The Evolution of the C# Language: The Impact of Syntactic Sugar and Language Integrated Query on Performance

by

Ahmad Emad Mageed

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 14, 2010

Keywords: C#, .NET, Language Integrated Query,
LINQ, performance, coding styles

Abstract

The C# language has seen a healthy adoption rate for a fairly young language. Each released version has introduced features that have addressed pain points in the version preceding it, as well as providing greater flexibility and expressiveness. Although each successive release offers beneficial features, this study is focused on the language features that are likely to be used in everyday development and problem solving, especially surrounding data manipulation.

The language is statically typed and is known to be similar to Java, another object-oriented language. As of C# 3.0 it is evident that the language designers are borrowing from other popular languages to evolve the language. C# 3.0 introduced lambda expressions, implicit typing, and a set of other features which, collectively, gave rise to the Language Integrated Query (LINQ). With LINQ and lambda expressions the language has borrowed from functional programming languages, increasing the power and brevity of programming. C# is a multi-paradigm language which enables programmers to write code in numerous styles and mix expressions belonging to different paradigms, thereby increasing the flexibility and descriptiveness of code. The paradigms supported by C# include being imperative, generic, reflective, object-oriented, and functional.

This thesis aims to investigate the performance differences between conventional coding and code using LINQ. The focus is on the C# language features that have contributed to the advent of LINQ as well as LINQ itself. Once the results are obtained a well-formed opinion can be reached with regards to the adoption of the new coding styles made possible with LINQ.

Acknowledgments

Now that I can lift my head up from my books, and from writing, it is tempting to think I achieved this on my own. However, I definitely had help during this educational sojourn.

First and foremost I thank God for the blessings I have received, for being guided to the pursuit of knowledge, and for paving the way for this arduous undertaking.

I am indebted to Dr. David Umphress for his endless support and guidance during my thesis research. His easy-going yet thorough style of teaching, mentoring and engagement has been insightful and has helped enhance the quality of my work.

I am also grateful to Dr. Dean Hendrix and Dr. Hari Narayanan for their time and willingness to support my research as committee members.

I would like to commemorate my father, Emad Mageed, without whom I may not have had the opportunity to be exposed to technology at a young age. Thanks to him my first interaction with a computer was playing Parsec on the Texas Instruments TI-99/4A[1]. Eventually I followed his footsteps into the field of engineering.

Special thanks to my family for bearing the brunt of the time I spent preoccupied with my studies. To my mother for her endless encouragement and to my brothers, Sherif and Ayman, for their confidence and for putting up with my assiduous study excuses to skip out on fun activities.

Finally, thanks to my wife Sara for her ceaseless patience and support as I worked towards completing my degree.

[1] The TI-99/4A was a home computer released in June of 1981. Parsec was a popular side-scrolling space shooter game.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Syntactic sugar is a term coined by Peter J. Landin, the late British computer scientist whose insight into the usage of lambda calculus to model programming languages gave way to functional programming. The term refers to an alternative syntax to express language concepts and makes the language sweeter, providing choices to perform operations in a different way that is more concise and may be preferred [13]. The distinction between required syntax and syntactic sugar is that the latter is an extra approach to writing an operation that can otherwise be achieved by other syntax. Hence, dropping the feature would not result in lost language functionality or an inability to perform a particular task.

The more powerful a language becomes, the potential exists for fewer lines of code to be written to achieve a particular task. Syntactic sugar is simply an expression in code that the compiler has the burden of decomposing and inferring meaning from to determine the intent behind a statement, where such actions are known as “compiler tricks.” Despite the brevity gained from such expressions, the responsibility of maintaining readability and performance lies with the developer who must understand the tradeoffs and appropriateness of the techniques used. LINQ is comparable to regular expressions in the sense that a brief pattern is powerful enough to yield results for string parsing and validation, yet it is possible to derive terse and complex patterns that adversely affect clarity and comprehension.

1.1 Adopting New Language Enhancements

LINQ was designed to bridge the coding and data source worlds in a way that kept specific source data language separate from code. Interaction between data and code often involves translating object representations from one from to another. This disconnect is commonly referred to as an impedance mismatch. In brief, data is not equivalent to an object in code. LINQ offers the ability to write integrated code that can be used for querying data and treating results as objects [16].

Determining the appropriateness of embracing new techniques should be a concern of conscientious software professionals. Their promotion to first class citizens in the language should not automatically promote them to first choice tools. A developer should understand the consequences of the constructs and idioms employed while asking themselves a key question: "does writing the code in this manner affect performance?"

The central focus of the thesis is adequately summed up by this question. The question elicits a deeper look into the C# language history, highlighting the enhancements along the way, particularly the C# 3.0 features that make LINQ possible. A comparative study and benchmark of traditional approaches versus LINQ approaches will be conducted.

1.2 Hypothesis Definition

The expectation of this thesis is expressed by the following hypothesis: "Traditional coding techniques perform better than LINQ equivalent code."

The hypothesis will be tested with regards to in-memory objects, SQL interaction, and XML interaction. The effort involved in the acceptance or rejection of the null hypothesis is detailed in chapter three.

Chapter 2: Previous Work

Previous work related to the topic of LINQ performance is scarce. Formal study backed by comparative empirical analysis is not readily available. Books on the subject allude to performance in a brusque manner, skipping opportunities to provide benchmark data. This is not surprising, as LINQ is a fairly new technology which was added to the C# 3.0 language and .NET 3.5 framework and was released with Visual Studio 2008 in late 2007 [24]. Thus, performance related material is not widespread.

A series of blog articles related to this topic was posted by Rico Mariani, a senior software architect at Microsoft. The articles focused on performance of LINQ to SQL during its beta 2 release. The conclusion of the benchmarks was that LINQ to SQL yielded a 93% throughput in comparison to traditional code [17]. While the benchmark was informative, it investigated a limited number of operations on a pre-release version of LINQ which may have been tweaked since then and, in turn, may yield different performance results. Moreover, the series did not address the other LINQ providers.

Another motivation for the thesis topic was born from two complementary books, (1) LINQ in Action, and (2) C# in Depth. The former focuses on all flavors of LINQ with a brief introduction on the language enhancements that enabled LINQ. It covered small examples of traditional and LINQ code and a small section showing performance comparisons, however it was not exhaustive. The latter book focused on the chronological enhancements of C# from versions 1.0 to 3.0, especially the C# 3.0 features that makes LINQ possible.

2.1 Anticipated Benefits

The focus of this research is to shed light on the changes that have culminated in the C# 3.0 release. By doing so and understanding how powerful the language has become over time, the benefits anticipated include understanding the relative simplicity of performing tasks in C# 3.0 and, naturally, future versions of the language.

Another potential benefit is providing reluctant organizations with an understanding of the language so that adoption rates increase. No doubt migration and upgrades are not always seamless, yet keeping up with the fast-paced changes in the software engineering field has a benefit that pays dividends thrice. Firstly, keeping up with technology will preserve the system's relevance and ensure that future developers and system maintainers are familiar with popular languages of today, thereby avoiding obsolescence that leads to costly legacy system maintenance. Secondly, development time is reduced, allowing for quicker prototyping as well as gaining an edge in competition and customer satisfaction by achieving a lower time-to-market. Lastly, language adoption can enhance an organization's developers' knowledge, careers, productivity, excitement and appreciation when embracing cutting edge features that are built on a solid foundation. Granted not all existing systems need to be updated to the latest and greatest, depending on need and complexity, but new systems or areas of an existing system can benefit from incremental changes.

It is important to note that some C# 3.0 features can work on existing .NET 2.0 frameworks since the Common Language Runtime (CLR) was unchanged between it and .NET 3.5. By targeting the .NET 2.0 framework, via a compiler option or through Visual Studio 2008 (or other IDEs), the C# 3.0 compiler can translate new language syntax features to allow them to function on the targeted framework. Therefore, adopting .NET 3.5 is not necessary to reap the rewards of some of the features. While LINQ is not available in .NET 2.0, it is possible make use of it through the LINQBridge project. LINQBridge is beyond the scope of this thesis, and only supports LINQ to Objects, but it is worth mentioning that it makes it possible to use a flavor of LINQ on the .NET 2.0 framework.

2.2 Foundational Background

A number of new features were added to C# 3.0 in order to provide LINQ. Individually most of these features can be classified as syntactic sugar that, while helpful in cutting down on the tediousness of repetitive code, have viable – albeit more verbose – workarounds in the language. This section will cover some foundational background by introducing implicitly typed local variables, automatic properties, object and collection initializers, anonymous types, extension methods, and lambda expressions.

The final portion of this section will demonstrate how all the aforementioned features come together when using LINQ.

2.2.1 Implicitly Typed Local Variables

C# 3.0 introduced the *var* keyword to declare an implicitly typed local variable. It represents a strongly typed variable that the compiler will determine at compile time. Using it helps with reducing the amount of repetitive declaration code that explicitly declares a variable's type. For straightforward usage it is entirely optional, however it is required in certain LINQ query scenarios that return anonymous types. Figure 2.1 shows an example of splitting a string and initializing a *Person* object using pre-C# 3.0 code.

```
// string splitting
string hello = "Hello, World!";
string[] split = hello.Split(' ');
foreach (string item in split)
{
   Console.WriteLine(item);
}

// Person object
Person ahmad = new Person(42, "Ahmad");
```

Figure 2.1 – Pre-C# 3.0 operations using explicitly declared variables.

The equivalent code using the *var* keyword is shown in figure 2.2.

```
// string splitting
var hello = "Hello, World!";
var split = hello.Split(' ');
foreach (var item in split)
{
   Console.WriteLine(item);
}

// Person object
var ahmad = new Person { Id = 42, Name = "Ahmad" };
```

Figure 2.2 – C# 3.0 operations using implicitly declared variables.

While this demonstrates the usage of the *var* keyword, excessive or inappropriate usage that leads to unreadable code is undesirable. For example, using it for numeric types is confusing and does not save much typing [28]. In contrast, the usage of the *var* keyword for the *ahmad* variable is ideal since the variable type is easily discernible by looking at the right-hand part of the declaration. Similarly, most developers are familiar with the *String.Split* method and can figure out that the return type is *string[]* (a string array) [10]. It is clear that *ahmad* is of type *Person*.

In certain scenarios it is best to use a *var* and allow the compiler to determine the best type instead of being explicit. LINQ to SQL offers such scenarios, where queries return an *IQueryable<T>* and are evaluated on the database end. Forcing a query to be placed into an

IEnumerable<T> typed variable results in all the data being sent from the database and evaluated in memory on the local client end. Such an oversight is a performance issue [31].

2.2.2 Automatic Properties and Object and Collection Initializers

Prior to C# 3.0 initializing a class and settings its properties was achieved by passing the parameters to the class constructor, if an appropriate overload was made available, or by declaring the class then setting each individual property. This could become repetitive and setting up a number of objects could span a number of lines of code. In addition, each class property declared also involved declaring an associated private backing field with the sole purpose of storing the property's value. To demonstrate the described syntax a *Person* class will be used as shown in figure 2.3.

```
public class Person
{
    private int _id;    // Id property's private backing field
    // Id property
    public int Id
    {
        get { return _id; }
        set { _id = value; }
    }

    private string _name;    // Name property's private backing field
    // Name property
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public Person() { }    // default constructor with no parameters

    // overloaded constructor that accepts parameters
    public Person(int id, string name)
    {
        _id = id;
        _name = name;
    }
}
```

Figure 2.3 – Pre-C# 3.0 Person class.

Before C# 3.0, using the *Person* class from figure 2.3 could be done using code similar to figure 2.4.

```
// set properties via parameters
Person ahmad = new Person(42, "Ahmad");

// set properties individually
Person emad = new Person();
emad.Id = 1;
emad.Name = "Emad";
```

Figure 2.4 – Code using the Pre-C# 3.0 Person class.

In C# 3.0 two new features drastically reduce the amount of code needed to serve the same purpose. Firstly, automatic properties remove the need to declare the private backing field for properties that directly get and set values. Most properties are setup for precisely this purpose. If any additional code is used during the get and set actions then a private backing field would still be needed. Secondly, object initializers allow classes to be initialized with a single expression without the need for an overloaded constructor. An added benefit to object initializers is the ability to specify the properties to set by name, which enhances the developer's understanding of a property's purpose (of course properly XML documented methods enhance the IntelliSense experience for constructors too). Moreover, the order of the properties does not matter, unlike an overloaded constructor which enforces a specific signature. With C# 3.0 the *Person* class can be reduced to the code in figure 2.5.

```
public class Person
{
   public int Id { get; set; }
   public string Name { get; set; }
}
```

Figure 2.5 – The updated C# 3.0 Person class using automatic properties.

The class can then be used by the code in figure 2.6.

```
var ahmad = new Person { Id = 42, Name = "Ahmad" };
Person emad = new Person { Name = "Emad", Id = 1 }; // order does not matter
```

Figure 2.6 – Using the Person class with object initializers in C# 3.0.

It is possible to support all three declaration approaches in C# 3.0. To do so, the *Person* class in figure 2.5 would need to add the overloaded constructor from figure 2.3 and set the properties instead of the private backing fields. However, by adding an overloaded constructor, the class would also need to support object initialization by explicitly adding a default parameterless constructor. This requirement arises from the fact that the compiler automatically generates a default constructor when a class does not provide one, as is the case in figure 2.5 [23]. Adding the constructor from figure 2.3 would no longer cause the compiler to add the default constructor, thus it must be defined explicitly. Finally, given a suitable constructor, it is possible to specify a parameter followed by the remaining public properties specified via object initialization, thereby mixing both approaches.

Collection initializers allow objects to be added to a collection in a single expression and cut down on repetitive calls to the collection's *Add* method. Instead, the compiler will call *Add* on our behalf provided the collection implements the *IEnumerable* interface and provides an *Add* method with the appropriate parameters used in the initialize [28]. Figure 2.7 shows the code used to add items to a collection prior to C# 3.0.

```
List<int> list = new List<int>();
list.Add(1);
list.Add(2);

Dictionary<int, Person> dictionary = new Dictionary<int, Person>();
Person p = new Person(42, "Ahmad");
dictionary.Add(1, p);
p = new Person(1, "Emad");
dictionary.Add(2, p);
```

Figure 2.7 – Using collections prior to C# 3.0.

With collection initializers, the C# 3.0 code to achieve the same goal is shown in figure 2.8.

```
var list = new List<int> { 1, 2 };

var dictionary = new Dictionary<int, Person>
{
   { 1, new Person { Id = 42, Name = "Ahmad" } },  // object initializer
   { 2, new Person(1, "Emad") }  // constructor
};
```

Figure 2.8 – Using collection initializers with C# 3.0.

2.2.3 Anonymous Types

Anonymous types represent objects created without specifying a type, hence they are unnamed and anonymous. Their declaration resembles that of object initializers without the type. The compiler creates a type for the anonymous type using the properties specified in the anonymous object initializer and infers each property's types based on the values assigned [16]. While they can be used on their own, their real purpose is to be used in conjunction with LINQ. Typically an existing class would be used since anonymous types are used in specific scopes and are not a substitute for regular classes. Figure 2.9 shows an anonymous type that represents a person's name, gender, and age.

```
var person = new
{
   Name = "Ahmad Mageed",
   Gender = 'M',
   Age = 28
};

// prints Name: Ahmad Mageed, Gender: M, Age: 28
Console.WriteLine("Name: {0}, Gender: {1}, Age: {2}", person.Name, person.Gender, person.Age);
```

Figure 2.9 – Anonymous type example.

Other anonymous types can be generated and, as long as the properties are of the same type and are in the same order, they will be considered to be of the same type [28].

2.2.4 Extension Methods

C# extension methods provide a way to extend a type with a method that can called off of the type as though it were an instance method [28]. The methods must be static and declared in a static class. The .NET framework provides a method to check if a string is null or empty (*String.IsNullOrEmpty*), however the method only checks if the string is empty without trimming it. It is possible that after trimming the string it would be empty. Figure 2.10 shows how a new *IsNullOrWhiteSpace* method could be implemented and used.

```
void Main()
{
   string input = " Hello, World! ";
   if (IsNullOrWhiteSpace(input))
   {
      throw new ArgumentException("input");
   }
   Console.WriteLine(input);
}

public static bool IsNullOrWhiteSpace(string input)
{
   if (input != null)
   {
      for (int index = 0; index < input.Length; index++)
      {
         if (!Char.IsWhiteSpace(input[index]))
         {
            return false;
         }
      }
   }
   return true;
}
```

Figure 2.10 – Using a custom *IsNullOrWhiteSpace* method.

Transforming the method to an extension method is achieved by prefixing the first parameter of the method with the *this* keyword. Doing so indicates that the method is an extension method that acts upon the type of the first parameter. In this case, the extension method becomes available to any *string*. Figure 2.11 shows the updated code and usage.

```
void Main()
{
   string input = " Hello, World! ";
   if (input.IsNullOrWhiteSpace()) // extension method used differently
   {
      throw new ArgumentException("input");
   }
   Console.WriteLine(input);
}

public static class StringUtilities
{
   // usage of "this" keyword indicates an extension method
   public static bool IsNullOrWhiteSpace(this string input)
   {
      if (input != null)
      {
         for (int index = 0; index < input.Length; index++)
         {
            if (!Char.IsWhiteSpace(input[index]))
               return false;
         }
      }
      return true;
   }
}
```

Figure 2.11 – Using *IsNullOrWhiteSpace* as an extension method in C# 3.0.

Notice that the extension method is now called by using *input.IsNullOrWhiteSpace()* instead of passing it as a parameter to the method, although it can still be used in that manner. Extension methods can also be chained together, which is commonly done when using LINQ.

2.2.5 Lambda Expressions

C# 2.0 offered anonymous methods to create delegates which acted as inline functions. Lambda expressions are similar to delegates and offer a less verbose syntax. A lambda expression is not a delegate type, although it can be converted to one depending on the usage [28]. For example, a lambda expression (or lambda for brevity) can be used for *Func*, *Action*, and *Predicate* types of delegates. Figure 2.12 demonstrates the usage of anonymous methods to provide an inline *Predicate<int>* and *Action<int>* when working with a list of integers.

```
var list = new List<int>(Enumerable.Range(1, 10));  // list of integers 1-10

// Predicate<int> anonymous delegate to find integers greater than 5
var result = list.FindAll(delegate(int n) { return n > 5; });

// Action<int> anonymous delegate that prints 6, 7, 8, 9, 10
result.ForEach(delegate(int n) { Console.WriteLine(n); });
```

Figure 2.12 – Anonymous delegates used for Predicate and Action delegates.

The equivalent code using lambdas is shown in figure 2.13. The usage of the => symbol indicates a lambda expression. Lambda expressions are used by specifying the input parameters, the => symbol (known as the lambda operator), followed by the expression or statement block. The lambda operator is read as “goes to” [20]. In figure 2.13 the lambda used with the *FindAll* method checks whether the integer *n* is greater than five and returns a Boolean indicating the logical condition’s result. It can be read as, “n goes to n greater than five.” Similarly, the lambda used in the *ForEach* method passes each integer in the *result* variable to the *Console.WriteLine* method to be displayed. Compared to the anonymous delegates used in figure 2.12, the lambdas omit the usage of the return statement, curly braces for single expressions, and semicolons to terminate the expression.

```
var list = new List<int>(Enumerable.Range(1, 10));  // list of integers 1-10

// Predicate<int> lambda to find integers greater than 5
var result = list.FindAll(n => n > 5);

// Action<int> lambda that prints 6, 7, 8, 9, 10
result.ForEach(n => Console.WriteLine(n));
```

Figure 2.13 – Lambda expressions used for Predicate and Action delegates.

2.2.6 LINQ and the New C# 3.0 Language Enhancements

Collectively the C# 3.0 language enhancements enable the usage of LINQ and the related methods available in the .NET 3.5 base class libraries. An important aspect of LINQ is its lazily evaluated nature known as deferred query execution (or evaluation). This concept enables most queries to be built in memory without accessing the source data, thereby enhancing performance of queries, especially when multiple operators are chained together. Data is not accessed by assigning the query to a variable. The variable merely represents the query to execute once it is iterated over in a *foreach* loop or a request for the results is made.

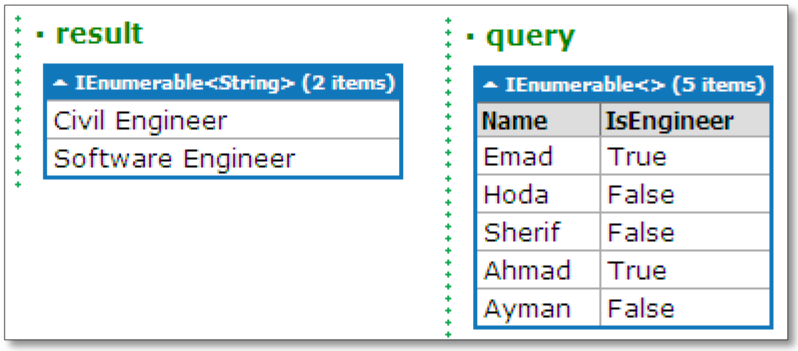
A few examples demonstrating the three LINQ providers benchmarked follow. Traditional approaches are not demonstrated in this section. To contrast traditional and LINQ approaches refer to the benchmark source code in Appendix A.

2.2.6.1 LINQ to Objects

Figure 2.14 demonstrates an example of LINQ to Objects. The representations of the queries in the *result* and *query* variables are shown in Figure 2.15 and are taken from the LINQPad tool [4]. The first LINQ query is stored in the *result* variable and uses the dot notation syntax. The *Enumerable.Where* method filters the collection for people that are engineers based on their occupation. At that point the query still has *Person* objects and is an *IEnumerable<Person>*. The chained *Enumerable.Select* method selects the occupations, yielding a final result of *IEnumerable<string>*. Next a loop is used to display the occupations.

```
public class Person
{
    public string Name { get; set; }
    public string Occupation { get; set; }
}

void Main()
{
    // populate list of Person objects
    var list = new List<Person>
    {
        new Person { Name = "Emad", Occupation = "Civil Engineer" },
        new Person { Name = "Hoda", Occupation = "Teacher" },
        new Person { Name = "Sherif", Occupation = "Pharmacist" },
        new Person { Name = "Ahmad", Occupation = "Software Engineer" },
        new Person { Name = "Ayman", Occupation = "Student" }
    };

    // filter for people that are engineers and select their occupations (dot notation syntax)
    var result = list.Where(person => person.Occupation.Contains("Engineer"))
                     .Select(person => person.Occupation);

    // prints "Civil Engineer" and "Software Engineer"
    foreach(string occupation in result)
    {
        Console.WriteLine(occupation);
    }

    // project into anonymous type to determine IsEngineer status
    var query = from person in list // query expression syntax
                let isEngineer = person.Occupation.Contains("Engineer")
                select new { person.Name, IsEngineer = isEngineer };

    // prints each person's name and their status as an engineer
    foreach(var item in query)
    {
        Console.WriteLine("{0} {1} an engineer", item.Name, item.IsEngineer ? "is" : "is not");
    }

    // Emad is an engineer
    // Hoda is not an engineer
    // Sherif is not an engineer
    // Ahmad is an engineer
    // Ayman is not an engineer
}
```

Figure 2.14 – LINQ to Objects example.



result

IEnumerable<String> (2 items)

Civil Engineer
Software Engineer

query

IEnumerable<> (5 items)

Name	IsEngineer
Emad	True
Hoda	False
Sherif	False
Ahmad	True
Ayman	False

Figure 2.15 – Representation of the *result* and *query* variables.

In the second example the result is stored in the *query* variable and the approach demonstrates the query expression syntax. The query expression syntax resembles a SQL query in reverse and is syntactic sugar that is ultimately compiled to the dot notation syntax. Certain queries are easier to think about and express in this syntax, such as queries involving grouping, joins, and temporary items declared with the *let* keyword. A projection using an anonymous type is made to capture the person's name and status as an engineer. The *Person* class does not have an *IsEngineer* property and is not being used in the projection. The anonymous type is created at compile time and consists of the *Name* and *IsEngineer* properties. The loop is used to reference each anonymous type's properties in a strongly typed manner, displaying the commented results shown at the end of the figure.

The alternative to using LINQ to Objects is to code the algorithms in a traditional manner to achieve the same results. Doing so involves declaring the appropriate collections to hold results, looping over collections with logical conditions to check for required criteria, and using variables for temporary storage of calculations.

2.2.6.2 LINQ to SQL

Working with LINQ to SQL entails defining a *DataContext* that represents the database tables to use and map objects to. To generate this object relational mapping either the Visual Studio visual designer or SqlMetal command line tool can be used. Figure 2.16 shows the class generated by dragging the *People* table from the custom AdventureWorksPeople database on to the visual designer of a LINQ to SQL class. The DataContext provides the mappings for it and each of its properties.

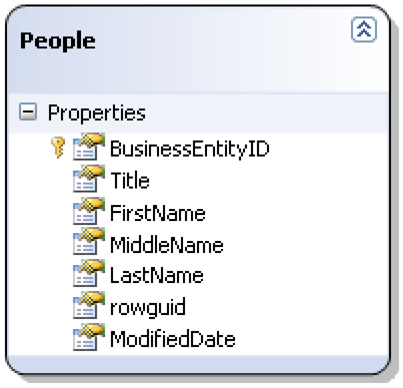


Figure 2.16 – The People table.

Querying the *People* table can be achieved by writing code similar to figure 2.17.

```
using (var dc = new AWPeopleDataContext())
{
   // get the first 5 people with a title of "Mr."
   var query = dc.Peoples.Where(person => person.Title == "Mr.")
                    .Take(5);

   // update ModifiedDate property
   foreach (var person in query)
   {
      person.ModifiedDate = DateTime.Now;
   }

   dc.SubmitChanges();
}
```

Figure 2.17 – LINQ to SQL example.

The code in figure 2.17 selects the first five people that match the filtration criterion. Next, the *ModifiedDate* property is updated for each result returned by the initial query. Finally, the changes are submitted and the necessary updates are performed. These commands generate a select statement and five update statements. The select statement and one of the update statements generated are shown in figure 2.18.

```
DECLARE @p0 NVarChar(3) = 'Mr.'
SELECT TOP (5) [t0].[BusinessEntityID], [t0].[Title], [t0].[FirstName], [t0].[MiddleName], [t0].[LastName],
[t0].[rowguid] AS [Rowguid], [t0].[ModifiedDate]
FROM [People] AS [t0]
WHERE [t0].[Title] = @p0
GO

DECLARE @p0 Int = 6
DECLARE @p1 DateTime = '2010-02-24 23:56:18.312'
UPDATE [People]
SET [ModifiedDate] = @p1
WHERE [BusinessEntityID] = @p0
GO
```

Figure 2.18 – SQL statements generated by the code in figure 2.17.

The alternative to using LINQ to SQL is to use the *SqlConnection* or *SqlDataAdapter* classes and use SQL statements and stored procedures directly. For benchmark purposes the *SqlConnection* class is used since it is the most light-weight option in the .NET framework.

2.2.6.3 LINQ to XML

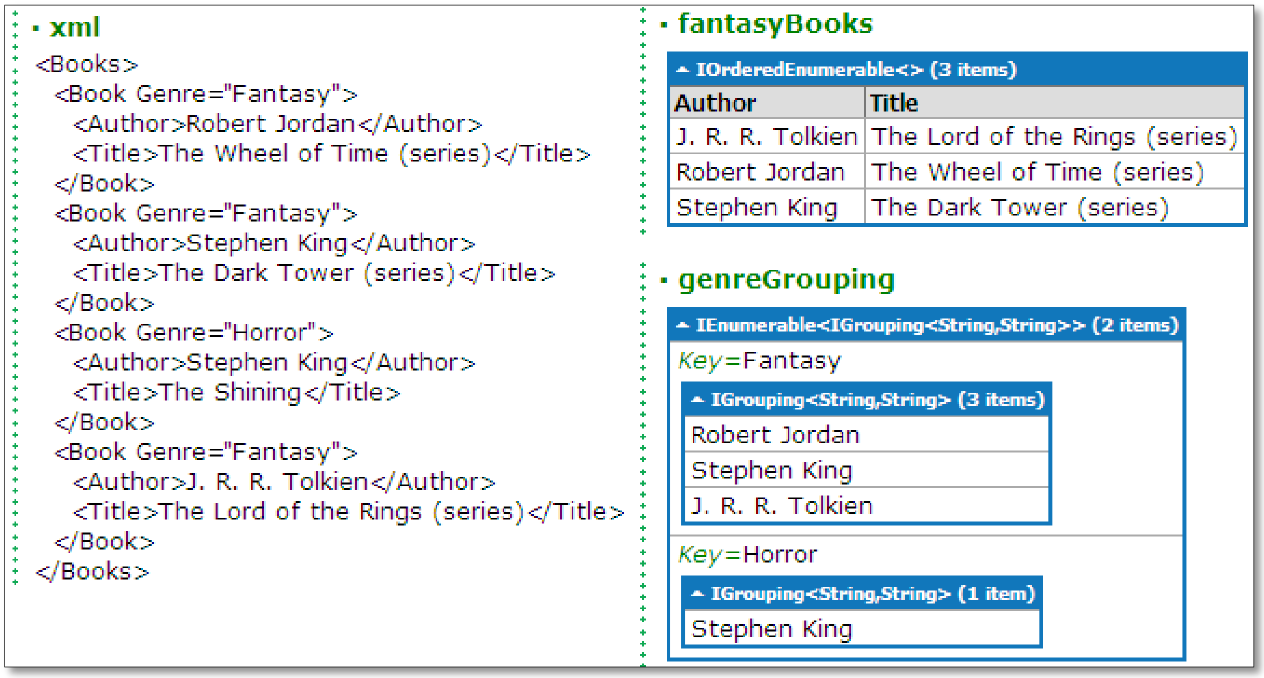
XPath is the alternative approach used for benchmark comparison against LINQ to XML. In figure 2.19 LINQ to XML is used to generate and query XML data of books. Alternately, XML data can be loaded from files and parsed from text. The results of the queries are shown in the comments.

```
XElement xml = new XElement("Books",
            new XElement("Book", new XAttribute("Genre", "Fantasy"),
               new XElement("Author", "Robert Jordan"),
               new XElement("Title", "The Wheel of Time (series)")),
            new XElement("Book", new XAttribute("Genre", "Fantasy"),
               new XElement("Author", "Stephen King"),
               new XElement("Title", "The Dark Tower (series)")),
            new XElement("Book", new XAttribute("Genre", "Horror"),
               new XElement("Author", "Stephen King"),
               new XElement("Title", "The Shining")),
            new XElement("Book", new XAttribute("Genre", "Fantasy"),
               new XElement("Author", "J. R. R. Tolkien"),
               new XElement("Title", "The Lord of the Rings (series)"))
           );

var fantasyBooks = xml.Elements("Book")
                .Where(book => book.Attribute("Genre").Value == "Fantasy")
                .Select(book => new { Author = book.Element("Author").Value, Title =
book.Element("Title").Value })
                .OrderBy(item => item.Author);

foreach (var item in fantasyBooks)
{
   Console.WriteLine("{0}: {1}", item.Author, item.Title);
}

// J. R. R. Tolkien: The Lord of the Rings (series)
// Robert Jordan: The Wheel of Time (series)
// Stephen King: The Dark Tower (series)

var genreGrouping = from book in xml.Elements("Book")
               group book.Element("Author").Value by book.Attribute("Genre").Value into grouping
               select grouping;

foreach (var group in genreGrouping)
{
   Console.WriteLine("Genre: " + group.Key);
   foreach (var item in group)
   {
      Console.WriteLine("- " + item);
   }
}
```

Figure 2.19 – LINQ to XML example.

