



Adding native data querying capabilities to .NET languages: Language Integrated Query

White Paper – Language Integrated Query

Contents

LINQ- An Introduction	2
LINQ	3
LINQ to Objects	4
LINQ to ADO.NET.....	4
LINQ to XML	5
LINQ to SQL	5
Creating Entity Classes.....	5
The DataContext.....	6
Defining Relationships	7
Querying Across Relationships	9
Modifying and Saving Entities	9
Entity Life Cycle	10
Using a read only DataContext	13
Submitting Changes.....	13
Simultaneous Changes.....	14
Transactions	15
Stored Procedures.....	17
Conclusion.....	19

While every attempt has been made to ensure that the information in this document is accurate and complete, some typographical errors or technical inaccuracies may exist. Scalable Systems does not accept responsibility for any kind of loss resulting from the use of information contained in this document.

This page shows the publication date. The information contained in this document is subject to change without notice.

This text contains proprietary information, which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, stored in a retrieval system, transmitted in any form or by any means, or translated into another language without the prior written consent of Scalable Systems Inc.

LINQ: An Introduction

LINQ (Language Integrated Query) - A new technology comes with the NET Framework 3.5. While using the features of LINQ, the programmer can query multiple types of data structures with a query language similar to SQL; that is native to the source code. By using LINQ, the Programmer can write a compact query expression that can be applied to all in-memory information.

LINQ to SQL, an element of Visual Studio, provides a run-time infrastructure for managing relational data as objects without giving up the ability to query. It does this by translating language-integrated queries into SQL for execution by the database and then translating the tabular results back into objects you define. Your application is then free to manipulate the objects while LINQ to SQL stays in the background tracking your changes automatically.

LINQ can be used with any of the different languages that run under .NET. This means that, when using the features provided by LINQ, the developer now has the capability to query different types of data structures with a SQL-like query language that is native to the source code. Using LINQ, the developer can write a compact query expression that can be applied to all in-memory information.

LINQ to SQL is crafted to be non intrusive to your application. It is possible to transfer current ADO.NET solutions to LINQ to SQL in a gradual fashion, sharing the exact connections and transactions, since LINQ to SQL is quite another component in the ADO.NET family. LINQ to SQL also has extensive support for stored procedures, allowing reuse of the existing enterprise assets.

LINQ to SQL applications are easy to get started. Objects linked to relational data can be defined just like normal objects, only decorated with attributes to identify how properties correspond to columns. Of course, it is not even necessary to do this by hand. A design-time tool is provided to automate translating pre-existing relational database schemas into object definitions for you.

The .NET Language Integrated Query provides a number of operators that permit filtering, traversal, grouping and ordering of data. Datasets, user-defined objects, arrays, dictionaries and lists are all examples of data structures from which information can easily be derived. In fact, any type that implements IEnumerable can be queried.

LINQ

LINQ is a technology that covers many data domains.

<p>LINQ to Objects</p>	<p>LINQ to ADO.NET</p> <p><i>1.LINQ to SQL</i></p> <p><i>2.LINQ to Dataset</i></p>	<p>LINQ to XML</p>
-------------------------------	---	---------------------------

	<i>3.LINQ to Entities</i>	
--	---------------------------	--

LINQ to Objects

LINQ to Objects has the aspects of manipulating collections of objects, which can be related to each other to form a graph or a hierarchy. LINQ to Objects is the default implementation used by a LINQ query. LINQ to Objects is enabled including the **System.Linq** namespace.

LINQ to Objects queries are not limited to collections of user-generated data but LINQ query extracted over information from the file system.

The list of all files in a given directory is read in memory before being filtered by the LINQ query. LINQ query that gets temporary files greater than 10,000 bytes, ordered by size

```
string tempPath = Path.GetTempPath();
DirectoryInfo dirInfo = new DirectoryInfo( tempPath );
var query = from f in dirInfo.GetFiles()
            where f.Length > 10000
            orderby f.Length descending
            select f;
```

LINQ to ADO.NET

LINQ to ADO.NET contains many LINQ implementations that share the requirement to manipulate relational data. It comprises of other technologies i.e. specific to each particular persistence layer:

- **LINQ to SQL** Handles the mapping between custom types in C# and the physical table schema.
- **LINQ to Entities** Is in many ways similar to LINQ to SQL. However, instead of using the physical database as a persistence layer, it uses a conceptual Entity Data Model (EDM). The result is an abstraction layer that is independent from the physical data layer.
- **LINQ to DataSet** Makes it possible to query a DataSet using LINQ.

