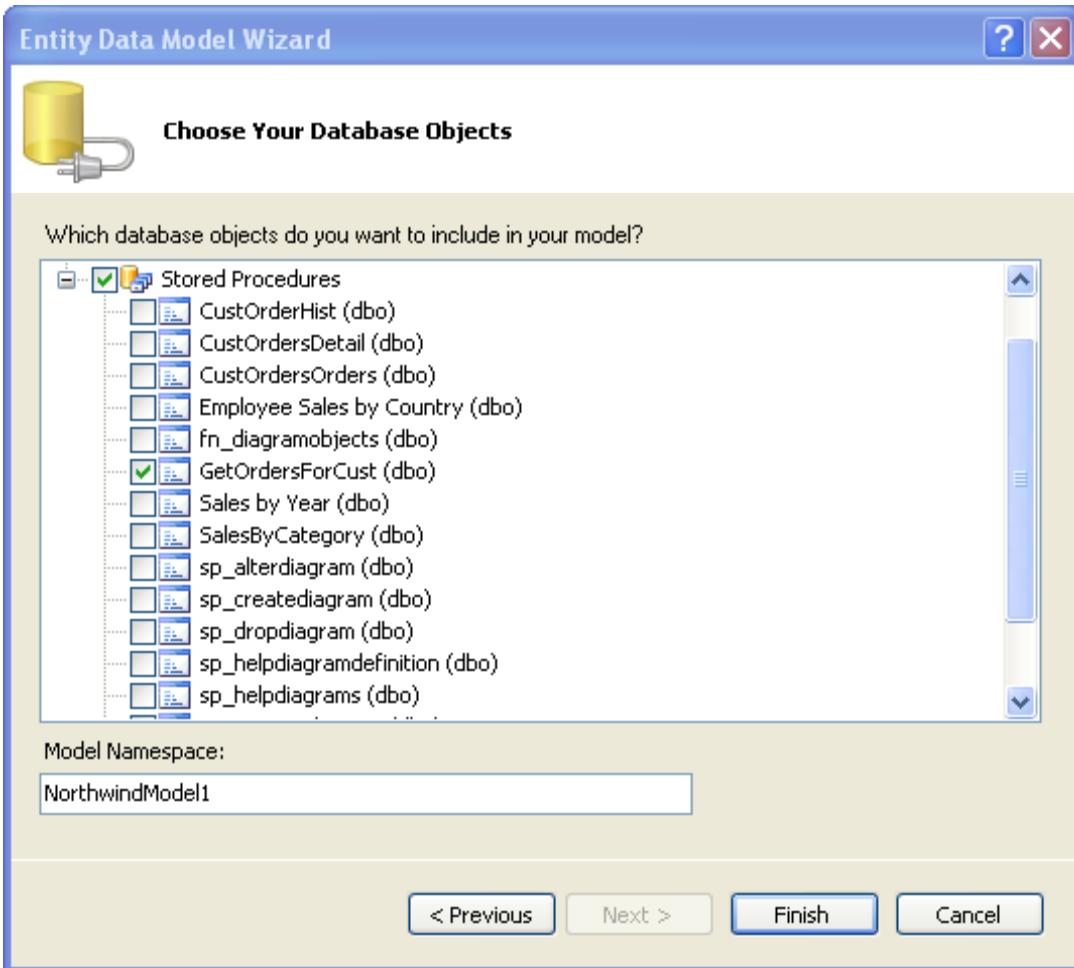
Following scenarios are covered in the store procedure.

- 1) Stored Procedure returning entities
- 2) Loading related child entities for a parent loaded from stored procedure.
- 3) How to disable tracking for data returned from stored procedure.
- 4) Stored Procedure returning scalar values.
- 5) Stored Procedure returning arbitrary number of columns that does not map to an entity on the conceptual model.
- 6) Stored Procedure performing some activity on the database and does not return anything.
- 7) Stored Procedure returning partially filled entity. (Not all columns returned as available on the original entity.)

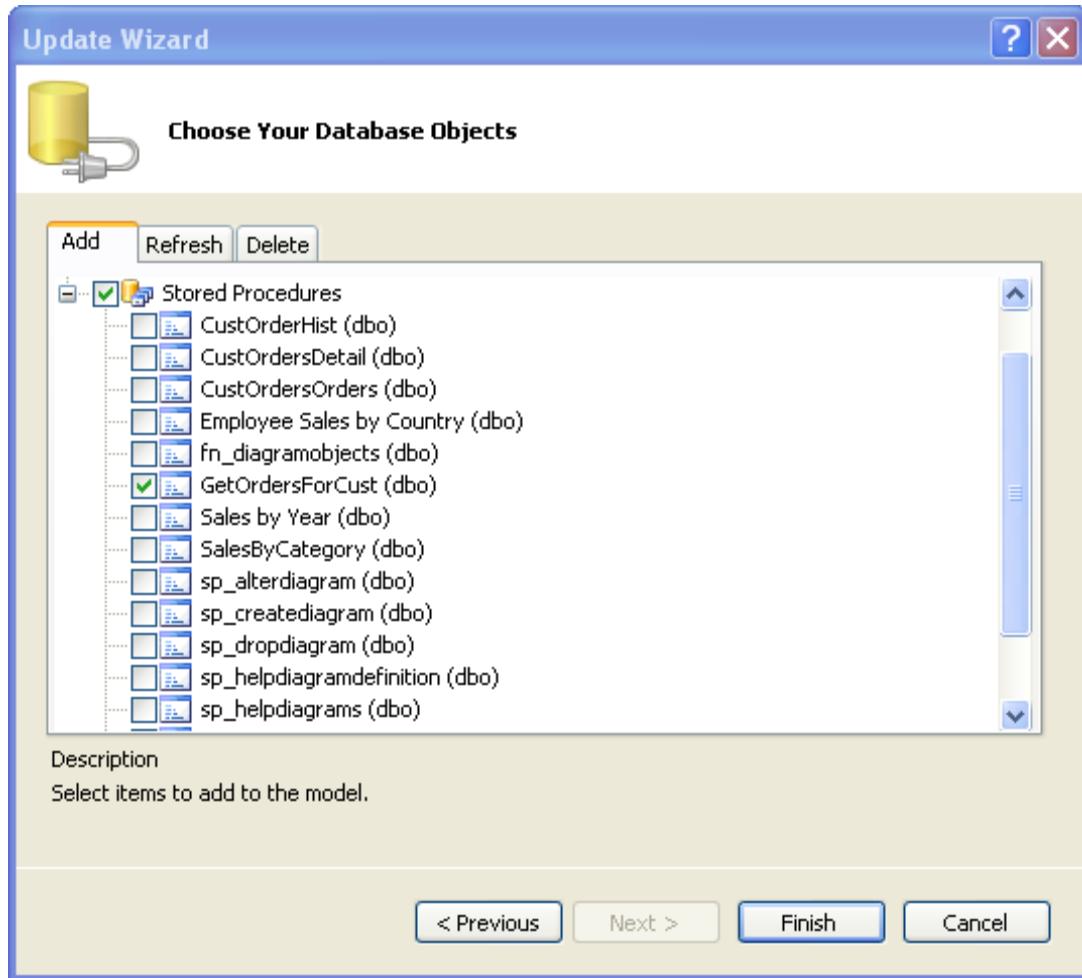
11.1.1 Stored Procedure Returning entities

Problem: You have stored procedure on the database that returns customer entity defined on entity data model. You want to import procedure in your EDM model using the designer support in visual studio. Customer entity also has additional relations such as OrderDetails that you want to access. You want to know different ways to load Order Details for customers that are fetched using a stored procedure and what are the pros and cons for each approach. Since results returned from the stored procedure may be too large, you want to ensure that customers retrieved from the stored procedure are not being tracked by the ObjectStateManager which could lead to performance issues.

Solution: To call a stored procedure in entity framework, you have to first bring the stored procedure in your store model. If you are generating the entity data model the first time, you can use entity data model wizard which lets you import database objects such as stored procedures and functions from the database. If you have already generated your model, you can right click anywhere on the model and select update model from the database to import the stored procedure. Following screen shot shows an example of importing the stored procedure GetOrdersForCust.

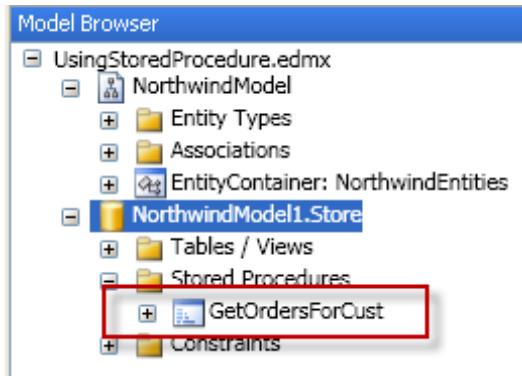


Generating the model from the database first time

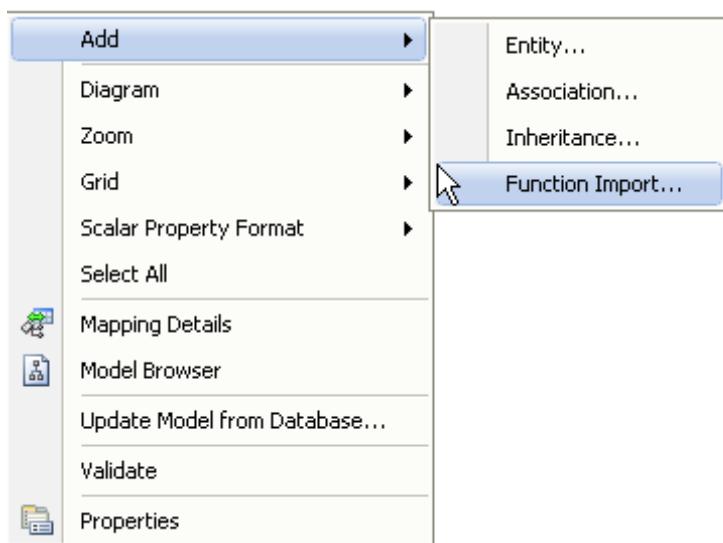


Updating the model by importing stored procedure after the model is created

To confirm that our stored procedure got imported in our model, we can open up model browser window and expand the stored procedures node to see our stored procedure. Screen shot below confirms that GetOrdersForCust stored procedure got successfully imported into our entity data model.

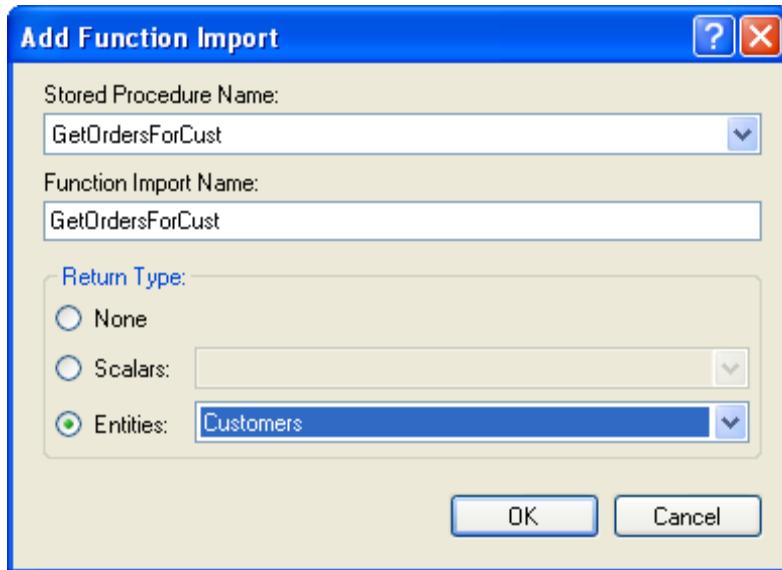


Importing the stored procedure is only the first step in being able to use the stored procedure with our entity data model. The next step is to import the stored procedure in our conceptual model. You can do that in 2 different ways. You can either right click the designer, click add, and select Add Functional Import or open up the entity model browser and right click the stored procedure and select create function import. Screen shot below shows how to access Add Functional Import window.



On the Add Function Import window, I have selected my stored procedure GetOrdersForCust and assigned GetOrdersForCust as the name I will use to reference the method that will call the actual stored procedure on the database. Since the stored procedure returns an entity of type Orders, I am choosing the

return type to be entities and selecting Customer entity. Screen shot below shows the configuration for my stored procedure.



After completing the function import, object context will expose method GetOrdersForCust which takes customerid as parameter and returns orders for the customer. Code below shows how to call the stored procedure to return orders for the customer.

```
var db = new NorthwindEntities();
    var orders = db.GetOrdersForCust("ALFKI");
    foreach (var order in orders)
    {
        Console.WriteLine(order.OrderID);
    }
```

Running the code above generates the following output

```
10643  
10692  
10702  
10835  
10952  
11011  
11078  
Press any key to continue . . .
```

Discussion:

Unlike linq to entity queries that are late bound and are only executed when you iterate over the results, calling stored procedure executes the query immediately. In the above example, when I call GetOrdersForCust, the result returned is of type ObjectResult of T where T in this case is an Order entity. If you are going to be iterating over orders collection multiple time, you have to use a ToList or other operators that can force the result into a collection. Failing to do so would raise an exception as follows.

The result of query cannot be enumerated more than once.

When you call GetOrdersForCust stored procedure, it uses data reader to fetch the results and by calling it once you reach to end of the data reader. Therefore when you call the stored procedure second time, you cannot reiterate the reader starting from top since data readers are only move forward. As a result you get an exception stating that query can only be enumerated once.

When we imported stored procedure GetOrdersForCust into our store model, following xml is written to storage model file or ssdl file.

```
<Function Name="GetOrdersForCust" Aggregate="false"  
         BuiltIn="false" NiladicFunction="false"  
         IsComposable="false"  
  
ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo">  
    <Parameter Name="custid" Type="varchar" Mode="In" />  
</Function>
```

In the above xml, we are declaring a stored procedure by the name GetOrdersForCust that exist in dbo schema. The reason we know it is a stored procedure is because IsComposable is set to false which means that the function cannot be called in from clause of sql statement which stored procedures cannot. We are also specifying that it is not a build in stored procedure by setting BuiltIn to false. Stored procedure takes an input parameter by name custid that has to be varchar data type which we are defining by using Parameter element inside of the function declaration.

When we imported the stored procedure into our conceptual model, function import calls are written in our conceptual model that causes a method to be declared with the correct parameters our stored procedure needs. Xml below shows the functionimport calls declared in our conceptual model.

```
<FunctionImport Name="GetOrdersForCust" EntitySet="Orders"
ReturnType="Collection(NorthwindModel.StoredProcedure.Orders)">
    <Parameter Name="custid" Mode="In" Type="String">
/></FunctionImport></EntityContainer>
<EntityType Name="Customers">
```

The above xml creates a method GetOrdersForCust on our object context that returns a collection of Orders belonging to Orders EntitySet. The method takes in a parameter custid with string data type. The designer also writes an entry in the mapping file that associate our method call in our conceptual model to a call to the stored procedure declared in our store definition. Following entry is written on the mapping file that associates that maps the method to the stored procedure. On the FunctionImportMapping element we specify the functionimportname declared in our conceptual model and specify the functionname defined in our store model to create the mapping.

```
<FunctionImportMapping FunctionImportName="GetOrdersForCust"
FunctionName="NorthwindModel.StoredProcedure.Store.GetOrdersForCu
st" />
```

Based on Function and import calls defined in our model, following code is generated by the designer to execute call to our stored procedure. I have simplified the actual code for clarity and explanation.

```
public ObjectResult<Orders> GetOrdersForCust(string custid)
```

```

    {
        ObjectParameter custidparam = new
ObjectParameter("custid", custid);
        return
base.ExecuteFunction<Orders>("GetOrdersForCust", custidparam);
    }

```

In the above example, we had a stored procedure that returned orders for a customer. What if you wanted to return order details for those orders in a single query? If you try to update the stored procedure to return multiple result set eg Orders and Order Details, entity framework would not allow that. Currently in the v1 release of entity framework stored procedures cannot return multiple result set. The result set can only be a single entity type. To get around this you can create another stored procedure that returns all the order details for a given customer and then manually match attach the OrderDetails that belong to a given Order.

Example below shows how to do this.

```

var db = new NorthwindEntities();
    var orders = db.GetOrdersForCust("ALFKI").ToList();
    var orderdetails =
db.GetOrderDetailsForCust("ALFKI").ToList();
    // one way to attach order.
    orders.ForEach(o =>
o.OrderDetails.Attach(orderdetails.Where(od => od.OrderID ==
o.OrderID)));
    foreach (var order in orders)
{
    Console.WriteLine(order.OrderDetails.Count());
}

```

On the above code, I am getting orders for ALFKI customer by calling our stored proc GetOrdersForCust and forcing the result of the query by calling ToList. Next I am retrieving all the Order Details for all the orders placed by ALFKI customer. Although I have two collections in memory orders and OrderDetails, they are completely disjoined from each other. To create a relationship between the OrderDetails and the orders, I am enumerating the orders in memory and for each order; I query the OrderDetails collection to find only OrderDetails entities that belong to the given order by filtering the results based on ordered. After finding the OrderDetails that belong to the order, I attach the matched OrderDetails to the

OrderDetails collection of the order. Once I have synced up my orders with order details, I am printing the count of the OrderDetails for each order and output is as follows.

On the above example that used stored procs, both orders and order details were disjoined from each other and you had to manually attach the order details to their order. If you had queried the data using Object services than entity framework would take care of merging order Details to their given orders as if they had been retrieved together. Example below show that attach is implicit since Entity framework knows more about the query and is therefore able to get enough information to fix up the relationship entries in the object statemanager for OrderDetails retrieved from the query.

```
var db = new NorthwindEntities();
    var orders = db.GetOrdersForCust("ALFKI").ToList();
    var orderdetails = db.OrderDetails.
        Where(od =>
od.Order.Customer.CustomerID == "ALFKI")
        .ToList();
    foreach (var order in orders)
    {
        Console.WriteLine(order.OrderDetails.Count());
    }
```

The above code, makes separate database calls to first retrieve Orders for ALFKI customer and followed by another query to retrieve all the OrderDetails for ALFKI customer. Although two queries are completely disjoined from each other, entity framework has enough information from the query being executed to determine

that OrderDetails retrieved actually belong to orders that are available in memory and therefore automatically performs the attach for us.