The first query filters the books based on the fantasy genre, showing the matching authors and book titles, while the second query groups the authors based on the genres.

The LINQ to XML axis methods are used to provide a sequence from the XML that can then be used by the standard query operators [16]. Some of the axis methods are the *Elements*, *Descendants*, and *Attribute* methods. Once a sequence is available working with the data is as straightforward as working with LINQ to Objects. Figure 2.20 shows the representation of the *xml*, *fantasyBooks*, and *genreGrouping* variables using in figure 2.19.



▪ xml

```
<Books>
  <Book Genre="Fantasy">
    <Author>Robert Jordan</Author>
    <Title>The Wheel of Time (series)</Title>
  </Book>
  <Book Genre="Fantasy">
    <Author>Stephen King</Author>
    <Title>The Dark Tower (series)</Title>
  </Book>
  <Book Genre="Horror">
    <Author>Stephen King</Author>
    <Title>The Shining</Title>
  </Book>
  <Book Genre="Fantasy">
    <Author>J. R. R. Tolkien</Author>
    <Title>The Lord of the Rings (series)</Title>
  </Book>
</Books>
```

▪ fantasyBooks

IOrderedEnumerable<> (3 items)

Author	Title
J. R. R. Tolkien	The Lord of the Rings (series)
Robert Jordan	The Wheel of Time (series)
Stephen King	The Dark Tower (series)

▪ genreGrouping

IEnumerable<IGrouping<String,String>> (2 items)

Key=Fantasy

IGrouping<String,String> (3 items)

Robert Jordan
Stephen King
J. R. R. Tolkien

Key=Horror

IGrouping<String,String> (1 item)

Stephen King

Figure 2.20 – Representation of the *xml*, *fantasyBooks* and *genreGrouping* variables.

Chapter 3: Benchmark Solution

The scope of the thesis is to explore LINQ and the features comprised by it, comparing them to traditional approaches prior to the language enhancements, while being mindful of performance. The objective is to focus on the evolving features that provide alternate methods of writing code which produces identical results.

Performance is quantifiable and can be determined through a series of benchmark testing. However, proper care should be taken to ensure accurate measurements are reached. Such considerations will include using similar test environments and recording results of multiple runs in order to minimize variable impact.

3.1 Performance Artifacts

In order to compare traditional code with LINQ code, a number of artifacts will be introduced. These artifacts represent common scenarios that practitioners deal with and are areas where LINQ excels at simplifying in terms of reduced code verbosity.

The LINQ standard query operators defined by Microsoft are suitable artifacts [22]. Some artifacts are common between the LINQ providers measured, however not all benchmarks will demonstrate the use of the same artifacts since tasks differ for each technology. The LINQ to Objects benchmark will measure the artifacts shown in Table 3.1.

Category	**Query Operators**
Sorting Data	OrderBy, Reverse, ThenBy
Set Operations	Distinct, Except
Filtering Data	Where
Quantifier Operations	All, Any
Projection Operations	Select, SelectMany
Partitioning Data	Skip, SkipWhile, Take, TakeWhile
Join Operations	Join
Grouping Data	GroupBy
Equality Operations	SequenceEqual
Element Operations	ElementAt
Converting Data Types	ToArray, ToList, ToDictionary
Concatenation Operations	Concat
Aggregation Operations	Aggregate, Average, Count, Max, Min, Sum

Table 3.1 – LINQ to Object Benchmark Operations.

LINQ to SQL will benchmark common database interactions and utilize the operations mentioned in Table 3.2.

Category	Query Operators
SQL Specific Operations	Insert, Update, Delete
Sorting Data	OrderBy
Filtering Data	Where
Projection Operations	Select, SelectMany
Partitioning Data	Take
Join Operations	Join
Grouping Data	GroupBy
Converting Data Types	ToList, ToDictionary
Aggregation Operations	Count, Sum

Table 3.2 – LINQ to SQL Benchmark Operations.

LINQ to XML will benchmark common XML document actions and utilize the operations displayed in Table 3.3.

Category	**Query Operators**
XML Specific Operations	Generate, Update, Remove
Sorting Data	OrderBy
Filtering Data	Where
Projection Operations	Select, SelectMany
Partitioning Data	Take, SkipWhile
Grouping Data	GroupBy
Element Operations	ElementAt, Last
Converting Data Types	ToList, ToDictionary
Aggregation Operations	Average, Count, Sum

Table 3.3 – LINQ to XML Benchmark Operations.

3.2 Performance Comparison Approach

The approach taken to demonstrate performance will be conducted as follows:

1. Select an artifact item to use in the comparison.
2. Identify the expected result of applying the artifact on the sample data.
3. Write code to implement the functionality using traditional code.
4. Write code to implement the functionality using LINQ inclusive code.
5. Verify equivalence.
6. Run each code snippet and measure the time taken (in milliseconds), repeating the process for a total of 16 runs.
7. Perform analysis and comparison of the results for each artifact.

Equivalence, as identified by step five, is achieved by verifying that both traditional code and LINQ inclusive code produce the same end result (further details are available in chapter four).

The code will be compiled and run in *Release* mode to take advantage of optimized code compilation which does not apply to *Debug* mode [18].

3.3 Performance Calculation Methods

Calculating performance will be achieved in the following manner:

1. Run the program demonstrating a particular artifact for 16 times while collecting the time taken for each run.
2. Perform a two population means hypothesis test to accept or reject the null hypothesis.

3.4 Considerations for Performance Benchmarking

This section briefly explains some considerations for performance benchmarking that influenced how the benchmarking code was written. In particular the focus is on the Common Language Runtime's (CLR) Just-In-Time (JIT) compiler and .NET garbage collector.

3.4.1 Just-In-Time Compilation

In .NET the first time a method is called during runtime is the costliest. This is because the method needs to be translated from Intermediate Language (IL) to native CPU instructions. This translation is carried out by the JIT compiler. The JIT compiler searches the assembly's metadata for the calling method's IL, verifies it and compiles it to native CPU instructions. These instructions are saved in dynamic memory. This process represents the performance hit incurred by calling a method for the first time. Subsequent usage of the method performs better since the code has been previously translated and stored in memory [27].

Being aware of the JIT compiler and the initial performance hit, it is important to account for this when benchmarking. Vance Morrison, the Compiler Architect for the CLR team at Microsoft, stated that the "benchmark should be run once before taking a measurement to ensure that any just-in-time (JIT) compilation and other one-time initialization has completed." He also mentioned that "the benchmark should be run several times and statistics should be gathered to determine the stability of the measurement" [25].

To account for the initial JIT performance hit the benchmark application executes the actual benchmark methods five times. These five runs are throwaway JIT warm up runs. Once completed, the actual benchmark runs are executed. While the warm up runs are included in the logged results, they are excluded from the statistical analysis.

3.4.2 Garbage Collection

The C# garbage collector operates sufficiently and it is rare to force a garbage collection. Nonetheless, two scenarios where doing so may be favorable are (1) when the application has created a large number of objects, and (2) to avoid a potential garbage collection prior to executing a particular code block [30].

These two scenarios fit for the benchmarking application since the desire is to record accurate and consistent results. In order to provide each benchmark method with a clean slate, an explicit garbage collection request will be made. This is achieved by calling the *GC.Collect()* method which reclaims all inaccessible memory by forcing an immediate garbage collection of all generations [19]. In the benchmark application the code responsible for calling the *GC.Collect()* function is the *Common.LogMethod* function which is executed at the beginning of each individual benchmark method. Prior to settling on the decision to force garbage collection, the benchmarks were run with and without it. The results showed longer execution times when explicit garbage collection was omitted. That was expected since an accumulation of object allocations made by previously executed methods produced objects that were waiting to be collected.

3.5 LINQ to SQL Shortcomings and Optimizations

The LINQ to SQL object relational mapper (ORM) offers an abstracted approach to database interaction by generating strongly typed classes that implement the necessary *IQueryable* interface that, in turn, allows developers to use strongly typed code and operate on tables in a fluent manner. The lambda expressions used with the standard LINQ operators are ultimately translated to T-SQL statements. For simple statements the generated T-SQL is normally close to what developers would write. For other scenarios this may not always be the case. As a substitute regular T-SQL statements and stored procedures can be executed by the LINQ to SQL *DataContext*, although doing so results in the loss of type safety granted by the strongly typed approach.

One of the major shortcomings of the provider occurs in two of the basic and frequently used database operations, namely updates and deletes. Furthermore, whenever a *DataContext* object is created certain features are used by default for consistency and ease of setup. Depending on the task to perform, these settings can be modified to improve performance. Another area where performance gains are possible is with frequently used queries. These points will be addressed further in the subsequent sections.

3.5.1 Pain Points of Update and Delete Statements

LINQ to SQL generally exhibits consistent performance in most queries. However, the operations which suffer the most with a staggering difference in performance compared to traditional code are the insert (or create), update, and delete statements (henceforth referred to as CUD operations). Select (or read) statements do not vary much.

In particular, the reason for the performance discrepancy in an update or delete statement is due to the way the LINQ to SQL provider translates such operations into the final SQL statement. Normally these operations act on all records that match specific criteria expressed in a single SQL statement. Yet, with LINQ to SQL, a statement is generated for each matching record. To elaborate, a walkthrough comparison of a SQL delete statement will be explained. The same explanation applies to an update statement as well. Consider the traditional SQL delete approach in figure 3.1.

```
using (SqlConnection conn = AdventureWorksPeopleConn)
{
   string query = @"DELETE FROM People WHERE LastName=@LastNameCriteria";
   SqlParameter lastNameCriteria = new SqlParameter("@LastNameCriteria", SqlDbType.NVarChar);
   lastNameCriteria.Value = "Mageed";

   conn.Open();
   SqlCommand command = new SqlCommand(query, conn);
   command.Parameters.Add(lastNameCriteria);
   traditionalAffectedRecords = command.ExecuteNonQuery();
}
```

Figure 3.1 – SQL delete statement example.

When *SqlCommand.ExecuteNonQuery* is executed exactly one SQL statement is performed by the database. Figure 3.2 shows the equivalent LINQ to SQL code.

```
using (var dc = new AWPeopleDataContext())
{
   dc.Log = Console.Out; // displays executed SQL queries in the Console window
   var peopleToDelete = dc.Peoples.Where(p => p.LastName == "Mageed");
   dc.Peoples.DeleteAllOnSubmit(peopleToDelete);
   dc.SubmitChanges();
}
```

Figure 3.2 – LINQ to SQL delete statement example.

Once the *Table.DeleteAllOnSubmit* method is called using the *Peoples* table all records will be retrieved, filtered, and stored in the *peopleToDelete* variable using a select statement. Next, the *DataContext.SubmitChanges* method executes a delete statement for each record held in the *peopleToDelete* variable. By using the *DataContext.Log* property, as shown in figure 3.2, the described sequence of events can be confirmed as the actual queries are displayed in the console window upon being executed. A partial view of the generated queries is shown in figure 3.3.

```
SELECT [t0].[BusinessEntityID], [t0].[Title], [t0].[FirstName], [t0].[MiddleName], [t0].[LastName],
[t0].[rowguid], [t0].[ModifiedDate]
FROM [dbo].[People] AS [t0]
WHERE [t0].[LastName] = @p0
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [Mageed]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

DELETE FROM [dbo].[People] WHERE ([BusinessEntityID] = @p0) AND ([Title] = @p1)
AND ([FirstName] = @p2) AND ([MiddleName] = @p3) AND ([LastName] = @p4) AND ([rowguid] = @p5)
AND ([ModifiedDate] = @p6)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [394004]
-- @p1: Input NVarChar (Size = 3; Prec = 0; Scale = 0) [Mr.]
-- @p2: Input NVarChar (Size = 4; Prec = 0; Scale = 0) [Emad]
-- @p3: Input NVarChar (Size = 3; Prec = 0; Scale = 0) [1]
-- @p4: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [Mageed]
-- @p5: Input UniqueIdentifier (Size = 0; Prec = 0; Scale = 0) [5fafa8ec-d82b-41b2-a413-f624ad120378]
-- @p6: Input DateTime (Size = 0; Prec = 0; Scale = 0) [2/17/2010 9:10:10 PM]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

...

DELETE FROM [dbo].[People] WHERE ([BusinessEntityID] = @p0) AND ([Title] = @p1)
AND ([FirstName] = @p2) AND ([MiddleName] = @p3) AND ([LastName] = @p4) AND ([rowguid] = @p5)
AND ([ModifiedDate] = @p6)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [394503]
-- @p1: Input NVarChar (Size = 3; Prec = 0; Scale = 0) [Mr.]
-- @p2: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [Ahmad]
-- @p3: Input NVarChar (Size = 3; Prec = 0; Scale = 0) [500]
-- @p4: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [Mageed]
-- @p5: Input UniqueIdentifier (Size = 0; Prec = 0; Scale = 0) [6dd143d4-d628-4099-a1b1-29fc6564e8a1]
-- @p6: Input DateTime (Size = 0; Prec = 0; Scale = 0) [2/17/2010 9:10:10 PM]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1
```

Figure 3.3 – Partial results of generated queries from the *DataContext.Log* property.

From figure 3.3 the first query issued is the select statement that filters on parameter *p0* which has the value of "Mageed." The number of matching records in this example was 500. The first and last records are shown, with the records in between elided. In the first delete statement the *BusinessEntityID* value is given by *p0* with a value of 394004. The last delete statement uses a *BusinessEntityID* value of 394503, which occurs 500 records later since the data was prepared in sequential order for this example (the *MiddleName* column was used as a counter and reflects this as well).

Due to the demonstrated behavior the LINQ to SQL approach would be increasingly inefficient as the number of affected records increase. In contrast to the traditional *SqlCommand* approach, this issue presents a performance concern for database heavy applications that perform many updates and deletes.

To address these shortcomings Terry Aney developed a freely available library (with public source code) which has been used to perform an additional run of the LINQ to SQL benchmarks in an optimized manner. An official library name is not apparent, however the downloadable project filename is "LinqBatchPost.zip" and is linked to from the blog post Terry made on the subject. The library introduces extension methods that perform batch updates and deletes while remaining true to the strongly typed object oriented usage of LINQ to SQL. In short, the library relies on using expression trees and uses the SQL generated by the LINQ to SQL provider to modify the statement and use an inner join to wrap the action in a single statement [6]. The library improves performance as evidenced by the significantly reduced execution time experienced and the results of the statistical analysis available in chapter four.

3.5.2 DataContext Settings and Compiled Queries

To boost performance when using the LINQ to SQL provider a read-only *DataContext* can be used when its sole purpose will be for data retrieval. This is achieved by disabling the

DataContext.ObjectTrackingEnabled property, which is enabled by default [26]. Doing so effectively disables change- and identity-tracking services at the expense of preventing change updates [16]. This option was used for the optimized benchmarks that were strictly read-only.

Another option for enhancing performance is to compile frequently used queries. The process of running a query involves creating an expression tree, translating it to SQL, executing the query, fetching the data, and finally representing them as objects [17]. This process generates unnecessary overhead for frequently used queries. By compiling the query the overhead of expression tree creation and SQL translation are avoided upon subsequent uses of the compiled function [26]. Essentially the expression tree and SQL translation are determined once and stored, which allows SQL Server to use the same execution plan for similar queries. Parameters are passed in during execution, thereby allowing for flexible use of compiled queries. The *System.Data.Linq.CompiledQuery.Compile* method is used to compile the query for reuse as a generic function definition. It is recommended to define the compiled query function as a static method to have it evaluated once in the *AppDomain* lifetime [16].

For the optimized benchmarks all queries were compiled with the exception of CUD operations. Inserts are performed via a *Table.InsertOnSubmit* method so queries are not involved. Updates and deletes do not qualify in terms of syntax but the select statement used to retrieve the data to be modified can be compiled. Compilation of the select statement was not carried out, however, since the batch updates and deletion library featured no support for operating on compiled queries. Since the compiled queries were specific to a particular benchmark method they were included as part of that method's scope and evaluated once before the benchmark operations were carried out.

3.6 Benchmark Setup for Replication

In accordance with the Microsoft .NET benchmark testing terms [21], this section details all the needed information to comply with the terms set forth by Microsoft.

Benchmark testing occurred on February 1st, 2010. The results were achieved on a Lenovo T61 laptop with an Intel Core 2 Duo CPU (T8300 @ 2.40 GHz, 2.39 GHz) and 2.99 GB of RAM. The benchmark was compiled in release mode and the executable was run from the Visual Studio Command Prompt. All applications were shutdown to provide as much memory as possible to the process.

3.6.1 Resources

The following resources were used to develop the benchmark and can be used for replication purposes:

- Source code:
    - See Appendix A.
    - Terry Aney’s LinqBatchPost library (included in Appendix A).
- Databases:
    - AdventureWorks2008 Refresh 1 available from http://msftdbprodsamples.codeplex.com/.
    - AdventureWorksPeople: this is a custom database consisting of a single *People* table used for the LINQ to SQL insert, update, and delete related benchmarks. Its schema is similar to the AdventureWorks2008 Persons table excluding a few fields. The identical fields were populated with the data from the Persons table. The schema creation script is shown in figure 3.4.

```
CREATE TABLE [dbo].[People](
	[BusinessEntityID] [int] IDENTITY(1,1) NOT NULL,
	[Title] [nvarchar](8) NULL,
	[FirstName] [nvarchar](50) NOT NULL,
	[MiddleName] [nvarchar](50) NULL,
	[LastName] [nvarchar](50) NOT NULL,
	[rowguid] [uniqueidentifier] NOT NULL,
	[ModifiedDate] [datetime] NOT NULL,
 CONSTRAINT [PK_People] PRIMARY KEY CLUSTERED
(
	[BusinessEntityID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Figure 3.4 – The schema of the AdventureWorksPeople database's *People* table.

3.6.2 Source Code Overview

All benchmarks will operate on data retrieved from the AdventureWorks2008 database. Appendix C shows the mapped files generated by the visual designer. A service class is used to retrieve commonly used data with the help of a LINQ to SQL *DataContext*. Randomizing the data is achieved by implementing a *Random* method in a partial class that relies on the SQL Server *NEWID* function (see AWDataContextPartial.cs in Appendix A).

The LINQ to Objects benchmarks will retrieve data from a service class that returns an in-memory .NET collection of the appropriate class or Plain Old CLR Object (POCO). For LINQ to SQL the data will be queried directly then represented as POCOs. The LINQ to XML provider will work with generated XML files (based on random data) as well as the service class for XML generation benchmarks.

Chapter 4: Solution Validation

This chapter validates the solution described in chapter three by covering the statistical analysis performed and describing how equivalence was verified in the benchmark application.

4.1 Statistical Analysis Results

To validate the benchmark results statistical analysis was performed using a two population mean hypothesis test at the five percent significance level for each LINQ provider. The benchmark application's source code can be found in Appendix A. The benchmark results recorded by the application are reproduced in Appendix B.

For reference, the hypothesis is restated: "Traditional coding techniques perform better than LINQ equivalent code." In terms of performance, "perform better" indicates that less time is taken to execute traditional code. Therefore, when setting up the hypothesis test, the mean for traditional code (μ_1) is expected to be less than the mean of LINQ equivalent code (μ_2): $\mu_1 < \mu_2$. Thus, a left-tailed two population mean hypothesis test will be conducted.

The two population mean hypothesis test assumes unknown standard deviations and uses sample population variances given a large population size ($n \geq 30$) [14]. The test statistic is expressed by the following formula where $\overline{x_n}$ is the mean and s_n^2 is the sample standard deviation of the n^{th} population, while n_x is the sample size of the x^{th} population:

$$z = \frac{\overline{x_1} - \overline{x_2}}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

4.2 LINQ to Objects

The components of the LINQ to Objects hypothesis testing and benchmark results are shown in table 4.1.

	Traditional	LINQ to Objects
Sample Size	416	416
Sample Mean	439.13	546.82
Sample Standard Deviation	724.60	1263.92

Table 4.1 – LINQ to Objects benchmark results.

Based on these details n_1 = 416, n_2 = 416, $\overline{x_1}$ = 439.13, $\overline{x_2}$ = 546.82, s_1 = 724.60, and s_2 = 1263.92. Thus, the hypothesis test is computed as follows:

H_0: $\mu_1 \geq \mu_2$

H_1: $\mu_1 < \mu_2$

$$z = \frac{\overline{x_1} - \overline{x_2}}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} = \frac{439.13 - 546.82}{\sqrt{\frac{724.60^2}{416} + \frac{1263.92^2}{416}}} = -1.508$$

Decision rule: for a significance level of $\alpha = 0.05$, reject the null hypothesis if the calculated test statistic z = -1.508 < -z_α = -1.645.

Conclusion: since -1.508 > -1.645, fail to reject the null hypothesis. There is insufficient sample evidence to claim that traditional code performs better than LINQ to Objects equivalent code.

4.3 LINQ to SQL

The LINQ to SQL benchmarks covered two setups: (1) regular LINQ to SQL queries, and (2) optimized LINQ to SQL queries.

4.3.1 Regular LINQ to SQL Benchmarks

The components of the regular LINQ to SQL hypothesis testing and benchmark results are shown in table 4.2.

	Traditional	**LINQ to SQL**
Sample Size	176	176
Sample Mean	107.72	4,182.74
Sample Standard Deviation	106.61	12,124.57

Table 4.2 – Regular LINQ to SQL benchmark results.

Based on these details n_1 = 176, n_2 = 176, $\overline{x_1}$ = 107.72, $\overline{x_2}$ = 4,182.74, s_1 = 106.61, and s_2 = 12,124.57. Thus, the hypothesis test is computed as follows:

H_0: $\mu_1 \geq \mu_2$

H_1: $\mu_1 < \mu_2$

$$z = \frac{\overline{x_1} - \overline{x_2}}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} = \frac{107.72 - 4182.74}{\sqrt{\frac{106.61^2}{176} + \frac{12124.57^2}{176}}} = -4.459$$

Decision rule: for a significance level of $\alpha = 0.05$, reject the null hypothesis if the calculated test statistic $z = -4.459 < -z_\alpha = -1.645$.

Conclusion: since -4.459 < -1.645, reject the null hypothesis. At the 5 percent significance level we conclude that traditional code performs better than the LINQ to SQL code. Stated differently, the average time for traditional code is less than the average time for LINQ to SQL code.

4.3.2 Optimized LINQ to SQL Benchmarks

The components of the optimized LINQ to SQL hypothesis testing and benchmark results are shown in table 4.3.

	Traditional	**Optimized LINQ to SQL**
Sample Size	176	176
Sample Mean	107.72	261.63
Sample Standard Deviation	106.61	371.07

Table 4.3 – Optimized LINQ to SQL benchmark results.

Based on these details n_1 = 176, n_2 = 176, $\overline{x_1}$ = 107.72, $\overline{x_2}$ = 261.63, s_1 = 106.61, and s_2 = 371.07. Thus, the hypothesis test is computed as follows:

H_0: $\mu_1 \geq \mu_2$

H_1: $\mu_1 < \mu_2$

$$z = \frac{\overline{x_1} - \overline{x_2}}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} = \frac{107.72 - 261.63}{\sqrt{\frac{106.61^2}{176} + \frac{371.07^2}{176}}} = -5.288$$

Decision rule: for a significance level of $\alpha = 0.05$, reject the null hypothesis if the calculated test statistic $z = -5.288 < -z_\alpha = -1.645$.

Conclusion: since -5.288 < -1.645, reject the null hypothesis. At the 5 percent significance level we conclude that traditional code performs better than the optimized LINQ to SQL code. That is, the average time for traditional code is less than the average time for optimized LINQ to SQL code.

4.3.3 LINQ versus Optimized LINQ

Although traditional code was found to perform better than both optimized and non-optimized LINQ to SQL, it is important to realize the benefits of optimizing LINQ to SQL for certain tasks. In order to gauge the worthiness of optimizing queries, an additional statistical comparison of regular queries versus optimized queries was computed. For this particular analysis, the hypothesis is: “Optimized queries perform better than non-optimized queries with regards to the LINQ to SQL provider.”

The components of the optimized and regular LINQ to SQL hypothesis testing and benchmark results are shown in table 4.4.

	Optimized LINQ to SQL	**Regular LINQ to SQL**
Sample Size	176	176
Sample Mean	261.63	4,182.74
Sample Standard Deviation	371.07	12,124.57

Table 4.4 – Optimized and Regular LINQ to SQL benchmark results.

Based on these details n_1 = 176, n_2 = 176, $\overline{x_1}$ = 261.63, $\overline{x_2}$ = 4,182.74, s_1 = 371.07, and s_2 = 12,124.57. Thus, the hypothesis test is computed as follows:

H_0: $\mu_1 \geq \mu_2$

H_1: $\mu_1 < \mu_2$

$$z = \frac{\overline{x_1} - \overline{x_2}}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} = \frac{261.63 - 4182.74}{\sqrt{\frac{371.07^2}{176} + \frac{12124.57^2}{176}}} = -4.288$$

Decision rule: for a significance level of $\alpha = 0.05$, reject the null hypothesis if the calculated test statistic $z = -4.288 < -z_\alpha = -1.645$.

Conclusion: since -4.288 < -1.645, reject the null hypothesis. At the 5 percent significance level we conclude that optimized LINQ to SQL code performs better than regular LINQ to SQL code. Thus, the average time for optimized LINQ to SQL code is less than the average time for regular LINQ to SQL code.

4.4 LINQ to XML

The components of the LINQ to XML hypothesis testing and benchmark results are shown in table 4.5.

	Traditional	**LINQ to XML**
Sample Size	208	208
Sample Mean	60.90	49.04
Sample Standard Deviation	106.05	103.37

Table 4.5 – LINQ to XML benchmark results.

Based on these details n_1 = 208, n_2 = 208, $\overline{x_1}$ = 60.90, $\overline{x_2}$ = 49.04, s_1 = 106.05, and s_2 = 103.37. Thus, the hypothesis test is computed as follows:

H_0: $\mu_1 \geq \mu_2$

H_1: $\mu_1 < \mu_2$

$$z = \frac{\overline{x_1} - \overline{x_2}}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} = \frac{60.90 - 49.04}{\sqrt{\frac{106.05^2}{208} + \frac{103.37^2}{208}}} = 1.155$$

Decision rule: for a significance level of α = 0.05, reject the null hypothesis if the calculated test statistic z = 1.155 < $-z_{\alpha}$ = -1.645.

Conclusion: since 1.155 > -1.645, fail to reject the null hypothesis. There is insufficient sample evidence to claim that traditional code performs better than LINQ to XML equivalent code.

4.5 Benchmark Equivalence Verification Approach

To verify the equivalence of the traditional and LINQ approaches all benchmarked operations were compared to ensure that the results were identical. The verification process is part of the benchmarked code (Appendix A) using a *BenchmarkNameVerifyEquivalence* method naming format. In fact, the *VerifyEquivalence* method was the original starting point for both approaches, which, once verified, were then placed into their own respective methods along with the benchmark timer and loop setup.

Most results are in the form of lists or objects that implement *IEnumerable<T>*. For such scenarios the verification process usually involved a call to the *Enumerable.SequenceEqual* method to verify that both sequences are identical by its Boolean return value. Other benchmarks, such as sorting, may have used a manual comparison of items with a Boolean flag that would be set to *false* if values were incorrect. The Boolean values of both approaches would need to be *true* and equal to each other to ascertain equivalence.

To conduct proper statistical analysis the benchmarks used randomized data samples. However, to verify equivalence, one randomized data sample was retrieved then used by both approaches to ensure the results matched by operating on the same set of data. Once equivalence was verified, the approaches were broken into their respective methods which retrieved randomized data on each iteration of the benchmark loop.

To illustrate, consider the *TakeWhileVerifyEquivalence* method. This benchmark test partitions the data by collecting people with first names that are longer than five characters in length. As soon as a person that does not meet this criterion is encountered in the source collection, the *TakeWhile* operation halts processing the collection and returns the matches found. The method is reproduced in figure 4.1.

```
/// <summary>
/// TakeWhile equivalence verification.
/// </summary>
private static void TakeWhileVerifyEquivalence()
{
   var list = Service.GetPeopleRandom();

   // Traditional
   var traditionalList = new List<Person>();
   foreach (var person in list)
   {
      if (person.FirstName.Length > 5)
      {
         traditionalList.Add(person);
      }
      else
      {
         break;
      }
   }

   // LINQ
   var linqList = list.TakeWhile(person => person.FirstName.Length > 5).ToList();

   // Verification
   Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList));
}
```

Figure 4.1 – The method used to verify equivalence of the *TakeWhile* benchmark.

First, the random collection of people is retrieved and stored in the *list* variable. Next, the traditional code is implemented, followed by the LINQ equivalent code. Finally, verification is performed and the results are logged. This order is used for all verification methods.

The *Common.LogVerification* method takes the method's name via reflection, in this case "TakeWhileVerifyEquivalence," and the Boolean result returned by the

Enumerable.SequenceEqual method. An example of the logged result is “12:10:27 AM 13. TakeWhileVerifyEquivalence: True.” Final results are available in Appendix B.

Once the approach is verified as accurate, the approaches are placed into their own methods as shown in figure 4.2.