LINQ to Entities and LINQ to SQL have similarities as they both access information kept in a relational database and operate on object entities that represents external data in memory. The major difference is that they operate at a different level of abstraction. LINQ to SQL is tied to the physical database structure, while LINQ to Entities operates over a conceptual model (business entities) that might be distant from the physical structure (database tables).

The obvious reason behind these different options to access relational data through LINQ is, different models for database access are used in current days. Some implement all access through stored procedures, including any kind of database query, without the use of dynamic queries. Some others use stored procedures to insert, update, or delete data and dynamically built SELECT statements to query data. Some see the database as a simple object persistence layer, while others set some business logic into the database using triggers, stored procedures, or both. LINQ offers help and improvements in database access without forcing everyone to adopt a single comprehensive model.

LINQ to XML

LINQ to XML offers a slightly different syntax that operates on XML data, allowing query and data manipulation. A particular type of support for LINQ to XML is offered by Visual Basic 9.0, which includes XML literals in the language. This enhanced support simplifies the code necessary to manipulate XML data. In fact, you can write such a query in Visual Basic 9.0:

```
Dim book = _
    <Book Title="Introducing LINQ">
        <%= From person In team _
            Where person.Role = "Author" _
            Select <Author><%= person.Name %></Author> %>
    </Book>
```

This query corresponds to the following C# 3.0 syntax:

```
dim book =
    new XElement( "Book",
        new XAttribute( "Title", "Introducing LINQ" ),
        from person in team
        where person.Role == "Author"
        select new XElement( "Author", person.Name ) );
```

LINQ to SQL

The number one step to build a LINQ to SQL application is declaring the object classes you will use to represent your application data. Let's walk through an example.

Creating Entity Classes

We will start with a simple class Customer and associate it with the customers table in the Northwind sample database. To do this, we need only apply a custom attribute to the top of the class declaration. LINQ to SQL defines the Table attribute for this purpose.

```
[Table(Name="Customers")]
public class Customer
{
    public string CustomerID;
```

```

        public string City;
    }

```

The `Table` attribute has a `Name` property that you can use to specify the exact name of the database table. If no `Name` property is supplied LINQ to SQL will assume the database table has the same name as the class. Only instances of classes declared as tables will be able to be stored in the database. Instances of these types of classes are known as entities, and the classes themselves, entity classes.

In addition to associating classes to tables you will need to denote each field or property you intend to associate with a database column. For this, LINQ to SQL defines the `Column` attribute.

```

[Table(Name="Customers")]
public class Customer
{
    [Column(IsPrimaryKey=true)]
    public string CustomerID;

    [Column]
    public string City;
}

```

The `Column` attribute has a variety of properties you can use to customize the exact mapping between your fields and the database's columns. One property of note is the `Id` property. It tells LINQ to SQL that the database column is part of the table's primary key.

As with the `Table` attribute, you only need to supply information in the `Column` attribute if it differs from what can be deduced from your field or property declaration. In this example, you need to tell LINQ to SQL that the `CustomerID` field is part of the table's primary key yet you don't have to specify the exact name or type.

Only fields and properties declared as columns will be persisted to or retrieved from the database. Others will be considered as transient parts of your application logic.

The DataContext

The `DataContext` is the main tool by which you retrieve objects from the database and submit your changes back. You use it in the same way that you would use an ADO.NET Connection. In fact, the `DataContext` is initialized with a connection or connection string you supply. The purpose of the `DataContext` is to translate your requests for objects into SQL queries made against the database and then assemble objects out of the results. The `DataContext` enables Language-Integrated Query by implementing the same operator pattern as the Standard Query Operators such as `Where` and `Select`.