One of the ways to improve application performance in read only scenarios is to query for objects with No Tracking option. Typically if you are not going to be performing inserts and updates on an entity retrieved from the query, you do not want the tracking burden imposed by the ObjectStateManager. When you apply MergeOption.NoTracking on queries issues using ObjectServices whether it is using esql or linq to entities, there is no tracking applied on the entities retrieved. However with stored procedure there is no way to turn this feature off. If stored procedures return entities which are only used for readonly scenarios, you will still incur the tracking cost. The only option to minimize the impact is, when the objects are returned from the stored procedure you can detach them from the context, which will remove tracking information from ObjectStateManager. In the example below, I am calling the stored procedure to return all orders for alfk1 customer. Then grabbing the first order in the list, I query for the object again by calling GetObjectByKey passing the entity key of my order entity.

GetObjectByKey method is because it supports caching concept where it will first see if ObjectStateManager is tracking the object for a given entity. If the object is not found, a request is issued to the database to get the object. The initial call to GetObjectByKey returns the object from the tracking service where as the second call to GetObjectByKey does hit the database because I have detached the orders collection from the object context which forces the order entities to be not tracked.

```
var db = new NorthwindEntities();

    //cant call the stored procedure with no tracking option.
    var orders = db.GetOrdersForCust("ALFKI").ToList();

    //no database call is made because objects are being tracked.
    var fetchedorder = db.GetObjectByKey(orders.First().EntityKey);

    //detach the objects
    orders.ForEach(o => db.Detach(o));
    //no database call is made
    db.GetObjectByKey(orders.First().EntityKey);
```

11.1.2 Stored Procedure Returning Scalar Types

Problem: You have stored procedures that return scalar values. You want to know how to map these stored procedures in your storage model , import them into conceptual model and call them from you application.

Solution: Stored procedures returning scalar value is not fully supported. To consume a stored procedure requires 4 steps. First step is updating the store model from the database. Second step is import the procedure into the conceptual model. Third step is mapping the function on the conceptual model to function definition on the storage model. The last step is generating a method on the objectcontext that calls the stored procedure in the database. This method is generated based on the definition of FunctionImport in the conceptual model. The only step that is not supported in the version 1 of entity framework is the code generation of the method on the object context. To accomplish calling a scalar function, we have to expose a method on ObjectContext that calls the function defined on our conceptual model.

Discussion:

To illustrate how to consume a stored procedure returning scalar values, I have created a stored proc which returns TotalSales for a customer based on customerid parameter. Code below shows the stored procedure TotalSalesForCust that returns a decimal value indicating the total sales generated for a given customer.

```
create proc dbo.TotalSalesForCust
(@custid varchar(5))
as
begin
select sum(Quantity * UnitPrice) TotalSales
from [Order Details] od join orders o on od.OrderID = o.OrderID
join Customers c on o.CustomerID = c.CustomerID
where c.CustomerID = 'ALFKI'
end
```

To use the stored procedure I have to perform my usual step of importing the stored procedure into my store model and from the store model, use the import function screen to add a functionImport call in our conceptual model. Xml below shows definitions created in ssdl, mapping and store model.

SSDL

```

<Function Name="TotalSalesForCust" Aggregate="false" BuiltIn="false"
NiladicFunction="false" IsComposable="false"
ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo">
    <Parameter Name="custid" Type="varchar" Mode="In" />
</Function>

```

CSDL

```

<FunctionImport Name="TotalSalesForCust"
ReturnType="Collection(Decimal)">
    <Parameter Name="custid" Mode="In" Type="String" />
</FunctionImport>

```

MSL

```

<FunctionImportMapping FunctionImportName="TotalSalesForCust"
FunctionName="NorthwindModel.StoredProcedure.Store.TotalSalesForCust" />

```

Although all the entries are created in the model, there is no method created in the generated files to call the stored proc. Due to time constraints this feature did not completely made it version 1 of entity framework and we should expect a complete support to return a scalar value from the stored proc in the next version. In the mean time, you can write some code to open a connection and manually execute the datareader to get value from the stored procedure.

Code below shows how to do that.

```

public partial class NorthwindEntities
{
    private T ExecuteFunction<T>(string functionname, DbParameter[] parameters) where T:struct
    {
        DbCommand cmd =
((EntityConnection)this.Connection).CreateCommand();
        cmd.CommandType = System.Data.CommandType.StoredProcedure;
        cmd.Parameters.AddRange(parameters);
        cmd.CommandText = this.DefaultContainerName + "." +
functionname;
        try
        {
            if (cmd.Connection.State ==
System.Data.ConnectionState.Closed)
            {
                cmd.Connection.Open();
            }
            var obj = cmd.ExecuteScalar();
            return (T)obj;
        }
        catch (Exception)
        {

            throw;
        }
    }
}

```

```

        }
        finally
        {
            cmd.Connection.Close();
        }
    }

    public decimal TotalSalesForCust(string custid)
    {
        var param = new EntityParameter("custid",
System.Data.DbType.String);
        param.Value = custid;
        return ExecuteFunction<decimal>("TotalSalesForCust", new[] {
param });
    }
}

```

The above class is extending NorthWindEntities partial class which derives from Objectcontext. Inside it, I have created a generic method ExecuteFunction that executes a function and returns T where T is any value type. To execute a stored procedure, I first get access to my connection object and cast it to EntityConnection. Using the connection, I create a command object and assign its Commandtext and CommandType to the stored procedure name passed into the method. I am also adding parameters collection to my command since TotalSalesForCust requires customerid as a parameter. After opening the connection I simply call execute scalar and cast it the value returned to my generic type.

To consume the generic function, I have created another method which passes ExeucteFunction the name of the stored procedure to call and the parameters required by the stored procedure. Having done this, consuming the stored procedure requires creating an instance of our objectContext and calling TotalSalesForCust method passing in the name of the customerid as shown below

```

var db = new NorthwindEntities();
decimal totalsales = db.TotalSalesForCust("ALFKI");

```

11.1.3 Stored Procedure Returning Anonymous Type

Problem: You have stored procedures that return summary data. The number of columns returned in the summary data does not match to any entity defined on your conceptual model. You want to know how to map the stored procedure to your conceptual model and access the rows returned in a strongly typed manner.

Solution: Entity framework only supports stored procedures that return entities defined on the conceptual model. If the stored procedure return arbitrary number of columns such as returning summary columns for reporting data, importing procedure into the conceptual and mapping the result to any CLR object is not possible. To overcome this problem, you have to create an entity that has same number of properties and data type as the result returned by the stored procedure. Since an entity defined on the conceptual modal has to be mapped to the store model, you have to fake an entity on the store model by using Defining query. Once that's set, you will use the mapping model to map store entity to conceptual entity.

Discussion: First step to using the stored procedure with entity framework is to define the stored procedure in the database. GetCusSales stored procedure below, returns top 5 customers with highest sales. The result returned includes CustomerId and TotalSales column.

```
create proc dbo.GetCusSales
as
begin
select top 5 c.CustomerID,sum(od.UnitPrice * od.Quantity) TotalSales
from Customers c join orders o on c.CustomerID = o.CustomerID
join [Order Details] od on o.OrderID = od.OrderID
group by c.CustomerID
order by 2 desc
end
```

To use the stored procedure with EDM, we will update the model from the database which will write following entry in our ssdl.

```
<Function Name="GetCusSales" Aggregate="false" BuiltIn="false"
NiladicFunction="false" IsComposable="false"
ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo" />
```

To import the function into our conceptual, the function needs to map to an entity. Therefore we will create an entity that matches the column and data type returned from the result of the stored procedure. Using the designer to

create CustomerSale entity generates the following entity definition on my conceptual model.

```
<EntitySet Name="CustomerSales"
EntityType="NorthwindModel.StoredProcedure.CustomerSale" />

<EntityType Name="CustomerSale">
    <Key>
        <PropertyRef Name="CustomerID" /></Key>
        <Property Name="CustomerID" Type="String" Nullable="false" />
    </EntityType>
```

We then need to import our stored procedure from SSDL into conceptual model with return type being the new entity called CustomerSale. Using the Add function import dialog on the designer I have imported the stored procedure and it generated the following statement in our conceptual model

```
<FunctionImport Name="GetCusSales" EntitySet="CustomerSales"
ReturnType="Collection(NorthwindModel.StoredProcedure.CustomerSale)" />
```

The above statement creates a method GetCusSales on our objectcontext that return CustomerSales entity set containing CustomerSale entities.

If you try to compile the edmx file at this stage, you will get validation errors. This is because entity framework requires an entity defined on the conceptual model to be mapped to something on the storage model. To achieve that purpose we will fake out an entity set and entity type on the storage model (SSDL) as follows.

```
<EntitySet Name="CustomerSales"
EntityType="NorthwindModel.StoredProcedure.Store.CustomerSale">
    <DefiningQuery>
        faking out entityset
    </DefiningQuery>
</EntitySet>

<EntityType Name="CustomerSale">
    <Key>
        <PropertyRef Name="CustomerID" />
    </Key>
    <Property Name="CustomerID" Type="nvarchar"
Nullable="false" />
    <Property Name="TotalSales" Type="decimal"
Nullable="false" />
</EntityType>
```

The above xml creates an entity set using a defining query. Since we are faking a query, defining query does not contain anything. We are also defining an entity type Customer Sale which we will use in our mapping layer to map Customer Sale on the conceptual model to storage model. EntitySetMapping section defined below in our msl layer is what maps our customerSales Entityset on the conceptual model to customerSales on entityset.

```
<EntityTypeMapping  
TypeName="IsTypeOf(NorthwindModel.StoredProcedure.CustomerSale)">  
    <MappingFragment  
        StoreEntitySet="CustomerSales">  
            <ScalarProperty  
                Name="CustomerID" ColumnName="CustomerID" />  
            <ScalarProperty  
                Name="TotalSales" ColumnName="TotalSales" />  
        </MappingFragment>  
    </EntityTypeMapping>  
</EntitySetMapping>
```

Once we have the mapping set correctly, we can consume the stored procedure by calling GetCusSales method available on our object context. Code below loops through CusSales object returned by GetCusSales method and prints the customerid and TotalSales on the console window.

```
private static void StoredProcedureReturningAnonymousType()  
{  
    var db = new NorthwindEntities();  
    foreach (var cussale in db.GetCusSales())  
    {  
        Console.WriteLine("CustID {0} Sales  
{1}", cussale.CustomerID, cussale.TotalSales.ToString("c"));  
    }  
}
```

```
C:\WINDOWS\system32\cmd.exe
CustID  QUICK Sales $117,483.39
CustID  SAUER Sales $115,673.39
CustID  ERNSH Sales $113,236.68
CustID  HUNGO Sales $57,317.39
CustID  RATTC Sales $52,245.90
Press any key to continue . . .
```

Stored Procedure returning entities that are partially filled will appear to work but when you execute them you get a runtime exception complaining the columns missing from the entity. Unlike entity framework, linq to sql allows you to return partially filled entities such as customer from both stored procedures and dynamic sql queries. The only constraint it imposes on either option is that the result set returned must have the primary key column included. In addition, if the stored procedure returns an arbitrary number of columns such as columns containing summary data, dragging the stored procedure on linq to sql designer would generate a class that matches columns and the return type defined on the stored procedure.

11.1.4 Stored Procedure with Command Text Option

Problem: Instead of defining a stored procedure to reuse sql logic, you want to use command Text option for functions definition to embed the sql statement inside the SSDL layer.

Solution: In most scenarios, if you have some complex sql statements that requires various joins and makes use of operators that are only available on sql server, you would prefer taking the route of stored procedure. In the case

where you do not have permissions to create stored procedure or do not want to manage separate deployments for application and database, you can use the commandText property on the function definition on the ssdl layer to embed stored procedure logic. This allows reuse of certain database logics without actually defining a stored procedure on the database. Example below embeds a sql statement to get customer with the highest total purchase.

```

<Function Name="CustomerWithHighestSales"
Aggregate="false"
           BuiltIn="false" NiladicFunction="false"
IsComposable="false"

ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo">
    <!-- cannot use command text with functions.
allows to execute multiple statement.-->
    <CommandText>
        select *
        from Customers
        where CustomerID = (
            select top 1 o.CustomerID
            from Orders o join [Order Details] od on
o.OrderID = od.OrderID
                group by o.CustomerID
                order by SUM(od.UnitPrice * od.Quantity)
desc
        )
    </CommandText>
</Function>
```

Discussion: On the above example, I am declaring a function CustomerWithHighestSale and embedding my sql logic to get the customer with the highest purchase inside of the CommandText property. Since the function is not defined on the database and the logic of the code resides inside the CommandText property, I have not defined any value for StoreFunctionName property available on the Function element. If the function exists on the database and has a different name than what is defined by the Name property on the function element than you have to use StoreFunctionName property to specify the name of defined on the database. After defining the function on the ssdl model, function is imported into conceptual model using FunctionImport statement declared on the csdl as follows.

```
<FunctionImport Name="CustomerWithHighestSales"
EntitySet="Customers"
ReturnType="Collection(NorthwindModel.StoredProcedure.Customer)">
```

The above function import defines the name that would be created on the objectcontext and will be used by the application layer to consume the stored procedure. Using the return type and entity set, we are also defining that stored procedure would return a collection of Customer entities that would be mapped to Customers entityset. Entityset is required because an entity customer could be part of many entity sets. To map the method definition on the object context to the store definition of the function, we have to create a mapping by using FunctionImportMapping element on msdl as shown in the example below. FunctionImportMapping specifies the functionImportname defined on our conceptual model that will map to the function name defined on our storage model.

```
<FunctionImportMapping
FunctionImportName="CustomerWithHighestSales"
FunctionName="NorthwindModel.StoredProcedure.Store.CustomerWithHighestSales" />
```

In the code below, I am calling CustomerWithHighestSales method defined on the object context to retrieve a collection of customers returned from the stored procedure. Preferably, I want the method to return a single entity instead of returning a collection since I know my stored procedure returns a single an entity. Currently there is no way to configure on the conceptual model to only return a single entity. The return type declaration on the FunctionImport has to be a collection and if you change that to be a single entity, EF validation engine will complain that return type must be a collection. To solve the problem, I use First operator to return a single entity and print the customerId on the console window.

```
var db = new NorthwindEntities();
var valuedcustomer = db.CustomerWithHighestSales().First();
Console.WriteLine(valuedcustomer.CustomerID);
```

Example that you saw with CustomerWithHighestSales should not indicate that you can only call put select statements inside of a commandText. You

can leverage everything that is available on the database such as calling other stored procs and declaring variable. In the example below, inside of UpdateCustomerSummary function's commandText property, I am declaring two variables totalorders and totalpurchases. These variables are populated by executing Custstats procedure which returns the total orders and total purchases of a given customer as an outparameter. Using the values obtained from the stored procedure, I am updating CustomerSummary table using merge statement. Merge statement was introduced with sql server 2008. It allows you to perform insert, updates and delete using a single statement. In my case, I am checking to see CustomerSummary table has a record for customerid. If the customerid is not found, I am inserting into the CustomerSummary table with customerid, total orders and total purchases. If a record is found matching a customerid, I am updating my summary information with the updated data received from the stored procedure. In case if you are wondering where is customerid variable is coming from, it is declared as a parameter to the function and passed in as an input parameter to command text property of the function.

```

<Function Name="UpdateCustomerSummary" IsComposable="false"
Schema="dbo">
    <CommandText>
        DECLARE
            @TotalOrders int,
            @TotalPurchases decimal(18, 0)

        EXEC [dbo].[CustStats]
            @custid,
            @TotalOrders = @TotalOrders OUTPUT,
            @TotalPurchases = @TotalPurchases

        OUTPUT

        MERGE CustomerSummary as cs
        USING (
            SELECT @custid CustomerID,
            @TotalOrders TotalOrders,
            @TotalPurchases TotalPurchases
        ) as csr
        on cs.CustomerID = csr.CustomerID
        WHEN MATCHED THEN
        UPDATE SET
            cs.TotalOrders = csr.TotalOrders,

```

```

        cs.TotalPurchases = csr.TotalPurchases
        WHEN NOT MATCHED THEN

    INSERT(CustomerID,TotalOrders,TotalPurchases)
        VALUES
    (csr.CustomerID,csr.TotalOrders,csr.TotalPurchases);
        </CommandText>
        <Parameter Name="custid" Type="nvarchar"
Mode="In" />
        </Function>

```

As we have done in the past, we have to also define the function on the conceptual layer and specify mapping in msdl to map to the stored procedure. In the v1 release of the EF there is no code generated on the object context to call a stored procedure with no result set; therefore we have to write little bit of boiler plate code to execute the stored procedure. Code below shows the completed version with all the declarations for msl, csdl and executing the method using ObjectContext.

CSDL

```

<FunctionImport Name="UpdateCustomerSummary">
    <Parameter Name="custid" Mode="In"
Type="String" />
    </FunctionImport>

```

MSL

```

<FunctionImportMapping
FunctionImportName="UpdateCustomerSummary"
FunctionName="NorthwindModel.StoredProcedure.Store.UpdateCustomerSummary" />

```

Object Context

```

private void ExecuteNonQuery(string functionname,
DbParameter[] parameters)
{
    DbCommand cmd = this.Connection.CreateCommand();
    cmd.CommandType =
System.Data.CommandType.StoredProcedure;
    cmd.Parameters.AddRange(parameters);
    cmd.CommandText = this.DefaultContainerName + "." +
functionname;

```

```

        if (cmd.Connection.State ==
System.Data.ConnectionState.Closed)
{
    cmd.Connection.Open();
}
cmd.ExecuteNonQuery();
}
public void UpdateCustTotal(string custid)
{
    var param = new EntityParameter("custid",
System.Data.DbType.String);
param.Value = custid;
this.ExecuteNonQuery("UpdateCusTotal",
new[] { param });
}

```

Calling the method on the objectcontext from our application

```

var db = new NorthwindEntities();
db.UpdateCustomerSummary("ALFKI");

```

Output below confirms that ALFKI record in Customer Summary table got updated.