```
/// <summary>
/// TakeWhile - Traditional.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void TakeWhileTraditional(int loops)
{
    Common.LogMethod(MethodBase.GetCurrentMethod().Name);
    for (int i = 1; i <= loops; i++)
    {
        using (var timer = new BenchmarkTimer())
        {
            var list = Service.GetPeopleRandom();
            var result = new List<Person>();
            foreach (var person in list)
            {
                if (person.FirstName.Length > 5)
                    result.Add(person);
                else
                    break;
            }
        }
    }
}

/// <summary>
/// TakeWhile - LINQ.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void TakeWhileLINQ(int loops)
{
    Common.LogMethod(MethodBase.GetCurrentMethod().Name);
    for (int i = 1; i <= loops; i++)
    {
        using (var timer = new BenchmarkTimer())
        {
            var list = Service.GetPeopleRandom();
            var result = list.TakeWhile(person => person.FirstName.Length > 5).ToList();
        }
    }
}
```

Figure 4.2 – The traditional and LINQ implementations of the *TakeWhile* benchmark.

The same lines of code that made up the traditional and LINQ approaches can be seen at the heart of these new methods. Surrounding each method is additional code to log the method's name, followed by a for loop that executes the code to benchmark a number of times based on the *loops* variable. Chapter three details the number of loops used for JIT warmup and actual benchmark testing. Each benchmark is encapsulated within a using block that uses a custom *BenchmarkTimer* object responsible for logging the amount of time taken by the encapsulated code.

Chapter 5: Conclusion and Future Efforts

The addition of LINQ has introduced a functional programming style to C#. Accomplishing a set of tasks can be achieved either traditionally in an imperative style of coding, or via LINQ for a functional style of coding. A hypothesis was stated claiming that traditional coding performed better than LINQ equivalent code. The new language features that make LINQ possible were described and a plan for benchmarking three LINQ providers was detailed.

5.1 Performance Conclusions and Development Considerations

In fulfillment of the goals set forth in chapter three, a series of benchmarks were carried out to measure the performance of traditional code versus the equivalent code using LINQ. Three LINQ providers were used for the basis of the performance comparison, namely LINQ to Objects, LINQ to SQL, and LINQ to XML. Additional comparisons were made for LINQ to SQL to account for performance after identifying and implementing relevant optimizations to the provider.

A summary of the statistical analysis results based on the gathered performance benchmarks is represented in table 5.1. Unless otherwise noted, all benchmark types in table 5.1 are compared against their equivalent traditional approaches as described in chapter two.

Benchmark Type	**Hypothesis Accepted / Rejected**	**Conclusion**
LINQ to Objects	Accepted	Traditional code is comparable in performance to LINQ to Objects.
LINQ to SQL (Non-optimized)	Rejected	Traditional code outperforms non-optimized LINQ to SQL.
LINQ to SQL (Optimized)	Rejected	Traditional code outperforms optimized LINQ to SQL.
LINQ to SQL: (Optimized versus non-optimized)	Rejected	Optimized LINQ to SQL outperforms non-optimized LINQ to SQL.
LINQ to XML	Accepted	Traditional code is comparable in performance to LINQ to XML.

Table 5.1 – Summary results of the statistical analysis of the benchmarks performed.

Based on the benchmark results it is clear that the performance differences in LINQ to Objects and LINQ to XML are not adequately significant to prevent developers from opting to use them as an alternative to – or in conjunction with – traditional code. An alluring quality of LINQ is the expressiveness of the API that keeps the syntax similar across providers. This similarity offers a strongly typed querying approach that may be easier to pick up than knowing how to write XPath queries for XML interaction or SQL queries for database usage. Also, at this time, XPath 2.0 is not supported in .NET which prevents simpler XPath functions from being used. For such situations LINQ makes it capable to achieve the desired results. Furthermore, LINQ keeps data source specific language such as XSL and SQL separate from the code.

With regards to LINQ to SQL, there exists a significant difference in performance that developers should recognize prior to favoring it over traditional approaches. Despite its shortcomings it offers attractive rapid application development (RAD) benefits with its ORM capabilities and strongly typed class generation. Optimizations are crucial in boosting performance, as evidenced by the results of the optimized and non-optimized LINQ to SQL benchmarks. In particular, the data modification statements are where LINQ to SQL loses its luster. Nonetheless, an organization that wishes to use it can work around these shortcomings by mixing and matching queries by using stored procedures. Although Microsoft supports LINQ to SQL, its recommendation for future usage of LINQ over relational databases is to use the Entity Framework [2].

Awareness of the underlying technology is crucial to deciding when to use a particular library or platform. The extra layer of abstraction that enhances the development experience usually comes at the expense of less performance. Depending on the application this could be acceptable. Organizations tend to favor development productivity to benefit from a quicker time to market and RAD instead of adding complexity to development that enhances performance and takes longer to be released [1]. Ultimately development efforts should not stagnate due to preoccupations with performance, nor should it adversely affect code clarity. During development optimal performance should be kept in mind, yet it should not be done at the expense of available time by implementing micro-optimizations that may yield no performance gains. Furthermore, critical areas should be identified as candidates for optimization with the help of performance profilers. As Donald Knuth has stated, “we should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil” [15].

5.2 LINQ to SQL Performance in Real World Applications

Although the performance is less than traditional approaches, LINQ to SQL has been used in production applications with favorable results. As an example, StackOverflow.com is a popular programming question and answer website that uses LINQ to SQL [8]. The website receives in the hundreds of thousands of page views per day and has reached a million page views in the past [7]. Launched in the later part of 2008, it has achieved a worldwide Alexa traffic rank of 546 as of March 4th, 2010, earning it a spot in the top one thousand websites visited [5]. StackOverflow.com offers a positive case study of LINQ to SQL being successful in a real world application.

5.3 Future Efforts

While researching and utilizing LINQ a number of tangential topics offer further opportunities for complementary investigation. Interest in these topics could spawn additional research and are recommended as areas that are similar to the underlying motivation of this thesis. Specifically, this thesis focused on “the impact of syntactic sugar and language integrated query” on performance. The work can be extended by investigating the impact on (1) performance via parallel programming, (2) readability and (3) performance of ADO.NET and LINQ to SQL compared to the Entity Framework.

5.3.1 Parallel Programming with PLINQ

The ubiquitousness of multi-core processors is increasing and hardware advances are continually appearing in mobile devices. The benefit of multiple cores is lost when applications are developed in a single threaded fashion. Moreover, LINQ operates on a single processor, which is slower than utilizing the available processors on a machine. Future development will likely focus on parallel programming to increase performance, especially for computationally intensive code, however parallel programming is a complicated practice.

To address these issues Parallel Language Integrated Query (PLINQ) will be available in the .NET 4.0 framework. PLINQ works with LINQ to Objects and LINQ to XML queries and automatically makes use of available multiple processors or cores. A benefit of PLINQ is that it does not require a developer to have extensive knowledge in handling concurrency since locking and threading are handled by the Parallel Framework (PFX) which includes PLINQ [11].

Normally a developer would need to partition their code into small chunks, execute the chunks in parallel threads, and then collate the results. PLINQ frees a developer from performing these complicated steps [3]. Furthermore, as the amount of data increases the solution will scale automatically due to the parallelized processing [11].

Determining the performance gains capable with PLINQ presents a potential research topic that is closely related to this thesis. The challenge will be in determining the types of scenarios that would benefit from being parallelized, specifically computational intensive areas of code, then setting up appropriate benchmarks. Conversely, scenarios where performance deteriorates when parallel programming is applied can be documented. Utilizing PLINQ is not complicated since parallel versions of the familiar LINQ methods are provided. There are some scenarios where parallelization is not possible. An example of a regular LINQ query and its PLINQ version are shown in figure 5.1. The only difference is the usage of the *AsParallel* method.

```
var numbers = Enumerable.Range(1, 10000); // 1 to 10000

// regular LINQ: get even numbers
int[] linqResult = numbers.Where(n => n % 2 == 0).ToArray();

// PLINQ: get even numbers
int[] plinqResult = numbers.AsParallel().Where(n => n % 2 == 0).ToArray();
```

Figure 5.1 – Code snippet demonstrating LINQ and PLINQ queries.

5.3.2 Readability

Determining the impact on readability is a subjective topic and does not present a quantifiable unit of measure. LINQ involves a shift in the way problems are expressed. LINQ queries describe what the desired result is rather than how the result is reached as is the case with the imperative style of coding. This key difference in coding style and the ability to pipeline (or chain) query methods with extension methods provides a composable style that is expressive and readable [9]. Flexibility is gained by the ability to write queries using either the dot notation or the query expression syntax, as well as mixing them.

Research in readability might focus on the conciseness of the code using LINQ and lambda expressions. A survey can be conducted to gather feedback on predefined tasks with solutions provided using traditional code and LINQ equivalent. Survey questions can focus on the ease of understanding the code, the verbosity involved in either approach, the length of the solution and the ability to follow the code in a linear manner. Survey participants could also develop simple solutions using LINQ and traditional code. The time taken to achieve the results, the number of mistakes and the clarity of their code (when presented to them at a later date) can be recorded. Additionally, a metric to measure the length of code can be used to compare solutions using both coding styles. Lines of code might not be ideal depending on how a line or statement is defined in the metric since LINQ can span multiple lines and still be considered one statement until a semicolon is reached. Conversely, logical operators contribute to the lines of code counted in the imperative style. This could also be compared against the feedback from participants since shorter code is not necessarily better in terms of readability.

5.3.3 Benchmarking the Entity Framework

Benchmarking the Entity Framework (EF) would serve to round out the database related comparisons. EF supports the use of other databases and the ability to use custom classes that do not resemble the database schema. For consistency SQL Server would need to be used since LINQ to SQL works exclusively with SQL Server [12].

Bibliography

[1] .NET Rocks! “Catching up with Juval Löwy.” http://www.dotnetrocks.com/text/0520/index.html. January 28, 2010.

[2] ADO.NET Team Blog. “Update on LINQ to SQL and LINQ to Entities Roadmap.” http://blogs.msdn.com/adonet/archive/2008/10/29/update-on-linq-to-sql-and-linq-to-entities-roadmap.aspx. September 7, 2009.

[3] Albahari, J. and Albahari, B. *C# 4.0 in a Nutshell*. Sebastopol, CA: O’Reilly, 2010.

[4] Albahari, J. LINQPad. http://www.linqpad.net/. September 1, 2009.

[5] Alexa: The Web Information Company. “stackoverflow.com.” http://www.alexa.com/siteinfo/stackoverflow.com. March 4, 2010.

[6] Aney, T. “Batch Updates and Deletes with LINQ to SQL.” http://www.aneyfamily.com/terryandann/post/2008/04/Batch-Updates-and-Deletes-with-LINQ-to-SQL.aspx. October 7, 2009.

[7] Atwood, J. “One Million Pageviews.” http://blog.stackoverflow.com/2009/09/one-million-pageviews/. September 30, 2009.

[8] Atwood, J. “What Was Stack Overflow Built With?” http://blog.stackoverflow.com/2008/09/what-was-stack-overflow-built-with/. September 21, 2008.

[9] Calvert, C. and Hejlsberg, A. “Anders Hejlsberg on LINQ and Functional Programming.” http://blogs.msdn.com/charlie/archive/2007/01/26/anders-hejlsberg-on-linq-and-functional-programming.aspx. January 26, 2007.

[10] Cwalina, K. and Abrams, B. *Framework Design Guidelines*. 2nd ed. Crawfordsville, IN: Addison Wesley, 2009.

[11] Duffy, J. and Essey, E. “Parallel LINQ: Running Queries on Multi-Core Processors.” http://msdn.microsoft.com/en-us/magazine/cc163329.aspx. *MSDN Magazine*, October 2007.

[12] Flasko, E. and Microsoft Corporation. “Introducing LINQ to Relational Data.” http://msdn.microsoft.com/en-us/library/cc161164.aspx. February 28, 2010.

[13] Howe, D. “Syntactic sugar.” http://dictionary.reference.com/browse/syntactic sugar. August 30, 2009.

[14] Jaisingh, L. *Statistics for the Utterly Confused*. 2nd ed. New York, NY: McGraw-Hill, 2005.

[15] Knuth, D. Structured Programming with go to Statements. *Computing Surveys*, Vol. 6, No. 4, December 1974, pp. 261-301.

[16] Marguerie, F., Eichert, S., and Wooley, J. *LINQ in Action*. Greenwich, CT: Manning Publications Co., 2008.

[17] Mariani, Rico. “DLinq (Linq to SQL) Performance (Part 5).” http://blogs.msdn.com/ricom/archive/2007/07/16/dlinq-linq-to-sql-performance-part-5.aspx. July 16, 2007.

[18] Microsoft MSDN Library. “Building and Debugging (Visual C#).” http://msdn.microsoft.com/en-us/library/ms173083.aspx. September 26, 2009.

[19] Microsoft MSDN Library. "GC.Collect Method (System)." http://msdn.microsoft.com/en-us/library/xe0c2357.aspx. February 14, 2010.
[20] Microsoft MSDN Library. "Lambda Expressions (C# Programming Guide)." http://msdn.microsoft.com/en-us/library/bb397687.aspx. February 15, 2010.
[21] Microsoft MSDN Library. "Microsoft .NET Framework Benchmark Testing Terms." http://msdn.microsoft.com/en-us/library/ms973265.aspx. February 14, 2010.
[22] Microsoft MSDN Library. "Standard Query Operators Overview." http://msdn.microsoft.com/en-us/library/bb397896.aspx. September 26, 2009.
[23] Microsoft MSDN Library. "Using Constructors (C# Programming Guide)." http://msdn.microsoft.com/en-us/library/ms173115.aspx. February 16, 2010.
[24] Microsoft PressPass. "Microsoft Commits to November Release Date for Visual Studio 2008 and the .NET Framework 3.5." http://www.microsoft.com/presspass/press/2007/nov07/11-05TechEdDevelopersPR.mspx. 30 Sep. 2009.
[25] Morrison, V. "Measure Early and Often for Performance, Part 1." http://msdn.microsoft.com/en-us/magazine/cc500596.aspx. Apr. 2008.
[26] Ok, S. "10 Tips to Improve your LINQ to SQL Application Performance." http://www.sidarok.com/web/blog/content/2008/05/02/10-tips-to-improve-your-linq-to-sql-application-performance.html. December 30, 2009.
[27] Richter, J. *CLR via C#*. 2nd. ed. Redmond, WA: Microsoft Press, 2006.
[28] Skeet, J. *C# in Depth*. Greenwich, CT: Manning Publications Co., 2008.
[29] Tiobe Software: Tiobe Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. 22 July 2009.
[30] Troelsen, A. *Pro C# 2005 and the .NET 2.0 Platform*. 3rd. ed. Berkley, CA: Apress, 2005.
[31] Wagner, B. *More Effective C#: 50 Specific Ways to Improve Your C#*. Crawfordsville, IN: Donnelley, 2008.

Appendix A: Source Code

AWDataContext.cs

```
using System;
using System.Data.Linq.Mapping;

namespace ThesisResearch
{
    public partial class AWDataContext
    {
        /// <summary>
        /// Returns a unique identifier used for randomizing results.
        /// </summary>
        [Function(Name = "NEWID", IsComposable = true)]
        public Guid Random()
        {
            throw new NotImplementedException();
        }
    }
}
```

Program.cs

```
using System;
using System.IO;
using System.Xml;

namespace ThesisResearch
{
    class Program
    {
        static void Main(string[] args)
        {
            GenerateRandomXmlData();
            BenchmarkTimer.PurgeResults();
            LinqToObjectsBenchmark.Start();
            LinqToSqlBenchmark.Start();
            LinqToSqlOptimizedBenchmark.Start();
            LinqToXmlBenchmark.Start();

            Console.WriteLine("Benchmark tests complete...");
            Console.ReadKey();
        }

        /// <summary>
        /// Generates random Xml data files from the SQL database.
        /// </summary>
        static void GenerateRandomXmlData()
        {
            for (int i = 1; i <= Common.XmlFilesCount; i++)
            {
                XmlWriterSettings settings = new XmlWriterSettings();
                settings.Indent = true;
                settings.IndentChars = "    ";
                string filename = "persons" + i + ".xml";
                File.Delete(filename);
                using (XmlWriter writer = XmlWriter.Create(filename, settings))
                {
                    writer.WriteStartElement("Persons");
                    foreach (var p in Service.GetPeopleRandom())
                    {
                        writer.WriteStartElement("Person");
                        writer.WriteAttributeString("BusinessEntityID", p.BusinessEntityID.ToString());
                        writer.WriteAttributeString("ModifiedDate", p.ModifiedDate.ToString());
                        writer.WriteElementString("FirstName", p.FirstName);
                        writer.WriteElementString("MiddleName", p.MiddleName);
                        writer.WriteElementString("LastName", p.LastName);
                        writer.WriteElementString("Suffix", p.Suffix);
                        writer.WriteElementString("Rowguid", p.rowguid.ToString());
                        writer.WriteEndElement();
                    }
                    writer.WriteEndElement();
                    writer.Flush();
                }
            }

            Console.WriteLine("Completed XML data generation...");
        }
    }
}
```

Service.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ThesisResearch
{
    public static class Service
    {
        /// <summary>
        /// Sample size of rows to return.
        /// </summary>
        public static readonly int SampleSize = 1000;

        /// <summary>
        /// Gets a random list of people.
        /// </summary>
        /// <returns>Person list.</returns>
        public static List<Person> GetPeopleRandom()
        {
            using (var dc = new AWDataContext())
            {
                var people = dc.Persons.OrderBy(r => dc.Random()).Take(SampleSize).ToList();
                return people;
            }
        }

        /// <summary>
        /// Gets a random list of PersonInfo items.
        /// </summary>
        /// <returns>PersonInfo list.</returns>
        public static List<PersonInfo> GetPersonInfoList()
        {
            using (var dc = new AWDataContext())
            {
                return dc.Persons.Select(p => new PersonInfo(p.BusinessEntityID, p.FirstName, p.LastName))
                                 .OrderBy(r => dc.Random()).Take(SampleSize).ToList();
            }
        }

        /// <summary>
        /// Gets a random list of first and last names.
        /// </summary>
        /// <returns>String list.</returns>
        public static List<string> GetPeopleFirstNames()
        {
            using (var dc = new AWDataContext())
            {
                return dc.Persons.Select(p => p.FirstName).OrderBy(r =>
dc.Random()).Take(SampleSize).ToList();
            }
        }

        /// <summary>
        /// Gets a random list of modified dates from Person records.
        /// </summary>
        /// <returns>DateTime list.</returns>
        public static List<DateTime> GetPeopleModifiedDates()
        {
            using (var dc = new AWDataContext())
            {
                return dc.Persons.Select(p => p.ModifiedDate).OrderBy(r =>
dc.Random()).Take(SampleSize).ToList();
            }
        }

        /// <summary>
        /// Gets a list of all Employee records.
        /// </summary>
        /// <returns>Employee list.</returns>
        public static List<Employee> GetEmployees()
        {
            using (var dc = new AWDataContext())
            {
                return dc.Employees.ToList();
            }
        }

        /// <summary>
        /// Gets a list of all EmployeeDepartmentHistories records.
        /// </summary>
        /// <returns>EmployeeDepartmentHistory list.</returns>
        public static List<EmployeeDepartmentHistory> GetEmployeeDepartmentHistories()
        {
            using (var dc = new AWDataContext())
            {
                return dc.EmployeeDepartmentHistories.ToList();
            }
        }

        /// <summary>
        /// Gets a list of all departments.
        /// </summary>
        /// <returns>Department list.</returns>
        public static List<Department> GetDepartments()
        {
            using (var dc = new AWDataContext())
            {
                return dc.Departments.ToList();
            }
        }

        /// <summary>
        /// Returns a name based on the loopindex being even or odd.
        /// </summary>
        /// <param name="loopIndex">The current loop index.</param>
        /// <returns>string</returns>
        public static string GetFirstName(int loopIndex)
        {
            return (loopIndex % 2 == 0) ? "Ahmad" : "Emad";
        }

        /// <summary>
        /// Cleans existing records and inserts records that will be deleted by Delete functions.
        /// </summary>
        public static void PrepareDataForDeletion()
        {
            using (var dc = new AWPeopleDataContext())
            {
                var peopleToDelete = dc.Peoples.Where(p => p.LastName == "Mageed");
                dc.Peoples.DeleteAllOnSubmit(peopleToDelete);
                dc.SubmitChanges();

                for (int i = 1; i <= Common.InsertCount; i++)
                {
                    People person = new People();
                    person.Title = "Mr.";
                    person.FirstName = GetFirstName(i);
                    person.MiddleName = i.ToString();
                    person.LastName = "Mageed";
                    person.rowguid = Guid.NewGuid();
                    person.ModifiedDate = DateTime.Now;

                    dc.Peoples.InsertOnSubmit(person);
                    dc.SubmitChanges();
                }
            }
        }
    }
}
```

BenchmarkTimer.cs

```
using System;
using System.Diagnostics;
using System.IO;
using System.Threading;

namespace ThesisResearch
{
    public class BenchmarkTimer : IDisposable
    {
        /// <summary>
        /// The file to store benchmark results.
        /// </summary>
        public static readonly string BenchmarkFilename = "BenchmarkResults.txt";

        /// <summary>
        /// The I/O stream for the benchmark file.
        /// </summary>
        private FileStream benchmarkFile = new FileStream(BenchmarkFilename, FileMode.Append);
        /// <summary>
        /// The stopwatch object.
        /// </summary>
        private Stopwatch stopwatch { get; set; }

        /// <summary>
        /// The constructor for the BenchmarkTimer class.
        /// </summary>
        public BenchmarkTimer()
        {
            Thread.Sleep(3000);
            TextWriterTraceListener myTextListener = new TextWriterTraceListener(benchmarkFile);
            Trace.Listeners.Add(myTextListener);
            stopwatch = Stopwatch.StartNew();
        }

        /// <summary>
        /// Deletes the benchmark results file.
        /// </summary>
        public static void PurgeResults()
        {
            File.Delete(BenchmarkFilename);
        }

        #region IDisposable Members

        /// <summary>
        /// Disposes of the BenchmarkTimer class by recording the stopwatch results to the results file and
cleaning up resources.
        /// </summary>
        public void Dispose()
        {
            stopwatch.Stop();
            Trace.WriteLine(stopwatch.ElapsedMilliseconds);
            Trace.Unindent();
            Trace.Flush();
            Trace.Close();
            benchmarkFile.Close();
        }

        #endregion
    }
}
```

Common.cs

```
using System;
using System.IO;

namespace ThesisResearch
{
    public static class Common
    {
        /// <summary>
        /// Counter used to number benchmark tests conducted.
        /// </summary>
        private static int counter = 1;

        /// <summary>
        /// The number of database INSERT statements to perform.
        /// </summary>
        public static readonly int InsertCount = 500;

        /// <summary>
        /// The number of generated XML files.
        /// </summary>
        public static readonly int XmlFilesCount = 50;

        /// <summary>
        /// The number of JIT warmup repetitions to run.
        /// </summary>
        public static readonly int JitWarmupRepetitions = 5;

        /// <summary>
        /// The total number of benchmark repititions to run.
        /// </summary>
        public static readonly int Repetitions = 16;

        /// <summary>
        /// Random generated seeded with repetitions for reproducibility.
        /// </summary>
        public static readonly Random Rand = new Random(Repetitions);

        /// <summary>
        /// Resets the counter to start from one.
        /// </summary>
        public static void ResetCounter()
        {
            counter = 1;
        }

        /// <summary>
        /// Logs verification information for the benchmark results.
        /// </summary>
        /// <param name="caption">The caption of the verification test performed.</param>
        /// <param name="result">The result of the verification.</param>
        public static void LogVerification(string caption, bool result)
        {
            string text = String.Format("{0}{1} {2}. {3}: {4}", Environment.NewLine,
DateTime.Now.ToLongTimeString(),
                                                counter++, caption, result);
            using (var sw = new StreamWriter(BenchmarkTimer.BenchmarkFilename, true))
            {
                sw.WriteLine(text);
            }
            Console.WriteLine(text);
        }

        /// <summary>
        /// Logs a benchmark method's name for the benchmark results.
        /// </summary>
        /// <param name="methodName"></param>
        public static void LogMethod(string methodName)
        {
            using (var sw = new StreamWriter(BenchmarkTimer.BenchmarkFilename, true))
            {
                sw.WriteLine(DateTime.Now.ToLongTimeString() + " " + methodName);
            }
            GC.Collect();
        }
    }
}
```

DateTimeComparer.cs

```
using System;
using System.Collections;

namespace ThesisResearch
{
    public class DateTimeComparer : IComparer
    {
        /// <summary>
        /// Comparer for DateTime objects.
        /// </summary>
        /// <param name="x">The first DateTime object to compare.</param>
        /// <param name="y">The second DateTime object to compare.</param>
        /// <returns>Integer indicating whether the first DateTime is earlier than, same as, or later than the
second DateTime.</returns>
        public int Compare(object x, object y)
        {
            DateTime xDate = DateTime.Parse(x.ToString());
            DateTime yDate = DateTime.Parse(y.ToString());
            return xDate.CompareTo(yDate);
        }
    }
}
```

EmployeeComparer.cs

```
using System;
using System.Collections.Generic;

namespace ThesisResearch
{
    public class EmployeeComparer : IEqualityComparer<EmployeeInfo>, IComparer<EmployeeInfo>
    {
        #region IEqualityComparer<EmployeeInfo> Members

        /// <summary>
        /// Determines whether two EmployeeInfo objects are equal.
        /// </summary>
        /// <param name="x">The first EmployeeInfo object.</param>
        /// <param name="y">The second EmployeeInfo object.</param>
        /// <returns>True if equal, false otherwise.</returns>
        public bool Equals(EmployeeInfo x, EmployeeInfo y)
        {
            // Check whether the compared objects reference the same data.
            if (Object.ReferenceEquals(x, y))
                return true;

            // Check whether any of the compared objects is null.
            if (Object.ReferenceEquals(x, null) || Object.ReferenceEquals(y, null))
                return false;

            // Check whether the employees' properties are equal.
            return x.ID == y.ID;
        }

        /// <summary>
        /// Gets the hashcode value for an EmployeeInfo object.
        /// </summary>
        /// <param name="e">EmployeeInfo object.</param>
        /// <returns>Integer representing the hashcode.</returns>
        public int GetHashCode(EmployeeInfo e)
        {
            // Check whether the object is null.
            if (Object.ReferenceEquals(e, null))
                return 0;

            // Get the hash code for the FirstName field if it is not null.
            int hashFirstName = e.FirstName == null ? 0 : e.FirstName.GetHashCode();

            // Get the hash code for the LastName field if it is not null.
            int hashLastName = e.LastName == null ? 0 : e.LastName.GetHashCode();

            // Calculate the hash code for the employee.
            return hashFirstName ^ hashLastName ^ e.ID.GetHashCode();
        }

        #endregion

        #region IComparer<EmployeeInfo> Members

        /// <summary>
        /// Comparer for EmployeeInfo objects.
        /// </summary>
        /// <param name="x">The first EmployeeInfo object.</param>
        /// <param name="y">The second EmployeeInfo object.</param>
        /// <returns>Integer result indicating the relative values of the EmployeeInfo objects based on
ID.</returns>
        public int Compare(EmployeeInfo x, EmployeeInfo y)
        {
            if (x == null)
            {
                if (y == null)
                {
                    return 0;   // equal
                }
                else
                {
                    // x is null, y is not null, y is greater
                    return -1;
                }
            }
            else
            {
                // x is not null...

                if (y == null)
                {
                    // y is null, x is greater
                    return 1;
                }
                else
                {
                    // y is not null, compare values
                    return x.ID.CompareTo(y.ID);
                }
            }
        }

        #endregion
    }
}
```

EmployeeInfo.cs

```
using System;

namespace ThesisResearch
{
    /// <summary>
    /// EmployeeInfo class.
    /// </summary>
    public class EmployeeInfo
    {
        /// <summary>
        /// The ID of the employee.
        /// </summary>
        public int ID { get; set; }

        /// <summary>
        /// The firstname of the employee.
        /// </summary>
        public string FirstName { get; set; }

        /// <summary>
        /// The lastname of the employee.
        /// </summary>
        public string LastName { get; set; }

        /// <summary>
        /// The department the employee belongs to.
        /// </summary>
        public string Department { get; set; }
        /// <summary>
        /// The gender of the employee.
        /// </summary>
        public char Gender { get; set; }

        /// <summary>
        /// The start date of the employee.
        /// </summary>
        public DateTime StartDate { get; set; }

        /// <summary>
        /// The end date of the employee.
        /// </summary>
        public DateTime? EndDate { get; set; }

        /// <summary>
        /// Empty EmployeeInfo constructor.
        /// </summary>
        public EmployeeInfo() { }

        /// <summary>
        /// EmployeeInfo constructor.
        /// </summary>
        /// <param name="id">The ID of the employee.</param>
        /// <param name="firstName">The firstname of the employee.</param>
        /// <param name="lastName">The lastname of the employee.</param>
        /// <param name="department">The department the employee belongs to.</param>
        /// <param name="gender">The gender of the employee.</param>
        /// <param name="startDate">The start date of the employee.</param>
        /// <param name="endDate">The end date of the employee.</param>
        public EmployeeInfo(int id, string firstName, string lastName,
            string department, char gender, DateTime startDate, DateTime? endDate)
        {
            ID = id;
            FirstName = firstName;
            LastName = lastName;
            Department = department;
            Gender = gender;
            StartDate = startDate;
            EndDate = endDate;
        }
    }
}
```

PersonComparer.cs

```
using System;
using System.Collections.Generic;

namespace ThesisResearch
{
    public class PersonComparer : IEqualityComparer<PersonInfo>, IComparer<PersonInfo>
    {
        #region IEqualityComparer<Person> Members
        /// <summary>
        /// Determines whether two PersonInfo objects are equal.
        /// </summary>
        /// <param name="x">The first PersonInfo object.</param>
        /// <param name="y">The second PersonInfo object.</param>
        /// <returns>True if equal, false otherwise.</returns>
        public bool Equals(PersonInfo x, PersonInfo y)
        {
            // Check whether the compared objects reference the same data.
            if (Object.ReferenceEquals(x, y))
                return true;

            // Check whether any of the compared objects is null.
            if (Object.ReferenceEquals(x, null) || Object.ReferenceEquals(y, null))
                return false;

            // Check whether the persons' properties are equal.
            return x.Id == y.Id;
        }

        /// <summary>
        /// Gets the hashcode value for a PersonInfo object.
        /// </summary>
        /// <param name="p">PersonInfo object.</param>
        /// <returns>Integer representing the hashcode.</returns>
        public int GetHashCode(PersonInfo p)
        {
            // Check whether the object is null.
            if (Object.ReferenceEquals(p, null))
                return 0;

            // Calculate the hash code for the person.
            return p.Id.GetHashCode();
        }
        #endregion

        #region IComparer<Person> Members

        /// <summary>
        /// Comparer for PersonInfo objects.
        /// </summary>
        /// <param name="x">The first PersonInfo object.</param>
        /// <param name="y">The second PersonInfo object.</param>
        /// <returns>Integer result indicating the relative values of the PersonInfo objects based lastname,
firstname, then ID.</returns>
        public int Compare(PersonInfo x, PersonInfo y)
        {
            if (x == null)
            {
                if (y == null)
                {
                    return 0;   // equal
                }
                else
                {
                    // x is null, y is not null, y is greater
                    return -1;
                }
            }
            else
            {
                // x is not null...

                if (y == null)
                {
                    // y is null, x is greater
                    return 1;
                }
                else
                {
                    // y is not null, compare LastNames
                    int retval = x.LastName.CompareTo(y.LastName);

                    if (retval != 0)
                    {
                        return retval; // not equal
                    }
                    else
                    {
                        // LastNames are equal, compare FirstNames
                        int firstNameCompare = x.FirstName.CompareTo(y.FirstName);

                        if (firstNameCompare != 0)
                        {
                            return firstNameCompare; // not equal
                        }

                        // FirstNames equal, compare IDs
                        return x.Id.CompareTo(y.Id);
                    }
                }
            }
        }
        #endregion
    }
}
```

PersonInfo.cs

```
namespace ThesisResearch
{
    public class PersonInfo
    {
        /// <summary>
        /// The ID of the person.
        /// </summary>
        public int Id { get; set; }

        /// <summary>
        /// The firstname of the person.
        /// </summary>
        public string FirstName { get; set; }

        /// <summary>
        /// The lastname of the person.
        /// </summary>
        public string LastName { get; set; }

        /// <summary>
        /// Empty PersonInfo constructor.
        /// </summary>
        public PersonInfo()
        {
        }

        /// <summary>
        /// PersonInfo constructor.
        /// </summary>
        /// <param name="id">The ID of the person.</param>
        /// <param name="firstName">The firstname of the person.</param>
        /// <param name="lastName">The lastname of the person.</param>
        public PersonInfo(int id, string firstName, string lastName)
        {
            Id = id;
            FirstName = firstName;
            LastName = lastName;
        }
    }
}
```