For example, you can use the `DataContext` to retrieve customer objects whose city is London as follows:

```

// DataContext takes a connection string
DataContext db = new    DataContext("c:\\northwind\\northwnd.mdf");

// Get a typed table to run queries
Table<Customer> Customers = db.GetTable<Customer>();

```

```
// Query for customers from London
var q =
    from c in Customers
    where c.City == "London"
    select c;

foreach (var cust in q)
    Console.WriteLine("id = {0}, City = {1}", cust.CustomerID, cust.City);
```

Each database table is represented as a Table collection accessible via the GetTable() method using its entity class to identify it. It is recommended that you declare a strongly typed DataContext instead of relying on the basic DataContext class and the GetTable() method. A strongly-typed DataContext declares all Table collections as members of the context.

```
public partial class Northwind : DataContext
{
    public Table<Customer> Customers;
    public Table<Order> Orders;

    public Northwind(string connection): base(connection) {}
}
```

The query for customers from London can then be expressed more simply as:

```
Northwind db = new Northwind("c:\\northwind\\northwnd.mdf");

var q =
    from c in db.Customers
    where c.City == "London"
    select c;

foreach (var cust in q)
    Console.WriteLine("id = {0}, City = {1}",cust.CustomerID, cust.City);
```

We will continue to use the strongly-typed Northwind class for the remainder of the overview document.

Defining Relationships

Relationships in relational databases are typically modeled as foreign key values referring to primary keys in other tables. To navigate between them you must explicitly bring the two tables together using a relational join operation. Objects, on the other hand, refer to each other using property references or collections of references navigated using 'dot' notation. Obviously, dotting is simpler than joining, since you need not recall the explicit join condition each time you navigate.

For such data relationships that will always be the same, it is quite convenient to encode them as property references in your entity class. LINQ to SQL defines an Association characteristic you can apply to a member used to symbolize a relationship. An association relationship is one like a foreign-key to primary-key relationship that is made by matching column values between tables.

```
[Table(Name="Customers")]
public class Customer
{
    [Column(Id=true)]
    public string CustomerID;
    ...
    private EntitySet<Order> _Orders;

    [Association(Storage="_Orders", OtherKey="CustomerID")]
    public EntitySet<Order> Orders {
        get { return this._Orders; }
        set { this._Orders.Assign(value); }
    }
}
```

The Customer class now has a property that declares the relationship between customers and their orders. The Orders property is of type EntitySet because the relationship is one-to-many. We use the OtherKey property in the Association attribute to describe how this association is done. It specifies the names of the properties in the related class to be compared with this one. There was also a ThisKey property we did not specify. Normally, we would use it to list the members on this side of the relationship. However, by omitting it we allow LINQ to SQL to infer them from the members that make up the primary key.

Notice how this is reversed in the definition for the Order class.

```
[Table(Name="Orders")]
public class Order
{
    [Column(Id=true)]
    public int OrderID;

    [Column]
    public string CustomerID;

    private EntityRef<Customer> _Customer;

    [Association(Storage="_Customer", ThisKey="CustomerID")]
    public Customer Customer {
        get { return this._Customer.Entity; }
        set { this._Customer.Entity = value; }
    }
}
```

The Order class uses the EntityRef type to describe the relationship back to the customer. The use of the EntityRef class is required to support deferred loading (discussed later). The Association attribute for the Customer property specifies the ThisKey property, since the non-inferable members are now on this side of the relationship.

Also take a look at the `Storage` property. It tells LINQ to SQL which private member is used to hold the value of the property. This allows LINQ to SQL to by-pass your public property accessors when it stores and retrieves their value. This is essential if you want LINQ to SQL to avoid any custom business logic written into your accessors. If the storage property is not specified, the public accessors will be used instead. You may use the `Storage` property with `Column` attributes as well.

Once you start introducing relationships in your entity classes, the amount of code you need to write grows as you introduce support for notifications and graph consistency. Fortunately, there is a tool (described later) that can be used to generate all the necessary definitions as partial classes, allowing you to use a mix of generated code and custom business logic.

For the rest of this document, we assume the tool has been used to generate a complete Northwind data context and all entity classes.

Querying Across Relationships

Now that you have relationships you can use them when you write queries simply by referring to the relationship properties defined in your class.

```
var q =
    from c in db.Customers
    from o in c.Orders
    where c.City == "London"
    select new { c, o };
```

The above query uses the `Orders` property to form the cross product between customers and orders, producing a new sequence of Customer and Order pairs.

It's also possible to do the reverse.

```
var q =
    from o in db.Orders
    where o.Customer.City == "London"
    select new { c = o.Customer, o };
```

In this example, the orders are queried and the `Customer` relationship is used to access information on the associated Customer object.