	CustomerID	TotalOrders	TotalPurchases
1	ALFKI	14	4862

11.1.5 Stored Procedure with output parameters

Problem: You have stored procedures defined on the database that has input and output parameters. You want to learn how to call these procedures and get output parameters back using entity framework.

Soloution: In the v1 release of entity framework, stored procedure with output parameters are not supported directly. You can define the stored procedure on the storage model, import it into the conceptual model and then

applying mapping, however there is no method generated on the object context based on the definition in the conceptual modal. To call a stored procedure that has no results set, you have to access the EntityCommand object exposed on the objectcontext and execute the stored procedure from there. Example below shows how the ssdl definition would look like for a stored procedure with both input and output parameters.

```
<Function Name="CustStats" Aggregate="false" BuiltIn="false"
NiladicFunction="false" IsComposable="false"
ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo">
    <Parameter Name="custid" Type="nchar"
Mode="In" />
    <Parameter Name="TotalOrders" Type="int"
Mode="InOut" />
    <Parameter Name="TotalPurchases"
Type="decimal" Mode="InOut" />
</Function>
```

Discussion:

On the above example, I have declared CustStats function with 3 parameters; first parameter custid is set as input by setting the mode to In and rest two parameters are defined as InOut parameter. To call the stored procedure from object context, a function must be defined in the conceptual model and mapped to the stored procedure using the mapping. Code below shows the conceptual and mapping definitions.

CSDL

```
<FunctionImport Name="CustStats">
    <Parameter Name="custid" Mode="In"
Type="String" />
    <Parameter Name="TotalOrders"
Mode="InOut" Type="Int32" />
    <Parameter Name="TotalPurchases"
Mode="InOut" Type="Decimal" />
</FunctionImport>
```

MSL

```
<FunctionImportMapping FunctionImportName="CustStats"
FunctionName="NorthwindModel.StoredProcedure.Store.CustStats"
/>
```

As mentioned earlier when you define a function on the conceptual model there is no method generated on the object context if the stored procedure returns no result. To call the stored procedure we can create a partial class for the objectcontext and create a method that calls the stored procedure. Example below creates GetCustStats method that executes our stored procedure in the database.

```

public void GetCustStats(string custid, ref int totalorders,
ref decimal totalpurchases)
{
    var dbparams = new DbParameter[]
    {
        new
EntityParameter{ParameterName="custid", DbType=
DbType.String, Value=custid},
        new
EntityParameter{ParameterName="TotalOrders", DbType=
System.Data.DbType.Int32, Direction =
ParameterDirection.Output},
        new
EntityParameter{ParameterName="TotalPurchases", DbType=
System.Data.DbType.Decimal, Direction =
ParameterDirection.Output}
    };
    ExecuteNonQuery("CustStats", dbparams);
    totalorders =
Convert.ToInt32(dbparams[1].Value);
    totalpurchases =
Convert.ToDecimal(dbparams[2].Value);
}

private void ExecuteNonQuery(string functionname,
DbParameter[] parameters)
{
    DbCommand cmd = this.Connection.CreateCommand();
    cmd.CommandType =
System.Data.CommandType.StoredProcedure;
    cmd.Parameters.AddRange(parameters);
    cmd.CommandText = this.DefaultContainerName +
" ." + functionname;
    if (cmd.Connection.State ==
System.Data.ConnectionState.Closed)
    {
        cmd.Connection.Open();
    }
}

```

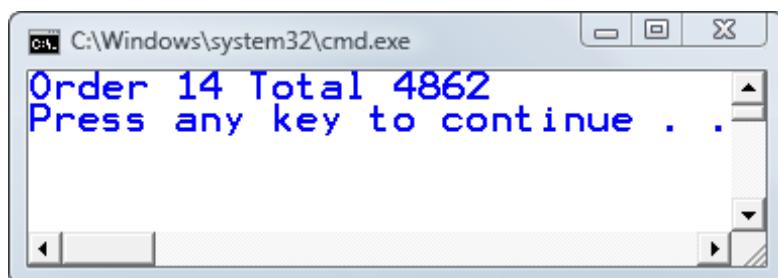
```
        cmd.ExecuteNonQuery();
    }
```

In the GetCustStats method, I am creating 3 entity parameters with the names that match the definition on the ssdl model. For totalorders and totalpurchases, I have declared them as output parameters. To call the stored procedure, I have created a generic method Execute NonQuery that takes a function name and parameters to the function.

ExecuteNonQuery method, gets a reference the EntityCommand object, sets the command to the stored procedure, assigns the CommandText property of the stored procedure, adds parameters to the parameters collection of the command object and finally calls ExecuteNonQuery to execute the stored procedures.

After executing the procedure, I am getting the output values returned on the parameter objects and assigning it to variables passed in as reference to GetCustStats method. To call GetCustStats from application code, we can create an instance of the ObjectContext and call the method with the appropriate parameters. Code below shows how to do that.

```
private static void StoreProcedureWithOutPutParamters()
{
    var db = new NorthwindEntities();
    int totalorders = 0;
    decimal totalpurchases = 0.0M;
    db.GetCustStats("ALFKI", ref totalorders,
ref totalpurchases);
    Console.WriteLine("Order {0} Total
{1}", totalorders, totalpurchases);
}
```



On the above code, I am creating instance of NorthWindEntities, our object context and calling GetCustStats method we just created, passing

totalorders and totalpurchases parameters as reference that are assigned correct values returned from the output parameters of the stored procedure. To confirm the results, I am printing the result to output window.

11.1.6 Stored Procedure Returning Inheritance Hierarchy

Problem: You have modeled product table in your conceptual model using table per hierarchy inheritance. Any products that are discontinued is defined using DiscontinuedProduct entity which inherits from Product entity. You have a stored procedure that returns products that also contains discontinued products. You want to ensure that products returned from the stored procedure gets correctly mapped to the inheritance structure defined on the conceptual model.

Solution: If products returned from the stored procedure are mapped to inheritance hierarchy on the conceptual model, the mapping needs to be correctly updated to take the inheritance into account. If you are using the designer, you can complete the entire process of updating the store model from the database and then import the stored procedure into the conceptual model which will generate a method on the datacontext to call the stored procedure. However the mapping the designer generates is agnostic of the inheritance hierarchy you have defined. Therefore you need to manually fix the mapping definition on the msdl layer.

Discussion:

To walk through the example, we will create a stored procedure that returns two products with second product being discontinued. Stored procedure below achieves our requirement.

```
ALTER proc [dbo].[GetSimpleProds]
as
begin
select top 1 * from SimpleProduct where Discontinued = 1
union
select top 1 * from SimpleProduct where Discontinued = 0
end
```

On the above stored procedure, I am returning the top 1 product for both regular products and discontinued products. To define the stored procedure into our storage model and import the procedure into our conceptual model, we will update ssdl and csdl layer as follows.

SSDL

```
<Function Name="GetSimpleProds" Aggregate="false"
BuiltIn="false" NiladicFunction="false" IsComposable="false"
ParameterTypeSemantics="AllowImplicitPromotion" Schema="dbo"
/>
```

CSDL

```
<FunctionImport Name="GetSimpleProds"
EntitySet="SimpleProducts"
ReturnType="Collection(NorthwindModel.StoredProcedure.SimpleProduct)" />
```

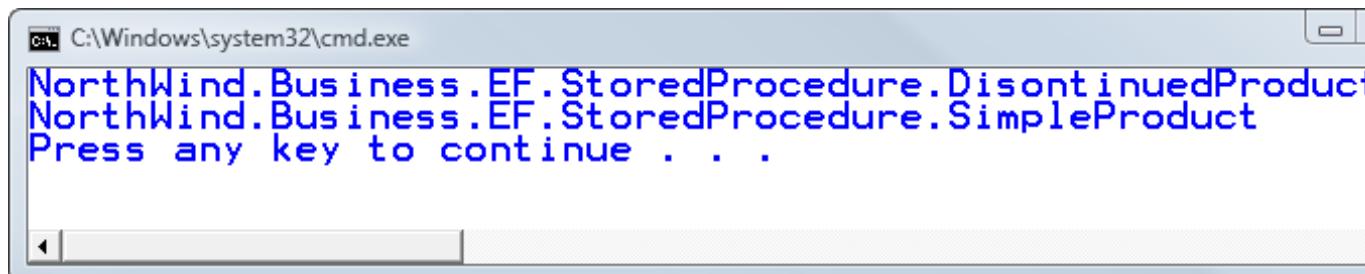
On the above code, in our conceptual model, we are setting the return type to be collection of SimpleProducts which is the base class for all products. To map the conceptual model to the storage model, mapping definition is updated to reflect inheritance structure.

```
<FunctionImportMapping FunctionImportName="GetSimpleProds"
FunctionName="NorthwindModel.StoredProcedure.Store.GetSimpleProds"
">
    <ResultMapping>
        <EntityTypeMapping
            TypeName="NorthwindModel.StoredProcedure.SimpleProduct">
            <Condition
                ColumnName="Discontinued" Value="0"/>
            </EntityTypeMapping>
            <EntityTypeMapping
                TypeName="NorthwindModel.StoredProcedure.DiscontinuedProduct">
                <Condition
                    ColumnName="Discontinued" Value="1"/>
                </EntityTypeMapping>
            </ResultMapping>
    </FunctionImportMapping>
```

On functionimportmapping element, we are mapping the result returned from the stored procedure using conditions. If the Discontinued column has a value

of 0, the record needs to be mapped to SimpleProduct entity; if the column has discontinued set to 1, we map the record to Discontinued Product entity. To call the stored procedure from our code, we have to create an instance of object context and call GetSimpleProds method generated on the objectcontext. The result returned is products with base class reference of SimpleProduct class. Code below shows an example of requesting products from the stored procedure.

```
private static void StoredProcedureReturningBaseClass()
{
    var db = new NorthwindEntities();
    var prods = db.GetSimpleProds();
    foreach (var prod in prods)
    {
        Console.WriteLine(prod.GetType().FullName);
    }
}
```



The screenshot shows a Windows Command Prompt window titled 'cmd' with the path 'C:\Windows\system32\cmd.exe'. The output of the program is displayed, showing two class names: 'NorthWind.Business.EF.StoredProcedure.DiscontinuedProduct' and 'NorthWind.Business.EF.StoredProcedure.SimpleProduct'. Below the output, the message 'Press any key to continue . . .' is visible.

```
C:\Windows\system32\cmd.exe
NorthWind.Business.EF.StoredProcedure.DiscontinuedProduct
NorthWind.Business.EF.StoredProcedure.SimpleProduct
Press any key to continue . . .
```

On the above code, I am calling GetSimpleProds method to return all products from the stored procedure. To confirm that correct product types are returned, I am looping through the products collection and printing the type of the product. Output on the console confirms that one of the products is a DiscontinuedProduct.

12. Mapping Crud Operations to Stored Procedure

When an entity is dragged on the EDM designer, it gets out of the box support for inserts, updates and deletes. However if your requirements mandates you to write stored procedures for performing crud operations, Entity framework provides various options to map your entity cruds operations to stored procedures defined on the database. Following scenarios would be covered.

1. Mapping entity crud operations to stored procedures using Edm designer.

12.1.1 Using EDM designer to Map Crud Operations to Stored Procedures

Problem: You have created stored procedures to insert, update and delete categories in the database. You want to make sure that when a category is inserted, updated or deleted using entity data model, EF, instead of generating a dynamic sql statement to perform the operation should use your stored procedures to carry out the operation.

Solution: I would personally prefer to use dynamic sql statements generated by entity framework because this is one less code that I have to maintain and plus it is the same code that gets repeated over and over so why not let the framework handle crud for you. On the contrary store procedures would offer better performance because they are compiled. In addition an application may not have the direct privileges to insert, update or delete directly into the table and dba requires you to use stored procedures to perform operations on a table. One of the other reasons to use a stored procedure is to apply database level security where depending upon the privileges of a user; he/she may not have privileges to delete items from a table but would have the ability to insert records into the table. To use stored procedures defined on the database to perform crud, you have to import the procedures into SSDL model and then use the stored procedure mapping dialog to map inserts, updates and deletes for an entity. Currently EF requires all operations for crud to be performed using stored procedures or let the framework handle the crud for you. There is no middle ground where you can use stored procedures for inserts and updates and delete be done using entity framework. Example

below shows category crud being mapped to insert, update and delete stored procedure.

The screenshot displays the EntityDataSource Configuration tool interface. At the top, there is a tree view of the 'Category' entity, which contains 'Scalar Properties' (CategoryID, CategoryName, Description) and 'Navigation Properties'. Below this, the 'Mapping Details - Category' tab is active. The mapping table lists three stored procedures: 'Insert Using InsertCategory', 'Update Using UpdateCategory', and 'Delete Using DeleteCategory'. Each procedure maps parameters to entity properties. For 'Insert Using InsertCategory', parameters @catname and @description map to CategoryName and Description respectively, and the result columncatid maps to CategoryID. For 'Update Using UpdateCategory', parameters @catid, @catname, and @description map to CategoryID, CategoryName, and Description respectively. For 'Delete Using DeleteCategory', the parameter @catid maps to CategoryID.

Parameter / Column	Operator	Property
Functions		
Insert Using InsertCategory		
Parameters		
@catname : varchar	←	CategoryName : String
@description : varchar	←	Description : String
Result Column Bindings		
catid	→	CategoryID : Int32
<Add Result Binding>		
Update Using UpdateCategory		
Parameters		
catid : int	←	CategoryID : Int32
@catname : varchar	←	CategoryName : String
@description : varchar	←	Description : String
Result Column Bindings		
<Add Result Binding>		
Delete Using DeleteCategory		
Parameters		
catid : int	←	CategoryID : Int32

Discussion: To map Category crud operations to stored procedures, we will create 3 stored procedures that will do the job of inserting, updating and deleting. Code below shows our stored procedures.

```
create proc dbo.InsertCategory
(
@catname varchar(50),
@description varchar(100)
)
as
begin
insert into categories(categoryname,[description]) values
(@catname,@description)
```

```

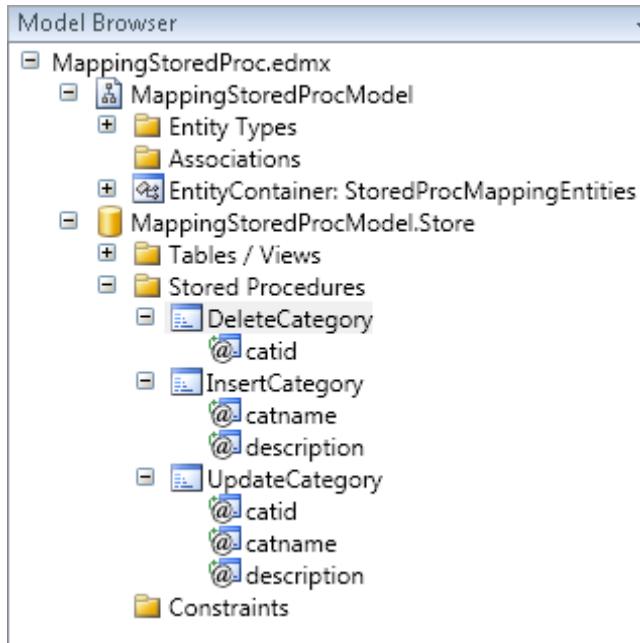
select SCOPE_IDENTITY() as catid
end

create proc dbo.UpdateCategory
(
@catid int,
@catname varchar(50),
@description varchar(100)
)
as
begin
update categories
set categoryname = @catname,[description] = @description where CategoryID =
@catid
end

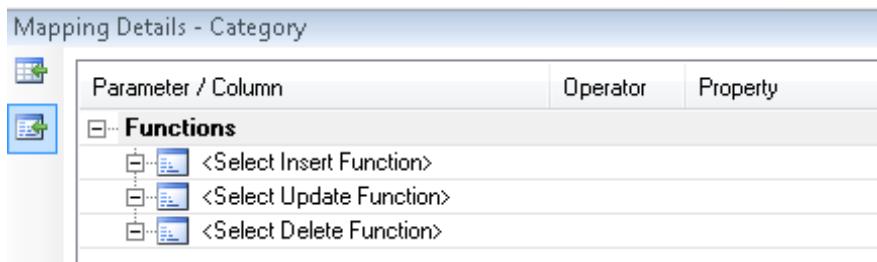
create proc dbo.DeleteCategory(@catid int)
as
begin
delete categories where categoryid = @catid
end

```

On the above example, InsertCategory stored procedure takes catname and description and inserts it into the category table. To send the categoryid just inserted, I am doing a select on scope_identity to get the id of the last category inserted. It is mandated by the Entity framework to return the id of record inserted using select. If you have stored procedure that returns the id as an output parameter, EF won't be able to retrieve the id and assign it back to the property on the entity. It is expected that version 2 of the entity framework would relax those restrictions and allow you to use stored procedure without having restrictions on how the parameters are returned. If you have fair amount of stored procedures that rely on out parameters, then there is a work around that I will show that will enable you to consume stored procedures that return output parameters. These work around are not supported by the designer and may get overwritten when you update the model form the database. For update stored procedures, I am updating the category columns from the values passed in the parameters with category record filtered based on categoryid. For delete, I am deleting category record where categoryid matches the categoryid passed into the parameter. The next step is importing the stored procedures into the entity data model. Screen shot below confirms that we have added our procedure into SSDL layer.