LinqToObjectsBenchmark.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace ThesisResearch
{
    public static class LinqToObjectsBenchmark
    {
        /// <summary>
        /// Starts the Linq to Objects benchmark tests.
        /// </summary>
        public static void Start()
        {
            Console.WriteLine("Started LinqToObjectsBenchmark...");
            RunBenchmarks(false);    // JIT run
            RunBenchmarks(true);     // actual run
        }

        /// <summary>
        /// Runs the benchmarks.
        /// </summary>
        /// <param name="isBenchmark">Indicates whether the run is a real benchmark or a
```

```
warmup.</param>
        private static void RunBenchmarks(bool isBenchmark)
        {
            Common.ResetCounter();
            int loops = isBenchmark ? Common.Repetitions : Common.JitWarmupRepetitions;

            SortVerifyEquivalence();
            SortTraditional(loops);
            SortLINQ(loops);

            SortComplexVerifyEquivalence();
            SortComplexTraditional(loops);
            SortComplexLINQ(loops);

            DistinctVerifyEquivalence();
            DistinctTraditional(loops);
            DistinctLINQ(loops);

            ExceptVerifyEquivalence();
            ExceptTraditional(loops);
            ExceptLINQ(loops);

            WhereVerifyEquivalence();
            WhereTraditional(loops);
            WhereLINQ(loops);

            AllVerifyEquivalence();
            AllTraditional(loops);
            AllLINQ(loops);

            AnyVerifyEquivalence();
            AnyTraditional(loops);
            AnyLINQ(loops);

            SelectVerifyEquivalence();
            SelectTraditional(loops);
            SelectLINQ(loops);

            SelectManyVerifyEquivalence();
            SelectManyTraditional(loops);
            SelectManyLINQ(loops);

            SkipVerifyEquivalence();
            SkipTraditional(loops);
            SkipLINQ(loops);

            SkipWhileVerifyEquivalence();
            SkipWhileTraditional(loops);
            SkipWhileLINQ(loops);

            TakeVerifyEquivalence();
            TakeTraditional(loops);
            TakeLINQ(loops);

            TakeWhileVerifyEquivalence();
            TakeWhileTraditional(loops);
            TakeWhileLINQ(loops);

            MaxVerifyEquivalence();
            MaxTraditional(loops);
            MaxLINQ(loops);

            MinVerifyEquivalence();
            MinTraditional(loops);
            MinLINQ(loops);

            CountVerifyEquivalence();
            CountTraditional(loops);
            CountLINQ(loops);

            AverageVerifyEquivalence();
            AverageTraditional(loops);
            AverageLINQ(loops);

            JoinVerifyEquivalence();
            JoinTraditional(loops);
            JoinLINQ(loops);

            SequenceEqualTrueVerifyEquivalence();
            SequenceEqualTrueTraditional(loops);
            SequenceEqualTrueLINQ(loops);

            SequenceEqualVerifyEquivalence();
            SequenceEqualTraditional(loops);
            SequenceEqualLINQ(loops);

            ElementAtVerifyEquivalence();
            ElementAtTraditional(loops);
            ElementAtLINQ(loops);

            ToArrayVerifyEquivalence();
            ToArrayTraditional(loops);
            ToArrayLINQ(loops);

            ToDictionaryVerifyEquivalence();
            ToDictionaryTraditional(loops);
            ToDictionaryLINQ(loops);

            ToListVerifyEquivalence();
            ToListTraditional(loops);
            ToListLINQ(loops);

            ConcatenateVerifyEquivalence();
            ConcatenateTraditional(loops);
            ConcatenateLINQ(loops);

            AggregateVerifyEquivalence();
            AggregateTraditional(loops);
            AggregateLINQ(loops);

            string text = String.Format("{0}*** {1} complete: {2} iterations ***{0}",
                Environment.NewLine, isBenchmark ? "Benchmark" : "Warmup", loops);
```

```
        using (var sw = new StreamWriter(BenchmarkTimer.BenchmarkFilename, true))
        {
            sw.WriteLine(text);
        }
        Console.WriteLine(text);
    }

    #region Artifact: Sorting

    #region simple sort

    /// <summary>
    /// Sort - Traditional.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void SortTraditional(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                var list = Service.GetPeopleFirstNames();
                list.Sort();
            }
        }
    }

    /// <summary>
    /// Sort - LINQ.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void SortLINQ(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                var list = Service.GetPeopleFirstNames();
                list.OrderBy(person => person).ToList();
            }
        }
    }

    /// <summary>
    /// Sort equivalence verification.
    /// </summary>
    private static void SortVerifyEquivalence()
    {
        var traditional = Service.GetPeopleFirstNames();
        traditional.Sort();
        var linq = Service.GetPeopleFirstNames().OrderBy(person => person).ToList();
        bool isSortedTraditional = true;
        bool isSortedLINQ = true;
        for (int index = 0; index < traditional.Count - 1; index++)
```

```
            {
                if (String.Compare(traditional[index], traditional[index + 1]) > 0)
                {
                    isSortedTraditional = false;
                    break;
                }
            }
            for (int index = 0; index < linq.Count() - 1; index++)
            {
                if (String.Compare(linq[index], linq[index + 1]) > 0)
                {
                    isSortedLINQ = false;
                    break;
                }
            }
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, (isSortedLINQ == true &&
isSortedTraditional == true));
        }
        #endregion

        #region complex sort (LastName, FirstName)

        /// <summary>
        /// Sort Complex (multiple properties) - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void SortComplexTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPersonInfoList();
                    list.Sort(new PersonComparer());
                }
            }
        }

        /// <summary>
        /// Sort Complex (multiple properties) - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void SortComplexLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPersonInfoList();
                    var q = list.OrderBy(person => person.LastName)
                            .ThenBy(person => person.FirstName)
                            .ThenBy(person => person.Id)
                            .ToList();
                }
            }
        }

        /// <summary>
        /// Sort Complex equivalence verification.
        /// </summary>
        private static void SortComplexVerifyEquivalence()
        {
            var list = Service.GetPersonInfoList();
            var traditionalList = new List<PersonInfo>(list);
            traditionalList.Sort(new PersonComparer());
            var linqList = list.OrderBy(person => person.LastName)
                               .ThenBy(person => person.FirstName)
                               .ThenBy(person => person.Id)
                               .ToList();
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList, new PersonComparer()));
        }
        #endregion

        #endregion

        #region Artifact: Set Operations

        /// <summary>
        /// Distinct - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void DistinctTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var hashSet = new HashSet<string>();
                    var list = Service.GetPeopleFirstNames();
                    foreach (string name in list)
                    {
                        hashSet.Add(name);
                    }

                    list.Clear();
                    foreach (string name in hashSet)
                    {
                        list.Add(name);
                    }
                }
            }
        }

        /// <summary>
        /// Distinct - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void DistinctLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleFirstNames();
                    list = list.Distinct().ToList();
                }
            }
        }

        /// <summary>
        /// Distinct equivalence verification.
        /// </summary>
        private static void DistinctVerifyEquivalence()
        {
            var masterList = Service.GetPeopleFirstNames();
            var traditionalList = masterList.ToList();
            var linqList = masterList.ToList();
            var hashSet = new HashSet<string>();
            foreach (string name in traditionalList)
            {
                hashSet.Add(name);
            }

            traditionalList.Clear();
            foreach (string name in hashSet)
            {
                traditionalList.Add(name);
            }

            linqList = linqList.Distinct().ToList();
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList));
        }

        /// <summary>
        /// Except - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ExceptTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleFirstNames();
                    var otherList = Service.GetPeopleFirstNames();
                    var exceptList = new List<string>();
                    var hashSet = new HashSet<string>();
                    foreach (string name in list)
                    {
                        if (!hashSet.Add(name))
                        {
```

```
                        exceptList.Add(name);
                    }
                }

                exceptList.RemoveAll(item => otherList.Contains(item));
            }
        }
    }

    /// <summary>
    /// Except - LINQ.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void ExceptLINQ(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                var list = Service.GetPeopleFirstNames();
                var otherList = Service.GetPeopleFirstNames();
                List<string> exceptList = list.Except(otherList).ToList();
            }
        }
    }

    /// <summary>
    /// Except equivalence verification.
    /// </summary>
    private static void ExceptVerifyEquivalence()
    {
        var list = Service.GetPeopleFirstNames();
        var otherList = Service.GetPeopleFirstNames();
        var traditionalExceptList = new List<string>();
        var hashSet = new HashSet<string>();
        foreach (string name in list)
        {
            if (hashSet.Add(name))
            {
                traditionalExceptList.Add(name);
            }
        }

        traditionalExceptList.RemoveAll(item => otherList.Contains(item));

        List<string> linqExceptList = list.Except(otherList).ToList();
        Common.LogVerification(MethodBase.GetCurrentMethod().Name,
traditionalExceptList.SequenceEqual(linqExceptList));
    }
    #endregion

    #region Artifact: Filtering

    /// <summary>
    /// Where filtering - Traditional.
```

```
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void WhereTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    var filteredList = list.FindAll(person => person.FirstName.StartsWith("C"));
                }
            }
        }

        /// <summary>
        /// Where filtering - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void WhereLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    var filteredList = list.Where(person => person.FirstName.StartsWith("C")).ToList();
                }
            }
        }

        /// <summary>
        /// Where filtering equivalence verification.
        /// </summary>
        private static void WhereVerifyEquivalence()
        {
            var list = Service.GetPeopleRandom();
            var traditionalListFindAll = list.FindAll(person => person.FirstName.StartsWith("C"));
            var linqList = list.Where(person => person.FirstName.StartsWith("C")).ToList();
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalListFindAll));
        }
        #endregion

        #region Artifact: Quantifier Operations

        /// <summary>
        /// All criteria - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void AllTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
```

```
            using (var timer = new BenchmarkTimer())
            {
                var list = Service.GetPeopleRandom();
                bool result = true;
                foreach (Person p in list)
                {
                    if (p.FirstName.Length <= 5)
                    {
                        result = false;
                        break;
                    }
                }
            }
        }
    }

    /// <summary>
    /// All criteria - LINQ.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void AllLINQ(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                var list = Service.GetPeopleRandom();
                bool result = list.All(person => person.FirstName.Length > 5);
            }
        }
    }

    /// <summary>
    /// All criteria equivalence verification.
    /// </summary>
    private static void AllVerifyEquivalence()
    {
        var list = Service.GetPeopleRandom();
        bool linqResult = list.All(person => person.FirstName.Length > 5);
        bool traditionalAll = true;
        foreach (Person p in list)
        {
            if (p.FirstName.Length <= 5)
            {
                traditionalAll = false;
                break;
            }
        }

        Common.LogVerification(MethodBase.GetCurrentMethod().Name, (linqResult == traditionalAll));
    }

    /// <summary>
    /// Any criteria - Traditional.
    /// </summary>
```

```
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void AnyTraditional(int loops)
{
    Common.LogMethod(MethodBase.GetCurrentMethod().Name);
    for (int i = 1; i <= loops; i++)
    {
        using (var timer = new BenchmarkTimer())
        {
            var list = Service.GetPeopleRandom();
            bool result = list.Exists(person => person.FirstName.Length > 5);
        }
    }
}

/// <summary>
/// Any criteria - LINQ.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void AnyLINQ(int loops)
{
    Common.LogMethod(MethodBase.GetCurrentMethod().Name);
    for (int i = 1; i <= loops; i++)
    {
        using (var timer = new BenchmarkTimer())
        {
            var list = Service.GetPeopleRandom();
            bool result = list.Any(person => person.FirstName.Length > 5);
        }
    }
}

/// <summary>
/// Any criteria equivalence verification.
/// </summary>
private static void AnyVerifyEquivalence()
{
    var list = Service.GetPeopleRandom();
    bool linqResult = list.Any(person => person.FirstName.Length > 5);
    bool traditionalAll = list.Exists(person => person.FirstName.Length > 5);
    Common.LogVerification(MethodBase.GetCurrentMethod().Name, (linqResult == traditionalAll));
}
#endregion

#region Artifact: Projection Operations

/// <summary>
/// Select - Traditional.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void SelectTraditional(int loops)
{
    Common.LogMethod(MethodBase.GetCurrentMethod().Name);
    for (int i = 1; i <= loops; i++)
    {
        using (var timer = new BenchmarkTimer())
        {
                List<Person> list = Service.GetPeopleRandom();
                List<PersonInfo> result = new List<PersonInfo>();
                foreach (var person in list)
                {
                    var personInfo = new PersonInfo(person.BusinessEntityID, person.FirstName,
person.LastName);
                    result.Add(personInfo);
                }
            }
        }
    }

    /// <summary>
    /// Select - LINQ.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void SelectLINQ(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                List<Person> list = Service.GetPeopleRandom();
                List<PersonInfo> result = list.Select(person =>
                                                new PersonInfo(person.BusinessEntityID, person.FirstName,
person.LastName))
                                              .ToList();
            }
        }
    }

    /// <summary>
    /// Select equivalence verification.
    /// </summary>
    private static void SelectVerifyEquivalence()
    {
        List<Person> list = Service.GetPeopleRandom();
        List<PersonInfo> traditionalList = new List<PersonInfo>();
        foreach (var person in list)
        {
            var personInfo = new PersonInfo(person.BusinessEntityID, person.FirstName,
person.LastName);
            traditionalList.Add(personInfo);
        }
        List<PersonInfo> linqList = list.Select(person => new PersonInfo(person.BusinessEntityID,
person.FirstName, person.LastName))
                                        .ToList();
        Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList, new PersonComparer()));
    }

    /// <summary>
    /// SelectMany - Traditional.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
```

```
        private static void SelectManyTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<string> list = Service.GetPeopleFirstNames();
                    List<string> result = new List<string>();
                    foreach (var name1 in list)
                    {
                        foreach (var name2 in list)
                        {
                            if (name1.CompareTo(name2) < 0)
                            {
                                string pair = name1 + " & " + name2;
                                result.Add(pair);
                            }
                        }
                    }
                    result.Sort();
                }
            }
        }

        /// <summary>
        /// SelectMany - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void SelectManyLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<string> list = Service.GetPeopleFirstNames();
                    List<string> result = list.SelectMany(name1 => list, (name1, name2) => new { name1, name2
})
                                          .Where(item => item.name1.CompareTo(item.name2) < 0)
                                          .OrderBy(item => item.name1).ThenBy(item => item.name2)
                                          .Select(item => item.name1 + " & " + item.name2)
                                          .ToList();
                }
            }
        }

        /// <summary>
        /// SelectMany equivalence verification.
        /// </summary>
        private static void SelectManyVerifyEquivalence()
        {
            List<string> list = Service.GetPeopleFirstNames();
            List<string> traditionalList = new List<string>();
            foreach (var name1 in list)
            {
```

```
            foreach (var name2 in list)
            {
                if (name1.CompareTo(name2) < 0)
                {
                    string pair = name1 + " & " + name2;
                    traditionalList.Add(pair);
                }
            }
        }
        traditionalList.Sort();

        var linqList = list.SelectMany(name1 => list, (name1, name2) => new { name1, name2 })
             .Where(item => item.name1.CompareTo(item.name2) < 0)
             .Select(item => item.name1 + " & " + item.name2)
             .OrderBy(item => item)
             .ToList();

        Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList));
    }
    #endregion

    #region Artifact: Partitioning Data

    /// <summary>
    /// Skip - Traditional.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void SkipTraditional(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                int skip = 100;
                var list = Service.GetPeopleRandom();
                var result = new List<Person>();
                for (int index = skip; index < list.Count; index++)
                {
                    result.Add(list[index]);
                }
            }
        }
    }

    /// <summary>
    /// Skip - LINQ.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void SkipLINQ(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                int skip = 100;
                var list = Service.GetPeopleRandom();
                var result = list.Skip(skip).ToList();
            }
        }
    }

    /// <summary>
    /// Skip equivalence verification.
    /// </summary>
    private static void SkipVerifyEquivalence()
    {
        int skip = 100;
        var list = Service.GetPeopleRandom();
        var traditionalList = new List<Person>();
        for (int index = skip; index < list.Count; index++)
        {
            traditionalList.Add(list[index]);
        }

        var linqList = list.Skip(skip).ToList();
        Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList));
    }

    /// <summary>
    /// SkipWhile - Traditional.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void SkipWhileTraditional(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                var list = Service.GetPeopleRandom();
                var result = new List<Person>();
                int startIndex = 0;
                for (int index = 0; index < list.Count; index++)
                {
                    Person person = list[index];
                    if (!(person.FirstName.Length > 5))
                    {
                        startIndex = index;
                        break;
                    }
                }
                for (int index = startIndex; index < list.Count; index++)
                {
                    result.Add(list[index]);
                }
            }
        }
    }
```

```
        /// <summary>
        /// SkipWhile - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void SkipWhileLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    var result = list.SkipWhile(person => person.FirstName.Length > 5).ToList();
                }
            }
        }

        /// <summary>
        /// SkipWhile equivalence verification.
        /// </summary>
        private static void SkipWhileVerifyEquivalence()
        {
            var list = Service.GetPeopleRandom();
            int startIndex = 0;
            for (int index = 0; index < list.Count; index++)
            {
                Person person = list[index];
                if (!(person.FirstName.Length > 5))
                {
                    startIndex = index;
                    break;
                }
            }
            var traditionalList = new List<Person>();
            for (int index = startIndex; index < list.Count; index++)
            {
                traditionalList.Add(list[index]);
            }

            var linqList = list.SkipWhile(person => person.FirstName.Length > 5).ToList();
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList));
        }

        /// <summary>
        /// Take - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void TakeTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
```

```
                int take = 100;
                var list = Service.GetPeopleRandom();
                var result = new List<Person>();
                for (int index = 0; index < take; index++)
                {
                    result.Add(list[index]);
                }
            }
        }
    }

    /// <summary>
    /// Take - LINQ.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void TakeLINQ(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                int take = 100;
                var list = Service.GetPeopleRandom();
                var result = list.Take(take).ToList();
            }
        }
    }

    /// <summary>
    /// Take equivalence verification.
    /// </summary>
    private static void TakeVerifyEquivalence()
    {
        int take = 100;
        var list = Service.GetPeopleRandom();
        var traditionalList = new List<Person>();
        for (int index = 0; index < take; index++)
        {
            traditionalList.Add(list[index]);
        }

        var linqList = list.Take(take).ToList();
        Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList));
    }

    /// <summary>
    /// TakeWhile - Traditional.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void TakeWhileTraditional(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                var list = Service.GetPeopleRandom();
                var result = new List<Person>();
                foreach (var person in list)
                {
                    if (person.FirstName.Length > 5)
                    {
                        result.Add(person);
                    }
                    else
                    {
                        break;
                    }
                }
            }
        }
    }

    /// <summary>
    /// TakeWhile - LINQ.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void TakeWhileLINQ(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                var list = Service.GetPeopleRandom();
                var result = list.TakeWhile(person => person.FirstName.Length > 5).ToList();
            }
        }
    }

    /// <summary>
    /// TakeWhile equivalence verification.
    /// </summary>
    private static void TakeWhileVerifyEquivalence()
    {
        var list = Service.GetPeopleRandom();
        var traditionalList = new List<Person>();
        foreach (var person in list)
        {
            if (person.FirstName.Length > 5)
            {
                traditionalList.Add(person);
            }
            else
            {
                break;
            }
        }
        var linqList = list.TakeWhile(person => person.FirstName.Length > 5).ToList();
        Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList));
        }
        #endregion

        #region Artifact: Join Operations

        /// <summary>
        /// Join - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void JoinTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<Person> people = Service.GetPeopleRandom();
                    List<Employee> employees = Service.GetEmployees();
                    List<EmployeeDepartmentHistory> employeeDepartmentHistories =
Service.GetEmployeeDepartmentHistories();
                    List<Department> departments = Service.GetDepartments();
                    List<EmployeeInfo> result = new List<EmployeeInfo>();

                    foreach (Person person in people)
                    {
                        var employee = employees.Find(e => e.BusinessEntityID == person.BusinessEntityID);
                        if (employee == null)
                        {
                            continue;
                        }

                        var history = employeeDepartmentHistories.Find(h => h.BusinessEntityID ==
employee.BusinessEntityID);
                        var department = departments.Find(d => d.DepartmentID == history.DepartmentID);
                        var employeeInfo = new EmployeeInfo(
                                        person.BusinessEntityID,
                                        person.FirstName,
                                        person.LastName,
                                        department.Name,
                                        employee.Gender,
                                        history.StartDate,
                                        history.EndDate
                                    );
                        result.Add(employeeInfo);
                    }
                }
            }
        }

        /// <summary>
        /// Join - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void JoinLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<Person> people = Service.GetPeopleRandom();
                    List<Employee> employees = Service.GetEmployees();
                    List<EmployeeDepartmentHistory> employeeDepartmentHistories =
Service.GetEmployeeDepartmentHistories();
                    List<Department> departments = Service.GetDepartments();

                    List<EmployeeInfo> result = (from e in employees
                                                 join edh in employeeDepartmentHistories on e.BusinessEntityID equals
edh.BusinessEntityID
                                                 join d in departments on edh.DepartmentID equals d.DepartmentID
                                                 join p in people on e.BusinessEntityID equals p.BusinessEntityID
                                                 select new EmployeeInfo(
                                                     p.BusinessEntityID,
                                                     p.FirstName,
                                                     p.LastName,
                                                     d.Name,
                                                     e.Gender,
                                                     edh.StartDate,
                                                     edh.EndDate
                                                 )).Distinct(new EmployeeComparer()).ToList();
                }
            }
        }

        /// <summary>
        /// Join equivalence verification.
        /// </summary>
        private static void JoinVerifyEquivalence()
        {
            List<Person> people = Service.GetPeopleRandom();
            List<Employee> employees = Service.GetEmployees();
            List<EmployeeDepartmentHistory> employeeDepartmentHistories =
Service.GetEmployeeDepartmentHistories();
            List<Department> departments = Service.GetDepartments();
            List<EmployeeInfo> traditionalList = new List<EmployeeInfo>();
            foreach (Person person in people)
            {
                var employee = employees.Find(e => e.BusinessEntityID == person.BusinessEntityID);
                if (employee == null)
                {
                    continue;
                }

                var history = employeeDepartmentHistories.Find(h => h.BusinessEntityID ==
employee.BusinessEntityID);
                var department = departments.Find(d => d.DepartmentID == history.DepartmentID);
                var employeeInfo = new EmployeeInfo(
                                    person.BusinessEntityID,
                                    person.FirstName,
                                    person.LastName,
                                    department.Name,
                                employee.Gender,
                                history.StartDate,
                                history.EndDate
                            );
                traditionalList.Add(employeeInfo);
            }
            List<EmployeeInfo> linqList = (from e in employees
                                      join edh in employeeDepartmentHistories on e.BusinessEntityID equals
edh.BusinessEntityID
                                      join d in departments on edh.DepartmentID equals d.DepartmentID
                                      join p in people on e.BusinessEntityID equals p.BusinessEntityID
                                      select new EmployeeInfo(
                                          p.BusinessEntityID,
                                          p.FirstName,
                                          p.LastName,
                                          d.Name,
                                          e.Gender,
                                          edh.StartDate,
                                          edh.EndDate
                                      )).Distinct(new EmployeeComparer()).ToList();
            linqList.Sort(new EmployeeComparer());
            traditionalList.Sort(new EmployeeComparer());
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList, new EmployeeComparer()));
        }
        #endregion

        #region Artifact: Equality Operations

        /// <summary>
        /// SequenceEqual (True) - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void SequenceEqualTrueTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    var otherList = new List<Person>(list);
                    bool isEqual = true;
                    for (int index = 0; index < list.Count; index++)
                    {
                        if (!list[index].Equals(otherList[index]))
                        {
                            isEqual = false;
                            break;
                        }
                    }
                }
            }
        }

        /// <summary>
        /// SequenceEqual (True) - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void SequenceEqualTrueLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    var otherList = new List<Person>(list);
                    bool isEqual = list.SequenceEqual(otherList);
                }
            }
        }

        /// <summary>
        /// SequenceEqual (True) equivalence verification.
        /// </summary>
        private static void SequenceEqualTrueVerifyEquivalence()
        {
            var list = Service.GetPeopleRandom();
            var otherList = new List<Person>(list);
            bool traditionalIsEqual = true;
            for (int index = 0; index < list.Count; index++)
            {
                if (!list[index].Equals(otherList[index]))
                {
                    traditionalIsEqual = false;
                    break;
                }
            }
            bool linqIsEqual = list.SequenceEqual(otherList);
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, (traditionalIsEqual ==
linqIsEqual));
        }

        /// <summary>
        /// SequenceEqual - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void SequenceEqualTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    var otherList = Service.GetPeopleRandom();
                    bool isEqual = true;
                    for (int index = 0; index < list.Count; index++)
                    {
                        if (!list[index].Equals(otherList[index]))
                        {
```

```
                    isEqual = false;
                    break;
                }
            }
        }
    }
}

/// <summary>
/// SequenceEqual - LINQ.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void SequenceEqualLINQ(int loops)
{
    Common.LogMethod(MethodBase.GetCurrentMethod().Name);
    for (int i = 1; i <= loops; i++)
    {
        using (var timer = new BenchmarkTimer())
        {
            var list = Service.GetPeopleRandom();
            var otherList = Service.GetPeopleRandom();
            bool isEqual = list.SequenceEqual(otherList);
        }
    }
}

/// <summary>
/// SequenceEqual equivalence verification.
/// </summary>
private static void SequenceEqualVerifyEquivalence()
{
    var list = Service.GetPeopleRandom();
    var otherList = Service.GetPeopleRandom();
    bool traditionalIsEqual = true;
    for (int index = 0; index < list.Count; index++)
    {
        if (!list[index].Equals(otherList[index]))
        {
            traditionalIsEqual = false;
            break;
        }
    }
    bool linqIsEqual = list.SequenceEqual(otherList);
    Common.LogVerification(MethodBase.GetCurrentMethod().Name, (traditionalIsEqual ==
linqIsEqual));
}
#endregion

#region Artifact: Element Operations

/// <summary>
/// ElementAt - Traditional.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void ElementAtTraditional(int loops)
{
```

```
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    Person person = list[Common.Rand.Next(0, Service.SampleSize)];
                }
            }
        }

        /// <summary>
        /// ElementAt - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ElementAtLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    Person person = list.ElementAt(Common.Rand.Next(0, Service.SampleSize));
                }
            }
        }

        /// <summary>
        /// ElementAt equivalence verification.
        /// </summary>
        private static void ElementAtVerifyEquivalence()
        {
            int randomIndex = Common.Rand.Next(0, Service.SampleSize);
            var list = Service.GetPeopleRandom();
            Person traditionalPerson = list[randomIndex];
            Person linqPerson = list.ElementAt(randomIndex);
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqPerson.Equals(traditionalPerson));
        }
        #endregion

        #region Artifact: Conversion

        /// <summary>
        /// ToArray - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ToArrayTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    Person[] array = new Person[list.Count];
                    for (int index = 0; index < list.Count; index++)
                    {
                        array[index] = list[index];
                    }
                }
            }
        }

        /// <summary>
        /// ToArray - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ToArrayLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    Person[] array = list.ToArray();
                }
            }
        }

        /// <summary>
        /// ToArray equivalence verification.
        /// </summary>
        private static void ToArrayVerifyEquivalence()
        {
            var list = Service.GetPeopleRandom();
            Person[] traditionalArray = new Person[list.Count];
            for (int index = 0; index < list.Count; index++)
            {
                traditionalArray[index] = list[index];
            }
            Person[] linqArray = list.ToArray();
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqArray.SequenceEqual(traditionalArray));
        }

        /// <summary>
        /// ToList - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ToListTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = new List<string>();
                    Array.ForEach(Directory.GetFiles(Path.GetTempPath()), list.Add);
                }
            }
        }

        /// <summary>
        /// ToList - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ToListLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Directory.GetFiles(Path.GetTempPath()).ToList();
                }
            }
        }

        /// <summary>
        /// ToList equivalence verification.
        /// </summary>
        private static void ToListVerifyEquivalence()
        {
            string[] tempFiles = Directory.GetFiles(Path.GetTempPath());
            var traditionalList = new List<string>();
            Array.ForEach(tempFiles, traditionalList.Add);
            var linqList = tempFiles.ToList();
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList));
        }

        /// <summary>
        /// ToDictionary - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ToDictionaryTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    Dictionary<string, List<Person>> dictionary = new Dictionary<string, List<Person>>();
                    for (int index = 0; index < list.Count; index++)
                    {
                        Person person = list[index];
                        string lastNameInitial = person.LastName[0].ToString();
                        if (dictionary.ContainsKey(lastNameInitial))
                        {
                            dictionary[lastNameInitial].Add(person);
                        }
                        else
                        {
                            List<Person> personList = new List<Person>();
                            personList.Add(person);
                            dictionary.Add(lastNameInitial, personList);
                        }
                    }
                }
            }
        }

        /// <summary>
        /// ToDictionary - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ToDictionaryLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    var dictionary = list.GroupBy(person => person.LastName[0].ToString())
                                        .ToDictionary(group => group.Key, group => group.ToList());
                }
            }
        }

        /// <summary>
        /// ToDictionary equivalence verification.
        /// </summary>
        private static void ToDictionaryVerifyEquivalence()
        {
            var list = Service.GetPeopleRandom();
            var traditionalDictionary = new Dictionary<string, List<Person>>();
            for (int index = 0; index < list.Count; index++)
            {
                Person person = list[index];
                string lastNameInitial = person.LastName[0].ToString();
                if (traditionalDictionary.ContainsKey(lastNameInitial))
                {
                    traditionalDictionary[lastNameInitial].Add(person);
                }
                else
                {
                    List<Person> personList = new List<Person>();
                    personList.Add(person);
                    traditionalDictionary.Add(lastNameInitial, personList);
                }
            }

            var linqDictionary = list.GroupBy(person => person.LastName[0].ToString())
                                    .ToDictionary(group => group.Key, group => group.ToList());
            bool isEqual = true;
            foreach (var key in linqDictionary.Keys)
            {
                if (traditionalDictionary.ContainsKey(key))
                {
                    var traditionalList = traditionalDictionary[key];
                    var linqList = linqDictionary[key];
                    if (!linqList.All(person => traditionalList.Find(person2 => person.BusinessEntityID ==
    person2.BusinessEntityID) != null))
                    {
                        isEqual = false;
                        break;
                    }
                }
                else
                {
                    isEqual = false;
                    break;
                }
            }
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, isEqual);
        }
        #endregion

        #region Artifact: Concatenation

        /// <summary>
        /// Concatenate - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ConcatenateTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            List<string> concatList = new List<string>();
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    foreach (var person in list)
                        concatList.Add(person.BusinessEntityID.ToString());
                    foreach (var person in list)
                        concatList.Add(person.FirstName);
                    foreach (var person in list)
                        concatList.Add(person.LastName);
                }
            }
        }

        /// <summary>
        /// Concatenate - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ConcatenateLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    var result = list.Select(p => p.BusinessEntityID.ToString())
                                     .Concat(list.Select(p => p.FirstName))
                                     .Concat(list.Select(p => p.LastName))
                                     .ToList();
                }
            }
        }