Modifying and Saving Entities

Some applications are built with single query in mind. Data must be created and modified too. LINQ to SQL is intended to provide maximum flexibility in manipulating and keep on making changes made to your objects. As soon as entity objects are there, either by retrieving them by a query or developed a new one, you may manipulate them as normal objects in your application, shifting their values or adding and removing them from collections as you see fit. LINQ to SQL track all your changes and is ready to broadcast them back to the database as soon as you are done.

The example below uses the Customer and Order classes generated by a tool from the metadata of the entire Northwind sample database. The class definitions have not been shown for brevity.

```
Northwind db = new Northwind("c:\\northwind\\northwnd.mdf");

// Query for a specific customer
string id = "ALFKI";
var cust = db.Customers.Single(c => c.CustomerID == id);

// Change the name of the contact
cust.ContactName = "New Contact";

// Create and add a new Order to Orders collection
Order ord = new Order { OrderDate = DateTime.Now };
cust.Orders.Add(ord);

// Ask the DataContext to save all the changes
db.SubmitChanges();
```

When `SubmitChanges()` is called, LINQ to SQL automatically generates and executes SQL commands in order to transmit the changes back to the database. It is also possible to override this behavior with custom logic. The custom logic may call a database stored procedure.

Entity Life Cycle

After entities are recovered from the database you are liberated to alter them as you like. They are your objects; use them as you need. As you do this LINQ to SQL tracks changes so it can persist them into the database when `SubmitChanges()` is called.

LINQ to SQL starts tracking your entities the moment they are retrieved from the database, before you ever lay your hands on them. Indeed the identity management service discussed earlier has already kicked in as well. Change tracking costs very little in additional overhead until you actually start making changes.

```
Customer cust = db.Customers.Single(c => c.CustomerID == "ALFKI");
cust.CompanyName = "Dr. Frogg's Croakers";
```

As soon as the `CompanyName` is assigned in the example above, LINQ to SQL becomes aware of the change and is able to record it. The original values of all data members are retained by the change tracking service

The change tracking service also records all manipulations of relationship properties. You use relationship properties to establish the links between your entities, even though they may be linked by key values in the database. There is no need to directly modify the members associated with the key columns. LINQ to SQL automatically synchronizes them for you before the changes are submitted.

```
Customer cust1 = db.Customers.Single(c => c.CustomerID == custId1);
```

```

foreach (Order o in db.Orders.Where(o => o.CustomerID == custId2)) {
    o.Customer = cust1;
}

```

You can move orders from one customer to another by simply making an assignment to their `Customer` property. Since the relationship exists between the customer and the order, you can change the relationship by modifying either side. You could have just as easily removed them from `cust2`'s `Orders` collection and added them to `cust1`'s collection, as shown below.

```

Customer cust1 = db.Customers.Single(c => c.CustomerID == custId1);
Customer cust2 = db.Customers.Single(c => c.CustomerID == custId2);

// Pick some order
Order o = cust2.Orders[0];

// Remove from one, add to the other
cust2.Orders.Remove(o);
cust1.Orders.Add(o);

// Displays 'true'
Console.WriteLine(o.Customer == cust1);

```

Of course, if you assign a relationship the value `null`, you are in fact getting rid of the relationship all together. Assigning a `Customer` property of an order to `null` actually removes the order from the customer's list.

```

Customer cust = db.Customers.Single(c => c.CustomerID == custId1);

// Pick some order
Order o = cust.Orders[0];

// Assign null value
o.Customer = null;

// Displays 'false'
Console.WriteLine(cust.Orders.Contains(o));

```

Automatic updating of both sides of a relationship is essential for maintaining consistency of your object graph. Unlike normal objects, relationships between data are often bi-directional. LINQ to SQL allow you to use properties to represent relationships, however, it does not offer a service to automatically keep these bi-directional properties in sync. This is a level of service that must be baked directly into your class definitions. Entity classes generated using the code generation tool have this capability. In the next chapter we will show you how to do this to your own hand written classes.