From the screen shot above, you can see that we have imported InsertCategory, UpdateCategory and Delete Category stored procedures. As mentioned earlier, EF requires you to map all stored procedures to each of the three operations on category entity. Failing to perform mapping on one of the actions would invalidate the model and raise compilation errors. To map the stored procedure, right click on the entity and select stored procedure mappings. This will open up a dialog as shown below.

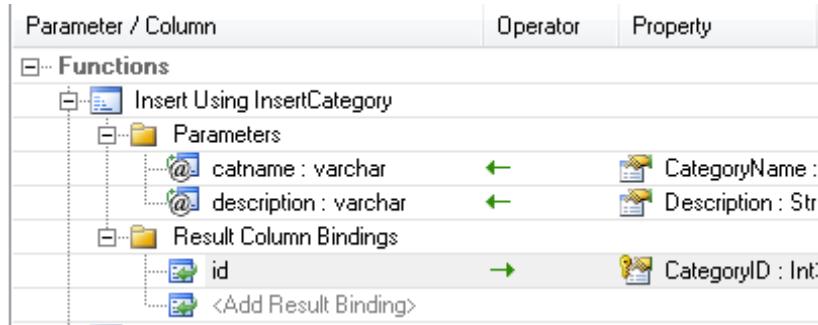


To map insert procedure, select insert function dropdown and choose InsertCategory procedure. If the columns defined on your stored procedure have the exact names defined on the property, mapping will be performed automatically. If not than you have to manually select the properties that map to the stored procedure columns. To map the id returned from inserting the category, select the result column binding and type id for column returned from the select statement performed on the stored procedure. The Id is mapped to CategoryId property on Category entity. The reason we are using Id column for the result column binding is because, we aliased the column returned from select of scope_identity to be Id.

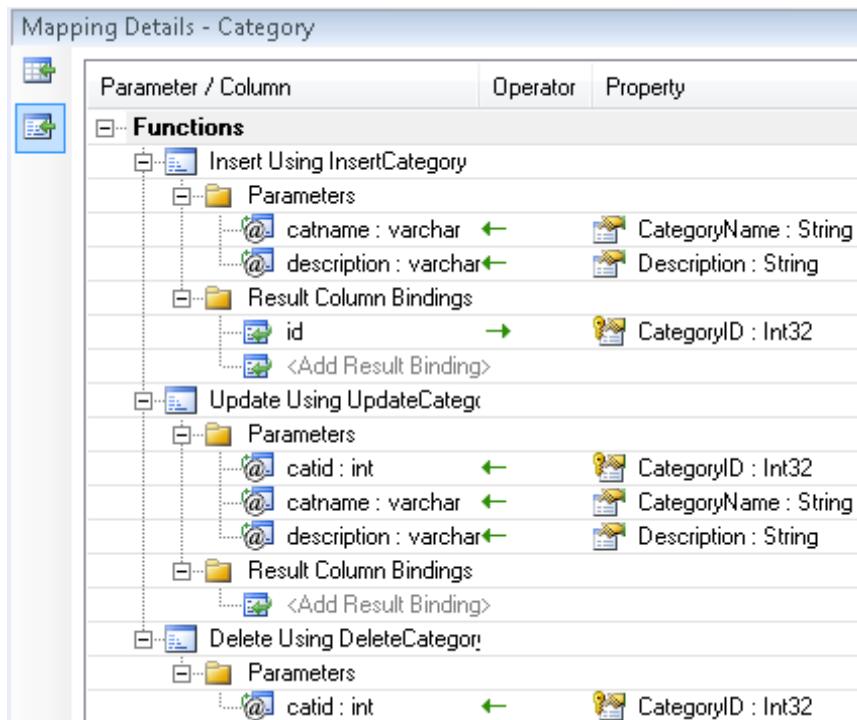
Code below shows the select statement used in the stored procedure to return the id of the category inserted.

```
select SCOPE_IDENTITY() as catid
```

Screen shot below shows how the mapping is configured for insert category stored procedure.



To map the update procedure, select update function from the dropdown which will automatically map the procedure column names to properties on the category entity where the names match. Since there is no return value to capture for update we do not have to worry about result binding. Similarly to map delete procedure, select delete function from the dropdown and map the catid column on the procedure to categoryid property on the category entity. Screen shot below shows the completed mapping for category crud operations.



To test that our crud operations issues calls to our stored procedure instead of generating dynamic sql statements , we can create a new category, update its field and delete the category from the database. During this operation, we can open our profiler to confirm that correct stored procedures are getting executed. Code below performs various crud operations on the category entity.

```
var db = new StoredProcedureMappingEntities();
        var cat = new Category
{
    CategoryName = "HomeStyle Food",
    Description = "Different homemade foods"
};
//calls InsertCategory stored proc
db.AddToCategories(cat);
db.SaveChanges();
//calls UpdateCategory stored proc
cat.Description = "HomeStyle Food delicious";
db.SaveChanges();

//calls delete stored proc.
db.DeleteObject(cat);
db.SaveChanges();
```

On the above example, I am creating a new category instance and marking the category to be added by calling AddToCategories. Calling SaveChanges triggers our InsertCategory stored procedure to be called. To call our update stored procedure, I am simply changing the description of the category and calling savechanges again. Since ObjectStateManager was already tracking the category we just inserted, it knew that changing the description was update operation. To call our deletecategory stored procedure, I am calling DeleteObject method available on the ObjectContext. Calling SaveChanges, triggers the delete process and causes DeleteCategory stored procedure to be called. Profiler screen shot below confirms that operations above caused stored procedure to be called which mapped to Category entity.

EventClass	TextData
Trace Start	
RPC:Completed	exec [dbo].[InsertCategory] @catnam...
RPC:Completed	exec sp_reset_connection
RPC:Completed	exec [dbo].[UpdateCategory] @catid=...
RPC:Completed	exec sp_reset_connection
RPC:Completed	exec [dbo].[DeleteCategory] @catid=16

Earlier, I mentioned that if you have existing stored procedures that return primary key generated value using output parameters, you can work around this limitation in v1 by tweaking the ssdl file directly. When you import a stored procedure in the ssdl file, EF creates a function in ssdl that calls your stored procedure. One of the options available on the function is commandtext which allows you to execute pretty much anything you can do at the database level. We can leverage the commandtext to declare an output parameter to the stored procedure, call the stored procedure with the output parameter declared and after executing the stored procedure we can return the output parameter value as select clause which Entity framework can understand. Following function declaration on the ssdl file shows how to do that.

```
<Function Name="InsertCategory" Aggregate="false"
BuiltIn="false" NiladicFunction="false" IsComposable="false"
ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo">
    <CommandText>
        declare @catid int
        exec dbo.InsertCategoryUsingOutput
        @catid = @catid output,
        @catname = @catname,
        @description = @description
        select @catid as catid
    </CommandText>
    <Parameter Name="catname" Type="varchar" Mode="In" />
    <Parameter Name="description" Type="varchar" Mode="In" />
/>
</Function>
```

On the above function, I am declaring catid parameter and passing that to my InsertCategoryUsingOuptut stored procedure which inserts the category and assigns the cateogryid inserted to the catid output parameter of the stored procedure. Code below shows how InsertCategoryUsingOutput stored procedure looks like.

```
create proc dbo.InsertCategoryUsingOutput
(
@catid int output,
@catname varchar(50),
@description varchar(100)
)
as
begin
insert into categories(categoryname,[description]) values
(@catname,@description)
select @catid = SCOPE_IDENTITY()
```

```
end
```

After executing the stored procedure the output parameter is populated with the categoryid of the inserted category. Since Ef does not understand output parameters, I am performing select to output the categoryid from the output parameter. Notice that my select uses as alias for catid to return the categoryid. This is the same alias name that we will use to apply binding on ResultBinding section of the InsertFunction defined on the mapping layer. Code below shows how we are mapping the parameters on the stored procedure to properties on the entity and also binding the result of CategoryId returned from the select operation to CategoryId property on Category entity.

```
<InsertFunction  
  FunctionName="MappingStoredProcModel.Store.InsertCategory">  
    <ScalarProperty Name="CategoryName"  
      ParameterName="catname" />  
    <ScalarProperty Name="Description"  
      ParameterName="description" />  
    <ResultBinding Name="CategoryID"  
      ColumnName="catid" />  
  </InsertFunction>
```

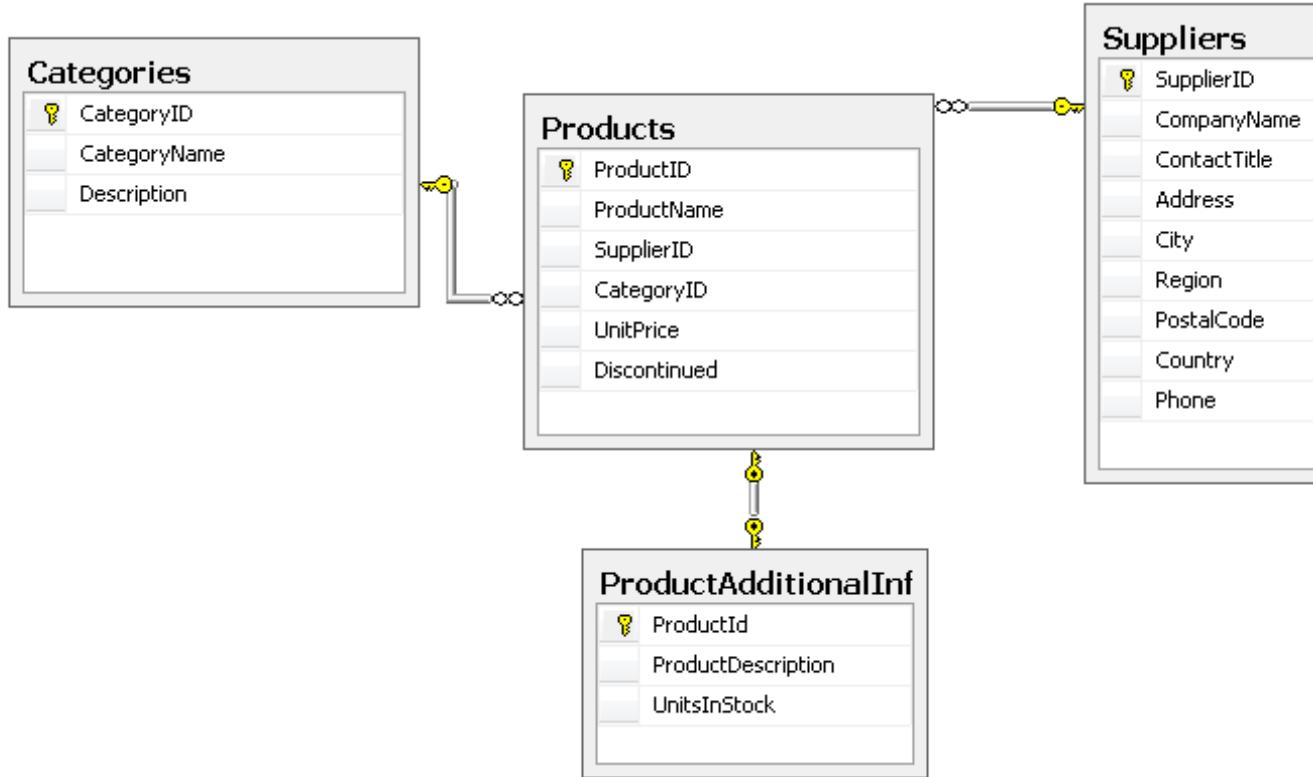
12.1.2 Mapping Associations to Stored Procedure

Problem: You want to map insert, update and delete for product entity defined on the entity data model to stored procedures defined on the database. The product table on the database has column for supplierid that associates the product to a supplier. The product also belongs to a category which is defined by categoryid column on the products table. However product entity on the entity data model does not expose productid and supplierid as its properties. However product entity exposes two navigation relationships Supplier and Category. You want to know how to map the navigation relationships exposed on the product entity to parameters defined on the stored procedure.

Solution: To map the stored procedures to crud operations on product entity, you need to right click on the product entity and select stored procedure mapping. To bind supplierid parameter on the store procedure, select the

property supplierid exposed on the navigation relationship supplier. To bind the categoryid parameter, select the categoryid property on the category navigation relationship exposed on the product entity.

Discussion: To start the discussion we will first see how our database diagram looks like.



On the above screen shot, I have a product table which has many to 1 association with Categories and suppliers. This means that a given category can have many products and a product is provided by a supplier. Additionally product has a 1 to 1 mapping with ProductAdditionalInfo which means if the product has additional info it would be inserted into productadditional info table with using the productid defined by the product table. To enable crud operation on product table, I have created 3 stored procedures shown below.

```

/*insert proc */
ALTER proc [dbo].[InsertProd]
(
@prodname varchar(50),
@unitprice decimal,
@catid int,

```

```

@suppid int
)
as
begin
insert into Products(ProductName,UnitPrice,CategoryID,SupplierID)
values (@prodname,@unitprice,@catid,@suppid)
select SCOPE_IDENTITY() as prodid
end

/* update proc */
ALTER proc [dbo].[UpdateProduct]
(@prodname varchar(50),
@unitprice decimal,
@supplierid int,
@categoryid int,
@prodid int)
as
begin
update Products
set ProductName = @prodname,
UnitPrice = @unitprice,
SupplierID = @supplierid,
CategoryID = @categoryid
where ProductID = @prodid
end

/* delete proc */
ALTER proc [dbo].[DeleteProduct]
(@prodid int,
@supplierid int,
@categoryid int
)
as
begin
delete Products where ProductID = @prodid
end

```

On the InsertProd stored procedure, I am taking parameters required to insert a product row. After the record is inserted, I am doing the select to return productid inserted using scope_identity. The select operation with an alias of prodid is later used by EF to map to ProductID property on the product entity. The updateProd procedures performs update on a row with values form the parameters. What if you have a requirement that when updating a product only unitprice and productname can be changed and supplierid and categoryid cannot be changed once it's set the first time when an insert happens? This scenario is not covered by entity framework. Entity framework requires insert and update stored procedures to have the same number of parameters if you map a stored procedure that does not have all the columns on the product table, you will get an exception as follow.

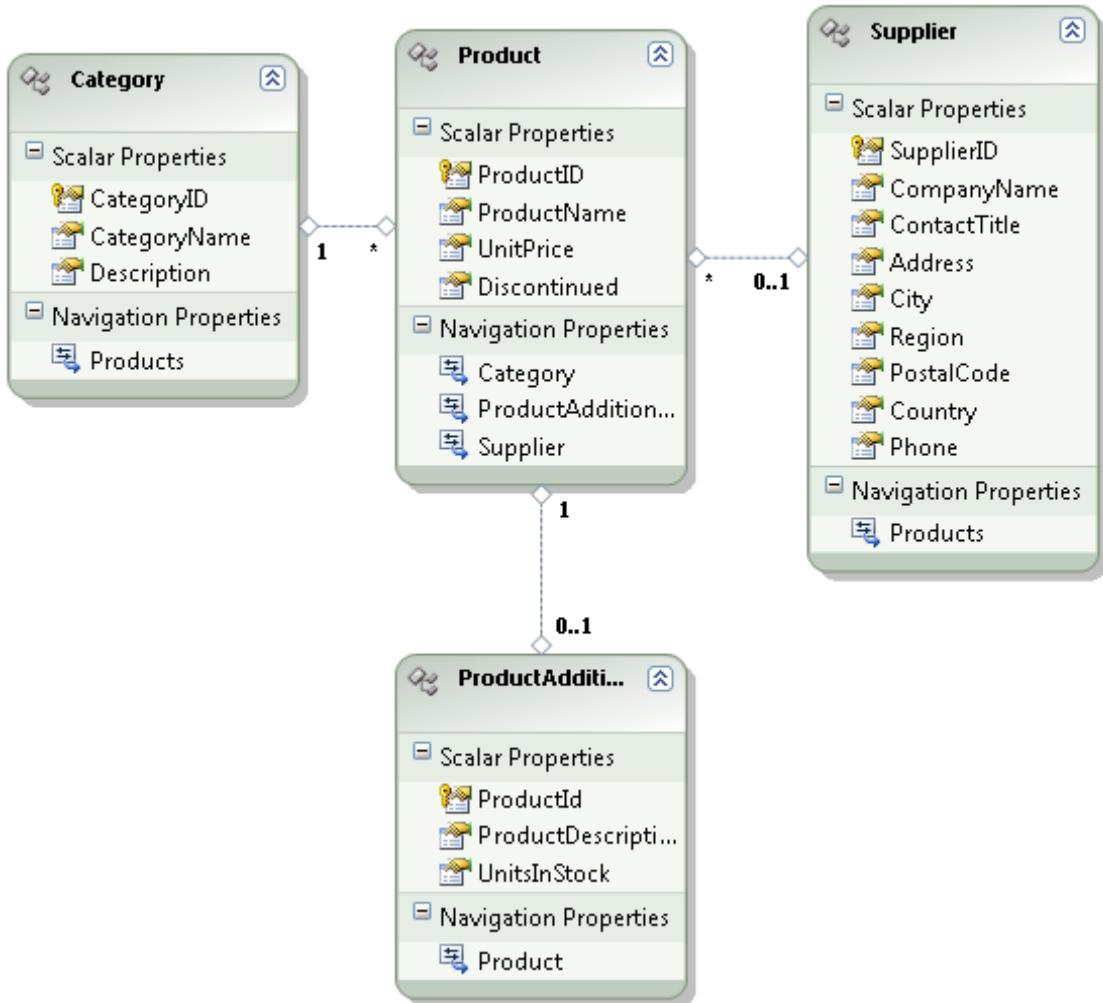
The EntitySet 'Products' includes function mappings for AssociationSet 'FK_Products_Suppliers', but none exists in element 'UpdateFunction' for type 'MappingStoredProcModel.Product'. AssociationSets must be consistently mapped for all operations.