        /// <summary>
        /// Concatenate equivalence verification.
        /// </summary>
        private static void ConcatenateVerifyEquivalence()
        {
            var list = Service.GetPeopleRandom();
            List<string> traditionalList = new List<string>();
            foreach (var person in list)
                traditionalList.Add(person.BusinessEntityID.ToString());
            foreach (var person in list)
                traditionalList.Add(person.FirstName);
            foreach (var person in list)
                traditionalList.Add(person.LastName);

            var linqList = list.Select(p => p.BusinessEntityID.ToString())
                               .Concat(list.Select(p => p.FirstName))
                               .Concat(list.Select(p => p.LastName))
                               .ToList();
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
    (traditionalList.SequenceEqual(linqList)));
        }
        #endregion

        #region Artifact: Aggregation

        /// <summary>
        /// Max - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void MaxTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleModifiedDates();
                    DateTime currentMax = list[0];
                    DateTime currentMin = list[0];
                    for (int c = 1; c < list.Count; c++)
                    {
                        if (list[c] > currentMax)
                        {
                            currentMax = list[c];
                        }
                        if (list[c] < currentMin)
                        {
                            currentMin = list[c];
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Max - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void MaxLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleModifiedDates();
                    list.Max();
                    list.Min();
                }
            }
        }

        /// <summary>
        /// Max equivalence verification.
        /// </summary>
        private static void MaxVerifyEquivalence()
        {
            var list = Service.GetPeopleModifiedDates();
            DateTime traditionalMax = list[0];
            DateTime traditionalMin = list[0];
            for (int c = 1; c < list.Count; c++)
            {
                if (list[c] > traditionalMax)
                {
                    traditionalMax = list[c];
                }
                else if (list[c] < traditionalMin)
                {
                    traditionalMin = list[c];
                }
            }

            var linqMax = list.Max();
            var linqMin = list.Min();
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, (linqMax == traditionalMax &&
    linqMin == traditionalMin));
        }

        /// <summary>
        /// Min - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void MinTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleModifiedDates();
                    DateTime current = list[0];
                    for (int c = 1; c < list.Count; c++)
                    {
                        if (list[c] < current)
                        {
                            current = list[c];
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Min - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void MinLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleModifiedDates();
                    list.Min();
                }
            }
        }

        /// <summary>
        /// Min equivalence verification.
        /// </summary>
        private static void MinVerifyEquivalence()
        {
            var list = Service.GetPeopleModifiedDates();
            DateTime traditionalMin = list[0];
            for (int c = 1; c < list.Count; c++)
            {
                if (list[c] < traditionalMin)
                {
                    traditionalMin = list[c];
                }
            }

            var linq = list.Min();
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, (linq == traditionalMin));
        }

        /// <summary>
        /// Count - Traditional.
```

```
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void CountTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleModifiedDates();
                    int count = list.Count;
                }
            }
        }

        /// <summary>
        /// Count - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void CountLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleModifiedDates();
                    int count = list.Count();
                }
            }
        }

        /// <summary>
        /// Count equivalence verification.
        /// </summary>
        private static void CountVerifyEquivalence()
        {
            var list = Service.GetPeopleModifiedDates();
            int traditionalCount = list.Count;
            int linqCount = list.Count();
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, (linqCount ==
traditionalCount));
        }

        /// <summary>
        /// Average - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void AverageTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    double sum = 0;
                    foreach (Person person in list)
                        sum += person.BusinessEntityID;

                    double average = sum / list.Count;
                }
            }
        }

        /// <summary>
        /// Average - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void AverageLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    double average = list.Average(person => person.BusinessEntityID);
                }
            }
        }

        /// <summary>
        /// Average equivalence verification.
        /// </summary>
        private static void AverageVerifyEquivalence()
        {
            var list = Service.GetPeopleRandom();
            double traditionalSum = 0;
            foreach (Person person in list)
                traditionalSum += person.BusinessEntityID;
            double average = traditionalSum / list.Count;

            double linqAverage = list.Average(person => person.BusinessEntityID);
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, (linqAverage == average));
        }

        /// <summary>
        /// Aggregate - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void AggregateTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    string names = "";
                    for (int index = 0; index < list.Count; index++)
                    {
                        if (index != list.Count - 1)
                        {
                            names += list[index].FirstName + ", ";
                        }
                        else
                        {
                            names += list[index].FirstName;
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Aggregate - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void AggregateLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var list = Service.GetPeopleRandom();
                    var query = list.Select(p => p.FirstName).Aggregate((current, next) => current + ", " + next);
                }
            }
        }

        /// <summary>
        /// Aggregate equivalence verification.
        /// </summary>
        private static void AggregateVerifyEquivalence()
        {
            var list = Service.GetPeopleRandom();
            string traditionalNames = "";
            for (int index = 0; index < list.Count; index++)
            {
                if (index != list.Count - 1)
                {
                    traditionalNames += list[index].FirstName + ", ";
                }
                else
                {
                    traditionalNames += list[index].FirstName;
                }
            }

            var linqNames = list.Select(p => p.FirstName).Aggregate((current, next) => current + ", " + next);
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, (linqNames ==
    traditionalNames));
        }
        #endregion
      }
    }
```

LinqToSqlBenchmark.cs

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.IO;
using System.Linq;
using System.Reflection;

namespace ThesisResearch
{
    public static class LinqToSqlBenchmark
    {
        /// <summary>
        /// Returns a new SqlConnection using the AdventureWorks2008 database.
        /// </summary>
        private static SqlConnection AdventureWorksConn
        {
            get
            {
                return new SqlConnection(@"Data Source=S03957;Initial
Catalog=AdventureWorks2008;Integrated Security=True");
            }
        }

        /// <summary>
        /// Returns a new SqlConnection using the AdventureWorksPeople database.
        /// </summary>
        private static SqlConnection AdventureWorksPeopleConn
        {
            get
            {
                return new SqlConnection(@"Data Source=S03957;Initial
Catalog=AdventureWorksPeople;Integrated Security=True");
            }
        }

        /// <summary>
        /// Starts the Linq to SQL benchmark tests.
        /// </summary>
        public static void Start()
        {
            Console.WriteLine("Started LinqToSqlBenchmark...");
            RunBenchmarks(false);    // JIT run
            RunBenchmarks(true);     // actual run
        }

        /// <summary>
        /// Runs the benchmarks.
        /// </summary>
        /// <param name="isBenchmark">Indicates whether the run is a real benchmark or a
warmup.</param>
        private static void RunBenchmarks(bool isBenchmark)
        {
            Common.ResetCounter();
            int loops = isBenchmark ? Common.Repetitions : Common.JitWarmupRepetitions;

            SelectVerifyEquivalence();
            SelectTraditional(loops);
            SelectLINQ(loops);

            CountVerifyEquivalence();
            CountTraditional(loops);
            CountLINQ(loops);

            WhereVerifyEquivalence();
            WhereTraditional(loops);
            WhereLINQ(loops);

            WhereNoMatchVerifyEquivalence();
            WhereNoMatchTraditional(loops);
            WhereNoMatchLINQ(loops);

            JoinVerifyEquivalence();
            JoinTraditional(loops);
            JoinLINQ(loops);

            OrderVerifyEquivalence();
            OrderTraditional(loops);
            OrderLINQ(loops);

            GroupVerifyEquivalence();
            GroupTraditional(loops);
            GroupLINQ(loops);

            SumVerifyEquivalence();
            SumTraditional(loops);
            SumLINQ(loops);

            InsertVerifyEquivalence();
            InsertTraditional(loops);
            InsertLINQ(loops);

            UpdateVerifyEquivalence();
            UpdateTraditional(loops);
            UpdateLINQ(loops);

            DeleteVerifyEquivalence();
            DeleteTraditional(loops);
            DeleteLINQ(loops);

            string text = String.Format("{0}*** {1} complete: {2} iterations ***{0}",
                Environment.NewLine, isBenchmark ? "Benchmark" : "Warmup", loops);
            using (var sw = new StreamWriter(BenchmarkTimer.BenchmarkFilename, true))
            {
                sw.WriteLine(text);
            }
            Console.WriteLine(text);
        }
```

```
/// <summary>
/// Select - Traditional.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void SelectTraditional(int loops)
{
   Common.LogMethod(MethodBase.GetCurrentMethod().Name);
   for (int i = 1; i <= loops; i++)
   {
      using (var timer = new BenchmarkTimer())
      {
         List<PersonInfo> traditionalList = new List<PersonInfo>();
         using (SqlConnection conn = AdventureWorksConn)
         {
            string query = @"SELECT TOP 500 BusinessEntityID, FirstName, LastName
                        FROM Person.Person
                        ORDER BY NEWID()";
            conn.Open();
            SqlCommand command = new SqlCommand(query, conn);
            using (SqlDataReader dr = command.ExecuteReader())
            {
               while (dr.Read())
               {
                  PersonInfo person = new PersonInfo
                  {
                     Id = Int32.Parse(dr["BusinessEntityID"].ToString()),
                     FirstName = dr["FirstName"].ToString(),
                     LastName = dr["LastName"].ToString()
                  };
                  traditionalList.Add(person);
               }
            }
         }
      }
   }
}

/// <summary>
/// Select - LINQ.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void SelectLINQ(int loops)
{
   Common.LogMethod(MethodBase.GetCurrentMethod().Name);
   for (int i = 1; i <= loops; i++)
   {
      using (var timer = new BenchmarkTimer())
      {
         List<PersonInfo> linqList = new List<PersonInfo>();
         using (var dc = new AWDataContext())
         {
            var linqQuery = dc.Persons
                          .Select(person => new PersonInfo
                          {
                             Id = person.BusinessEntityID,
                                    FirstName = person.FirstName,
                                    LastName = person.LastName
                                })
                                .OrderBy(item => dc.Random())
                                .Take(500);
                    linqList = linqQuery.ToList();
                }
            }
        }
    }

    /// <summary>
    /// Select equivalence verification.
    /// </summary>
    private static void SelectVerifyEquivalence()
    {
        List<PersonInfo> traditionalList = new List<PersonInfo>();
        using (SqlConnection conn = AdventureWorksConn)
        {
            conn.Open();
            string query = @"SELECT TOP 500 BusinessEntityID, FirstName, LastName
                        FROM Person.Person
                        --ORDER BY NEWID()";
            SqlCommand command = new SqlCommand(query, conn);
            using (SqlDataReader dr = command.ExecuteReader())
            {
                while (dr.Read())
                {
                    PersonInfo person = new PersonInfo
                    {
                        Id = Int32.Parse(dr["BusinessEntityID"].ToString()),
                        FirstName = dr["FirstName"].ToString(),
                        LastName = dr["LastName"].ToString()
                    };
                    traditionalList.Add(person);
                }
            }
        }

        List<PersonInfo> linqList = new List<PersonInfo>();
        using (var dc = new AWDataContext())
        {
            var linqQuery = dc.Persons
                         .Select(person => new PersonInfo
                         {
                             Id = person.BusinessEntityID,
                             FirstName = person.FirstName,
                             LastName = person.LastName
                         })
                        //.OrderBy(item => dc.Random())
                        .Take(500);
            linqList = linqQuery.ToList();
        }

        Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList, new PersonComparer()));
```

```
        }

        /// <summary>
        /// Count - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void CountTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (SqlConnection conn = AdventureWorksConn)
                    {
                        conn.Open();
                        string query = @"SELECT COUNT(*) AS [Results] FROM
                                (SELECT TOP (500) *
                                    FROM [Person].[Person]
                                    ORDER BY NEWID()
                                ) AS [query]";
                        SqlCommand command = new SqlCommand(query, conn);
                        int count = (int)command.ExecuteScalar();
                    }
                }
            }
        }

        /// <summary>
        /// Count - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void CountLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    int linqCount;
                    using (var dc = new AWDataContext())
                    {
                        linqCount = dc.Persons.OrderBy(item => dc.Random()).Take(500).Count();
                    }
                }
            }
        }

        /// <summary>
        /// Count equivalence verification.
        /// </summary>
        private static void CountVerifyEquivalence()
        {
            int traditionalCount;
            using (SqlConnection conn = AdventureWorksConn)
            {
```

```
            conn.Open();
            string query = @"SELECT COUNT(*) AS [Results] FROM
                            (SELECT TOP (500) *
                                FROM [Person].[Person]
                                ORDER BY NEWID()
                            ) AS [query]";
            SqlCommand command = new SqlCommand(query, conn);
            traditionalCount = (int)command.ExecuteScalar();
        }

        int linqCount;
        using (var dc = new AWDataContext())
        {
            linqCount = dc.Persons.OrderBy(item => dc.Random()).Take(500).Count();
        }

        Common.LogVerification(MethodBase.GetCurrentMethod().Name, (traditionalCount ==
linqCount));
    }

    /// <summary>
    /// Where filtering - Traditional.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void WhereTraditional(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                List<PersonInfo> traditionalList = new List<PersonInfo>();
                using (SqlConnection conn = AdventureWorksConn)
                {
                    conn.Open();
                    string query = @"SELECT TOP 500 * FROM Person.Person AS P
                            WHERE P.FirstName LIKE 'A%'
                            ORDER BY NEWID()";
                    SqlCommand command = new SqlCommand(query, conn);
                    using (SqlDataReader dr = command.ExecuteReader())
                    {
                        while (dr.Read())
                        {
                            PersonInfo person = new PersonInfo
                            {
                                Id = Int32.Parse(dr["BusinessEntityID"].ToString()),
                                FirstName = dr["FirstName"].ToString(),
                                LastName = dr["LastName"].ToString()
                            };
                            traditionalList.Add(person);
                        }
                    }
                }
            }
        }
    }
```

```
        /// <summary>
        /// Where filtering - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void WhereLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<PersonInfo> linqList = new List<PersonInfo>();
                    using (var dc = new AWDataContext())
                    {
                        linqList = dc.Persons
                                    .Where(person => person.FirstName.StartsWith("A"))
                                    .Select(person => new PersonInfo(person.BusinessEntityID, person.FirstName,
person.LastName))
                                    .OrderBy(item => dc.Random()).Take(500)
                                    .ToList();
                    }
                }
            }
        }

        /// <summary>
        /// Where filtering equivalence verification.
        /// </summary>
        private static void WhereVerifyEquivalence()
        {
            List<PersonInfo> traditionalList = new List<PersonInfo>();
            using (SqlConnection conn = AdventureWorksConn)
            {
                conn.Open();
                string query = @"SELECT TOP 500 BusinessEntityID, FirstName, LastName
                        FROM Person.Person
                        WHERE FirstName LIKE 'A%'";
                SqlCommand command = new SqlCommand(query, conn);
                using (SqlDataReader dr = command.ExecuteReader())
                {
                    while (dr.Read())
                    {
                        PersonInfo person = new PersonInfo
                        {
                            Id = Int32.Parse(dr["BusinessEntityID"].ToString()),
                            FirstName = dr["FirstName"].ToString(),
                            LastName = dr["LastName"].ToString()
                        };
                        traditionalList.Add(person);
                    }
                }
            }
            traditionalList = traditionalList.OrderBy(p => p.Id).ToList();

            List<PersonInfo> linqList = new List<PersonInfo>();
```

```
          using (var dc = new AWDataContext())
          {
             linqList = dc.Persons
                        .Where(person => person.FirstName.StartsWith("A"))
                        .Select(person => new PersonInfo(person.BusinessEntityID, person.FirstName,
person.LastName))
                        //.OrderBy(item => dc.Random())
                        .Take(500)
                        .ToList();
          }
          linqList = linqList.OrderBy(p => p.Id).ToList();

          Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList, new PersonComparer()));
       }

       /// <summary>
       /// Where filtering (no match) - Traditional.
       /// </summary>
       /// <param name="loops">The number of times to execute the benchmark.</param>
       private static void WhereNoMatchTraditional(int loops)
       {
          Common.LogMethod(MethodBase.GetCurrentMethod().Name);
          for (int i = 1; i <= loops; i++)
          {
             using (var timer = new BenchmarkTimer())
             {
                List<PersonInfo> traditionalList = new List<PersonInfo>();
                using (SqlConnection conn = AdventureWorksConn)
                {
                   conn.Open();
                   string query = @"SELECT TOP 500 * FROM Person.Person AS P
                               WHERE P.FirstName = 'Ahmad'
                               ORDER BY NEWID()";
                   SqlCommand command = new SqlCommand(query, conn);
                   using (SqlDataReader dr = command.ExecuteReader())
                   {
                      while (dr.Read())
                      {
                         PersonInfo person = new PersonInfo
                         {
                            Id = Int32.Parse(dr["BusinessEntityID"].ToString()),
                            FirstName = dr["FirstName"].ToString(),
                            LastName = dr["LastName"].ToString()
                         };
                         traditionalList.Add(person);
                      }
                   }
                }
             }
          }
       }

       /// <summary>
       /// Where filtering (no match) - LINQ.
       /// </summary>
```

```
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void WhereNoMatchLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<PersonInfo> linqList = new List<PersonInfo>();
                    using (var dc = new AWDataContext())
                    {
                        linqList = dc.Persons
                                    .Where(person => person.FirstName == "Ahmad")
                                    .Select(person => new PersonInfo(person.BusinessEntityID, person.FirstName,
person.LastName))
                                    .OrderBy(item => dc.Random()).Take(500)
                                    .ToList();
                    }
                }
            }
        }

        /// <summary>
        /// Where filtering (no match) equivalence verification.
        /// </summary>
        private static void WhereNoMatchVerifyEquivalence()
        {
            List<PersonInfo> traditionalList = new List<PersonInfo>();
            using (SqlConnection conn = AdventureWorksConn)
            {
                conn.Open();
                string query = @"SELECT TOP 500 * FROM Person.Person AS P
                        WHERE P.FirstName = 'Ahmad'
                        --ORDER BY NEWID()";
                SqlCommand command = new SqlCommand(query, conn);
                using (SqlDataReader dr = command.ExecuteReader())
                {
                    while (dr.Read())
                    {
                        PersonInfo person = new PersonInfo
                        {
                            Id = Int32.Parse(dr["BusinessEntityID"].ToString()),
                            FirstName = dr["FirstName"].ToString(),
                            LastName = dr["LastName"].ToString()
                        };
                        traditionalList.Add(person);
                    }
                }
            }
            traditionalList = traditionalList.OrderBy(p => p.Id).ToList();

            List<PersonInfo> linqList = new List<PersonInfo>();
            using (var dc = new AWDataContext())
            {
                linqList = dc.Persons
                            .Where(person => person.FirstName == "Ahmad")
                        .Select(person => new PersonInfo(person.BusinessEntityID, person.FirstName,
person.LastName))
                        //.OrderBy(item => dc.Random())
                        .Take(500)
                        .ToList();
          }
          linqList = linqList.OrderBy(p => p.Id).ToList();

          Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList, new PersonComparer()));
      }

      /// <summary>
      /// Join - Traditional.
      /// </summary>
      /// <param name="loops">The number of times to execute the benchmark.</param>
      private static void JoinTraditional(int loops)
      {
          Common.LogMethod(MethodBase.GetCurrentMethod().Name);
          for (int i = 1; i <= loops; i++)
          {
              using (var timer = new BenchmarkTimer())
              {
                  List<EmployeeInfo> traditionalList = new List<EmployeeInfo>();
                  using (SqlConnection conn = AdventureWorksConn)
                  {
                      conn.Open();
                      string query = @"SELECT TOP 500 P.BusinessEntityID, P.FirstName, P.LastName,
D.Name, E.Gender, EDH.StartDate, EDH.EndDate
                          FROM HumanResources.Employee AS E,
                              HumanResources.EmployeeDepartmentHistory AS EDH,
                              HumanResources.Department AS D,
                              Person.Person AS P
                          WHERE
                              E.BusinessEntityID = EDH.BusinessEntityID
                              AND EDH.DepartmentID = D.DepartmentID
                              AND E.BusinessEntityID = P.BusinessEntityID
                          ORDER BY NEWID()";
                      SqlCommand command = new SqlCommand(query, conn);
                      using (SqlDataReader dr = command.ExecuteReader())
                      {
                          while (dr.Read())
                          {
                              EmployeeInfo employee = new EmployeeInfo(
                                  Int32.Parse(dr["BusinessEntityID"].ToString()),
                                  dr["FirstName"].ToString(),
                                  dr["LastName"].ToString(),
                                  dr["Name"].ToString(),
                                  Char.Parse(dr["Gender"].ToString()),
                                  DateTime.Parse(dr["StartDate"].ToString()),
                                  (dr["EndDate"].ToString() == "" ? null :
(DateTime?)DateTime.Parse(dr["EndDate"].ToString()))
                              );
                              traditionalList.Add(employee);
                          }
                      }
```

```
                }
            }
        }
    }

    /// <summary>
    /// Join - LINQ.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void JoinLINQ(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                List<EmployeeInfo> linqList = new List<EmployeeInfo>();
                using (var dc = new AWDataContext())
                {
                    linqList = (from e in dc.Employees
                                join edh in dc.EmployeeDepartmentHistories
                                    on e.BusinessEntityID equals edh.BusinessEntityID
                                join d in dc.Departments on edh.DepartmentID equals d.DepartmentID
                                join p in dc.Persons on e.BusinessEntityID equals p.BusinessEntityID
                                select new EmployeeInfo(
                                    p.BusinessEntityID,
                                    p.FirstName,
                                    p.LastName,
                                    d.Name,
                                    e.Gender,
                                    edh.StartDate,
                                    edh.EndDate
                                ))
                                .OrderBy(item => dc.Random()).Take(500).ToList();
                }
            }
        }
    }

    /// <summary>
    /// Join equivalence verification.
    /// </summary>
    private static void JoinVerifyEquivalence()
    {
        List<EmployeeInfo> traditionalList = new List<EmployeeInfo>();
        using (SqlConnection conn = AdventureWorksConn)
        {
            conn.Open();
            string query = @"SELECT TOP 500 P.BusinessEntityID, P.FirstName, P.LastName, D.Name,
E.Gender, EDH.StartDate, EDH.EndDate
                            FROM HumanResources.Employee AS E,
                                HumanResources.EmployeeDepartmentHistory AS EDH,
                                HumanResources.Department AS D,
                                Person.Person AS P
                            WHERE
                                E.BusinessEntityID = EDH.BusinessEntityID
```

```
                              AND EDH.DepartmentID = D.DepartmentID
                              AND E.BusinessEntityID = P.BusinessEntityID
                           --ORDER BY NEWID()";
            SqlCommand command = new SqlCommand(query, conn);
            using (SqlDataReader dr = command.ExecuteReader())
            {
               while (dr.Read())
               {
                  EmployeeInfo employee = new EmployeeInfo(
                     Int32.Parse(dr["BusinessEntityID"].ToString()),
                     dr["FirstName"].ToString(),
                     dr["LastName"].ToString(),
                     dr["Name"].ToString(),
                     Char.Parse(dr["Gender"].ToString()),
                     DateTime.Parse(dr["StartDate"].ToString()),
                     (dr["EndDate"].ToString() == "" ? null :
(DateTime?)DateTime.Parse(dr["EndDate"].ToString()))
                  );
                  traditionalList.Add(employee);
               }
            }
         }

         List<EmployeeInfo> linqList = new List<EmployeeInfo>();
         using (var dc = new AWDataContext())
         {
            linqList = (from e in dc.Employees
                     join edh in dc.EmployeeDepartmentHistories
                         on e.BusinessEntityID equals edh.BusinessEntityID
                     join d in dc.Departments on edh.DepartmentID equals d.DepartmentID
                     join p in dc.Persons on e.BusinessEntityID equals p.BusinessEntityID
                     select new EmployeeInfo(
                        p.BusinessEntityID,
                        p.FirstName,
                        p.LastName,
                        d.Name,
                        e.Gender,
                        edh.StartDate,
                        edh.EndDate
                     ))
                     //.OrderBy(item => dc.Random())
                     .Take(500)
                     .ToList();
         }
         Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList, new EmployeeComparer()));
      }

      /// <summary>
      /// Order - Traditional.
      /// </summary>
      /// <param name="loops">The number of times to execute the benchmark.</param>
      private static void OrderTraditional(int loops)
      {
         Common.LogMethod(MethodBase.GetCurrentMethod().Name);
         for (int i = 1; i <= loops; i++)
```

```
        {
            using (var timer = new BenchmarkTimer())
            {
                using (SqlConnection conn = AdventureWorksConn)
                {
                    conn.Open();
                    string query = @"SELECT * FROM
                          (SELECT TOP 500 P.BusinessEntityID, P.FirstName, P.LastName
                                      FROM Person.Person AS P
                                      ORDER BY NEWID()
                              ) AS Result
                        ORDER BY Result.BusinessEntityID";
                    SqlCommand command = new SqlCommand(query, conn);
                    using (SqlDataReader dr = command.ExecuteReader())
                    {
                        List<PersonInfo> traditionalList = new List<PersonInfo>();
                        while (dr.Read())
                        {
                            PersonInfo person = new PersonInfo(
                                Int32.Parse(dr["BusinessEntityID"].ToString()),
                                dr["FirstName"].ToString(),
                                dr["LastName"].ToString()
                            );
                            traditionalList.Add(person);
                        }
                    }
                }
            }
        }
    }

    /// <summary>
    /// Order - LINQ.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void OrderLINQ(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                using (var dc = new AWDataContext())
                {
                    var linqQuery = dc.Persons
                                .Select(person => new PersonInfo
                                {
                                    Id = person.BusinessEntityID,
                                    FirstName = person.FirstName,
                                    LastName = person.LastName
                                })
                                .OrderBy(item => dc.Random())
                                .Take(500);

                    List<PersonInfo> linqList = linqQuery.OrderBy(person => person.Id).ToList();
                }
            }
        }
    }

    /// <summary>
    /// Order equivalence verification.
    /// </summary>
    private static void OrderVerifyEquivalence()
    {
        List<PersonInfo> traditionalList = new List<PersonInfo>();
        using (SqlConnection conn = AdventureWorksConn)
        {
            conn.Open();
            string query = @"SELECT * FROM
                        (SELECT TOP 500 P.BusinessEntityID, P.FirstName, P.LastName
                                    FROM Person.Person AS P
                                    --ORDER BY NEWID()
                              ) AS Result
                      ORDER BY Result.BusinessEntityID";
            SqlCommand command = new SqlCommand(query, conn);
            using (SqlDataReader dr = command.ExecuteReader())
            {
                while (dr.Read())
                {
                    PersonInfo person = new PersonInfo(
                        Int32.Parse(dr["BusinessEntityID"].ToString()),
                        dr["FirstName"].ToString(),
                        dr["LastName"].ToString()
                    );
                    traditionalList.Add(person);
                }
            }
        }

        List<PersonInfo> linqList = new List<PersonInfo>();
        using (var dc = new AWDataContext())
        {
            var linqQuery = dc.Persons
                        .Select(person => new PersonInfo
                        {
                            Id = person.BusinessEntityID,
                            FirstName = person.FirstName,
                            LastName = person.LastName
                        })
                       //.OrderBy(item => dc.Random())
                        .Take(500);

            linqList = linqQuery.OrderBy(person => person.Id).ToList();
        }

        Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqList.SequenceEqual(traditionalList, new PersonComparer()));
    }

    /// <summary>
    /// Group - Traditional.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void GroupTraditional(int loops)
{
   Common.LogMethod(MethodBase.GetCurrentMethod().Name);
   for (int i = 1; i <= loops; i++)
   {
      using (var timer = new BenchmarkTimer())
      {
         Dictionary<string, int> traditionalDictionary = new Dictionary<string, int>();
         using (SqlConnection conn = AdventureWorksConn)
         {
            conn.Open();
            string query = @"SELECT COUNT(*) AS Total, Title
                           FROM (
                                          SELECT TOP 500 Title
                                          FROM Person.Person
                                          ORDER BY NEWID()
                                    ) AS P
                           GROUP BY P.Title";
            SqlCommand command = new SqlCommand(query, conn);
            using (SqlDataReader dr = command.ExecuteReader())
            {
               while (dr.Read())
               {
                  string title = dr["Title"].ToString();
                  int total = Int32.Parse(dr["Total"].ToString());
                  traditionalDictionary.Add(title, total);
               }
            }
         }
      }
   }
}

/// <summary>
/// Group - LINQ.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void GroupLINQ(int loops)
{
   Common.LogMethod(MethodBase.GetCurrentMethod().Name);
   for (int i = 1; i <= loops; i++)
   {
      using (var timer = new BenchmarkTimer())
      {
         using (var dc = new AWDataContext())
         {
            Dictionary<string, int> linqDictionary = dc.Persons
                                         .OrderBy(item => dc.Random())
                                         .Take(500)
                                         .GroupBy(p => p.Title)
                                         .ToDictionary(g => g.Key ?? String.Empty, g => g.Count());
         }
      }
   }
```

```
        }

        /// <summary>
        /// Group equivalence verification.
        /// </summary>
        private static void GroupVerifyEquivalence()
        {
            Dictionary<string, int> traditionalDictionary = new Dictionary<string, int>();
            using (SqlConnection conn = AdventureWorksConn)
            {
                conn.Open();
                string query = @"SELECT COUNT(*) AS Total, Title
                        FROM (
                                    SELECT TOP 500 Title
                                    FROM Person.Person
                                    --ORDER BY NEWID()
                              ) AS P
                        GROUP BY P.Title";
                SqlCommand command = new SqlCommand(query, conn);
                using (SqlDataReader dr = command.ExecuteReader())
                {
                    while (dr.Read())
                    {
                        string title = dr["Title"].ToString();
                        int total = Int32.Parse(dr["Total"].ToString());
                        traditionalDictionary.Add(title, total);
                    }
                }
            }

            Dictionary<string, int> linqDictionary = new Dictionary<string, int>();
            using (var dc = new AWDataContext())
            {
                linqDictionary = dc.Persons
                         //.OrderBy(item => dc.Random())
                          .Take(500)
                          .GroupBy(p => p.Title)
                          .ToDictionary(g => g.Key ?? String.Empty, g => g.Count());
            }

            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqDictionary.SequenceEqual(traditionalDictionary));
        }

        /// <summary>
        /// Sum - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void SumTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (SqlConnection conn = AdventureWorksConn)
                    {
                        conn.Open();
                        string query = @"SELECT SUM(LineTotal) AS RandomTotal
                                FROM (
                                             SELECT TOP 500 LineTotal
                                             FROM Sales.SalesOrderDetail
                                             ORDER BY NEWID()
                                     ) AS S";
                        SqlCommand command = new SqlCommand(query, conn);
                        decimal total = Decimal.Parse(command.ExecuteScalar().ToString());
                    }
                }
            }
        }

        /// <summary>
        /// Sum - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void SumLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (var dc = new AWDataContext())
                    {
                        decimal total = dc.SalesOrderDetails
                                      .OrderBy(item => dc.Random())
                                      .Take(500)
                                      .Sum(s => s.LineTotal);
                    }
                }
            }
        }

        /// <summary>
        /// Sum equivalence verification.
        /// </summary>
        private static void SumVerifyEquivalence()
        {
            decimal traditionalTotal;
            using (SqlConnection conn = AdventureWorksConn)
            {
                conn.Open();
                string query = @"SELECT SUM(LineTotal) AS RandomTotal
                        FROM (
                            SELECT TOP 500 LineTotal
                            FROM Sales.SalesOrderDetail
                            --ORDER BY NEWID()
                          ) AS S";
                SqlCommand command = new SqlCommand(query, conn);
                traditionalTotal = Decimal.Parse(command.ExecuteScalar().ToString());
            }
```

```
            decimal linqTotal;
            using (var dc = new AWDataContext())
            {
                linqTotal = dc.SalesOrderDetails
                             //.OrderBy(item => dc.Random())
                              .Take(500)
                              .Sum(s => s.LineTotal);
            }

            Common.LogVerification(MethodBase.GetCurrentMethod().Name, traditionalTotal == linqTotal);
        }

        #region AdventureWorksPeople Methods

        /// <summary>
        /// Insert - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void InsertTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (SqlConnection conn = AdventureWorksPeopleConn)
                    {
                        conn.Open();
                        for (int count = 1; count <= Common.InsertCount; count++)
                        {
                            string query = @"INSERT INTO People
                                (Title, FirstName, MiddleName, LastName, rowguid, ModifiedDate)
                                VALUES(@Title, @FirstName, @MiddleName, @LastName, @rowguid,
@ModifiedDate)";
                            SqlParameter title = new SqlParameter("@Title", SqlDbType.Text);
                            title.Value = "Mr.";
                            SqlParameter firstName = new SqlParameter("@FirstName", SqlDbType.Text);
                            firstName.Value = Service.GetFirstName(i);
                            SqlParameter middleName = new SqlParameter("@MiddleName", SqlDbType.Text);
                            middleName.Value = i.ToString();
                            SqlParameter lastName = new SqlParameter("@LastName", SqlDbType.Text);
                            lastName.Value = "Mageed";
                            SqlParameter rowguid = new SqlParameter("@rowguid", SqlDbType.UniqueIdentifier);
                            rowguid.Value = Guid.NewGuid();
                            SqlParameter modifiedDate = new SqlParameter("@ModifiedDate",
SqlDbType.DateTime);
                            modifiedDate.Value = DateTime.Now;

                            SqlCommand command = new SqlCommand(query, conn);
                            command.Parameters.Add(title);
                            command.Parameters.Add(firstName);
                            command.Parameters.Add(middleName);
                            command.Parameters.Add(lastName);
                            command.Parameters.Add(rowguid);
                            command.Parameters.Add(modifiedDate);
                            command.ExecuteNonQuery();
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Insert - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void InsertLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (var dc = new AWPeopleDataContext())
                    {
                        for (int count = 1; count <= Common.InsertCount; count++)
                        {
                            People person = new People
                            {
                                Title = "Mr.",
                                FirstName = Service.GetFirstName(i),
                                MiddleName = count.ToString(),
                                LastName = "Mageed",
                                rowguid = Guid.NewGuid(),
                                ModifiedDate = DateTime.Now
                            };

                            dc.Peoples.InsertOnSubmit(person);
                            dc.SubmitChanges();
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Insert equivalence verification.
        /// </summary>
        private static void InsertVerifyEquivalence()
        {
            int traditionalAffectedRecords;
            using (SqlConnection conn = AdventureWorksPeopleConn)
            {
                string query = @"INSERT INTO People
                                (Title, FirstName, MiddleName, LastName, rowguid, ModifiedDate)
                                VALUES(@Title, @FirstName, @MiddleName, @LastName, @rowguid,
@ModifiedDate)";
                SqlParameter title = new SqlParameter("@Title", SqlDbType.Text);
                title.Value = "Mr.";
                SqlParameter firstName = new SqlParameter("@FirstName", SqlDbType.Text);
                firstName.Value = Service.GetFirstName(0);
                SqlParameter middleName = new SqlParameter("@MiddleName", SqlDbType.Text);
```

```
                middleName.Value = 1.ToString();
                SqlParameter lastName = new SqlParameter("@LastName", SqlDbType.Text);
                lastName.Value = "Mageed";
                SqlParameter rowguid = new SqlParameter("@rowguid", SqlDbType.UniqueIdentifier);
                rowguid.Value = Guid.NewGuid();
                SqlParameter modifiedDate = new SqlParameter("@ModifiedDate", SqlDbType.DateTime);
                modifiedDate.Value = DateTime.Now;

                conn.Open();
                SqlCommand command = new SqlCommand(query, conn);
                command.Parameters.Add(title);
                command.Parameters.Add(firstName);
                command.Parameters.Add(middleName);
                command.Parameters.Add(lastName);
                command.Parameters.Add(rowguid);
                command.Parameters.Add(modifiedDate);
                traditionalAffectedRecords = command.ExecuteNonQuery();
            }

            int linqAffectedRecords;
            using (var dc = new AWPeopleDataContext())
            {
                People person = new People();
                person.Title = "Mr.";
                person.FirstName = Service.GetFirstName(1);
                person.MiddleName = 1.ToString();
                person.LastName = "Mageed";
                person.rowguid = Guid.NewGuid();
                person.ModifiedDate = DateTime.Now;

                dc.Peoples.InsertOnSubmit(person);
                linqAffectedRecords = dc.GetChangeSet().Inserts.Count;
                dc.SubmitChanges();
            }

            // note: LINQ count is prior to actual submission, but an exception is thrown if it fails so it'll be
noticed
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, linqAffectedRecords ==
traditionalAffectedRecords);
        }

        /// <summary>
        /// Update - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void UpdateTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (SqlConnection conn = AdventureWorksPeopleConn)
                    {
                        string query = @"UPDATE People SET
                                Title=@Title, FirstName=@FirstName, MiddleName=@MiddleName,
                                LastName=@LastName, ModifiedDate=@ModifiedDate
                                WHERE LastName=@LastNameCriteria";
                        SqlParameter title = new SqlParameter("@Title", SqlDbType.Text);
                        title.Value = "Mr.";
                        SqlParameter firstName = new SqlParameter("@FirstName", SqlDbType.Text);
                        firstName.Value = Service.GetFirstName(i);
                        SqlParameter middleName = new SqlParameter("@MiddleName", SqlDbType.Text);
                        middleName.Value = i.ToString();
                        SqlParameter lastName = new SqlParameter("@LastName", SqlDbType.Text);
                        lastName.Value = "Mageed";
                        SqlParameter modifiedDate = new SqlParameter("@ModifiedDate",
SqlDbType.DateTime);
                        modifiedDate.Value = DateTime.Now;
                        SqlParameter lastNameCriteria = new SqlParameter("@LastNameCriteria",
SqlDbType.NVarChar);
                        lastNameCriteria.Value = "Mageed";

                        conn.Open();
                        SqlCommand command = new SqlCommand(query, conn);
                        command.Parameters.Add(title);
                        command.Parameters.Add(firstName);
                        command.Parameters.Add(middleName);
                        command.Parameters.Add(lastName);
                        command.Parameters.Add(modifiedDate);
                        command.Parameters.Add(lastNameCriteria);
                        command.ExecuteNonQuery();
                    }
                }
            }
        }

        /// <summary>
        /// Update - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void UpdateLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (var dc = new AWPeopleDataContext())
                    {
                        var peopleToUpdate = dc.Peoples.Where(p => p.LastName == "Mageed");
                        foreach (People person in peopleToUpdate)
                        {
                            person.Title = "Mr.";
                            person.FirstName = Service.GetFirstName(i);
                            person.MiddleName = i.ToString();
                            person.LastName = "Mageed";
                            person.ModifiedDate = DateTime.Now;
                        }
                        dc.SubmitChanges();
                    }
                }
            }
        }

        /// <summary>
        /// Update equivalence verification.
        /// </summary>
        private static void UpdateVerifyEquivalence()
        {
            int traditionalAffectedRecords;
            using (SqlConnection conn = AdventureWorksPeopleConn)
            {
                string query = @"UPDATE People SET
                            Title=@Title, FirstName=@FirstName, MiddleName=@MiddleName,
                            LastName=@LastName, ModifiedDate=@ModifiedDate
                            WHERE LastName=@LastNameCriteria";
                SqlParameter title = new SqlParameter("@Title", SqlDbType.Text);
                title.Value = "Mr.";
                SqlParameter firstName = new SqlParameter("@FirstName", SqlDbType.Text);
                firstName.Value = Service.GetFirstName(0);
                SqlParameter middleName = new SqlParameter("@MiddleName", SqlDbType.Text);
                middleName.Value = 1.ToString();
                SqlParameter lastName = new SqlParameter("@LastName", SqlDbType.Text);
                lastName.Value = "Mageed";
                SqlParameter modifiedDate = new SqlParameter("@ModifiedDate", SqlDbType.DateTime);
                modifiedDate.Value = DateTime.Now;
                SqlParameter lastNameCriteria = new SqlParameter("@LastNameCriteria",
SqlDbType.NVarChar);
                lastNameCriteria.Value = "Mageed";

                conn.Open();
                SqlCommand command = new SqlCommand(query, conn);
                command.Parameters.Add(title);
                command.Parameters.Add(firstName);
                command.Parameters.Add(middleName);
                command.Parameters.Add(lastName);
                command.Parameters.Add(modifiedDate);
                command.Parameters.Add(lastNameCriteria);
                traditionalAffectedRecords = command.ExecuteNonQuery();
            }

            int linqAffectedRecords;
            using (var dc = new AWPeopleDataContext())
            {
                var peopleToUpdate = dc.Peoples.Where(p => p.LastName == "Mageed");
                foreach (People person in peopleToUpdate)
                {
                    person.Title = "Mr.";
                    person.FirstName = Service.GetFirstName(1);
                    person.MiddleName = 1.ToString();
                    person.LastName = "Mageed";
                    person.ModifiedDate = DateTime.Now;
                }
                linqAffectedRecords = dc.GetChangeSet().Updates.Count;
                dc.SubmitChanges();
            }
```

```
          // note: LINQ count is prior to actual submission, but an exception is thrown if it fails so it'll be
noticed
          Common.LogVerification(MethodBase.GetCurrentMethod().Name, linqAffectedRecords ==
traditionalAffectedRecords);
      }

      /// <summary>
      /// Delete - Traditional.
      /// </summary>
      /// <param name="loops">The number of times to execute the benchmark.</param>
      private static void DeleteTraditional(int loops)
      {
          Common.LogMethod(MethodBase.GetCurrentMethod().Name);
          for (int i = 1; i <= loops; i++)
          {
              Service.PrepareDataForDeletion();
              using (var timer = new BenchmarkTimer())
              {
                  using (SqlConnection conn = AdventureWorksPeopleConn)
                  {
                      string query = @"DELETE FROM People WHERE LastName=@LastNameCriteria";
                      SqlParameter lastNameCriteria = new SqlParameter("@LastNameCriteria",
SqlDbType.NVarChar);
                      lastNameCriteria.Value = "Mageed";

                      conn.Open();
                      SqlCommand command = new SqlCommand(query, conn);
                      command.Parameters.Add(lastNameCriteria);
                      command.ExecuteNonQuery();
                  }
              }
          }
      }

      /// <summary>
      /// Delete - LINQ.
      /// </summary>
      /// <param name="loops">The number of times to execute the benchmark.</param>
      private static void DeleteLINQ(int loops)
      {
          Common.LogMethod(MethodBase.GetCurrentMethod().Name);
          for (int i = 1; i <= loops; i++)
          {
              Service.PrepareDataForDeletion();
              using (var timer = new BenchmarkTimer())
              {
                  using (var dc = new AWPeopleDataContext())
                  {
                      var peopleToDelete = dc.Peoples.Where(p => p.LastName == "Mageed");
                      dc.Peoples.DeleteAllOnSubmit(peopleToDelete);
                      dc.SubmitChanges();
                  }
              }
          }
      }
```

```
        /// <summary>
        /// Delete equivalence verification.
        /// </summary>
        private static void DeleteVerifyEquivalence()
        {
            Service.PrepareDataForDeletion();
            int traditionalAffectedRecords;
            using (SqlConnection conn = AdventureWorksPeopleConn)
            {
                string query = @"DELETE FROM People WHERE LastName=@LastNameCriteria";
                SqlParameter lastNameCriteria = new SqlParameter("@LastNameCriteria",
SqlDbType.NVarChar);
                lastNameCriteria.Value = "Mageed";

                conn.Open();
                SqlCommand command = new SqlCommand(query, conn);
                command.Parameters.Add(lastNameCriteria);
                traditionalAffectedRecords = command.ExecuteNonQuery();
            }

            Service.PrepareDataForDeletion();
            int linqAffectedRecords;
            using (var dc = new AWPeopleDataContext())
            {
                var peopleToDelete = dc.Peoples.Where(p => p.LastName == "Mageed");
                dc.Peoples.DeleteAllOnSubmit(peopleToDelete);
                linqAffectedRecords = dc.GetChangeSet().Deletes.Count;
                dc.SubmitChanges();
            }

            // note: LINQ count is prior to actual submission, but an exception is thrown if it fails so it'll be
noticed
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, linqAffectedRecords ==
traditionalAffectedRecords);
        }

        #endregion
    }
}
```