It is important to note, however, that removing a relationship does not imply that an object has been deleted from the database. Remember, the lifetime of the underlying data persists in the database until the row has been deleted from the table. The only way to actually delete an object is to remove it from its `Table` collection.

```

Customer cust = db.Customers.Single(c => c.CustomerID == custId1);

```

```

// Pick some order
Order o = cust.Orders[0];

// Remove it directly from the table (I want it gone!)
db.Orders.Remove(o);

// Displays `false`.. gone from customer's Orders
Console.WriteLine(cust.Orders.Contains(o));

// Displays `true`.. order is detached from its customer
Console.WriteLine(o.Customer == null);

```

Like with all other changes the order has not actually been deleted yet. It just looks that way to us since it has been removed and detached from the rest of our objects. When the order object was removed from the Orders table it was marked for deletion by the change tracking service. The actual deletion from the database will occur when the changes are submitted on a call to `SubmitChanges()`. Note that the object itself is never deleted. The runtime manages the lifetime of object instances, so it sticks around as long as you are still holding a reference to it. However, after an object has been removed from its Table and changes submitted it is no longer tracked by the change tracking service.

The only other time an entity is left untracked is when it exists before the `DataContext` is aware of it. This happens whenever you create new objects in your code. You are free to use instances of entity classes in your application without ever retrieving them from a database. Change tracking and identity management only apply to those objects that the `DataContext` is aware of. Therefore neither service is enabled for newly created instances until you add them to the `DataContext`.

This can occur in one of two ways. You can call the `Add()` method on the related Table collection manually.

```

Customer cust =
    new Customer {
        CustomerID = "ABCDE",
        ContactName = "FronD Smooty",
        CompanyTitle = "Eggbert's Eduware",
        Phone = "888-925-6000"
    };

// Add new customer to Customers table
db.Customers.Add(cust);

```

Or you can attach a new instance to an object that the `DataContext` is already aware of.

```

// Add an order to a customer's Orders
cust.Orders.Add(
    new Order { OrderDate = DateTime.Now }
);

```

The `DataContext` will discover your new object instances even if they are attached to other new instances.

```
// Add an order and details to a customer's Orders
Cust.Orders.Add(
    new Order {
        OrderDate = DateTime.Now,
        OrderDetails = {
            new OrderDetail {
                Quantity = 1,
                UnitPrice = 1.25M,
                Product = someProduct
            }
        }
    }
);
```

Basically, the DataContext will recognize any entity in your object graph that is not currently tracked as a new instance, whether or not you called the Add() method.

Using a read only DataContext

There are many cases that don't require updating the entities recovered from the database. Showing a table of Customers on a web page is one obvious example. In all such cases, it is possible to improve performance by instructing the DataContext not to track the changes to the entities. This is achieved by specifying the ObjectTracking property on the DataContext to be false as in the following code:

```
db.ObjectTracking = false;

var q = db.Customers.Where( c => c.City = "London");
foreach(Customer c in q)
    Display(c);
```

Submitting Changes

In spite of of changes you make to your objects, those changes were only made to in-memory replicas. Nothing yet has happened to the actual data in the database. Transmission of this information to the server will not happen until you explicitly request it by calling SubmitChanges() on the DataContext.

```
Northwind db = new Northwind("c:\\northwind\\northwnd.mdf");

// make changes here

db.SubmitChanges();
```

When you do call SubmitChanges() the DataContext will attempt to translate all your changes into equivalent SQL commands, inserting, updating or deleting rows in corresponding tables. These actions can be overridden by your own custom logic if you desire, however the order of submission is orchestrated by a service of the DataContext known as the change processor.

The first thing that happens when you call `SubmitChanges()` is that the set of known objects are examined to determine if new instances have been attached to them. These new instances are added to the set of tracked objects. Next, all objects with pending changes are ordered into a sequence of objects based on dependencies between them. Those objects whose changes depend on other objects are sequenced after their dependencies. Foreign key constraints and uniqueness constraints in the database play a big part in determining the correct ordering of changes. Then just before any actual changes are transmitted, a transaction is started to encapsulate the series of individual commands unless one is already in scope. Finally, one by one the changes to the objects are translated into SQL commands and sent to the server.

At this point, any errors detected by the database will cause the submission process to abort and an exception will be raised. All changes to the database will be rolled back as if none of the submissions ever took place. The `DataContext` will still have a full recording of all changes so it is possible to attempt to rectify the problem and resubmit them by calling `SubmitChanges()` again.

```
Northwind db = new Northwind("c:\\northwind\\northwnd.mdf");
// make changes here
try {
    db.SubmitChanges();
}
catch (Exception e) {
    // make some adjustments
    ...
    // try again
    db.SubmitChanges();
}
```

When the transaction around the submission completes successfully the `DataContext` will accept the changes to the objects by simply forgetting the change tracking information.