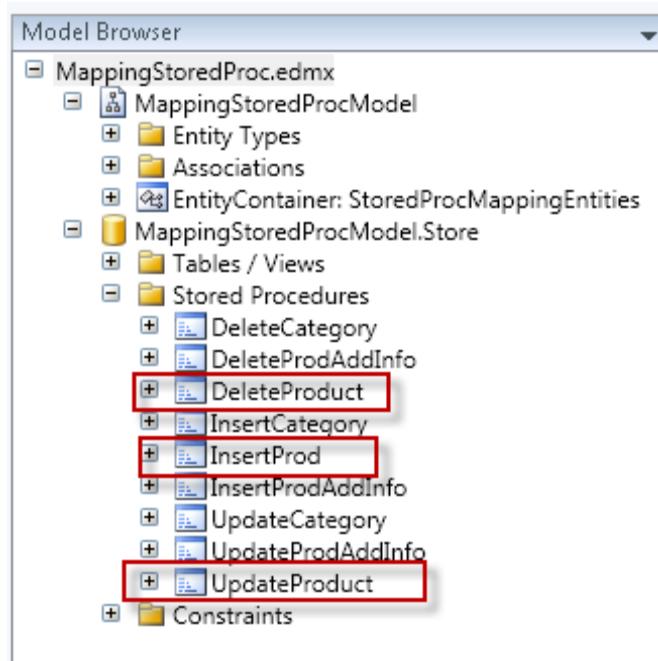
Providing update on limited number of columns is not directly supported by entity framework and the designer, but you can tweak the ssdl file manually if you are not going to overwriting the ssdl file from the database which is what happens when you update the model. Later on in the discussion I will demonstrate how to achieve that.

For delete stored proc, we are faced with the same constraint. Although for deleting a product all you need is the productid. Since EF requires parameters for navigation relationship, such supplier and category exposed on our product entity, we have to provide these parameters regardless if they will be used in the procedure or not. Once again, if you already have stored procedures on the database that does not confirm to EF rules, you have the ability to tweak the ssdl file to map the delete stored procedure with only productid parameter.

To perform the mapping we need to import 4 tables into our model; Category, Product, Supplier and ProductAdditional Info. We also have to import three stored procedures created to insert, update and delete product entity. Following screen show how the model looks like after importing 4 tables.

Entity Data Model





On the entity diagram, product entity has many to 1 and many to 0-1 association to category and supplier entity. However there is 0-1 association between product and productAdditionalInfo which is very close to one to one mapping. One of constraints enforced by entity framework is, if you map product entity using stored procedure, any related entity, product is tied to with 1 to 1 mapping or 1 to 0 mapping must also be mapped using stored procedure. So in our case, we have to provide stored procedure mapping to ProductAdditionalInfo since it is tied to product with 0-1 mapping. We are not obligated to provide mapping for supplier or category because product has many to 1 association with these entities. Failing to provide mapping for ProductAdditionalInfo raises the following error.

If an EntitySet or AssociationSet includes a function mapping, all related entity and AssociationSets in the EntityContainer must also define function mappings. The following sets require function mappings: FK_ProductAdditionalInfo_Products.

To map ProductAdditionalInfo to stored procedures on the database, I have created the following stored procedures.

```
/* insert for product info */
ALTER proc [dbo].[InsertProdAddInfo]
(@prodid int,
@desc varchar(100),
@unitsinstock int)
```

```

as
begin
insert into ProductAdditionalInfo(ProductId,ProductDescription,UnitsInStock)
values (@prodid,@desc,@unitsinstock)
end

/* update product info */
ALTER proc [dbo].[UpdateProdAddInfo]
(@prodid int,
@desc varchar(100),
@unitsinstock int)
as
begin
update ProductAdditionalInfo
set ProductDescription = @desc,
UnitsInStock = @unitsinstock
where ProductId = @prodid
end

/* delete product info */
ALTER proc [dbo].[DeleteProdAddInfo]
(@prodid int)
as
begin
delete ProductAdditionalInfo where ProductId = @prodid
end

```

The above stored procedures follow the same pattern as we used for product table stored procedures. Noticeable difference is, for insertProdAddInfo we are not returning the identity of the productid since productid was originally created by insertproduct stored procedure. We are in fact using the productid created after inserting the product, to use in inserting record inside ProdAdditionalInfo table. DeleteProdInfo stored procedure takes productid to delete additional info for a given product.

Mapping the stored procedures to Product entity is done by selecting the column on the procedure and mapping it to the property on the entity. For the case of supplierid and categoryid parameter, there is no property available on the product entity. Therefore we have to use the navigation relationship exposed on the product entity to map the correct parameter values.

Parameter / Column	Operator	Property
Functions		
Insert Using InsertProd		
Parameters		
@prodname : varchar	←	ProductName : String
@unitprice : decimal	←	UnitPrice : Decimal
@catid : int	←	Category.CategoryID : Int32
@suppid : int	←	Supplier.SupplierID : Int32
Result Column Bindings		
prodid	→	ProductID : Int32
<Add Result Binding>		
Update Using UpdateProduct		
Parameters		
@prodname : varchar	←	ProductName : String
@unitprice : decimal	←	UnitPrice : Decimal
@supplierid : int	←	Supplier.SupplierID : Int32
@categoryid : int	←	Category.CategoryID : Int32
@prodid : int	←	ProductID : Int32
Result Column Bindings		
<Add Result Binding>		
Delete Using DeleteProduct		
Parameters		
@prodid : int	←	ProductID : Int32
@supplierid : int	←	Supplier.SupplierID : Int32
@categoryid : int	←	Category.CategoryID : Int32

On the above screen shot, to map categoryid parameter, I have selected CategoryId property available on the Category navigation property. Similarly for supplierid, I have selected supplierid property on supplier entity. Entity framework requires all 3 operations on an entity to be mapped to stored procedure. Failing to map an operation to a stored procedure would cause the model to not compile. If you feel that delete is not allowed for an entity, you can tweak the ssdl layer using CommandText property to perform no operation when a delete is executed on an entity. Modifying the model is not supported by entity framework designer and would get overwritten if you were to update the model from the database. Example below shows no code executed for a delete scenario.

```

<Function Name="DeleteProdAddInfo" Aggregate="false"
BuiltIn="false" NiladicFunction="false" IsComposable="false"
ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo">
    <CommandText>

```

```

        raiserror(N'Operation not permitted', 16,
1) ;
    </CommandText>
    <Parameter Name="prodid" Type="int" Mode="In" />
</Function>
<Function Name="DeleteProduct" Aggregate="false"
BuiltIn="false" NiladicFunction="false" IsComposable="false"
ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo">
    <CommandText>
        raiserror(N'Operation not permitted', 16,
1) ;
    </CommandText>
    <Parameter Name="prodid" Type="int" Mode="In" />
    <Parameter Name="supplierid" Type="int" Mode="In" />
    <Parameter Name="categoryid" Type="int" Mode="In" />
</Function>

```

On the above code, I am raising an error inside of ComandText property to indicate that delete operation is not permitted. When you define commandText property for a function, EF instead of actually executing a stored procedure executes everything defined inside of ComandText. This is one of the ways you can get your model to compile without mapping to delete stored procedure.

Once the mapping for stored procedure is configured using the designer, following definition is written for product entity on the msdl layer.

```

<EntitySetMapping Name="Products">
    <EntityTypeMapping
TypeName="IsTypeOf(MappingStoredProcModel.Product)">
        <MappingFragment
StoreEntitySet="Products">
            <ScalarProperty
Name="Discontinued" ColumnName="Discontinued" />
            <ScalarProperty
Name="UnitPrice" ColumnName="UnitPrice" />
            <ScalarProperty
Name="ProductName" ColumnName="ProductName" />
            <ScalarProperty
Name="ProductID" ColumnName="ProductID" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping
TypeName="MappingStoredProcModel.Product">
        <ModificationFunctionMapping>

```

```

        <InsertFunction
    FunctionName="MappingStoredProcModel.Store.InsertProd">
                <AssociationEnd
    AssociationSet="FK_Products_Suppliers" From="Products"
    To="Suppliers">

        <ScalarProperty Name="SupplierID" ParameterName="suppid" />
                </AssociationEnd>
                <AssociationEnd
    AssociationSet="FK_Products_Categories" From="Products"
    To="Category">

        <ScalarProperty Name="CategoryID" ParameterName="catid" />
                </AssociationEnd>
                <ScalarProperty
    Name="ProductName" ParameterName="prodname" />
                <ScalarProperty
    Name="UnitPrice" ParameterName="unitprice" />
                <ResultBinding
    Name="ProductID" ColumnName="prodid" />
                </InsertFunction>
                <UpdateFunction
    FunctionName="MappingStoredProcModel.Store.UpdateProduct">
                    <ScalarProperty
    Name="ProductID" ParameterName="prodid" Version="Current" />
                    <AssociationEnd
    AssociationSet="FK_Products_Categories" From="Products"
    To="Category">

        <ScalarProperty Name="CategoryID"
    ParameterName="categoryid" Version="Current" />
                </AssociationEnd>
                <AssociationEnd
    AssociationSet="FK_Products_Suppliers" From="Products"
    To="Suppliers">

        <ScalarProperty Name="SupplierID"
    ParameterName="supplierid" Version="Current" />
                </AssociationEnd>
                <ScalarProperty
    Name="ProductName" ParameterName="prodname" Version="Current" />
                <ScalarProperty
    Name="UnitPrice" ParameterName="unitprice" Version="Current" />
                </UpdateFunction>
                <DeleteFunction
    FunctionName="MappingStoredProcModel.Store.DeleteProduct" >

```

```

        <AssociationEnd
AssociationSet="FK_Products_Categories" From="Products"
To="Category">

    <ScalarProperty Name="CategoryID"
ParameterName="categoryid" />
        </AssociationEnd>
        <AssociationEnd
AssociationSet="FK_Products_Suppliers" From="Products"
To="Suppliers">

    <ScalarProperty Name="SupplierID"
ParameterName="supplierid" />
        </AssociationEnd>
        <ScalarProperty
Name="ProductID" ParameterName="prodid" />
        </DeleteFunction>

    </ModificationFunctionMapping>
        </EntityTypeMapping>
    </EntitySetMapping>

```

Product Entity Type Mapping is split into 3 regions; InsertFunction, UpdateFunction and DeleteFunction. Inside each function we are mapping the scalar properties to the stored procedure parameters. For the case of supplierid and categoryid, we are using AssociationEnd and defining the scalar property of the association set that map to the stored procedure parameter name. In our case, product and supplier entity is mapped using single key. If the join between supplier and product required a composite key than Association End would contain both scalar properties to get mapped to the parameter names. We also have a declaration for ResultBinding for InsertFunction which is used to map the id returned from the stored procedure to be mapped to entity key defined on the product entity.

As mentioned earlier, ProductAdditionalInfo is another entity that is related to product entity; therefore we need to map the ProductAdditionalInfo using stored procedure as follows.

Mapping Details - ProductAdditionalInfo

Parameter / Column	Operator	Property
Functions		
Insert Using InsertProdAddInfo		
Parameters		
@prodid : int	←	Product.ProductID : Int32
@desc : varchar	←	ProductDescription : String
@unitsinstock : int	←	UnitsInStock : Int32
Result Column Bindings		
<Add Result Binding>	No binding	
Update Using UpdateProdAddInfo		
Parameters		
@prodid : int	←	Product.ProductID : Int32
@desc : varchar	←	ProductDescription : String
@unitsinstock : int	←	UnitsInStock : Int32
Result Column Bindings		
<Add Result Binding>		
Delete Using DeleteProdAddInfo		
Parameters		
@prodid : int	←	Product.ProductID : Int32

Mapping for ProductAdditionalInfo table is same as product mapping. The only difference is there is no results binding for insert stored procedure of the table. The reason is productid is not generated by ProdAdditionalInfo table; it is generated when product is inserted in the product table. InsertProdAddInfo stored procedure only uses the productid that was inserted to add additional information about the productid.

After configuring the mapping we can write code that forces the stored procedures to get called such as inserting, updating and deleting a product. Code below shows how to insert the product entity using stored procedure.

```
var db = new StoredProcedureEntities();
    var cat = new Category { CategoryName = "Makeups" };

    var supplier = new Supplier { CompanyName = "Hairs
LTD" };
    var product = new Product
{
    ProductName = "Hair Formula",
    UnitPrice = 12.0M,
    Supplier = supplier,
```

```

        Category = cat,
        ProductAdditionalInfo =
            new ProductAdditionalInfo
            {
                ProductDescription = "Hair Dye",
                UnitsInStock = 20
            }
        };
        db.AddToProducts(product);
        db.SaveChanges();
        Console.WriteLine("Insert Details");
        Console.WriteLine("Cat ID: {0}
Name:{1}", cat.CategoryID, cat.CategoryName);
        Console.WriteLine("Supp ID: {0}
Name:{1}", supplier.SupplierID, supplier.CompanyName);
        Console.WriteLine("Prod ID:{0}
Name:{1}", product.ProductID, product.ProductName);
    }
}

```

On the above example, I am creating a new category and supplier and assigning it to the entityreference on the product entity for Category and Supplier. Product also has a 1 to 1 mapping with ProductAddditionalInfo which I am creating an instance for inside the objectintializer. To add the product, I have to call AddToProducts passing in the newly created product followed by SaveChanges. SaveChanges is the method that actually triggers the insert process in the correct sequence defined by our model. So in this case, EF framework first issues a dynamic insert statement to insert Category and Supplier entity and then followed by Product entity.

Inserting the product entity uses our insert stored procedure which gets passed the categoryid and supplierid of the newly created supplier and Category entity. Since productAdditionalInfo requires a valid product to exist, entity framework is smart enough to execute insert for ProductAdditionalInfo to be the last. For executing insert for ProductAdditionalInfo table, entity framework uses Insert stored procedure to carry out the insert operation. Following profiler capture shows the execution of the code in the order performed.

EventClass	TextData
RPC:Completed	exec sp_reset_connection
RPC:Completed suppliers	exec sp_executesql N'insert [dbo].[Suppliers]([Co...]
RPC:Completed category	exec [dbo].[InsertCategory] @catname='Makeups',@d...
RPC:Completed Product	exec [dbo].[InsertProd] @prodname='Hair Formula',...
RPC:Completed ProdAddInfo	exec [dbo].[InsertProdAddInfo] @prodid=94,@desc='...

To update the product entity, I am using the following code.

```
//update product info.
    var db2 = new StoredProcedureEntities();
    var updateproduct = db2.Products.First(p =>
p.ProductName == "Hair Formula");
        //change the unit price price.
        updateproduct.UnitPrice = 15.0M;
        db2.SaveChanges();
        //change supplier and category information and
product additional info.

        updateproduct.Supplier = db2.Suppliers.First(sp =>
sp.SupplierID == 1);

        updateproduct.Category =
db2.Categories.FirstOrDefault(c => c.CategoryID == 1);
        updateproduct.ProductAdditionalInfoReference.Load();

        //load product additonal info.

updateproduct.ProductAdditionalInfo.ProductDescription = "Hair
formula";

db2.SaveChanges();
```

On the above example, I am updating product entity two times. For the first update only unit price is changed. Also notice that for the first update we have not explicitly loaded Category or supplier navigation relationship even though our stored procedure's categoryid and supplierid is mapped to properties on our navigation relationship. Does that mean our update will crash? The answer is no. Entity framework under the covers materializes the entity references using the metadata and therefore before the update stored procedure call is issued, entity references are assigned valid references. For the second update, I am replacing the existing category and supplier entity reference to a new category and supplier. To change description property on ProductAdditioanalInfo entity, I have to load the entity first and then change the description property. When SaveChanges is called several updates are triggered. First UpdateProcedure on product is called followed by update procedure defined for ProductAdditionalInfo. Profilers capture show below illustrates the order of updates issued.

	TExtdata
1	exec [dbo].[UpdateProduct] @prodname='Hair Formula',@unitprice=15.0,@supplierid=47,@categoryid=41,@prodid=96
2	exec [dbo].[UpdateProduct] @prodname='Hair Formula',@unitprice=15.0,@supplierid=1,@categoryid=1,@prodid=96
3	exec [dbo].[UpdateProdAddInfo] @prodid=96,@desc='Hair formula',@unitsinstock=20

To delete product and its additional info, all that we have to do is pass the product entity to DeleteObject method exposed on the ObjectContext. Code below shows the example.

```
//delete updated product and additionalinfo table
    db2.DeleteObject(updateproduct);
    db2.SaveChanges();
```

Notice on the above code we did not call delete on ProdAdditionalInfo. Since ProductAdditionalInfo has a relationship with product and cannot exists without a product entity, calling delete on the product entity triggers an automatic delete to be called on ProdAdditional entity as well. If we look at the profiler trace, we will see that entity framework first calls delete procedure for productadditionalinfo followed by product delete procedure to delete product from product table. The order in which entity framework performs this operation is important otherwise we can get a foreign key violation error. Profile capture shows the order in which delete procedure is called.

EventClass	TextData
RPC:Completed	exec sp_reset_connection
RPC:Completed	exec [dbo].[DeleteProdAddInfo] @prodid=98
RPC:Completed	exec [dbo].[DeleteProduct] @prodid=98,@supplierid=...

12.1.3 Deleting and Inserting Many to Many Relationship using Stored Procedures

Problem: You have two tables MusicalShows and Sponsor that are joined using a link table to portray many to many relations. A musical Show can have many sponsors and a sponsor can participate in many musical shows.

You want to map both entities defined on your model to use stored procedures for crud operations. You want to know how many stored procedures are required for this entire process and how to map these stored procedures to entity data model.

Solution: When you import tables that are defined as using a link table, entity framework will display two tables having many to many relationships.

Although there will be 2 entities on the designer, but crud operation would require 8 stored procedures meaning 3 stored procedures for MusicalShow, Sponsor and two stored procedure for link table called

MusicalShow_Sponsor. The two procedures for link table would be Insert and Delete because you cannot update a link table; you can either insert a many to many relationship or delete a many to many relationship. Mapping stored procedures for Musical Show and Sponsor can be achieved using the designer. However the link table mapping which immerges as an association set in entity framework cannot currently be mapped using the designer. This would require editing the msdl to map the association sets to stored procedure for the link table.