LinqToSqlOptimizedBenchmark.cs

```
using System;
using System.Collections.Generic;
using System.Data.Linq;
using System.Data.SqlClient;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Reflection;
using BTR.Core.Linq;

namespace ThesisResearch
{
    public static class LinqToSqlOptimizedBenchmark
    {
        /// <summary>
        /// Returns a new SqlConnection using the AdventureWorks2008 database.
        /// </summary>
        private static SqlConnection AdventureWorksConn
        {
            get
            {
                return new SqlConnection(@"Data Source=S03957;Initial
Catalog=AdventureWorks2008;Integrated Security=True");
            }
        }

        /// <summary>
        /// Returns a new SqlConnection using the AdventureWorksPeople database.
        /// </summary>
        private static SqlConnection AdventureWorksPeopleConn
        {
            get
            {
                return new SqlConnection(@"Data Source=S03957;Initial
Catalog=AdventureWorksPeople;Integrated Security=True");
            }
        }

        /// <summary>
        /// Starts the Linq to SQL Optimized benchmark tests.
        /// </summary>
        public static void Start()
        {
            Console.WriteLine("Started LinqToSqlCompiledBenchmark...");
            RunBenchmarks(false);    // JIT run
            RunBenchmarks(true);     // actual run
        }

        /// <summary>
        /// Runs the benchmarks.
        /// </summary>
        /// <param name="isBenchmark">Indicates whether the run is a real benchmark
or a warmup.</param>
        private static void RunBenchmarks(bool isBenchmark)
        {
            Common.ResetCounter();
            int loops = isBenchmark ? Common.Repetitions :
Common.JitWarmupRepetitions;

            SelectLINQ(loops);
            CountLINQ(loops);
            WhereLINQ(loops);
            WhereNoMatchLINQ(loops);
            JoinLINQ(loops);
            OrderLINQ(loops);
            GroupLINQ(loops);
            SumLINQ(loops);
            InsertLINQ(loops);
            UpdateLINQ(loops);
            DeleteLINQ(loops);

            string text = String.Format("{0}*** {1} complete: {2} iterations ***{0}",
                Environment.NewLine, isBenchmark ? "Benchmark" : "Warmup", loops);
            using (var sw = new StreamWriter(BenchmarkTimer.BenchmarkFilename, true))
            {
                sw.WriteLine(text);
            }
            Console.WriteLine(text);
        }

        /// <summary>
        /// Select - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void SelectLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            Stopwatch compileTimer = Stopwatch.StartNew();

            Func<AWDataContext, IQueryable<PersonInfo>> SelectCompiled =
CompiledQuery.Compile((AWDataContext dc) =>
                            dc.Persons
                                .Select(person => new PersonInfo
                                {
                                    Id = person.BusinessEntityID,
                                    FirstName = person.FirstName,
                                    LastName = person.LastName
                                })
                                .OrderBy(item => dc.Random())
                                .Take(500));

            compileTimer.Stop();
            Common.LogMethod(compileTimer.ElapsedMilliseconds.ToString());
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<PersonInfo> linqList = new List<PersonInfo>();
                    using (var dc = new AWDataContext())
                    {
                        dc.ObjectTrackingEnabled = false;
                        SelectCompiled(dc).ToList();
                    }
                }
            }
        }

        /// <summary>
        /// Count - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void CountLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            Stopwatch compileTimer = Stopwatch.StartNew();

            Func<AWDataContext, int> CountCompiled = 
CompiledQuery.Compile((AWDataContext dc) =>
                            dc.Persons.OrderBy(item => 
dc.Random()).Take(500).Count());

            compileTimer.Stop();
            Common.LogMethod(compileTimer.ElapsedMilliseconds.ToString());

            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    int linqCount;
                    using (var dc = new AWDataContext())
                    {
                        dc.ObjectTrackingEnabled = false;
                        linqCount = CountCompiled(dc);
                    }
                }
            }
        }

        /// <summary>
        /// Where filtering - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void WhereLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            Stopwatch compileTimer = Stopwatch.StartNew();

            Func<AWDataContext, IQueryable<PersonInfo>> WhereCompiled = 
CompiledQuery.Compile((AWDataContext dc) =>
                            dc.Persons.Where(person => 
person.FirstName.StartsWith("A"))
                                      .Select(person => new PersonInfo
                                      {
                                          Id = person.BusinessEntityID,
                                          FirstName = person.FirstName,
                                          LastName = person.LastName
                                      })
                                      .OrderBy(item => dc.Random()).Take(500));

            compileTimer.Stop();
            Common.LogMethod(compileTimer.ElapsedMilliseconds.ToString());

            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<PersonInfo> linqList = new List<PersonInfo>();
                    using (var dc = new AWDataContext())
                    {
                        dc.ObjectTrackingEnabled = false;
                        linqList = WhereCompiled(dc).ToList();
                    }
                }
            }
        }

        /// <summary>
        /// Where filtering (no match) - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void WhereNoMatchLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            Stopwatch compileTimer = Stopwatch.StartNew();

            Func<AWDataContext, IQueryable<PersonInfo>> WhereNoMatchCompiled =
CompiledQuery.Compile((AWDataContext dc) =>
                            dc.Persons.Where(person => person.FirstName == "Ahmad")
                                      .Select(person => new PersonInfo
                                      {
                                          Id = person.BusinessEntityID,
                                          FirstName = person.FirstName,
                                          LastName = person.LastName
                                      })
                                      .OrderBy(item => dc.Random()).Take(500));

            compileTimer.Stop();
            Common.LogMethod(compileTimer.ElapsedMilliseconds.ToString());

            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<PersonInfo> linqList = new List<PersonInfo>();
                    using (var dc = new AWDataContext())
                    {
                        dc.ObjectTrackingEnabled = false;
                        linqList = WhereNoMatchCompiled(dc).ToList();
                    }
                }
            }
        }

        /// <summary>
        /// Join - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void JoinLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            Stopwatch compileTimer = Stopwatch.StartNew();

            Func<AWDataContext, IQueryable<EmployeeInfo>> JoinCompiled =
CompiledQuery.Compile((AWDataContext dc) =>
                            (from e in dc.Employees
                             join edh in dc.EmployeeDepartmentHistories
                                  on e.BusinessEntityID equals edh.BusinessEntityID
                             join d in dc.Departments on edh.DepartmentID equals
d.DepartmentID
                             join p in dc.Persons on e.BusinessEntityID equals
p.BusinessEntityID
                             select new EmployeeInfo(
                                 p.BusinessEntityID,
                                 p.FirstName,
                                 p.LastName,
                                 d.Name,
                                 e.Gender,
                                 edh.StartDate,
                                 edh.EndDate
                             ))
                                    .OrderBy(item => dc.Random()).Take(500));

            compileTimer.Stop();
            Common.LogMethod(compileTimer.ElapsedMilliseconds.ToString());

            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<EmployeeInfo> linqList = new List<EmployeeInfo>();
                    using (var dc = new AWDataContext())
                    {
                        dc.ObjectTrackingEnabled = false;
                        linqList = JoinCompiled(dc).ToList();
                    }
                }
            }
        }

        /// <summary>
        /// Order - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void OrderLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            Stopwatch compileTimer = Stopwatch.StartNew();

            Func<AWDataContext, IQueryable<PersonInfo>> OrderCompiled =
CompiledQuery.Compile((AWDataContext dc) =>
                            dc.Persons.Select(person => new PersonInfo
                                        {
                                            Id = person.BusinessEntityID,
                                            FirstName = person.FirstName,
                                            LastName = person.LastName
                                        })
                                        .OrderBy(item => dc.Random())
                                        .Take(500));
            compileTimer.Stop();
            Common.LogMethod(compileTimer.ElapsedMilliseconds.ToString());

            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (var dc = new AWDataContext())
                    {
                        dc.ObjectTrackingEnabled = false;
                        var linqQuery = OrderCompiled(dc);
                        List<PersonInfo> linqList = linqQuery.OrderBy(person =>
person.Id).ToList();
                    }
                }
            }
        }

        /// <summary>
        /// Group - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void GroupLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            Stopwatch compileTimer = Stopwatch.StartNew();

            Func<AWDataContext, IQueryable<IGrouping<string, Person>>> GroupCompiled
= CompiledQuery.Compile((AWDataContext dc) =>
                                            dc.Persons.OrderBy(item => dc.Random())
                                                      .Take(500)
                                                      .GroupBy(p => p.Title));

            compileTimer.Stop();
            Common.LogMethod(compileTimer.ElapsedMilliseconds.ToString());

            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (var dc = new AWDataContext())
                    {
                        dc.ObjectTrackingEnabled = false;
                        Dictionary<string, int> linqDictionary =
GroupCompiled(dc).ToDictionary(g => g.Key ?? String.Empty, g => g.Count());
                    }
                }
            }
        }

        /// <summary>
        /// Sum - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void SumLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            Stopwatch compileTimer = Stopwatch.StartNew();

            Func<AWDataContext, decimal> SumCompiled =
CompiledQuery.Compile((AWDataContext dc) =>
                                        dc.SalesOrderDetails
                                          .OrderBy(item => dc.Random())
                                          .Take(500)
                                          .Sum(s => s.LineTotal));

            compileTimer.Stop();
            Common.LogMethod(compileTimer.ElapsedMilliseconds.ToString());

            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (var dc = new AWDataContext())
                    {
                        dc.ObjectTrackingEnabled = false;
                        decimal total = SumCompiled(dc);
                    }
                }
            }
        }

        #region AdventureWorksPeople Methods

        /// <summary>
        /// Insert - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void InsertLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (var dc = new AWPeopleDataContext())
                    {
                        for (int count = 1; count <= Common.InsertCount; count++)
                        {
                            People person = new People
                            {
                                Title = "Mr.",
                                FirstName = Service.GetFirstName(i),
                                MiddleName = count.ToString(),
                                LastName = "Mageed",
                                rowguid = Guid.NewGuid(),
                                ModifiedDate = DateTime.Now
                            };
                            dc.Peoples.InsertOnSubmit(person);
                            dc.SubmitChanges();
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Update - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void UpdateLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    using (var dc = new AWPeopleDataContext())
                    {
                        var peopleToUpdate = dc.Peoples.Where(p => p.LastName ==
"Mageed");
                        dc.Peoples.UpdateBatch(peopleToUpdate, p => new People
                        {
                            Title = "Mr.",
                            FirstName = Service.GetFirstName(i),
                            MiddleName = i.ToString(),
                            LastName = "Mageed",
                            ModifiedDate = DateTime.Now
                        });
                    }
                }
            }
        }

        /// <summary>
        /// Delete - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void DeleteLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                Service.PrepareDataForDeletion();
                using (var timer = new BenchmarkTimer())
                {
                    using (var dc = new AWPeopleDataContext())
                    {
                        var peopleToDelete = dc.Peoples.Where(p => p.LastName ==
"Mageed");
                        dc.Peoples.DeleteBatch(peopleToDelete);
                    }
                }
            }
        }

        #endregion
    }
}
```

LinqToXmlBenchmark.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Xml;
using System.Xml.Linq;
using System.Xml.XPath;

namespace ThesisResearch
{
    public static class LinqToXmlBenchmark
    {
        /// <summary>
        /// Returns a random XML filename.
        /// </summary>
        /// <returns>Filename string</returns>
        private static string GetRandomXmlFile()
        {
            return "persons" + Common.Rand.Next(1, Common.XmlFilesCount) + ".xml";
        }

        /// <summary>
        /// Returns an XPathNavigator loaded with a random XML file.
        /// </summary>
        /// <returns>XPathNavigator</returns>
        private static XPathNavigator GetRandomXPathNavigator()
        {
            XPathDocument xpathDocument = new XPathDocument(GetRandomXmlFile());
            return xpathDocument.CreateNavigator();
        }

        /// <summary>
        /// Returns an XmlDocument loaded with a random XML file.
        /// </summary>
        /// <returns></returns>
        private static XmlDocument GetRandomXmlDocument()
        {
            XmlDocument xmlDocument = new XmlDocument();
            xmlDocument.Load(GetRandomXmlFile());
            return xmlDocument;
        }

        /// <summary>
        /// Starts the Linq to XML benchmark tests.
        /// </summary>
        public static void Start()
        {
            Console.WriteLine("Started LinqToXmlBenchmark...");
            RunBenchmarks(false);    // JIT run
            RunBenchmarks(true);     // actual run
        }

        /// <summary>
        /// Runs the benchmarks.
        /// </summary>
        /// <param name="isBenchmark">Indicates whether the run is a real benchmark or a
warmup.</param>
        private static void RunBenchmarks(bool isBenchmark)
        {
            Common.ResetCounter();
            int loops = isBenchmark ? Common.Repetitions : Common.JitWarmupRepetitions;

            OrderByVerifyEquivalence();
            OrderByTraditional(loops);
            OrderByLINQ(loops);

            WhereVerifyEquivalence();
            WhereTraditional(loops);
            WhereLINQ(loops);

            FullnameListVerifyEquivalence();
            FullnameListTraditional(loops);
            FullnameListLINQ(loops);

            CountVerifyEquivalence();
            CountTraditional(loops);
            CountLINQ(loops);

            AverageVerifyEquivalence();
            AverageTraditional(loops);
            AverageLINQ(loops);

            RemoveVerifyEquivalence();
            RemoveTraditional(loops);
            RemoveLINQ(loops);

            UpdateVerifyEquivalence();
            UpdateTraditional(loops);
            UpdateLINQ(loops);

            GenerateVerifyEquivalence();
            GenerateTraditional(loops);
            GenerateLINQ(loops);

            ElementAtVerifyEquivalence();
            ElementAtTraditional(loops);
            ElementAtLINQ(loops);

            LastVerifyEquivalence();
            LastTraditional(loops);
            LastLINQ(loops);

            TakeVerifyEquivalence();
            TakeTraditional(loops);
            TakeLINQ(loops);

            SkipWhileVerifyEquivalence();
            SkipWhileTraditional(loops);
            SkipWhileLINQ(loops);

            GroupByAndToDictionaryVerifyEquivalence();
            GroupByAndToDictionaryTraditional(loops);
            GroupByAndToDictionaryLINQ(loops);

            string text = String.Format("{0}*** {1} complete: {2} iterations ***{0}",
                                  Environment.NewLine, isBenchmark ? "Benchmark" : "Warmup", loops);
            using (var sw = new StreamWriter(BenchmarkTimer.BenchmarkFilename, true))
            {
                sw.WriteLine(text);
            }
            Console.WriteLine(text);
        }

        /// <summary>
        /// OrderBy - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void OrderByTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    XPathNavigator navigator = GetRandomXPathNavigator();
                    XPathExpression expression = navigator.Compile("Persons/Person[@ModifiedDate]");
                    expression.AddSort("@ModifiedDate", new DateTimeComparer());
                    XPathNodeIterator xpathIterator = navigator.Select(expression);

                    StringBuilder traditionalResult = new StringBuilder();
                    while (xpathIterator.MoveNext())
                    {
                        traditionalResult.AppendLine(xpathIterator.Current.OuterXml);
                    }
                }
            }
        }

        /// <summary>
        /// OrderBy - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void OrderByLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
```

```
          {
             using (var timer = new BenchmarkTimer())
             {
                XDocument document = XDocument.Load(GetRandomXmlFile());
                var query = document.Root.Elements("Person")
                                  .OrderBy(p => DateTime.Parse(p.Attribute("ModifiedDate").Value));
                StringBuilder linqResult = new StringBuilder();
                foreach (var person in query)
                {
                   linqResult.AppendLine(person.ToString());
                }
             }
          }
       }

       /// <summary>
       /// OrderBy equivalence verification.
       /// </summary>
       private static void OrderByVerifyEquivalence()
       {
          string filename = "persons1.xml";
          // Traditional
          XPathDocument xpathDocument = new XPathDocument(filename);
          XPathNavigator navigator = xpathDocument.CreateNavigator();
          XPathExpression expression = navigator.Compile("Persons/Person[@ModifiedDate]");
          expression.AddSort("@ModifiedDate", new DateTimeComparer());
          XPathNodeIterator xpathIterator = navigator.Select(expression);

          StringBuilder traditionalResult = new StringBuilder();
          while (xpathIterator.MoveNext())
          {
             traditionalResult.AppendLine(xpathIterator.Current.OuterXml);
          }

          // LINQ
          XDocument document = XDocument.Load(filename);
          StringBuilder linqResult = new StringBuilder();
          var query = document.Root.Elements("Person")
                            .OrderBy(p => DateTime.Parse(p.Attribute("ModifiedDate").Value));
          foreach (var person in query)
          {
             linqResult.AppendLine(person.ToString());
          }

          // Verification
          Common.LogVerification(MethodBase.GetCurrentMethod().Name, linqResult.ToString() ==
traditionalResult.ToString());
       }

       /// <summary>
       /// Where filtering - Traditional.
       /// </summary>
       /// <param name="loops">The number of times to execute the benchmark.</param>
       private static void WhereTraditional(int loops)
       {
          Common.LogMethod(MethodBase.GetCurrentMethod().Name);
```

```
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    XPathNavigator navigator = GetRandomXPathNavigator();
                    XPathNodeIterator xpathIterator = navigator.Select("Persons/Person[FirstName[starts-
with(.,'A')]]");
                    StringBuilder traditionalResult = new StringBuilder();
                    while (xpathIterator.MoveNext())
                    {
                        traditionalResult.AppendLine(xpathIterator.Current.OuterXml);
                    }
                }
            }
        }

        /// <summary>
        /// Where filtering - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void WhereLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    XDocument document = XDocument.Load(GetRandomXmlFile());
                    var query = document.Root.Elements("Person")
                                         .Where(p => p.Element("FirstName").Value.StartsWith("A"));
                    StringBuilder linqResult = new StringBuilder();
                    foreach (var person in query)
                    {
                        linqResult.AppendLine(person.ToString());
                    }
                }
            }
        }

        /// <summary>
        /// Where filtering equivalence verification.
        /// </summary>
        private static void WhereVerifyEquivalence()
        {
            string filename = "persons1.xml";
            // Traditional
            XPathDocument xpathDocument = new XPathDocument(filename);
            XPathNavigator navigator = xpathDocument.CreateNavigator();
            XPathNodeIterator xpathIterator = navigator.Select("Persons/Person[FirstName[starts-
with(.,'A')]]");
            StringBuilder traditionalResult = new StringBuilder();
            while (xpathIterator.MoveNext())
            {
                traditionalResult.AppendLine(xpathIterator.Current.OuterXml);
            }
            // LINQ
            XDocument document = XDocument.Load(filename);
            var query = document.Root.Elements("Person")
                              .Where(p => p.Element("FirstName").Value.StartsWith("A"));
            StringBuilder linqResult = new StringBuilder();
            foreach (var person in query)
            {
                linqResult.AppendLine(person.ToString());
            }