Simultaneous Changes

There are many reasons for call to `SubmitChanges()` may fail. You have created an object with an invalid primary key, one that's already in use, or with a value that breached some check constraint of the database. These checks are difficult to fit into business logic since they often require complete knowledge of the entire database state. However, the most likely reason for failure is simply that somebody else made some alterations to the objects before you.

This would be next to impossible if you were locking each object in the database and using a fully sequential transaction. However, this style of programming (pessimistic concurrency) is barely used since it is expensive and true conflicts seldom occur. The most popular form of managing simultaneous changes is to employ a form of optimistic concurrency. In this model, no locks against the database rows are taken at all. That means any number of alterations to the database has happened between the time you first retrieved your objects and the time you submitted your changes.

So unless you want to go with a policy that the last update wins, wiping over whatever else occurred before you, you probably want to be alerted to the fact that the underlying data was changed by someone else.

The `DataContext` has built in support for optimistic concurrency by automatically detecting change conflicts. Individual updates only succeed if the database's current state matches the state you understood the data to be in when you first retrieved your objects. This happens on a per object basis, only alerting you to violations if they happen to objects you have made changes to.

You can control the degree to which the `DataContext` detects change conflicts when you define your entity classes. Each `Column` attribute has a property called `UpdateCheck` that can be assigned one of three values; `Always`, `Never`, and `WhenChanged`. If not set the default for a `Column` attribute is `Always`, meaning the data values represented by that member are always checked for conflicts. That is, unless there is an obvious tie-breaker like a version stamp. A `Column` attribute has an `IsVersion` property that allows you to specify whether the data value constitutes a version stamp maintained by the database. If a version exists, then the version is used alone to determine if a conflict has occurred.

When a change conflict does occur an exception will be thrown just as if it were any other error. The transaction surrounding the submission will abort yet the `DataContext` will remain the same allowing you the opportunity to rectify the problem and try again.

```
while (retries < maxRetries) {
    Northwind db = new Northwind("c:\\northwind\\northwnd.mdf");

    // fetch objects and make changes here

    try {
        db.SubmitChanges();
        break;
    }
    catch (ChangeConflictException e) {
        retries++;
    }
}
```

If you are making changes on a middle-tier or server, the easiest thing you can do to rectify a change conflict is to simply start over and try again, recreating the context and reapplying the changes.

Transactions

Transaction is a service provided by a databases or any other resource manager that can assure a series of individual actions happen atomically, meaning either they all succeed or they all don't, and if they don't then they are also all involuntarily undone before anything else is permitted to happen. If no transaction is already in scope, the `DataContext` will automatically start a database transaction to guard updates when you call `SubmitChanges()`.

You may choose to control the type of transaction used, its isolation level or what it actually encompasses by initiating it yourself. The transaction isolation that the `DataContext` will use is known as `ReadCommitted`.

```
Product prod = db.Products.Single(p => p.ProductID == 15);

if (prod.UnitsInStock > 0)
    prod.UnitsInStock--;

using(TransactionScope ts = new TransactionScope()) {
    db.SubmitChanges();
    ts.Complete();
}
```

The example above initiates a fully serialized transaction by creating a new transaction scope object. All database commands executed within the scope of the transaction will be guarded by the transaction.

```
Product prod = db.Products.Single(p => p.ProductId == 15);

if (prod.UnitsInStock > 0)
    prod.UnitsInStock--;

using(TransactionScope ts = new TransactionScope()) {
    db.ExecuteCommand("exec sp_BeforeSubmit");
    db.SubmitChanges();
    ts.Complete();
}
```

This modified version of the same example uses the `ExecuteCommand()` method on the `DataContext` to execute a stored procedure in the database right before the changes are submitted. Regardless of what the stored procedure does to the database, we can be certain its actions are part of the same transaction.

If the transaction completes successfully the `DataContext` throws out all the accumulated tracking information and treats the new states of the entities as unchanged. It does not, however, rollback the changes to your objects if the transaction fails. This allows you the maximum flexibility in dealing with problems during change submission.