Discussion: On the database, the only way to define many to many relationships between two database objects is to use a link table which contains primary key from both tables. To map MusicalShows, Sponsors and the link table to stored procedure we have to create 8 stored procedures. Code below shows our eight stored procedures.

```
/* spnonsor stored procedure */
CREATE proc [dbo].[InsertSponsor]
(@name varchar(50),@phone varchar(50))
as
begin
insert into Sponsors(name,phone) values (@name,@phone)
select SCOPE_IDENTITY() as sponsorid
end

CREATE proc [dbo].[UpdateSponsor]
(@name varchar(50),@phone varchar(50),@sponsorid int)
as
begin
update sponsors
set name = @name,phone = @phone
where sponsorid = @sponsorid
end
```

```

CREATE proc [dbo].[DeleteSponsor]
(@sponsorid int)
as
begin
delete sponsors where sponsorid = @sponsorid
end

/* Show stored procedure */
create proc [dbo].[InsertShow]
(@name varchar(50),@cost decimal)
as
begin
insert into MusicalShow(ShowName,cost) values (@name,@cost)
select SCOPE_IDENTITY() as showid
end

create proc [dbo].[UpdateShow]
(@name varchar(50),@cost decimal,@showid int)
as
begin
update MusicalShow
set ShowName = @name,cost = @cost
where showid = @showid
end

create proc [dbo].[DeleteShow]
(@showid int)
as
begin
delete MusicalShow where showid = @showid
end

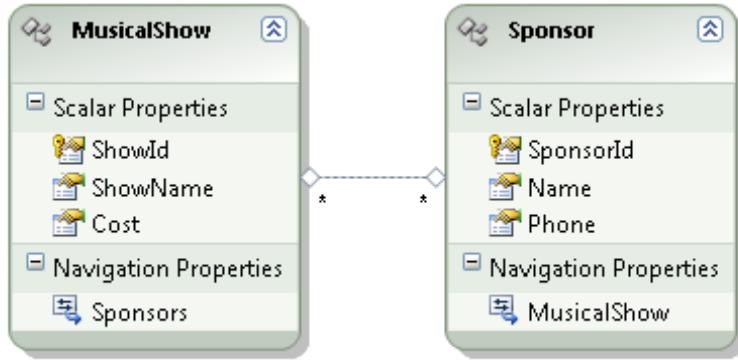
/* show_sponsor link table. */
create proc [dbo].[insertshowsponsor]
(@showid int,@sponsorid int)
as
begin
insert into show_sponsor values (@showid,@sponsorid)
end

create proc [dbo].[deleteshowsponsor]
(@showid int,@sponsorid int)
as
begin
delete show_sponsor where showid =@showid and sponsorid = @sponsorid
end

```

On the above stored procedures, we have 3 stored procedures that take cares of inserting, updating and deleting a sponsor. Similarly for inserting, updating and deleting sponsors we have defined three store procedures. To insert into our link table, show_sponsor, stored procedure takes sponsorid and showid to create a relationship between a sponsor and a show. Similarly to delete relationship between a sponsor and a show we have delete stored procedure for the link table that takes showid and sponsorid to delete the relationship. After importing the

stored procedure and selecting 3 tables from the entity model wizard, we get the following diagram that represents relationship between sponsor and musicalshow. Notice that the relationship between MusicalShow and Sponsor is denoted by many to many on both sides.



The next step is to map the Musicalshow entity using the stored procedures we have imported. StoredProcedured mapping window shows how we have configured the crud mapping for Musical Show.

Mapping Details - MusicalShow

Parameter / Column	Operator	Property
Functions		
Insert Using InsertShow		
Parameters		
@name : varchar	←	ShowName : String
@cost : decimal	←	Cost : Decimal
Result Column Bindings		
showid	→	ShowId : Int32
<Add Result Binding>		
Update Using UpdateShow		
Parameters		
@name : varchar	←	ShowName : String
@cost : decimal	←	Cost : Decimal
@showid : int	←	ShowId : Int32
Result Column Bindings		
<Add Result Binding>		
Delete Using DeleteShow		
Parameters		
@showid : int	←	ShowId : Int32

The mapping for Musical Show is very similar to the mappings we have done in other examples so I will not go into the details of how to configure it. To map

Sponsor entity to the stored procedure we have imported we can once again use the Stored Procedure mapping window. The final result for Sponsor stored procedure mapping is shown below.

Parameter / Column	Operator	Property
Functions		
Insert Using InsertSponsor		
Parameters		
@name : varchar	←	Name : String
@phone : varchar	←	Phone : String
Result Column Bindings		
sponsorid	→	SponsorId : Int32
<Add Result Binding>		
Update Using UpdateSponsor		
Parameters		
@name : varchar	←	Name : String
@phone : varchar	←	Phone : String
@sponsorid : int	←	SponsorId : Int32
Result Column Bindings		
<Add Result Binding>		
Delete Using DeleteSponsor		
Parameters		
@sponsorid : int	←	SponsorId : Int32

As discussed earlier, we have to map two stored procedures we had created earlier for the link table to the associationset defined by the entityframework. Current version of entity framework does not support mapping associationset to stored procedure using the designer. Therefore we need to go in msdl section of edmx file and modify the assocaitonset that maps to Show_Sponsor entityset that happens to be our table in the database. Code below shows the mapping required for the linked table.

```

<AssociationSetMapping Name="Show_Sponsor"
TypeName="MappingStoredProcModel.Show_Sponsor"
StoreEntitySet="Show_Sponsor">
    <EndProperty Name="Sponsors">
        <ScalarProperty Name="SponsorId"
ColumnName="SponsorId" /></EndProperty>
    <EndProperty Name="MusicalShow">

```

```

        <ScalarProperty Name="ShowId" ColumnName="ShowId" /></EndProperty>
            <ModificationFunctionMapping>
                <InsertFunction
                    FunctionName="MappingStoredProcModel.Store.insertshowsponsor">
                    <EndProperty Name="Sponsors">
                        <ScalarProperty Name="SponsorId" ParameterName="sponsorid"/>
                    </EndProperty>
                    <EndProperty Name="MusicalShow">
                        <ScalarProperty Name="ShowId" ParameterName="showid"/>
                    </EndProperty>
                </InsertFunction>
                <DeleteFunction
                    FunctionName="MappingStoredProcModel.Store.deleteshowsponsor">
                    <EndProperty Name="Sponsors">
                        <ScalarProperty Name="SponsorId" ParameterName="sponsorid"/>
                    </EndProperty>
                    <EndProperty Name="MusicalShow">
                        <ScalarProperty Name="ShowId" ParameterName="showid"/>
                    </EndProperty>
                </DeleteFunction>
            </ModificationFunctionMapping>
        </AssociationSetMapping>
    
```

The part that we added to the AssociationSetMapping is ModificationFunctionMapping. ModificationFunctionMapping consists of two functions; Insert and Delete. There is no update because a many to many relationship can either be inserted or delete but cannot be updated. The insert function maps to our insertstored procedure for link table and the parameters for function are read from the scalar properties on both sides of the associationset meaning Sponsor and MusicalShow. Similarly Delete function calls our delete function defined on the store with parameter mappings coming from scalar properties on both sides of the associationset. After configuring the mapping we can use our sponsor and musical show entity to cause our functions to be invoked. Code below shows how to insert many to many relationships between sponsor and musical show entity.

```

var db = new StoredProcedureEntities();
var alex = new Sponsor{Name="Alex Jones"};

```

```

var mark = new Sponsor{Name="Mark Walt"};
var musicalshow = new
MusicalShow
{
    ShowName = "Rock Concert",
    Sponsors = {alex,mark}
};
var rockandroll = new MusicalShow
{
    ShowName = "Rock And Roll",
    Sponsors = { alex }
};

//causes insert to sponsor table and the
show_sponsor link table.
db.AddToMusicalShow(musicalshow);
db.AddToMusicalShow(rockandroll);
db.SaveChanges();

```

On the above code, I am creating two sponsors followed by a new musical show instance. Using the collection initialize syntax, I am adding our two sponsors to the musical show. Next I create another rockandroll musical show and only add one of the sponsor alex to be part of the show. I then add the two sponsors to the object context followed by SaveChanges method that triggers insert process. Entity framework based on the model runs the stored procedures in the correct order as not cause any foreign key violations on the database. Profiler output below shows the order in which inserts happen.

EventClass	TextData
RPC:Completed	exec sp_reset_connection
Audit Login	-- network protocol: LPC set quote...
RPC:Completed	exec [dbo].[InsertShow] @name='Rock...
RPC:Completed	exec [dbo].[InsertShow] @name='Rock...
RPC:Completed	exec [dbo].[InsertSponsor] @name='M...
RPC:Completed	exec [dbo].[InsertSponsor] @name='A...
RPC:Completed	exec [dbo].[insertshowsponsor] @sho...
RPC:Completed	exec [dbo].[insertshowsponsor] @sho...
RPC:Completed	exec [dbo].[insertshowsponsor] @sho...

The profile capture shown above shows that we inserted two shows to MusicalShow table followed by calling InsertSponsor stored procedure twice to insert our sponsors. Then we add the three relationship between sponsor and the show by calling InsertShowSponsor three times.

To delete the relationship we can call clear to clear all sponsors for a given show or call remove to remove a specific sponsor from a show.

```
//removes alex from the sponsor_show
    rockandroll.Sponsors.Remove(alex);

    //removes all sponsors from sponsor_show for
musicalshow
    musicalshow.Sponsors.Clear();
    db.SaveChanges();
```

On the above code, I am removing alex from rockandroll show and clearing all the sponsors for musicalshow. Removing and clearing of Sponsors would not work unless we load all the sponsors for a given show ahead of time by either calling Load using Include to load sponsors with their shows. Profiler capture shows the execution path taken by entity framework to delete many to many relationships from our link table.

EventClass	TextData	SPID
RPC:Completed	exec sp_reset_connection	55
Audit Login	-- network protocol: LPC set quote...	55
RPC:Completed	exec [dbo].[deleteshowsponsor] @sho...	55
RPC:Completed	exec [dbo].[deleteshowsponsor] @sho...	55
RPC:Completed	exec [dbo].[deleteshowsponsor] @sho...	55

On the above profiler capture we can see that deleteshowsponsor was called three times; first for deleting alex from rockandroll show and then removing the two sponsors we assigned earlier to musicalshow.

12.1.4 Mapping Complex Type using Stored Procedure

Problem: Customer entity defined on the entity data model uses Address complex type to group Address, City, State and zip under a single class. You want to how to map the customer entity including the its complex type to stored procedures defined on the database.

Solution: In entity framework version1, if you define a complex type on an entity, there is no designer support to map the entity to stored procedures defined on the database. This process has to be done manually by editing the ssdl, msdl and csdl layers of the model. Even if you managed to edit the edmx file, you will end up getting validation errors for which there is no workaround. If you have successfully mapped your entity and the complex type to stored procedures, you will get following validation errors.

1. Complex Type not supported by the designer.
2. Complex Type property is not mapped.

Both validation errors mentioned above do not prevent you from building your solution and can be ignored. However if seeing those validation errors annoy you, you can either get rid of complex type or do use edmx file and generate your model using command line utility such edmgen.exe or edmgen2.exe.

To map the complex type and the entity to stored procedure, following steps must be followed.

1. Import stored procedures into ssdl layer.
2. Define Complex Type inside of csdl layer.
3. Remove properties on the Customer entity that will be grouped inside the complex address.
4. Define Address property on the customer that mapped to Complex Type address.
5. On the msdl layer, map the complex type Address property to columns defined on the customer table.
6. Map the Address Complex type property defined on the Customer entity to parameters defined on the stored procedure. This step needs to be performed for both Insert and Update stored procedure.

Discussion: To demonstrate complex type mapping, we will use customer table. Customer table has Address,City and Zip that we can group them under a complex type address. To map the customer and the complex we will need 3 stored procedures for inserting, updating and deleting customer entity. Following stored procedures takes care of crud operations on Customer entity.

```

/* insert stored procedure */
ALTER proc [dbo].[InsertCustomer]
(
@CustomerID nchar(5) ,
@ContactName nvarchar(30) ,
@Address nvarchar(60) ,
@City nvarchar(15) ,
@Zip nvarchar(10)
)
as
begin
insert into customers (CustomerID,ContactName,[Address],City,Zip)
values (@CustomerID,@ContactName,@Address,@City,@Zip)
end

/* update stored procedure */
ALTER proc [dbo].[UpdateCustomer]
(
@CustomerID nchar(5) ,
@ContactName nvarchar(30) ,
@Address nvarchar(60) ,
@City nvarchar(15) ,
@Zip nvarchar(10)
)
as
begin
update customers
set ContactName = @ContactName,
[Address] = @Address,
City = @City,
Zip = @Zip
where customerid = @CustomerID
end

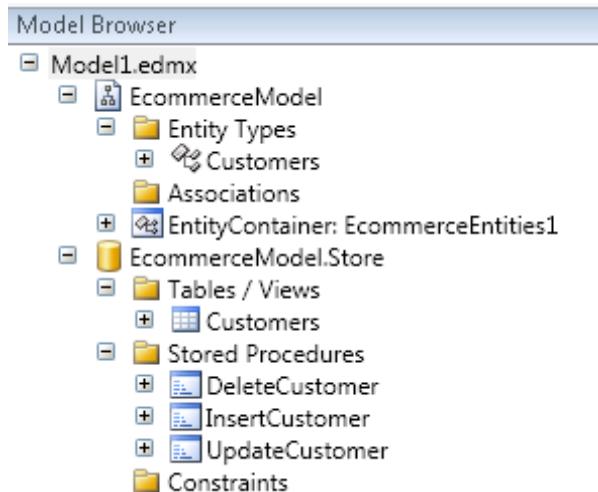
/* delete stored procedure. */
ALTER proc [dbo].[DeleteCustomer]
(@customerid nchar(5))
as
begin
delete customers where customerid = @customerid
end

```

Stored procedures defined above are fairly simple from the fact that they take in parameters and insert, update and delete customer record. When we define a complex type on Customer entity, we cannot use the designer and have to

manually edit the edmx file. Once you have edited the edmx file manually you lose the ability to open the edmx file in the designer. Because of this constraint you cannot even leverage the designer abilities to map crud operation to stored procedures. One of the ways I have gotten around to this is by not declaring the complex type initially and use the designer to import stored procedures and my entity. After importing the stored procedure, I went ahead and mapped customer entity to stored procedure. What this allowed me to do is get me 80 percent through the mapping and 20 percent I went ahead and manually edit the mapping files. Following steps I took to map the complex type to stored procedure.

1. Import the customer crud stored procedure and customer table into entity data model using the designer. Model browser window confirms our import from the database.



2. Map the stored procedure to customer entity as shown below

Mapping Details - Customers

Parameter / Column	Operator	Property
Functions		
Insert Using InsertCustomer		
Parameters		
@ CustomerID : nchar	←	CustomerID : String
@ ContactName : nvarchar	←	ContactName : String
@ Address : nvarchar	←	Address : String
@ City : nvarchar	←	City : String
@ Zip : nvarchar	←	Zip : String
Result Column Bindings		
<Add Result Binding>		
Update Using UpdateCustomer		
Parameters		
@ CustomerID : nchar	←	CustomerID : String
@ ContactName : nvarchar	←	ContactName : String
@ Address : nvarchar	←	Address : String
@ City : nvarchar	←	City : String
@ Zip : nvarchar	←	Zip : String
Result Column Bindings		
<Add Result Binding>		
Delete Using DeleteCustomer		
Parameters		
@ customerid : nchar	←	CustomerID : String

Using the designer till this stage would import the stored procedures on the ssdl, map the customer entity to customer table, map stored procedure parameters to properties defined on the customer entity. The next step is to define Address complex type and remove columns defined on the Customer entity that will be grouped inside of Address complex type as shown below.

```

<EntityType Name="Customers">
  <Key>
    <PropertyRef Name="CustomerID" />
  </Key>
  <Property Name="CustomerID" Type="String" Nullable="false"
  MaxLength="5" Unicode="true" FixedLength="true" />
  <Property Name="ContactName" Type="String" MaxLength="30"
  Unicode="true" FixedLength="false" />
  <Property Name="Address" Type="Self.CAddress"
  Nullable="false" />
</EntityType>
<ComplexType Name="CAddress">
  <Property Name="StreetAddress" Type="String" />

```

```

<Property Name="City" Type="String" />
<Property Name="Zip" Type="String" />
</ComplexType>

```

3. Update the customer entity mapping appropriately to include the mapping of Address complex type to columns on Customer table as shown below.

```

<MappingFragment StoreEntitySet="Customers">
    <ScalarProperty Name="CustomerID"
ColumnName="CustomerID" />
    <ScalarProperty Name="ContactName"
ColumnName="ContactName" />
    <ComplexProperty Name="Address"
TypeName="SpMappComplexType.CAddress">
        <ScalarProperty Name="StreetAddress"
ColumnName="Address" />
        <ScalarProperty Name="City" ColumnName="City" />
        <ScalarProperty Name="Zip" ColumnName="Zip" />
    </ComplexProperty>

```

4. Update Insert stored procedure parameter mapping to read address information from Address complex type as shown below.

```

<InsertFunction
FunctionName="SpMappComplexType.Store.InsertCustomer">
    <ComplexProperty Name="Address"
TypeName="SpMappComplexType.CAddress">
        <ScalarProperty Name="StreetAddress"
ParameterName="Address" />
        <ScalarProperty Name="City" ParameterName="City"
/>
        <ScalarProperty Name="Zip" ParameterName="Zip" />
    </ComplexProperty>
    <ScalarProperty Name="ContactName"
ParameterName="ContactName" />
    <ScalarProperty Name="CustomerID"
ParameterName="CustomerID" />
</InsertFunction>

```

5. Similarly update, UpdateCustomer stored procedure to read address information from Address complex type as shown below.

```

<UpdateFunction
FunctionName="SpMappComplexType.Store.UpdateCustomer">

```

```

        <ComplexProperty Name="Address"
TypeName="SpMappComplexType.CAddress">
            <ScalarProperty Name="StreetAddress"
ParameterName="Address" Version="Current" />
            <ScalarProperty Name="City" ParameterName="City"
Version="Current" />
            <ScalarProperty Name="Zip" ParameterName="Zip"
Version="Current" />
        </ComplexProperty>
        <ScalarProperty Name="ContactName"
ParameterName="ContactName" Version="Current" />
        <ScalarProperty Name="CustomerID"
ParameterName="CustomerID" Version="Current" />
    </UpdateFunction>

```

After completing the mapping when you build your project, you will still get validation errors such complex type property not mapped. Ignore these errors as they are bug in the designer and features not completely implemented in the designer. To insert customer entity with our complex type using stored procedure, we can create an instance of Customer entity, initialize Address complex type will correct values and add the customer entity to the objectcontext and call savechanges. Code below shows how to insert customer and complex type using stored procedures defined on the store model.

```

var db = new SpMappComplex();
    var customer = new Customers
    {
        CustomerID = "ALFK1",
        ContactName = "Zeeshan Hirani",
        Address = new CAddress { City = "Dallas",
StreetAddress = "123 Address", Zip = "76111" }
    };
    db.AddToCustomers(customer);
    db.SaveChanges();
    db.DeleteObject(customer);
    db.SaveChanges();

```

12.1.5 Mapping Crud Operations To Table Per Hierarchy

Problem: You have created an entity data model using table per hierarchy. Inheritance model consists of a base class Person with two derived classes Student and Employee. Student contains another derived class called Special

Student. So far you had successfully mapped inheritance hierarchy structure to People table in the database. You have created insert, update and delete stored procedure for people table in the database. You want to know how to map these procedures to inheritance hierarchy defined on Entity data Model.