            // Verification
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, traditionalResult.ToString() ==
linqResult.ToString());
        }

        /// <summary>
        /// Select (fullname list) - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void FullnameListTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    XPathNavigator navigator = GetRandomXPathNavigator();
                    XPathNodeIterator xpathIterator = navigator.Select("Persons/Person");
                    XPathExpression concatExpression = navigator.Compile("concat(FirstName, ' ',
LastName)");
                    List<string> traditionalResult = new List<string>();
                    while (xpathIterator.MoveNext())
                    {
                        traditionalResult.Add(xpathIterator.Current.Evaluate(concatExpression).ToString());
                    }
                }
            }
        }

        /// <summary>
        /// Select (fullname list) - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void FullnameListLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    XDocument document = XDocument.Load(GetRandomXmlFile());
                    var query = document.Root.Elements("Person")
                                  .Select(p => p.Element("FirstName").Value + " " +
p.Element("LastName").Value);
                    List<string> linqResult = query.ToList();
                }
            }
        }

        /// <summary>
        /// Select (fullname list) equivalence verification.
        /// </summary>
        private static void FullnameListVerifyEquivalence()
        {
            string filename = "persons1.xml";
            // Traditional
            XPathDocument xpathDocument = new XPathDocument(filename);
            XPathNavigator navigator = xpathDocument.CreateNavigator();
            XPathNodeIterator xpathIterator = navigator.Select("Persons/Person");
            XPathExpression concatExpression = navigator.Compile("concat(FirstName, ' ', LastName)");
            List<string> traditionalResult = new List<string>();
            while (xpathIterator.MoveNext())
            {
                traditionalResult.Add(xpathIterator.Current.Evaluate(concatExpression).ToString());
            }

            // LINQ
            XDocument document = XDocument.Load(filename);
            var query = document.Root.Elements("Person")
                                .Select(p => p.Element("FirstName").Value + " " + p.Element("LastName").Value);
            List<string> linqResult = query.ToList();

            // Verification
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqResult.SequenceEqual(traditionalResult));
        }

        /// <summary>
        /// Count - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void CountTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    XPathNavigator navigator = GetRandomXPathNavigator();
                    string count = navigator.Evaluate("count(Persons/Person)").ToString();
                    int traditionalResult = Int32.Parse(count);
                }
            }
        }

        /// <summary>
        /// Count - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void CountLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
```

```
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    XDocument document = XDocument.Load(GetRandomXmlFile());
                    int linqResult = document.Root.Elements("Person").Count();
                }
            }
        }

        /// <summary>
        /// Count equivalence verification.
        /// </summary>
        private static void CountVerifyEquivalence()
        {
            // Traditional
            string filename = "persons1.xml";
            XPathDocument xpathDocument = new XPathDocument(filename);
            XPathNavigator navigator = xpathDocument.CreateNavigator();
            string count = navigator.Evaluate("count(Persons/Person)").ToString();
            int traditionalResult = Int32.Parse(count);

            // LINQ
            XDocument document = XDocument.Load(filename);
            int linqResult = document.Root.Elements("Person").Count();

            // Verification
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, linqResult ==
traditionalResult);
        }

        /// <summary>
        /// Average - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void AverageTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    XPathNavigator navigator = GetRandomXPathNavigator();
                    string sum = navigator.Evaluate("sum(Persons/Person/@BusinessEntityID) div
count(Persons/Person/@BusinessEntityID)").ToString();
                    double traditionalResult = Double.Parse(sum);
                }
            }
        }

        /// <summary>
        /// Average - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void AverageLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    XDocument document = XDocument.Load(GetRandomXmlFile());
                    double linqResult = document.Root.Elements("Person")
                                            .Average(p => Double.Parse(p.Attribute("BusinessEntityID").Value));
                }
            }
        }

        /// <summary>
        /// Average equivalence verification.
        /// </summary>
        private static void AverageVerifyEquivalence()
        {
            // Traditional
            string filename = "persons1.xml";
            XPathDocument xpathDocument = new XPathDocument(filename);
            XPathNavigator navigator = xpathDocument.CreateNavigator();
            string sum = navigator.Evaluate("sum(Persons/Person/@BusinessEntityID) div
count(Persons/Person/@BusinessEntityID)").ToString();
            double traditionalResult = Double.Parse(sum);

            // LINQ
            XDocument document = XDocument.Load(filename);
            double linqResult = document.Root.Elements("Person")
                                    .Average(p => Double.Parse(p.Attribute("BusinessEntityID").Value));

            // Verification
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, linqResult ==
traditionalResult);
        }

        /// <summary>
        /// Remove - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void RemoveTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    XmlDocument xmlDocument = GetRandomXmlDocument();
                    XmlNode personsRoot = xmlDocument.SelectSingleNode("Persons");
                    var traditionalNodes = personsRoot.SelectNodes("Person[string-length(FirstName) > 6]");
                    foreach (XmlNode node in traditionalNodes)
                    {
                        personsRoot.RemoveChild(node);
                    }
                }
            }
        }
        /// <summary>
        /// Remove - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void RemoveLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    XDocument document = XDocument.Load(GetRandomXmlFile());
                    var linqNodes = document.Root.Elements("Person")
                                        .Where(p => p.Element("FirstName").Value.Length > 6);
                    linqNodes.Remove();
                }
            }
        }

        /// <summary>
        /// Remove equivalence verification.
        /// </summary>
        private static void RemoveVerifyEquivalence()
        {
            string filename = "persons1.xml";
            // Traditional
            XmlDocument xmlDocument = new XmlDocument();
            xmlDocument.Load(filename);
            XmlNode personsRoot = xmlDocument.SelectSingleNode("Persons");
            var traditionalNodes = personsRoot.SelectNodes("Person[string-length(FirstName) > 6]");
            int traditionalBeforeRemove = personsRoot.ChildNodes.Count;
            foreach (XmlNode node in traditionalNodes)
            {
                personsRoot.RemoveChild(node);
            }
            int traditionalAfterRemove = personsRoot.ChildNodes.Count;

            // LINQ
            XDocument document = XDocument.Load(filename);
            int linqBeforeRemove = document.Root.Nodes().Count();
            var linqNodes = document.Root.Elements("Person").Where(p =>
p.Element("FirstName").Value.Length > 6);
            linqNodes.Remove();
            int linqAfterRemove = document.Root.Nodes().Count();

            // Verification
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
                (traditionalBeforeRemove == linqBeforeRemove && traditionalAfterRemove ==
linqAfterRemove));
        }

        /// <summary>
        /// Update - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
```

```
private static void UpdateTraditional(int loops)
{
    Common.LogMethod(MethodBase.GetCurrentMethod().Name);
    for (int i = 1; i <= loops; i++)
    {
        using (var timer = new BenchmarkTimer())
        {
            XmlDocument xmlDocument = GetRandomXmlDocument();
            XmlNode personsRoot = xmlDocument.SelectSingleNode("Persons");
            XmlNodeList nodes = personsRoot.SelectNodes("Person/@ModifiedDate");
            foreach (XmlNode node in nodes)
            {
                node.InnerText = DateTime.Today.ToString();
            }
        }
    }
}

/// <summary>
/// Update - LINQ.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void UpdateLINQ(int loops)
{
    Common.LogMethod(MethodBase.GetCurrentMethod().Name);
    for (int i = 1; i <= loops; i++)
    {
        using (var timer = new BenchmarkTimer())
        {
            var xDocument = XDocument.Load(GetRandomXmlFile());
            foreach (var person in xDocument.Root.Elements("Person"))
            {
                person.Attribute("ModifiedDate").Value = DateTime.Today.ToString();
            }
        }
    }
}

/// <summary>
/// Update equivalence verification.
/// </summary>
private static void UpdateVerifyEquivalence()
{
    string filename = "persons1.xml";
    // Traditional
    XmlDocument xmlDocument = new XmlDocument();
    xmlDocument.Load(filename);
    XmlNode personsRoot = xmlDocument.SelectSingleNode("Persons");
    XmlNodeList nodes = personsRoot.SelectNodes("Person/@ModifiedDate");
    foreach (XmlNode node in nodes)
    {
        node.InnerText = DateTime.Today.ToString();
    }

    // LINQ
    XDocument document = XDocument.Load(filename);
```

```
            foreach (var person in document.Root.Elements("Person"))
            {
                person.Attribute("ModifiedDate").Value = DateTime.Today.ToString();
            }

            // Verification
            bool traditionalUpdated = true;
            foreach (XmlNode node in nodes)
            {
                if (node.InnerText != DateTime.Today.ToString())
                {
                    traditionalUpdated = false;
                    break;
                }
            }
            bool linqUpdated = document.Elements().Attributes().All(attribute => attribute.Value ==
DateTime.Today.ToString());
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, traditionalUpdated ==
linqUpdated);
        }

        /// <summary>
        /// Generate - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void GenerateTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<Person> people = Service.GetPeopleRandom();

                    XmlDocument xmlDocument = new XmlDocument();
                    XmlNode documentRoot = xmlDocument.CreateElement("root");
                    xmlDocument.AppendChild(documentRoot);
                    foreach (var person in people)
                    {
                        XmlElement element = xmlDocument.CreateElement("Person");
                        XmlAttribute id = xmlDocument.CreateAttribute("BusinessEntityID");
                        id.Value = person.BusinessEntityID.ToString();
                        element.Attributes.Append(id);

                        XmlAttribute modifiedDate = xmlDocument.CreateAttribute("ModifiedDate");
                        modifiedDate.Value = person.ModifiedDate.ToString();
                        element.Attributes.Append(modifiedDate);

                        XmlElement firstName = xmlDocument.CreateElement("FirstName");
                        firstName.InnerText = person.FirstName;
                        element.AppendChild(firstName);

                        XmlElement middleName = xmlDocument.CreateElement("MiddleName");
                        middleName.InnerText = person.MiddleName;
                        element.AppendChild(middleName);
                        XmlElement lastName = xmlDocument.CreateElement("LastName");
                        lastName.InnerText = person.LastName;
                        element.AppendChild(lastName);

                        XmlElement rowGuid = xmlDocument.CreateElement("Rowguid");
                        rowGuid.InnerText = person.rowguid.ToString();
                        element.AppendChild(rowGuid);

                        documentRoot.AppendChild(element);
                    }
                    string traditionalResult = xmlDocument.OuterXml;
                }
            }
        }

        /// <summary>
        /// Generate - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void GenerateLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    List<Person> people = Service.GetPeopleRandom();

                    var elements = people.Select(person => new XElement("Person",
                            new XAttribute("BusinessEntityID", person.BusinessEntityID),
                            new XAttribute("ModifiedDate", person.ModifiedDate.ToString()),
                            new XElement("FirstName", person.FirstName),
                            new XElement("MiddleName", person.MiddleName),
                            new XElement("LastName", person.LastName),
                            new XElement("Rowguid", person.rowguid)
                        )
                    );

                    XElement root = new XElement("root");
                    foreach (var person in elements)
                    {
                        root.Add(person);
                    }

                    string linqResult = root.ToString();
                }
            }
        }

        /// <summary>
        /// Generate equivalence verification.
        /// </summary>
        private static void GenerateVerifyEquivalence()
        {
            List<Person> people = Service.GetPeopleRandom();
```

```
// Traditional
XmlDocument xmlDocument = new XmlDocument();
XmlNode documentRoot = xmlDocument.CreateElement("root");
xmlDocument.AppendChild(documentRoot);
foreach (var person in people)
{
   XmlElement element = xmlDocument.CreateElement("Person");
   XmlAttribute id = xmlDocument.CreateAttribute("BusinessEntityID");
   id.Value = person.BusinessEntityID.ToString();
   element.Attributes.Append(id);

   XmlAttribute modifiedDate = xmlDocument.CreateAttribute("ModifiedDate");
   modifiedDate.Value = person.ModifiedDate.ToString();
   element.Attributes.Append(modifiedDate);

   XmlElement firstName = xmlDocument.CreateElement("FirstName");
   firstName.InnerText = person.FirstName;
   element.AppendChild(firstName);

   XmlElement middleName = xmlDocument.CreateElement("MiddleName");
   middleName.InnerText = person.MiddleName;
   element.AppendChild(middleName);

   XmlElement lastName = xmlDocument.CreateElement("LastName");
   lastName.InnerText = person.LastName;
   element.AppendChild(lastName);

   XmlElement rowGuid = xmlDocument.CreateElement("Rowguid");
   rowGuid.InnerText = person.rowguid.ToString();
   element.AppendChild(rowGuid);

   documentRoot.AppendChild(element);
}
string traditionalResult = xmlDocument.OuterXml;

// LINQ
var elements = people.Select(person => new XElement("Person",
            new XAttribute("BusinessEntityID", person.BusinessEntityID),
            new XAttribute("ModifiedDate", person.ModifiedDate.ToString()),
            new XElement("FirstName", person.FirstName),
            new XElement("MiddleName", person.MiddleName),
            new XElement("LastName", person.LastName),
            new XElement("Rowguid", person.rowguid)
         )
      );
XElement root = new XElement("root");
foreach (var person in elements)
{
   root.Add(person);
}
string linqResult = root.ToString();

// Verification
XDocument linqXml = XDocument.Parse(linqResult);
XDocument traditionalXml = XDocument.Parse(traditionalResult);
```

```
            bool isEqual = traditionalXml.Root.Elements().Count() == linqXml.Root.Elements().Count();
            if (isEqual)
            {
                for (int i = 0; i < linqXml.Root.Elements().Count(); i++)
                {
                    XElement linqElement = linqXml.Root.Elements().ElementAt(i);
                    XElement traditionalElement = traditionalXml.Root.Elements().ElementAt(i);
                    bool equalAttributes = linqElement.Attributes().All(a => a.Value ==
traditionalElement.Attribute(a.Name).Value);
                    bool equalElements = linqElement.Elements().All(e => e.Value ==
traditionalElement.Element(e.Name).Value);

                    if (!equalAttributes || !equalElements)
                    {
                        isEqual = false;
                        break;
                    }
                }
            }

            Common.LogVerification(MethodBase.GetCurrentMethod().Name, isEqual);
        }

        /// <summary>
        /// ElementAt - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ElementAtTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    int randomIndex = Common.Rand.Next(1, Service.SampleSize);
                    XPathNavigator navigator = GetRandomXPathNavigator();
                    XPathNavigator nodeResult = navigator.SelectSingleNode("Persons/Person[" + randomIndex
+ "]");
                    string traditionalResult = nodeResult.OuterXml;
                }
            }
        }

        /// <summary>
        /// ElementAt - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void ElementAtLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    int linqRandomIndex = Common.Rand.Next(1, Service.SampleSize) - 1;
                    var xDocument = XDocument.Load(GetRandomXmlFile());
                    string linqResult =
xDocument.Root.Elements("Person").ElementAt(linqRandomIndex).ToString();
                }
            }
        }

        /// <summary>
        /// ElementAt equivalence verification.
        /// </summary>
        private static void ElementAtVerifyEquivalence()
        {
            // random index minimum set to 1 since XPath is 1-based
            int randomIndex = Common.Rand.Next(1, Service.SampleSize);
            // LINQ ElementAt is 0-based, decrement to get same element
            int linqRandomIndex = randomIndex - 1;

            // Traditional
            string filename = "persons1.xml";
            XPathDocument xpathDocument = new XPathDocument(filename);
            XPathNavigator navigator = xpathDocument.CreateNavigator();
            XPathNavigator nodeResult = navigator.SelectSingleNode("Persons/Person[" + randomIndex +
"]");
            string traditionalResult = nodeResult.OuterXml;

            // LINQ
            XDocument document = XDocument.Load(filename);
            string linqResult = document.Root.Elements("Person").ElementAt(linqRandomIndex).ToString();

            // Verification
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, linqResult ==
traditionalResult);
        }

        /// <summary>
        /// Last - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void LastTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    int randomIndex = Common.Rand.Next(1, Service.SampleSize);
                    XPathNavigator navigator = GetRandomXPathNavigator();
                    string traditionalResult = navigator.SelectSingleNode("Persons/Person[last()]").OuterXml;
                }
            }
        }

        /// <summary>
        /// Last - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void LastLINQ(int loops)
```

```
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    var xDocument = XDocument.Load(GetRandomXmlFile());
                    string linqResult = xDocument.Root.Elements("Person").Last().ToString();
                }
            }
        }

        /// <summary>
        /// Last equivalence verification.
        /// </summary>
        private static void LastVerifyEquivalence()
        {
            // Traditional
            string filename = "persons1.xml";
            XPathDocument xpathDocument = new XPathDocument(filename);
            XPathNavigator navigator = xpathDocument.CreateNavigator();
            string traditionalResult = navigator.SelectSingleNode("Persons/Person[last()]").OuterXml;

            // LINQ
            XDocument document = XDocument.Load(filename);
            string linqResult = document.Root.Elements("Person").Last().ToString();

            // Verification
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, linqResult ==
traditionalResult);
        }

        /// <summary>
        /// Take - Traditional.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void TakeTraditional(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    int randomCount = Common.Rand.Next(1, Service.SampleSize);
                    XPathNavigator navigator = GetRandomXPathNavigator();
                    XPathNodeIterator xpathIterator = navigator.Select("Persons/Person[position() <= " +
randomCount + "]");
                    StringBuilder traditionalResult = new StringBuilder();
                    while (xpathIterator.MoveNext())
                    {
                        traditionalResult.AppendLine(xpathIterator.Current.OuterXml);
                    }
                }
            }
        }
        /// <summary>
        /// Take - LINQ.
        /// </summary>
        /// <param name="loops">The number of times to execute the benchmark.</param>
        private static void TakeLINQ(int loops)
        {
            Common.LogMethod(MethodBase.GetCurrentMethod().Name);
            for (int i = 1; i <= loops; i++)
            {
                using (var timer = new BenchmarkTimer())
                {
                    int randomCount = Common.Rand.Next(1, Service.SampleSize);
                    var document = XDocument.Load(GetRandomXmlFile());
                    var elements = document.Root.Elements("Person").Take(randomCount);
                    StringBuilder linqResult = new StringBuilder();
                    foreach (var item in elements)
                    {
                        linqResult.AppendLine(item.ToString());
                    }
                }
            }
        }

        /// <summary>
        /// Take equivalence verification.
        /// </summary>
        private static void TakeVerifyEquivalence()
        {
            int randomCount = Common.Rand.Next(1, Service.SampleSize);

            // Traditional
            string filename = "persons1.xml";
            XPathDocument xpathDocument = new XPathDocument(filename);
            XPathNavigator navigator = xpathDocument.CreateNavigator();
            XPathNodeIterator xpathIterator = navigator.Select("Persons/Person[position() <= " +
randomCount + "]");
            StringBuilder traditionalResult = new StringBuilder();
            while (xpathIterator.MoveNext())
            {
                traditionalResult.AppendLine(xpathIterator.Current.OuterXml);
            }

            // LINQ
            XDocument document = XDocument.Load(filename);
            var elements = document.Root.Elements("Person").Take(randomCount);
            StringBuilder linqResult = new StringBuilder();
            foreach (var item in elements)
            {
                linqResult.AppendLine(item.ToString());
            }

            // Verification
            Common.LogVerification(MethodBase.GetCurrentMethod().Name, linqResult.ToString() ==
traditionalResult.ToString());
        }
```

```
    /// <summary>
    /// SkipWhile - Traditional.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void SkipWhileTraditional(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                int nameLength = Common.Rand.Next(6, 10);
                XPathNavigator navigator = GetRandomXPathNavigator();
                XPathNodeIterator xpathIterator = navigator.Select("//Person");

                bool conditionMet = false;
                StringBuilder traditionalResult = new StringBuilder();
                while (xpathIterator.MoveNext())
                {
                    string firstName = xpathIterator.Current.SelectSingleNode("FirstName").Value;
                    if (!conditionMet && firstName.Length < nameLength)
                    {
                        // skip while the condition hasn't been met
                        continue;
                    }
                    else
                    {
                        // halt skipping, condition has been met
                        conditionMet = true;
                        traditionalResult.AppendLine(xpathIterator.Current.OuterXml);
                    }
                }
            }
        }
    }

    /// <summary>
    /// SkipWhile - LINQ.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
    private static void SkipWhileLINQ(int loops)
    {
        Common.LogMethod(MethodBase.GetCurrentMethod().Name);
        for (int i = 1; i <= loops; i++)
        {
            using (var timer = new BenchmarkTimer())
            {
                int nameLength = Common.Rand.Next(6, 10);
                var document = XDocument.Load(GetRandomXmlFile());
                var elements = document.Descendants("Person")
                                      .SkipWhile(person => person.Element("FirstName").Value.Length <
nameLength);

                StringBuilder linqResult = new StringBuilder();
                foreach (var item in elements)
                {
                    linqResult.AppendLine(item.ToString());
                }
            }
        }
    }

    /// <summary>
    /// SkipWhile equivalence verification.
    /// </summary>
    private static void SkipWhileVerifyEquivalence()
    {
        int nameLength = Common.Rand.Next(6, 10);

        // Traditional
        string filename = "persons1.xml";
        XPathDocument xpathDocument = new XPathDocument(filename);
        XPathNavigator navigator = xpathDocument.CreateNavigator();
        XPathNodeIterator xpathIterator = navigator.Select("//Person");

        bool conditionMet = false;
        StringBuilder traditionalResult = new StringBuilder();
        while (xpathIterator.MoveNext())
        {
            string firstName = xpathIterator.Current.SelectSingleNode("FirstName").Value;
            if (!conditionMet && firstName.Length < nameLength)
            {
                // skip while the condition hasn't been met
                continue;
            }
            else
            {
                // halt skipping, condition has been met
                conditionMet = true;
                traditionalResult.AppendLine(xpathIterator.Current.OuterXml);
            }
        }

        // LINQ
        XDocument document = XDocument.Load(filename);
        var elements = document.Descendants("Person")
                        .SkipWhile(person => person.Element("FirstName").Value.Length < nameLength);
        StringBuilder linqResult = new StringBuilder();
        foreach (var item in elements)
        {
            linqResult.AppendLine(item.ToString());
        }

        // Verification
        Common.LogVerification(MethodBase.GetCurrentMethod().Name, linqResult.ToString() ==
traditionalResult.ToString());
    }

    /// <summary>
    /// GroupByAndToDictionary - Traditional.
    /// </summary>
    /// <param name="loops">The number of times to execute the benchmark.</param>
```

```
private static void GroupByAndToDictionaryTraditional(int loops)
{
    Common.LogMethod(MethodBase.GetCurrentMethod().Name);
    for (int i = 1; i <= loops; i++)
    {
        using (var timer = new BenchmarkTimer())
        {
            XPathNavigator navigator = GetRandomXPathNavigator();
            XPathNodeIterator xpathIterator = navigator.Select("Persons/Person");
            Dictionary<string, int> traditionalResult = new Dictionary<string, int>();
            while (xpathIterator.MoveNext())
            {
                string initial = xpathIterator.Current.Evaluate("substring(LastName, 1, 1)").ToString();
                int currentCount;
                if (traditionalResult.TryGetValue(initial, out currentCount))
                {
                    traditionalResult[initial] = currentCount + 1;
                }
                else
                {
                    traditionalResult.Add(initial, 1);
                }
            }
        }
    }
}

/// <summary>
/// GroupByAndToDictionary - LINQ.
/// </summary>
/// <param name="loops">The number of times to execute the benchmark.</param>
private static void GroupByAndToDictionaryLINQ(int loops)
{
    Common.LogMethod(MethodBase.GetCurrentMethod().Name);
    for (int i = 1; i <= loops; i++)
    {
        using (var timer = new BenchmarkTimer())
        {
            var document = XDocument.Load(GetRandomXmlFile());
            var linqResult = document.Root.Elements("Person")
                                  .GroupBy(p => p.Element("LastName").Value.Substring(0, 1))
                                  .ToDictionary(g => g.Key, g => g.Count());
        }
    }
}

/// <summary>
/// GroupByAndToDictionary equivalence verification.
/// </summary>
private static void GroupByAndToDictionaryVerifyEquivalence()
{
    string filename = "persons1.xml";

    // Traditional
    XPathDocument xpathDocument = new XPathDocument(filename);
    XPathNavigator navigator = xpathDocument.CreateNavigator();
```

```
            XPathNodeIterator xpathIterator = navigator.Select("Persons/Person");
            Dictionary<string, int> traditionalResult = new Dictionary<string, int>();
            while (xpathIterator.MoveNext())
            {
                string initial = xpathIterator.Current.Evaluate("substring(LastName, 1, 1)").ToString();
                int currentCount;
                if (traditionalResult.TryGetValue(initial, out currentCount))
                {
                    traditionalResult[initial] = currentCount + 1;
                }
                else
                {
                    traditionalResult.Add(initial, 1);
                }
            }

            // LINQ
            XDocument document = XDocument.Load(filename);
            var linqResult = document.Root.Elements("Person")
                                .GroupBy(p => p.Element("LastName").Value.Substring(0, 1))
                                .ToDictionary(g => g.Key, g => g.Count());