It is also possible to use a local SQL transaction instead of the new `TransactionScope`. LINQ to SQL offers this capability to help you integrate LINQ to SQL features into pre-existing ADO.NET applications. However, if you go this route you will need to be responsible for much more.

```

Product prod = q.Single(p => p.ProductId == 15);

if (prod.UnitsInStock > 0)
    prod.UnitsInStock--;

db.Transaction = db.Connection.BeginTransaction();
try {
    db.SubmitChanges();
    db.Transaction.Commit();
}
catch {
    db.Transaction.Rollback();
    throw;
}
finally {
    db.Transaction = null;
}

```

As you can see, using a manually controlled database transaction is a bit more involved. Not only do you have to start it yourself, you have to tell the `DataContext` explicitly to use it by assigning it to the `Transaction` property. Then you must use a try-catch block to encase your submit logic, remembering to explicitly tell the transaction to commit and to explicitly tell the `DataContext` to accept changes, or to abort the transactions if there is failure at any point. Also, don't forget to set the `Transaction` property back to null when you are done.

Stored Procedures

When `SubmitChanges()` is called LINQ to SQL creates and executes SQL commands to insert, update and delete rows in the database. These actions can be overridden by application programmers and their custom code can be used to execute the desired actions. In this way, substitute facilities like database stored procedures can be called up automatically by the change processor.

Consider a stored procedure for updating the units in stock for the `Products` table in the Northwind sample database. The SQL declaration of the procedure is as follows.

```

create proc UpdateProductStock
    @id int,
    @originalUnits int,
    @decrement int
as

```

You can use the stored procedure instead of the normal auto-generated update command by defining a method on your strongly-typed `DataContext`. Even if the `DataContext` class is being auto-generated by the LINQ to SQL code generation tool, you can still specify these methods in a partial class of your own.

```

public partial class Northwind : DataContext
{
    ...

    public void UpdateProduct(Product original, Product current) {
        // Execute the stored procedure for UnitsInStock update
        if (original.UnitsInStock != current.UnitsInStock) {
            int rowCount = this.ExecuteCommand(
                "exec UpdateProductStock " +
                "@id={0}, @originalUnits={1}, @decrement={2}",
                original.ProductID,
                original.UnitsInStock,
                (original.UnitsInStock - current.UnitsInStock)
            );
            if (rowCount < 1)
                throw new Exception("Error updating");
        }
        ...
    }
}

```

The signature of the method and the generic parameter tells the `DataContext` to use this method in place of a generated update statement. The `original` and `current` parameters are used by LINQ to SQL for passing in the original and current copies of the object of the specified type. The two parameters are available for optimistic concurrency conflict detection. Note that if you override the default update logic, conflict detection is your responsibility.

The stored procedure `UpdateProductStock` is invoked using the `ExecuteCommand()` method of the `DataContext`. It returns the number of rows affected and has the following signature:

```
public int ExecuteCommand(string command, params object[] parameters);
```

The object array is used for passing parameters required for executing the command.

Similar to the update method, insert and delete methods may be specified. Insert and delete methods take only one parameter of the entity type to be updated. For example methods to insert and delete a `Product` instance can be specified as follows:

```

public void InsertProduct(Product prod) { ... }
public void DeleteProduct(Product prod) { ... }

```

Conclusion

LINQ is programming model that commence queries as a first-class concept into any Microsoft .NET language. LINQ to SQL can noticeably reduce the amount of code you need to inscribe when preparing database-driven web applications.

About Scalable Systems:.

Scalable Systems is a premier global IT consulting, development and outsourcing company providing both offshore and onshore software solutions and integration services to business enterprises around the globe. Scalable Systems has proven expertise in encompassing low cost, but high quality and reliable software solutions and services in areas like Data Warehousing, Business Intelligence, Enterprise Resource Planning, Web and Custom Application Development.

Scalable Systems Inc
525 Milltown Road, Suite 303
North Brunswick, NJ 08902, USA
Tel : (732) 993 4320
Fax: (732) 909 2732
Email: info@scalable-systems.com

Copyright © 2008 Scalable Systems. All Rights Reserved.

While every attempt has been made to ensure that the information in this document is accurate and complete, some typographical errors or technical inaccuracies may exist. Scalable Systems does not accept responsibility for any kind of loss resulting from the use of information contained in this document. The information contained in this document is subject to change without notice.

Scalable Systems logos, and trademarks or registered trademarks of Scalable Systems or its subsidiaries in the United States and other countries. Other names and brands may be claimed as the property of others. Information regarding third party products is provided solely for educational purposes. Scalable Systems is not responsible for the performance or support of third party products and does not make any representations or warranties whatsoever regarding quality, reliability, functionality, or compatibility of these devices or products.