Solution: There are couple of different approaches to map stored procedures to inheritance model defined in entity data model. First approach is to create stored procedure for every entity that is part of the inheritance tree and map them to their appropriate stored procedure. This is the easier approach but over the long run, you would be maintaining ample of stored procedures that basically insert, update and delete to same people table. The good point about adopting this approach is, you will get full designer support and this option is fully supported in version 1 of entity framework. Second approach to mapping is create one insert, update and delete procedure for the people table and set default values of null for parameters that are specific to each entity. When you try to map these procedures to every entity in the inheritance tree and not specify values for parameter not specific to an entity, the designer would complain that all parameters to procedure require a value. I think this is a limitation of the designer which should check to see that some parameters on the stored procedures have default values assigned and should not be required. Furthermore, designer only lets you map parameters to properties on the entity; there is no way to hard code specific default value. To get around the problem, you have to fake out stored procedures for each entity inside ssdl model and using CommandText property of a function. Inside of CommandText property, you can execute the same procedure passing in default or null values for parameters not specific to certain derived class.

Discussion: We have defined people table in the database that contains student, employee and special student. Each type of person is differentiated by Category column. Following screen shot shows how the table looks like.

	PersonId	Name	HireDate	EnrollmentDate	SpecialNeeds	Category
▶	1	James	NULL	2008-09-28	Hearing	3
	2	Zeeshan Hirani	NULL	2008-09-28	NULL	1
	3	Alex	2008-09-28	NULL	NULL	2
*	NULL	NULL	NULL	NULL	NULL	NULL

Name column is required for all 3 entities where HireDate is only required for Employee type. Enrollment date is only required for Student and Special needs is required for student with special need. To identify different person, Category column is assigned different integer value to different if a row is a student, employee or special student. To insert into people table different type of persons, we can create one generic insert, update and delete stored procedure. Code below shows how the stored procedure.

```

/* insert stored procedure */
ALTER proc [SPIH].[InsertPeople]
(
@name varchar(50),
@hiredate date = null,
@enrollmentdate date = null,
@specialneeds varchar(50) = null,
@category int
)
as
begin
insert into
SPIH.people(name,hiredate,EnrollmentDate,specialneeds,category)
values
(@name,@hiredate,@enrollmentdate,@specialneeds,@category)

select SCOPE_IDENTITY() as personid
end

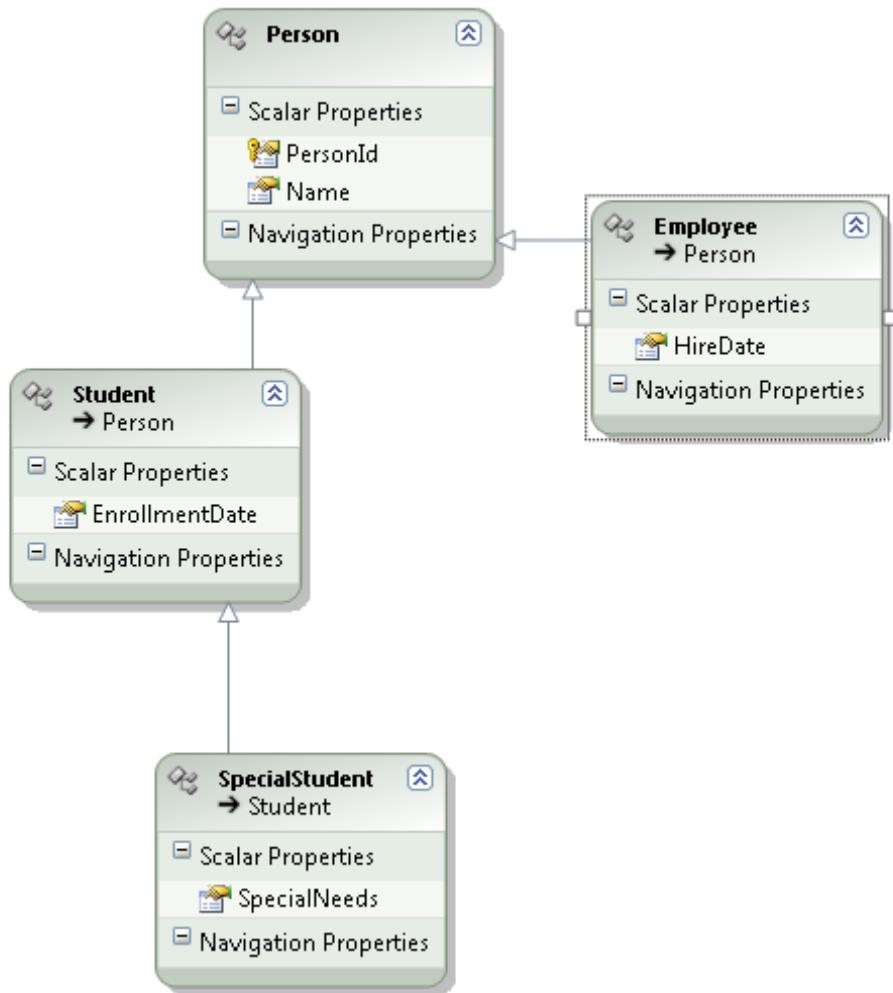
/* update stored procedure */
ALTER proc [SPIH].[UpdatePeople]
(
@name varchar(50),
@hiredate date = null,
@enrollmentdate date = null,
@specialneeds varchar(50) = null,
@personid int
)
as
begin
update SPIH.People
set name = @name,
hiredate = @hiredate,
EnrollmentDate = @enrollmentdate,
specialneeds = @specialneeds
from SPIH.People
where personid = @personid
end

/*delete procedure */
ALTER proc [SPIH].[DeletePeople]
(@personid int)
as
begin

```

```
delete SPIH.People where personid = @personid  
end
```

On the above stored procedure, I have set hiredate and enrollmentdate and specialneeds to have default values of null. We are setting default values for these parameters because all entities derived from people will use the same procedure for mapping their crud and will not have values defined for every parameter on the stored procedure. By setting default values ensure that stored procedure would not crash if a derived entity does not provide value for a parameter that is not specific for that entity. Both insert and update stored procedure takes Category parameter that is used to define what type of entity this record represents in the table. To map the stored procedures to our entity data model, we have to import the people table, the 3 stored procedures we have created and then model our people table in entity data model as table per hierarchy. Screen shot below shows how entity data model looks like after importing the table mapping the table to table per hierarchy structure.



After mapping the People table to inheritance hierarchy, we need to map stored procedure to each derived entity in the model. If we try to map the store procedures imported from the database to derived entity, the designer will complain that we are missing mapping for certain parameters of the procedure and mapping of all parameters for the stored procedure is required. From our perspective this is completely a valid situation because not all parameters are valid for a given derived entity. To get around this problem, we have to edit the ssdl model and fake out insert and update stored procedure for every derived entity defined on the model. We do not need a separate delete for every derived entity because our delete stored procedure takes in personid that is common to all derived entities. SSDL below shows our fake stored procedure that calls our original stored procedure with default values.

```

<Function Name="InsertStudent" IsComposable="false">
    <CommandText>
        exec [SPIH].InsertPeople @name =
@name,@enrollmentdate = @enrollmentdate,@category = 1
    </CommandText>
    <Parameter Name="name" Type="varchar" Mode="In"
/>
    <Parameter Name="enrollmentdate" Type="date"
Mode="In" />
    </Function>
    <Function Name="UpdateStudent" IsComposable="false">
        <CommandText>
            exec SPIH.[UpdatePeople]@name =
@name,@enrollmentdate = @enrollmentdate,@personid = @personid
        </CommandText>
        <Parameter Name="name" Type="varchar" Mode="In"
/>
        <Parameter Name="enrollmentdate" Type="date"
Mode="In" />
        <Parameter Name="personid" Type="int" Mode="In"
/>
        </Function>
        <Function Name="InsertSpecialStudent"
IsComposable="false">
            <CommandText>
                exec SPIH.InsertPeople
                    @name = @name,@enrollmentdate =
@enrollmentdate,@specialneeds = @specialneeds,@category = 3
            </CommandText>
            <Parameter Name="name" Type="varchar" Mode="In"
/>
            <Parameter Name="enrollmentdate" Type="date"
Mode="In" />
            <Parameter Name="specialneeds" Type="varchar"
Mode="In" />
            </Function>
            <Function Name="UpdateSpecialStudent"
IsComposable="false">
                <CommandText>
                    exec SPIH.UpdatePeople
                        @name = @name,@enrollmentdate =
@enrollmentdate,@specialneeds = @specialneeds,@category =
3,@personid = @personid
                </CommandText>
                <Parameter Name="name" Type="varchar" Mode="In"
/>

```

```

        <Parameter Name="enrollmentdate" Type="date"
Mode="In" />
        <Parameter Name="specialneeds" Type="varchar"
Mode="In" />
        <Parameter Name="personid" Type="int" Mode="In"
/>
    </Function>
<Function Name="InsertEmployee" IsComposable="false">
    <CommandText>
        exec SPIH.InsertPeople
        @name = @name,@hiredate =
@hiredate,@category = 2
    </CommandText>
    <Parameter Name="name" Type="varchar" Mode="In"
/>
    <Parameter Name="hiredate" Type="date" Mode="In"
/>
    </Function>
<Function Name="UpdateEmployee" IsComposable="false">
    <CommandText>
        exec SPIH.UpdatePeople
        @name = @name,@hiredate =
@hiredate,@category = 2,@personid = @personid
    </CommandText>
    <Parameter Name="name" Type="varchar" Mode="In"
/>
    <Parameter Name="hiredate" Type="date" Mode="In"
/>
    <Parameter Name="personid" Type="int" Mode="In"
/>
</Function>

```

For InsertStudent, I am calling InsertPeople actual stored procedure defined on the database with parameters only specific to student entity such as name and enrollment date. Since we are inserting student entity, I am hardcoding the category to be passed to the stored procedure as 1. Similarly update stored procedure only requires 3 parameters name, enrollmentdate and personid to update student entity in the database. For other entities, I am following a similar approach where each function has only parameters that are specific to given entity. Inside the commandText, I call the same Insert and Update procedure passing values only for parameters needed and hardcoding the Category valued based on the entity I am inserting. It is important to

clarify that these functions are really not defined on the database. They are faked as functions to EF by using CommandText property.

After creating these procedures, we can go back to the model in the designer and map these procedures to every derived entity. There will be no mapping for the base Person entity because it is marked as abstract and entity framework does not allow stored procedure mapping to abstract entity. Below is the screen shot for mapping Student entity.

The screenshot shows the 'Mapping Details - Student' window in the EntityDataSource Configuration tool. The left pane lists three functions: 'Insert Using InsertStudent', 'Update Using UpdateStudent', and 'Delete Using DeletePeople'. The 'Insert Using InsertStudent' function is expanded, showing its parameters (@name, @enrollmentdate) and result column binding (personid). The 'Parameters' section maps @name to 'Name : String' and @enrollmentdate to 'EnrollmentDate : DateTime'. The 'Result Column Bindings' section maps personid to 'PersonId : Int32'. The 'Update Using UpdateStudent' function is also expanded, showing similar mappings for its parameters (@name, @enrollmentdate, @personid) and result column binding (@personid). The 'Delete Using DeletePeople' function is shown with its single parameter (@personid).

On the above screen shot, I am mapping InsertFunction of student to InsertStudent stored procedure. The personid returned from the stored procedure is mapped to PersonId defined on the base class Person. Similarly UpdateFunction and DeleteFunction map to UpdateStudent and DeleteStudent stored procedure defined on the store model. The mapping of SpecialStudent is as follows

Parameter / Column	Operator	Property
Functions		
Insert Using InsertSpecialStudent		
Parameters		
@name : varchar	←	Name : String
@enrollmentdate : date	←	EnrollmentDate : DateTime
@specialneeds : varchar	←	SpecialNeeds : String
Result Column Bindings		
personid	→	PersonId : Int32
<Add Result Binding>		
Update Using UpdateSpecialStudent		
Parameters		
@name : varchar	←	Name : String
@enrollmentdate : date	←	EnrollmentDate : DateTime
@specialneeds : varchar	←	SpecialNeeds : String
@personid : int	←	PersonId : Int32
Result Column Bindings		
<Add Result Binding>		
Delete Using DeletePeople		
Parameters		
@personid : int	←	PersonId : Int32

For SpecialStudent, name parameter is mapped to Name property defined on the base Person class. Enrollment parameter is mapped to Enrollment property defined on the direct base Student entity and special needs parameter maps to SpecialNeeds property defined on SpecialStudent entity. The same process is applied to Update and Delete stored procedure mapping.

Stored procedure binding for Employee entity is shown below.

Mapping Details - Employee

Parameter / Column	Operator	Property
Functions		
Insert Using InsertEmployee		
Parameters		
@name : varchar	←	Name : String
@hiredate : date	←	HireDate : DateTime
Result Column Bindings		
personid	→	PersonId : Int32
<Add Result Binding>		
Update Using UpdateEmployee		
Parameters		
@name : varchar	←	Name : String
@hiredate : date	←	HireDate : DateTime
@personid : int	←	PersonId : Int32
Result Column Bindings		
<Add Result Binding>		
Delete Using DeletePeople		
Parameters		
@personid : int	←	PersonId : Int32

If we map one of the derived entities to stored procedure mapping, entity framework requires mapping for all the derived entities defined on the model that inherits from the same base entity. This means that you cannot map Student and Special Student entity and use defaults for Employee entity because all entities derive from Person entity. Doing so will raise validation error by the designer saying that all related entities must be mapped using stored procedure.

Now that we have completed our mapping we can create instance of derived entities and call save changes to save entities to the database. Saving entity would trigger Insert procedure specific to each entity to get called. Code below shows how to insert derived entities to the database.

```
var db = new SPInhMapping();
    var spstudent = new SpecialStudent
    {
        EnrollmentDate = DateTime.Now,
        Name = "James",
        SpecialNeeds = "Hearing"
    };
    var student = new Student
```

```

{
    EnrollmentDate = DateTime.Now,
    Name = "Zeeshan Hirani"
} ;
var employee = new Employee
{
    HireDate = DateTime.Now,
    Name = "Alex"
} ;
db.AddToPersons(spstudent);
db.AddToPersons(student);
db.AddToPersons(employee);
db.SaveChanges();

```

The above code creates 3 different derived entities and adds them to the Persons method followed by SaveChanges. This triggers the insert procedure to be called. If you open up the profiler, you will see that it is the same procedure InsertPeople getting called just with different parameters.

12.1.6 Managing concurrency using stored procedures

Problem: You have created an author entity that maps to author table in the database. Authors table contains a timestamp column that gets updated when a row is changed. You have created stored procedures for inserts, updates and delete to author's table. You want to map these procedures to author entity defined on the EDM but want to ensure that these stored procedures observe concurrency and do not update the row if the timestamp value passed in to the stored procedure does not match with the current time stamp value defined for the row being updated.

Solution: To use stored procedures with concurrency option, you need to map the time stamp parameter of the stored procedure to TimeStamp property on author entity using original value option. In addition on the Update stored procedure, the where clause needs to include the original timestamp value passed in the parameter along with the primary key column. The number of rows affected needs to be communicated back to entity framework using

output parameter. When entity framework sees that a row affected is zero, it throws concurrency violation exception.

Discussion: Since we will be mapping stored procedure for inserts, update and delete, it would be a good place to start exploring how we have written the stored procedures to leverage concurrency option. The first stored procedure, we will explore is InsertAuthor responsible for inserting author record.

```
alter proc dbo.InsertAuthor
(@name varchar(50),@bkpublished int)
as
begin

insert Authors (name,BooksPublished)
values (@name,@bkpublished)

select a.AuthorId,a.[TimeStamp]
from Authors a where a.AuthorId = SCOPE_IDENTITY()
end
```

The above InsertAuthor stored procedure, takes name and bookspublished parameter and inserts a record into the author's table. Once the row is inserted we need to inform entity framework of two values. First the authorid of the record just inserted. Second the timestamp value of the record to ensure optimistic concurrency when an update happens on the same record. To return these two values, I am doing selecting authored and timestamp column from authors with filter of primary key obtained from scope_identity which represents the id of the last record inserted. When we get to the mapping of the stored procedure, I will discuss how to map these returned values to properties on author entity.

To update an author record, I have created UpdateAuthor stored procedure which looks like this

```
alter proc dbo.UpdateAuthor
(
@email varchar(50),
@bkpublished int,
@timestamp timestamp,
@authorid int,
@rowsaffected int output
)
as
begin
```

```

update Authors
set name = @name,
BooksPublished = @bkpublished
from authors
where authorid = @authorid and [TimeStamp] = @timestamp
set @rowsaffected = @@ROWCOUNT
select [TimeStamp]
from Authors where AuthorId = @authorid

end

```

UpdateAuthor procedure takes name and bkpublished parameter and updates the author records based on the primary key valued . To ensure concurrency of the record being updated, the stored procedure is also passed the timestamp value of when the record was last edited. We apply that time stamp parameter as an additional filter along with the primary key. If there are no records affected after the update operation, this means that record was updated in between the last time it was read and the current update. To notify entity framework of the records updated we are taking an additional output parameter called rowsaffected which entity framework will question to find out if the update caused any rows to be affected. If the count value returned from this parameter is zero, entity framework would throw OptimisticConcurrencyException indicating that row has changed since the last retrieval. After setting rowsaffected parameter, I am returning the new timestamp value after the update operation has completed. Failing to provide new TimeStamp value back to author entity would cause subsequent update operations on author entity to fail because the new value for TimeStamp would not match to what is passed in the stored procedure.

To delete an author record, I have created DeleteAuthor stored procedure which looks like this.

```

create proc dbo.DeleteAuthor
(@authorid int,@timestamp timestamp)
as
begin
delete Authors where AuthorId = @authorid and [TimeStamp] = @timestamp
end

```

On the DeleteAuthor stored procedure, I am deleting the record based on primary key and the timestamp parameter. We do not want the delete

operation to succeed if the timestamp value passed in the parameter does not match to what is currently defined for the record being deleted.

To map the 3 stored procedures we can import the author's table along with 3 stored procedures. Screen shot below shows the insert, update and delete stored procedure mapping for author entity.

Parameter / Column	Operator	Property	Use Original Value
Functions			
Insert Using InsertAuthor			
Parameters			
@name : varchar	←	Name : String	
@bkpublished : int	←	BooksPublished : Int32	
Result Column Bindings			
AuthorId	→	AuthorId : Int32	
TimeStamp	→	TimeStamp : Binary	
<Add Result Binding>			
Update Using UpdateAuthor			
Parameters			
@name : varchar	←	Name : String	<input type="checkbox"/>
@bkpublished : int	←	BooksPublished : Int32	<input type="checkbox"/>
@timestamp : timestamp	←	TimeStamp : Binary	<input checked="" type="checkbox"/>
@authorid : int	←	AuthorId : Int32	<input checked="" type="checkbox"/>
@rowsaffected : int	←		<input type="checkbox"/>
Result Column Bindings			
TimeStamp	→	TimeStamp : Binary	
<Add Result Binding>			
Delete Using DeleteAuthor			
Parameters			
@authorid : int	←	AuthorId : Int32	
@timestamp : timestamp	←	TimeStamp : Binary	

For InsertAuthor stored procedure mapping, I am setting name and bkpublished parameter to Name and BooksPublished parameter on author entity. Since insert stored procedure returns authored and Timestamp value for optimistic concurrency, I am using ResultBinding to assign the valued received from the stored procedure to properties on author entity. For UpdateAuthor stored procedure, to ensure optimistic concurrency, I am passing the original value of the time stamp column by checking Use original

value checkbox to true. Additionally, I am also assigning the new value received for time stamp after an update operation succeeds to TimeStamp property on author entity. This would ensure that any subsequent updates will have the correct timestamp value. For DeleteAuthor stored procedure, I am mapping authorid and timestamp column to properties on the author entity. This way delete operation would fail if the timestamp value does not match with what's currently defined for the row being deleted.

To test that our stored procedure handles the optimistic concurrency we can insert author record and then update the record using a separate datacontext and on updating again using the original datacontext would raise OptimisticConcurrencyViolation. Code below shows exception being thrown when rowsaffected is returned zero from the stored procedure.

```
var db = new SpConcurrency();
    var author = new Author{Name =
"Zeeshan", BooksPublished = 1};
    db.AddToAuthors(author);
    db.SaveChanges();
    Console.WriteLine("authorid {0} timestamp
{1}", author.AuthorId,
    Convert.ToBase64String(author.TimeStamp));
    //cause concurrency violation
    db.Connection.Open();
    db.CreateCommand("update authors set
BooksPublished = 2 where name = 'Zeeshan' ")
    .ExecuteNonQuery();
    db.Connection.Close();
    author.Name = "Zeeshan Hirani";
    try
    {
        int result = db.SaveChanges();
    }
    catch (OptimisticConcurrencyException ex)
    {

        Console.WriteLine("Concurreny violation on
update " + ex.Message);
    }

    //update fails but let's try deleting the object.
    db.DeleteObject(author);
    try
```

```

{
    db.SaveChanges();
}
catch (OptimisticConcurrencyException ex)
{
    //delete also fails because of optimistic
concurrency.
    Console.WriteLine("Concurrent violation on
delete " + ex.Message);
    db.Detach(author);
    db.Attach(author);
}
//get a fresh copy from the database

db.Refresh(System.Data.Objects.RefreshMode.ClientWins, author);
author.Name = "Zeeshan Hirani";
//update author name succeeds
db.SaveChanges();
//delete to author succeeds.
db.DeleteObject(author);
db.SaveChanges();
Console.WriteLine("Author delete successfully");

```

On the above code, I am creating an instance of Author object and saving it to the database. Then using a CreateComand, I am updating booksPublished to 2 for author Zeeshan. This update causes the TimeStamp value for the record to get updated as well. Since entity framework is not aware of this update, the value of timestamp available on author's TimeStamp property does not match with what's defined on the database. Thus when I change the author name and save changes to the database, OptimisticConcurrencyException is raised indicating the record was modified since the last retrieval. Similarly when I try to delete the author entity, I also get the same exception because DeleteStoredprocedure checks to see if the timestamp value passed in matches to what is currently defined. To ensure a successful update, I refresh the author entity's original value from the database, and then once again update the author name and this time the update succeeds. Similarly delete operation on the author entity also succeeds. Screen shot below shows the result of the output printed on the console window.

```
cmd C:\Windows\system32\cmd.exe
authorid 1 timestamp AAAAAAAAJ14=
Concurrency violation on update Store update
tStateManager entries.
Concurrency violation on delete Store update
tStateManager entries.
Author delete successfully
Press any key to continue . . .
```

12.2 Exploring Entity Framework Extensions

Problem: Entity framework has several limitations in terms of executing stored procedures and arbitrary sql statements. Some of the limitations are as follows.

1. Executing arbitrary sql statement from ObjectContext.
2. Executing arbitrary stored procedure and materializing its result into entities without going through the entire process of declaring the stored procedure in ssdl, importing into conceptual model and declaring mapping for both models.
3. Be able to work at sql layer from ObjectContext and execute different commands such as ExecuteReader, ExecuteNonQuery and specify sql parameters instead of entity parameters.
4. Materialize data from data readers into entities.
5. Returning more than 1 result from stored procedure.

You need to know how entity framework extensions fill the gap by providing the above features and how you can use it to read data from database and execute arbitrary commands.

Solution:

Discussion: Entity framework Extensions extends Object Context class with extension methods and provides a new class EntitySet that inherits from ObjectContext class providing several utility methods. These methods help solve common tasks that are sometimes fairly trivial to do if you are directly coding against entity framework. In some cases EF extensions fills in the gap for features that did not make it to version 1 release of the product. In this segment we will explore different feature set that helps working with entity framework easier.

Setting EntityReference Key

When you assign a foreign key value to an entity such as specifying a CategoryId for product entity, you have to create a fully qualified EntityKey which consist of 3 parameters. First parameter is the fully qualified entityset name which includes the name of the entity container. For instance category entity resides inside of Categories entitySet which is configurable via EDM designer. Second parameter is the column which is defined as Key for the entity and third parameter is the value for the key. Code below shows how to use categoryid entity key to assign Category EntityRefernce to Product entity.

```
var db = new NorthwindEFEntities();
        var product = new Product { ProductName =
"MyProduct" };

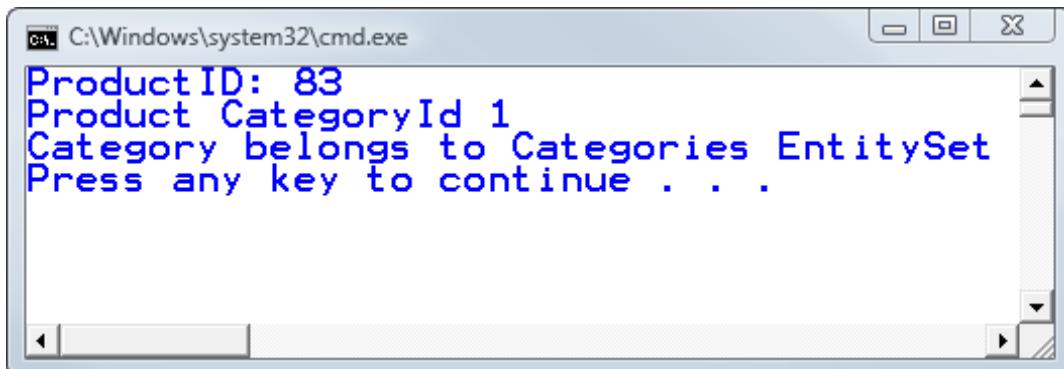
        //creating beverage entity key.
        var beveragekey = new
EntityKey("NorthwindEFEntities.Categories", "CategoryID", 1);
        product.CategoriesReference.EntityKey = beveragekey;
        db.AddToProducts(product);
        db.SaveChanges();
        Console.WriteLine("ProductID:
{0}", product.ProductID);
```

In the above code, the tedious work lies in creating an entity key. To create an entity key for category entity, I am specifying NorthWindEntities as my container and Categories as my entityset which contains my category entities. For the second parameter, I am specifying CategoryID as my key and third parameter being the value for the CategoryID. The above code is

redundant because Meta data in our conceptual model contains these definitions. When we use EF extensions, it gets rid of additional details by leveraging the conceptual model and allows us to simply specify a value for entity key. Code below shows how to achieve the same thing using EF extensions.

```
var product2 = new Product { ProductName = "Product2" };
    db.ProductSet.InsertOnSaveChanges(product2);
    //allow setting key without meta data.
    product2.CategoriesReference.SetKey(1);
    db.SaveChanges();
    //reading key value
    Console.WriteLine("Product CategoryId
{0}",product2.CategoriesReference.GetKey().ToString());
    //can also retrieve meta data and value back.
    Console.WriteLine("Category belongs to {0}
EntitySet",product.CategoriesReference.GetTargetEntitySet()
.Name);
```

By leveraging the extension method SetKey, we can simply pass the CategoryId value for the Entity Reference's CategoryID entity key. I am also using two other extension methods GetKey and GetTargetEntitySet which allows me retrieve the key value and the entityset, the Category entity belongs. Screen shot below shows the result of output window.



Executing Dynamic Sql and Stored Procedures

With entity framework you are required to import the stored procedure into SSDL, CSDL and then provider the mapping. In addition the stored procedure must return an entity defined on the conceptual modal. This is too

much work if you want to simply execute a stored procedure or arbitrary sql statement to get either an entity or any clr object back. Code below shows how to perform these simple tasks using EF extensions with few lines of code.

```
public class ProductPartial
{
    public int ProductID { get; set; }
    public string ProductName { get; set; }
}

//SettingEntityReference();
//get all categories using dynamic sql.
var db = new NorthwindEFEntities();
var categories = db.CreateStoreCommand("select * 
from categories").Materialize<Category>();
Console.WriteLine("Total Categories {0}
", categories.Count());

//returning plain clr type.
var prods = db.CreateStoreCommand("select 
ProductID,ProductName from Products where CategoryID = 5")
    .Materialize<ProductPartial>();
Console.WriteLine("Total Partial Products " +
prods.Count());
```

In the above code, I am executing a dynamic sql statement and materializing the results returned into collection of categories. Performing this code only required 1 only line of code as compared to manually doing it in entity framework would require at least 15 to 20 lines. Not only can you materialize the result to an entity, EF extensions allow you to map the results returned from sql statement into any clr type. In the case above, I have created ProductPartial class containing two properties. As long as I am matching the column names returned from the sql statement to properties on my class, materialization would be taken care by EF extensions. If columns returned from the query does not match with properties names, then you can use an overloaded Materialize method that lets you specify a lambda expression which is responsible for the translation. Code below shows how to use lambda expression to map ID and Name column to ProductID and ProductName on ProductPartial class.

```

//columns returned dont match with properties.
    var prods2 = db.CreateStoreCommand("select ProductID
as ID,ProductName as Name from Products where CategoryID = 5")
        .Materialize<ProductPartial>(r =>
            new ProductPartial {
                ProductID =
r.Field<int>("ID"),
                ProductName =
r.Field<string>("Name") })
        );

```

Entity framework extensions allow you to use any arbitrary stored procedure defined on the database without registering the stored procedure with entity data model. Example below is a stored procedure that returns two results.

First result returns suppliers for products with UnitPrice > 90 dollars.

Second result returns productid and ProductName of products with UnitPrice greater than 90 dollars.

```

create proc dbo.EFExMultipleResult
as
begin
select s.*
from Suppliers s join Products as p on s.SupplierID = p.SupplierID
where UnitPrice > 90

select ProductID,ProductName
from Products where UnitPrice > 90

end

```

In the version 1 release of entity framework, there is no out of box support for stored procedures that return multiple results. When using EF extensions, you can easily map multiple results to entity or any clr object defined in your project. Code below calls EFExMultipleResult stored procedure described above that returns suppliers and products. To call the stored procedure I am setting CommandType to StoredProcedure and calling ExecuteReader to get reader containing the results. Materializing the results of the reader must be performed in the order in which the results are returned from the stored procedure. Since the stored procedure returns suppliers followed by products, the first reader contains suppliers. To access the next result set returned, I am moving the reader to next result by calling NextResult.

```

//executing multiple resultset.
IEnumerable<Suppliers> suppliers;

```

```

IEnumerable<ProductPartial> prods3;
object quantityordered;
var multiplecmd = db.CreateStoreCommand("dbo.EFExMultipleResult",
CommandType.StoredProcedure);
using (multiplecmd.Connection.CreateConnectionScope())
using (var reader =
multiplecmd.ExecuteReader(CommandBehavior.CloseConnection))

{
    suppliers = multiplecmd.Materialize<Suppliers>();
    reader.NextResult();
    prods3 = multiplecmd.Materialize<ProductPartial>();
    reader.NextResult();
    // quantityordered = multiplecmd.ExecuteScalar();
}
Console.WriteLine("Results from multipleresult set");
Console.WriteLine("Total Suppliers {0}",suppliers.Count());
Console.WriteLine("Total Products {0}",prods3.Count());
Category cat;

```

Screen shot below shows the results returned from the stored procedure.

```

C:\WINDOWS\system32\cmd.exe
Total Categories 8
Total Partial Products 7
Results from multipleresult set
Total Suppliers 3
Total Products 3
Press any key to continue . . .

```

By default the entities returned using the extension methods are not tracked by the ObjectStateManager, therefore update and inserts and delete cannot be performed. To overcome this problem, EF extensions expose a Bind method on Entityset to attach these entities to the context passed in.

Some UtilityMethods on EF Extensions

Entity Framework extensions introduces a new class EntitySet that extends ObjectQuery class with some useful methods that helps working with EF framework a little easier. Following are some methods exposed by EntitySet that I have found useful.

EntitySet.Attach

If you have an entity that you received from WCF service, session or viewstate, in order to perform update, delete or insert you have first register the entity with EF. To register an entity you can either use Attach or AttachTo but you have to fully qualify the EntityKey that includes the EntityContainer, entity key property and entity key property value. With Attach method on EntitySet you can attach an entity without specifying an entity container or creating an entity key. Attach method creates an entity key if it is not provided. Code below shows the use of Attach method.

```
var db = new NorthwindEFEntities();
        var category = new Category { CategoryID = 1, CategoryName =
"Beverages" };
        //no need for entitykey or entity container.
        db.CategorySet.Attach(category);

Console.WriteLine(db.ObjectStateManager.GetObjectStateEntry(category).S
tate);
```

IQueryable Extensions

ToTraceString

When you write query using linq to entities or esql, ObjectQuery converts the command trees to into sql that is send to database for execution. To get the sql that is created ObjectQuery exposes ToTraceString method that returns the sql statement. However if there your query returns an IQueryable there is no way to access the sql statement unless you cast the results back to ObjectQuery and than get reference to ToTraceString method. With EF extensions, IQueryable also has a similar method which allows you call ToTraceString to get the sql generated. Code below calls ToTraceQuery extension method on IQueryable.

```
var db = new NorthwindEFEntities();
        //using ToTraceString on IQueryable.
        IQueryable<Product> products = db.Products.Where(p
=> p.UnitPrice > 20).Take(10);
        Console.WriteLine(products.ToTraceString());
```

Include

ObjectQuery has an include method that allows you to eagerly load related entities when fetching an entity or entities from the database. If the query returns an IQueryable, you do not get an option to specify an Include clause. One option is to cast IQueryable query to ObjectQuery and then specify and Include statement. EF extensions extends an IQueryable with an Include method so a query returning an IQueryable can specify addition related entities it wants to eagerly load. Example below calls Include on IQueryable of categories to load products for those categories. Since we are only interested in the first category, I use first operator to only take the first category. To confirm that are products are loaded as well I am printing the total products for the category retrieved from the database.

```
//Products included with category
    var category =
db.Categories.Take(2).Include("Products").First();
    Console.WriteLine("Total Products for category:
" + category.Products.Count());
```