            // Verification
            Common.LogVerification(MethodBase.GetCurrentMethod().Name,
linqResult.SequenceEqual(traditionalResult));
        }
    }
}
```

LINQBatchPost for LINQ to SQL – ExpressionVisitor.cs

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Linq.Expressions;
using System.Text;

namespace BTR.Core.Linq
{
	public static class ExpressionExtensions
	{
		public static Expression Visit<T>(
			this Expression exp,
			Func<T, Expression> visitor ) where T : Expression
		{
			return ExpressionVisitor<T>.Visit( exp, visitor );
		}

		public static TExp Visit<T, TExp>(
			this TExp exp,
			Func<T, Expression> visitor )
			where T : Expression
			where TExp : Expression
		{
				return (TExp)ExpressionVisitor<T>.Visit( exp, visitor );
			}

			public static Expression<TDelegate> Visit<T, TDelegate>(
				this Expression<TDelegate> exp,
				Func<T, Expression> visitor ) where T : Expression
			{
				return ExpressionVisitor<T>.Visit<TDelegate>( exp, visitor );
			}

			public static IQueryable<TSource> Visit<T, TSource>(
				this IQueryable<TSource> source,
				Func<T, Expression> visitor ) where T : Expression
			{
				return source.Provider.CreateQuery<TSource>( ExpressionVisitor<T>.Visit(
source.Expression, visitor ) );
			}
		}

		/// <summary>
		/// This class visits every Parameter expression in an expression tree and calls a delegate
		/// to optionally replace the parameter.  This is useful where two expression trees need to
		/// be merged (and they don't share the same ParameterExpressions).
		/// </summary>
		public class ExpressionVisitor<T> : ExpressionVisitor where T : Expression
		{
			Func<T, Expression> visitor;

			public ExpressionVisitor( Func<T, Expression> visitor )
			{
				this.visitor = visitor;
			}

			public static Expression Visit(
				Expression exp,
				Func<T, Expression> visitor )
			{
				return new ExpressionVisitor<T>( visitor ).Visit( exp );
			}

			public static Expression<TDelegate> Visit<TDelegate>(
				Expression<TDelegate> exp,
				Func<T, Expression> visitor )
			{
				return (Expression<TDelegate>)new ExpressionVisitor<T>( visitor ).Visit( exp );
			}

			protected override Expression Visit( Expression exp )
			{
				if ( exp is T && visitor != null ) exp = visitor( (T)exp );

				return base.Visit( exp );
			}
		}

		/// <summary>
```

```
        /// Expression visitor
        /// (from http://blogs.msdn.com/mattwar/archive/2007/07/31/linq-building-an-iqueryable-provider-
part-ii.aspx)
        /// </summary>
        public abstract class ExpressionVisitor
        {
                protected ExpressionVisitor()
                {
                }

                protected virtual Expression Visit( Expression exp )
                {
                        if ( exp == null )
                                return exp;
                        switch ( exp.NodeType )
                        {
                                case ExpressionType.Negate:
                                case ExpressionType.NegateChecked:
                                case ExpressionType.Not:
                                case ExpressionType.Convert:
                                case ExpressionType.ConvertChecked:
                                case ExpressionType.ArrayLength:
                                case ExpressionType.Quote:
                                case ExpressionType.TypeAs:
                                        return this.VisitUnary( (UnaryExpression)exp );
                                case ExpressionType.Add:
                                case ExpressionType.AddChecked:
                                case ExpressionType.Subtract:
                                case ExpressionType.SubtractChecked:
                                case ExpressionType.Multiply:
                                case ExpressionType.MultiplyChecked:
                                case ExpressionType.Divide:
                                case ExpressionType.Modulo:
                                case ExpressionType.And:
                                case ExpressionType.AndAlso:
                                case ExpressionType.Or:
                                case ExpressionType.OrElse:
                                case ExpressionType.LessThan:
                                case ExpressionType.LessThanOrEqual:
                                case ExpressionType.GreaterThan:
                                case ExpressionType.GreaterThanOrEqual:
                                case ExpressionType.Equal:
                                case ExpressionType.NotEqual:
                                case ExpressionType.Coalesce:
                                case ExpressionType.ArrayIndex:
                                case ExpressionType.RightShift:
                                case ExpressionType.LeftShift:
                                case ExpressionType.ExclusiveOr:
                                        return this.VisitBinary( (BinaryExpression)exp );
                                case ExpressionType.TypeIs:
                                        return this.VisitTypeIs( (TypeBinaryExpression)exp );
                                case ExpressionType.Conditional:
                                        return this.VisitConditional( (ConditionalExpression)exp );
                                case ExpressionType.Constant:
                                        return this.VisitConstant( (ConstantExpression)exp );
                                case ExpressionType.Parameter:
                                        return this.VisitParameter( (ParameterExpression)exp );
                                case ExpressionType.MemberAccess:
                                        return this.VisitMemberAccess( (MemberExpression)exp );
                                case ExpressionType.Call:
                                        return this.VisitMethodCall( (MethodCallExpression)exp );
                                case ExpressionType.Lambda:
                                        return this.VisitLambda( (LambdaExpression)exp );
                                case ExpressionType.New:
                                        return this.VisitNew( (NewExpression)exp );
                                case ExpressionType.NewArrayInit:
                                case ExpressionType.NewArrayBounds:
                                        return this.VisitNewArray( (NewArrayExpression)exp );
                                case ExpressionType.Invoke:
                                        return this.VisitInvocation( (InvocationExpression)exp );
                                case ExpressionType.MemberInit:
                                        return this.VisitMemberInit( (MemberInitExpression)exp );
                                case ExpressionType.ListInit:
                                        return this.VisitListInit( (ListInitExpression)exp );
                                default:
                                        throw new Exception( string.Format( "Unhandled expression
type: '{0}'", exp.NodeType ) );
                        }
                }

                protected virtual MemberBinding VisitBinding( MemberBinding binding )
                {
                        switch ( binding.BindingType )
                        {
                                case MemberBindingType.Assignment:
                                        return this.VisitMemberAssignment(
(MemberAssignment)binding );
                                case MemberBindingType.MemberBinding:
                                        return this.VisitMemberMemberBinding(
(MemberMemberBinding)binding );
                                case MemberBindingType.ListBinding:
                                        return this.VisitMemberListBinding( (MemberListBinding)binding
);
                                default:
                                        throw new Exception( string.Format( "Unhandled binding type
'{0}'", binding.BindingType ) );
                        }
                }

                protected virtual ElementInit VisitElementInitializer( ElementInit initializer )
                {
                        ReadOnlyCollection<Expression> arguments = this.VisitExpressionList(
initializer.Arguments );
                        if ( arguments != initializer.Arguments )
                        {
                                return Expression.ElementInit( initializer.AddMethod, arguments );
                        }
                        return initializer;
                }

                protected virtual Expression VisitUnary( UnaryExpression u )
                {
```

```
				Expression operand = this.Visit( u.Operand );
				if ( operand != u.Operand )
				{
					return Expression.MakeUnary( u.NodeType, operand, u.Type, u.Method
);
				}
				return u;
			}

			protected virtual Expression VisitBinary( BinaryExpression b )
			{
				Expression left = this.Visit( b.Left );
				Expression right = this.Visit( b.Right );
				Expression conversion = this.Visit( b.Conversion );
				if ( left != b.Left || right != b.Right || conversion != b.Conversion )
				{
					if ( b.NodeType == ExpressionType.Coalesce && b.Conversion != null )
						return Expression.Coalesce( left, right, conversion as
LambdaExpression );
					else
						return Expression.MakeBinary( b.NodeType, left, right,
b.IsLiftedToNull, b.Method );
				}
				return b;
			}

			protected virtual Expression VisitTypeIs( TypeBinaryExpression b )
			{
				Expression expr = this.Visit( b.Expression );
				if ( expr != b.Expression )
				{
					return Expression.TypeIs( expr, b.TypeOperand );
				}
				return b;
			}

			protected virtual Expression VisitConstant( ConstantExpression c )
			{
				return c;
			}

			protected virtual Expression VisitConditional( ConditionalExpression c )
			{
				Expression test = this.Visit( c.Test );
				Expression ifTrue = this.Visit( c.IfTrue );
				Expression ifFalse = this.Visit( c.IfFalse );
				if ( test != c.Test || ifTrue != c.IfTrue || ifFalse != c.IfFalse )
				{
					return Expression.Condition( test, ifTrue, ifFalse );
				}
				return c;
			}

			protected virtual Expression VisitParameter( ParameterExpression p )
			{
				return p;
```

```
        }

        protected virtual Expression VisitMemberAccess( MemberExpression m )
        {
            Expression exp = this.Visit( m.Expression );
            if ( exp != m.Expression )
            {
                return Expression.MakeMemberAccess( exp, m.Member );
            }
            return m;
        }

        protected virtual Expression VisitMethodCall( MethodCallExpression m )
        {
            Expression obj = this.Visit( m.Object );
            IEnumerable<Expression> args = this.VisitExpressionList( m.Arguments );
            if ( obj != m.Object || args != m.Arguments )
            {
                return Expression.Call( obj, m.Method, args );
            }
            return m;
        }

        protected virtual ReadOnlyCollection<Expression> VisitExpressionList(
ReadOnlyCollection<Expression> original )
        {
            List<Expression> list = null;
            for ( int i = 0, n = original.Count; i < n; i++ )
            {
                Expression p = this.Visit( original[ i ] );
                if ( list != null )
                {
                    list.Add( p );
                }
                else if ( p != original[ i ] )
                {
                    list = new List<Expression>( n );
                    for ( int j = 0; j < i; j++ )
                    {
                        list.Add( original[ j ] );
                    }
                    list.Add( p );
                }
            }
            if ( list != null )
            {
                return list.AsReadOnly();
            }
            return original;
        }

        protected virtual MemberAssignment VisitMemberAssignment( MemberAssignment
assignment )
        {
            Expression e = this.Visit( assignment.Expression );
            if ( e != assignment.Expression )
```

```
                {
                    return Expression.Bind( assignment.Member, e );
                }
                return assignment;
            }

            protected virtual MemberMemberBinding VisitMemberMemberBinding(
MemberMemberBinding binding )
            {
                IEnumerable<MemberBinding> bindings = this.VisitBindingList( binding.Bindings
);
                if ( bindings != binding.Bindings )
                {
                    return Expression.MemberBind( binding.Member, bindings );
                }
                return binding;
            }

            protected virtual MemberListBinding VisitMemberListBinding( MemberListBinding binding
)
            {
                IEnumerable<ElementInit> initializers = this.VisitElementInitializerList(
binding.Initializers );
                if ( initializers != binding.Initializers )
                {
                    return Expression.ListBind( binding.Member, initializers );
                }
                return binding;
            }

            protected virtual IEnumerable<MemberBinding> VisitBindingList(
ReadOnlyCollection<MemberBinding> original )
            {
                List<MemberBinding> list = null;
                for ( int i = 0, n = original.Count; i < n; i++ )
                {
                    MemberBinding b = this.VisitBinding( original[ i ] );
                    if ( list != null )
                    {
                        list.Add( b );
                    }
                    else if ( b != original[ i ] )
                    {
                        list = new List<MemberBinding>( n );
                        for ( int j = 0; j < i; j++ )
                        {
                            list.Add( original[ j ] );
                        }
                        list.Add( b );
                    }
                }
                if ( list != null )
                    return list;
                return original;
            }
```

```
        protected virtual IEnumerable<ElementInit> VisitElementInitializerList(
ReadOnlyCollection<ElementInit> original )
        {
            List<ElementInit> list = null;
            for ( int i = 0, n = original.Count; i < n; i++ )
            {
                ElementInit init = this.VisitElementInitializer( original[ i ] );
                if ( list != null )
                {
                    list.Add( init );
                }
                else if ( init != original[ i ] )
                {
                    list = new List<ElementInit>( n );
                    for ( int j = 0; j < i; j++ )
                    {
                        list.Add( original[ j ] );
                    }
                    list.Add( init );
                }
            }
            if ( list != null )
                return list;
            return original;
        }

        protected virtual Expression VisitLambda( LambdaExpression lambda )
        {
            Expression body = this.Visit( lambda.Body );
            if ( body != lambda.Body )
            {
                return Expression.Lambda( lambda.Type, body, lambda.Parameters );
            }
            return lambda;
        }

        protected virtual NewExpression VisitNew( NewExpression nex )
        {
            IEnumerable<Expression> args = this.VisitExpressionList( nex.Arguments );
            if ( args != nex.Arguments )
            {
                if ( nex.Members != null )
                    return Expression.New( nex.Constructor, args, nex.Members );
                else
                    return Expression.New( nex.Constructor, args );
            }
            return nex;
        }

        protected virtual Expression VisitMemberInit( MemberInitExpression init )
        {
            NewExpression n = this.VisitNew( init.NewExpression );
            IEnumerable<MemberBinding> bindings = this.VisitBindingList( init.Bindings );
            if ( n != init.NewExpression || bindings != init.Bindings )
            {
                return Expression.MemberInit( n, bindings );
                }
                return init;
        }

        protected virtual Expression VisitListInit( ListInitExpression init )
        {
                NewExpression n = this.VisitNew( init.NewExpression );
                IEnumerable<ElementInit> initializers = this.VisitElementInitializerList(
init.Initializers );
                if ( n != init.NewExpression || initializers != init.Initializers )
                {
                        return Expression.ListInit( n, initializers );
                }
                return init;
        }

        protected virtual Expression VisitNewArray( NewArrayExpression na )
        {
                IEnumerable<Expression> exprs = this.VisitExpressionList( na.Expressions );
                if ( exprs != na.Expressions )
                {
                        if ( na.NodeType == ExpressionType.NewArrayInit )
                        {
                                return Expression.NewArrayInit( na.Type.GetElementType(),
exprs );
                        }
                        else
                        {
                                return Expression.NewArrayBounds(
na.Type.GetElementType(), exprs );
                        }
                }
                return na;
        }

        protected virtual Expression VisitInvocation( InvocationExpression iv )
        {
                IEnumerable<Expression> args = this.VisitExpressionList( iv.Arguments );
                Expression expr = this.Visit( iv.Expression );
                if ( args != iv.Arguments || expr != iv.Expression )
                {
                        return Expression.Invoke( expr, args );
                }
                return iv;
        }
    }
}
```

LINQBatchPost for LINQ to SQL – LinqToSqlExtensions.cs

```
using System;
using System.Data.Common;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Data.SqlClient;
using System.Globalization;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.IO;
using System.Runtime.CompilerServices;

namespace BTR.Core.Linq
{
	public static class LinqToSqlExtensions
	{
		/// <summary>
		/// Creates a *.csv file from an IQueryable query, dumping out the 
'simple' properties/fields.
		/// </summary>
		/// <param name="query">Represents a SELECT query to execute.</param>
		/// <param name="fileName">The name of the file to create.</param>
		/// <remarks>
		/// <para>If the file specified by <paramref name="fileName"/> exists, 
it will be deleted.</para>
		/// <para>If the <paramref name="query"/> contains any properties that 
are entity sets (i.e. rows from a FK relationship) the values will not be dumped to 
the file.</para>
		/// <para>This method is useful for debugging purposes or when used in 
other utilities such as LINQPad.</para>
		/// </remarks>
		public static void DumpCSV( this IQueryable query, string fileName )
		{
			query.DumpCSV( fileName, true );
		}

		/// <summary>
		/// Creates a *.csv file from an IQueryable query, dumping out the 
'simple' properties/fields.
		/// </summary>
		/// <param name="query">Represents a SELECT query to execute.</param>
		/// <param name="fileName">The name of the file to create.</param>
		/// <param name="deleteFile">Whether or not to delete the file specified 
by <paramref name="fileName"/> if it exists.</param>
		/// <remarks>
		/// <para>If the <paramref name="query"/> contains any properties that 
are entity sets (i.e. rows from a FK relationship) the values will not be dumped to 
the file.</para>
		/// <para>This method is useful for debugging purposes or when used in 
other utilities such as LINQPad.</para>
		/// </remarks>
		public static void DumpCSV( this IQueryable query, string fileName, bool 
deleteFile )
		{
			if ( File.Exists( fileName ) && deleteFile )
			{
				File.Delete( fileName );
```

```
                    }

                    using ( var output = new FileStream( fileName, FileMode.CreateNew
) )
                    {
                          using ( var writer = new StreamWriter( output ) )
                          {
                                var firstRow = true;

                                PropertyInfo[] properties = null;
                                FieldInfo[] fields = null;
                                Type type = null;
                                bool typeIsAnonymous = false;

                                foreach ( var r in query )
                                {
                                      if ( type == null )
                                      {
                                            type = r.GetType();
                                            typeIsAnonymous = type.IsAnonymous();
                                            properties = type.GetProperties();
                                            fields = type.GetFields();
                                      }

                                      var firstCol = true;

                                      if ( typeIsAnonymous )
                                      {
                                            if ( firstRow )
                                            {
                                                  foreach ( var p in properties )
                                                  {
                                                        if ( !firstCol )
writer.Write( "," );
                                                        else { firstCol = false; }

                                                        writer.Write( p.Name );
                                                  }
                                                  writer.WriteLine();
                                            }
                                            firstRow = false;
                                            firstCol = true;

                                            foreach ( var p in properties )
                                            {
                                                  if ( !firstCol ) writer.Write(
"," );
                                                  else { firstCol = false; }
                                                  DumpValue( p.GetValue( r, null
), writer );
                                            }
                                      }
                                      else
                                      {
                                            if ( firstRow )
```

```
                            {
                                foreach ( var p in fields )
                                {
                                    if ( !firstCol )
writer.Write( "," );
                                    else { firstCol = false; }

                                    writer.Write( p.Name );
                                }
                                writer.WriteLine();
                            }
                            firstRow = false;
                            firstCol = true;

                            foreach ( var p in fields )
                            {
                                if ( !firstCol ) writer.Write(
"," );
                                else { firstCol = false; }

                                DumpValue( p.GetValue( r ),
writer );
                            }
                        }

                        writer.WriteLine();
                    }
                }
            }
        }

        private static void DumpValue( object v, StreamWriter writer )
        {
            if ( v != null )
            {
                switch ( Type.GetTypeCode( v.GetType() ) )
                {
                    // csv encode the value
                    case TypeCode.String:
                        string value = (string)v;
                        if ( value.Contains( "," ) || value.Contains(
'"' ) || value.Contains( "\n" ) )
                        {
                            value = value.Replace( "\"", "\"\"" );

                            if ( value.Length > 31735 )
                            {
                                value = value.Substring( 0,
31732 ) + "...";
                            }
                            writer.Write( "\"" + value + "\"" );
                        }
                        else
                        {
                            writer.Write( value );
```

```
                                        }
                                        break;

                                    default: writer.Write( v ); break;
                                }
                        }
                }

                private static bool IsAnonymous( this Type type )
                {
                        if ( type == null )
                                throw new ArgumentNullException( "type" );

                        // HACK: The only way to detect anonymous types right now.
                        return Attribute.IsDefined( type, typeof(
CompilerGeneratedAttribute ), false )
                                    && type.IsGenericType && type.Name.Contains(
"AnonymousType" )
                                    && ( type.Name.StartsWith( "<>" ) ||
type.Name.StartsWith( "VB$" ) )
                                    && ( type.Attributes & TypeAttributes.NotPublic )
== TypeAttributes.NotPublic;

                }

                /// <summary>
                /// Batches together multiple IQueryable queries into a single DbCommand
and returns all data in
                /// a single roundtrip to the database.
                /// </summary>
                /// <param name="context">The DataContext to execute the batch select
against.</param>
                /// <param name="queries">Represents a collections of SELECT queries to
execute.</param>
                /// <returns>Returns an IMultipleResults object containing all
results.</returns>
                public static IMultipleResults SelectMutlipleResults( this DataContext
context, IQueryable[] queries )
                {
                        var commandList = new List<DbCommand>();

                        foreach ( IQueryable query in queries )
                        {
                                var command = context.GetCommand( query );
                                commandList.Add( command );
                        }

                        SqlCommand batchCommand = CombineCommands( commandList );
                        batchCommand.Connection = context.Connection as SqlConnection;

                        DbDataReader dr = null;

                        if ( batchCommand.Connection.State == ConnectionState.Closed )
                        {
                                batchCommand.Connection.Open();
```

```
                    dr = batchCommand.ExecuteReader(
CommandBehavior.CloseConnection );
                }
                else
                {
                    dr = batchCommand.ExecuteReader();
                }

                IMultipleResults mr = context.Translate( dr );
                return mr;
            }

            /// <summary>
            /// Combines multiple SELECT commands into a single SqlCommand so that
all statements can be executed in a
            /// single roundtrip to the database and return multiple result sets.
            /// </summary>
            /// <param name="commandList">Represents a collection of commands to be
batched together.</param>
            /// <returns>Returns a single SqlCommand that executes all SELECT
statements at once.</returns>
            private static SqlCommand CombineCommands( List<DbCommand>
selectCommands )
            {
                SqlCommand batchCommand = new SqlCommand();
                SqlParameterCollection newParamList = batchCommand.Parameters;

                int commandCount = 0;

                foreach ( DbCommand cmd in selectCommands )
                {
                    string commandText = cmd.CommandText;
                    DbParameterCollection paramList = cmd.Parameters;
                    int paramCount = paramList.Count;

                    for ( int currentParam = paramCount - 1; currentParam >=
0; currentParam-- )
                    {
                        DbParameter param = paramList[ currentParam ];
                        DbParameter newParam = CloneParameter( param );
                        string newParamName = param.ParameterName.Replace(
"@", string.Format( "@{0}_", commandCount ) );
                        commandText = commandText.Replace(
param.ParameterName, newParamName );
                        newParam.ParameterName = newParamName;
                        newParamList.Add( newParam );
                    }
                    if ( batchCommand.CommandText.Length > 0 )
                    {
                        batchCommand.CommandText += ";";
                    }
                    batchCommand.CommandText += commandText;
                    commandCount++;
                }
```

```
				return batchCommand;
			}

			/// <summary>
			/// Returns a clone (via copying all properties) of an existing 
DbParameter.
			/// </summary>
			/// <param name="src">The DbParameter to clone.</param>
			/// <returns>Returns a clone (via copying all properties) of an existing 
DbParameter.</returns>
			private static DbParameter CloneParameter( DbParameter src )
			{
				SqlParameter source = (SqlParameter)src;
				SqlParameter destination = new SqlParameter();

				destination.Value = source.Value;
				destination.Direction = source.Direction;
				destination.Size = source.Size;
				destination.Offset = source.Offset;
				destination.SourceColumn = source.SourceColumn;
				destination.SourceVersion = source.SourceVersion;
				destination.SourceColumnNullMapping = 
source.SourceColumnNullMapping;
				destination.IsNullable = source.IsNullable;

				destination.CompareInfo = source.CompareInfo;
				destination.XmlSchemaCollectionDatabase = 
source.XmlSchemaCollectionDatabase;
				destination.XmlSchemaCollectionOwningSchema = 
source.XmlSchemaCollectionOwningSchema;
				destination.XmlSchemaCollectionName = 
source.XmlSchemaCollectionName;
				destination.UdtTypeName = source.UdtTypeName;
				destination.TypeName = source.TypeName;
				destination.ParameterName = source.ParameterName;
				destination.Precision = source.Precision;
				destination.Scale = source.Scale;

				return destination;
			}

			/// <summary>
			/// Immediately deletes all entities from the collection with a single 
delete command.
			/// </summary>
			/// <typeparam name="TEntity">Represents the object type for rows 
contained in <paramref name="table"/>.</typeparam>
			/// <param name="table">Represents a table for a particular type in the 
underlying database containing rows are to be deleted.</param>
			/// <typeparam name="TPrimaryKey">Represents the object type for the 
primary key of rows contained in <paramref name="table"/>.</typeparam>
			/// <param name="primaryKey">Represents the primary key of the item to 
be removed from <paramref name="table"/>.</param>
			/// <returns>The number of rows deleted from the database (maximum of 
1).</returns>
```

```
            /// <remarks>
            /// <para>If the primary key for <paramref name="table"/> is a composite
key, <paramref name="primaryKey"/> should be an anonymous type with property names
mapping to the property names of objects of type <typeparamref
name="TEntity"/>.</para>
            /// </remarks>
            public static int DeleteByPK<TEntity>( this Table<TEntity> table, object
primaryKey ) where TEntity : class
            {
                  DbCommand delete = table.GetDeleteByPKCommand<TEntity>(
primaryKey );

                  var parameters = from p in delete.Parameters.Cast<DbParameter>()
                                          select p.Value;

                  return table.Context.ExecuteCommand( delete.CommandText,
parameters.ToArray() );
            }

            /// <summary>Returns a string representation the LINQ <see
cref="IProvider"/> command text and parameters used that would be issued to delete a
entity row via the supplied primary key.</summary>
            /// <typeparam name="TEntity">Represents the object type for rows
contained in <paramref name="table"/>.</typeparam>
            /// <param name="table">Represents a table for a particular type in the
underlying database containing rows are to be deleted.</param>
            /// <typeparam name="TPrimaryKey">Represents the object type for the
primary key of rows contained in <paramref name="table"/>.</typeparam>
            /// <param name="primaryKey">Represents the primary key of the item to
be removed from <paramref name="table"/>.</param>
            /// <returns>Returns a string representation the LINQ <see
cref="IProvider"/> command text and parameters used that would be issued to delete a
entity row via the supplied primary key.</returns>
            /// <remarks>This method is useful for debugging purposes or when used
in other utilities such as LINQPad.</remarks>
            public static string DeleteByPKPreview<TEntity>( this Table<TEntity>
table, object primaryKey ) where TEntity : class
            {
                  DbCommand delete = table.GetDeleteByPKCommand<TEntity>(
primaryKey );
                  return delete.PreviewCommandText( false ) +
table.Context.GetLog();
            }

            private static DbCommand GetDeleteByPKCommand<TEntity>( this
Table<TEntity> table, object primaryKey ) where TEntity : class
            {
                  Type type = primaryKey.GetType();
                  bool typeIsAnonymous = type.IsAnonymous();
                  string dbName = table.GetDbName();

                  var metaTable = table.Context.Mapping.GetTable( typeof( TEntity )
);

                  var keys = from mdm in metaTable.RowType.DataMembers
```

```
                                    where mdm.IsPrimaryKey
                                    select new { mdm.MappedName, mdm.Name, mdm.Type
};

                    SqlCommand deleteCommand = new SqlCommand();
                    deleteCommand.Connection = table.Context.Connection as
SqlConnection;

                    var whereSB = new StringBuilder();

                    foreach ( var key in keys )
                    {
                          // Add new parameter with massaged name to avoid clashes.
                          whereSB.AppendFormat( "[{0}] = @p{1}, ", key.MappedName,
deleteCommand.Parameters.Count );

                          object value = primaryKey;
                          if ( typeIsAnonymous || ( type.IsClass && type != typeof(
string ) ) )
                          {
                                if ( typeIsAnonymous )
                                {
                                      PropertyInfo property = type.GetProperty(
key.Name );

                                      if ( property == null )
                                      {
                                            throw new ArgumentOutOfRangeException(
string.Format( "The property {0} which is defined as part of the primary key for {1}
was not supplied by the parameter primaryKey.", key.Name, metaTable.TableName ) );
                                      }

                                      value = property.GetValue( primaryKey, null
);
                                }
                                else
                                {
                                      FieldInfo field = type.GetField( key.Name );

                                      if ( field == null )
                                      {
                                            throw new ArgumentOutOfRangeException(
string.Format( "The property {0} which is defined as part of the primary key for {1}
was not supplied by the parameter primaryKey.", key.Name, metaTable.TableName ) );
                                      }

                                      value = field.GetValue( primaryKey );
                                }

                                if ( value.GetType() != key.Type )
                                {
                                      throw new InvalidCastException(
string.Format( "The property {0} ({1}) does not have the same type as {2} ({3}).",
key.Name, value.GetType(), key.MappedName, key.Type ) );
                                }
```

```
                            }
                            else if ( value.GetType() != key.Type )
                            {
                                    throw new InvalidCastException( string.Format( "The
value supplied in primaryKey ({0}) does not have the same type as {1} ({2}).",
value.GetType(), key.MappedName, key.Type ) );
                            }

                            deleteCommand.Parameters.Add( new SqlParameter(
string.Format( "@p{0}", deleteCommand.Parameters.Count ), value ) );
                    }

                    string wherePK = whereSB.ToString();

                    if ( wherePK == "" )
                    {
                            throw new MissingPrimaryKeyException( string.Format( "{0}
does not have a primary key defined.  Batch updating/deleting can not be used for
tables without a primary key.", metaTable.TableName ) );
                    }

                    deleteCommand.CommandText = string.Format( "DELETE {0}\r\nWHERE
{1}", dbName, wherePK.Substring( 0, wherePK.Length - 2 ) );

                    return deleteCommand;
            }

            /// <summary>
            /// Immediately deletes all entities from the collection with a single
delete command.
            /// </summary>
            /// <typeparam name="TEntity">Represents the object type for rows
contained in <paramref name="table"/>.</typeparam>
            /// <param name="table">Represents a table for a particular type in the
underlying database containing rows are to be deleted.</param>
            /// <param name="entities">Represents the collection of items which are
to be removed from <paramref name="table"/>.</param>
            /// <returns>The number of rows deleted from the database.</returns>
            /// <remarks>
            /// <para>Similiar to stored procedures, and opposite from
DeleteAllOnSubmit, rows provided in <paramref name="entities"/> will be deleted
immediately with no need to call <see cref="DataContext.SubmitChanges()"/>.</para>
            /// <para>Additionally, to improve performance, instead of creating a
delete command for each item in <paramref name="entities"/>, a single delete command
is created.</para>
            /// </remarks>
            public static int DeleteBatch<TEntity>( this Table<TEntity> table,
IQueryable<TEntity> entities ) where TEntity : class
            {
                    DbCommand delete = table.GetDeleteBatchCommand<TEntity>( entities
);

                    var parameters = from p in delete.Parameters.Cast<DbParameter>()
                                     select p.Value;
```

```
                        return table.Context.ExecuteCommand( delete.CommandText,
parameters.ToArray() );
                }

                /// <summary>
                /// Immediately deletes all entities from the collection with a single
delete command.
                /// </summary>
                /// <typeparam name="TEntity">Represents the object type for rows
contained in <paramref name="table"/>.</typeparam>
                /// <param name="table">Represents a table for a particular type in the
underlying database containing rows are to be deleted.</param>
                /// <param name="filter">Represents a filter of items to be updated in
<paramref name="table"/>.</param>
                /// <returns>The number of rows deleted from the database.</returns>
                /// <remarks>
                /// <para>Similiar to stored procedures, and opposite from
DeleteAllOnSubmit, rows provided in <paramref name="entities"/> will be deleted
immediately with no need to call <see cref="DataContext.SubmitChanges()"/>.</para>
                /// <para>Additionally, to improve performance, instead of creating a
delete command for each item in <paramref name="entities"/>, a single delete command
is created.</para>
                /// </remarks>
                public static int DeleteBatch<TEntity>( this Table<TEntity> table,
Expression<Func<TEntity, bool>> filter ) where TEntity : class
                {
                        return table.DeleteBatch( table.Where( filter ) );
                }

                /// <summary>
                /// Returns a string representation the LINQ <see cref="IProvider"/>
command text and parameters used that would be issued to delete all entities from the
collection with a single delete command.
                /// </summary>
                /// <typeparam name="TEntity">Represents the object type for rows
contained in <paramref name="table"/>.</typeparam>
                /// <param name="table">Represents a table for a particular type in the
underlying database containing rows are to be deleted.</param>
                /// <param name="entities">Represents the collection of items which are
to be removed from <paramref name="table"/>.</param>
                /// <returns>Returns a string representation the LINQ <see
cref="IProvider"/> command text and parameters used that would be issued to delete
all entities from the collection with a single delete command.</returns>
                /// <remarks>This method is useful for debugging purposes or when used
in other utilities such as LINQPad.</remarks>
                public static string DeleteBatchPreview<TEntity>( this Table<TEntity>
table, IQueryable<TEntity> entities ) where TEntity : class
                {
                        DbCommand delete = table.GetDeleteBatchCommand<TEntity>( entities
);
                        return delete.PreviewCommandText( false ) +
table.Context.GetLog();
                }

                /// <summary>
```

```
            /// Returns a string representation the LINQ <see cref="IProvider"/>
command text and parameters used that would be issued to delete all entities from the
collection with a single delete command.
            /// </summary>
            /// <typeparam name="TEntity">Represents the object type for rows
contained in <paramref name="table"/>.</typeparam>
            /// <param name="table">Represents a table for a particular type in the
underlying database containing rows are to be deleted.</param>
            /// <param name="filter">Represents a filter of items to be updated in
<paramref name="table"/>.</param>
            /// <returns>Returns a string representation the LINQ <see
cref="IProvider"/> command text and parameters used that would be issued to delete
all entities from the collection with a single delete command.</returns>
            /// <remarks>This method is useful for debugging purposes or when used
in other utilities such as LINQPad.</remarks>
            public static string DeleteBatchPreview<TEntity>( this Table<TEntity>
table, Expression<Func<TEntity, bool>> filter ) where TEntity : class
            {
                  return table.DeleteBatchPreview( table.Where( filter ) );
            }

            /// <summary>
            /// Returns the Transact SQL string representation the LINQ <see
cref="IProvider"/> command text and parameters used that would be issued to delete
all entities from the collection with a single delete command.
            /// </summary>
            /// <typeparam name="TEntity">Represents the object type for rows
contained in <paramref name="table"/>.</typeparam>
            /// <param name="table">Represents a table for a particular type in the
underlying database containing rows are to be deleted.</param>
            /// <param name="entities">Represents the collection of items which are
to be removed from <paramref name="table"/>.</param>
            /// <returns>Returns the Transact SQL string representation the LINQ
<see cref="IProvider"/> command text and parameters used that would be issued to
delete all entities from the collection with a single delete command.</returns>
            /// <remarks>This method is useful for debugging purposes or when LINQ
generated queries need to be passed to developers without LINQ/LINQPad.</remarks>
            public static string DeleteBatchSQL<TEntity>( this Table<TEntity> table,
IQueryable<TEntity> entities ) where TEntity : class
            {
                  DbCommand delete = table.GetDeleteBatchCommand<TEntity>( entities
);
                  return delete.PreviewCommandText( true );
            }

            /// <summary>
            /// Returns the Transact SQL string representation the LINQ <see
cref="IProvider"/> command text and parameters used that would be issued to delete
all entities from the collection with a single delete command.
            /// </summary>
            /// <typeparam name="TEntity">Represents the object type for rows
contained in <paramref name="table"/>.</typeparam>
            /// <param name="table">Represents a table for a particular type in the
underlying database containing rows are to be deleted.</param>
            /// <param name="filter">Represents a filter of items to be updated in
```

```
<paramref name="table"/>.</param>
		/// <returns>Returns the Transact SQL string representation the LINQ
<see cref="IProvider"/> command text and parameters used that would be issued to
delete all entities from the collection with a single delete command.</returns>
		/// <remarks>This method is useful for debugging purposes or when LINQ
generated queries need to be passed to developers without LINQ/LINQPad.</remarks>
		public static string DeleteBatchSQL<TEntity>( this Table<TEntity> table,
Expression<Func<TEntity, bool>> filter ) where TEntity : class
		{
			return table.DeleteBatchSQL( table.Where( filter ) );
		}

		/// <summary>
		/// Returns a string representation the LINQ <see cref="IProvider"/>
command text and parameters used that would be issued to update all entities from the
collection with a single update command.
		/// </summary>
		/// <typeparam name="TEntity">Represents the object type for rows
contained in <paramref name="table"/>.</typeparam>
		/// <param name="table">Represents a table for a particular type in the
underlying database containing rows are to be updated.</param>
		/// <param name="entities">Represents the collection of items which are
to be updated in <paramref name="table"/>.</param>
		/// <param name="evaluator">A Lambda expression returning a
<typeparamref name="TEntity"/> that defines the update assignments to be performed on
each item in <paramref name="entities"/>.</param>
		/// <returns>Returns a string representation the LINQ <see
cref="IProvider"/> command text and parameters used that would be issued to update
all entities from the collection with a single update command.</returns>
		/// <remarks>This method is useful for debugging purposes or when used
in other utilities such as LINQPad.</remarks>
		public static string UpdateBatchPreview<TEntity>( this Table<TEntity>
table, IQueryable<TEntity> entities, Expression<Func<TEntity, TEntity>> evaluator )
where TEntity : class
		{
			DbCommand update = table.GetUpdateBatchCommand<TEntity>(
entities, evaluator );
			return update.PreviewCommandText( false ) +
table.Context.GetLog();
		}

		/// <summary>
		/// Returns a string representation the LINQ <see cref="IProvider"/>
command text and parameters used that would be issued to update all entities from the
collection with a single update command.
		/// </summary>
		/// <typeparam name="TEntity">Represents the object type for rows
contained in <paramref name="table"/>.</typeparam>
		/// <param name="table">Represents a table for a particular type in the
underlying database containing rows are to be updated.</param>
		/// <param name="filter">Represents a filter of items to be updated in
<paramref name="table"/>.</param>
		/// <param name="evaluator">A Lambda expression returning a
<typeparamref name="TEntity"/> that defines the update assignments to be performed on
each item in <paramref name="entities"/>.</param>
```

```
		/// <returns>Returns a string representation the LINQ <see
cref="IProvider"/> command text and parameters used that would be issued to update
all entities from the collection with a single update command.</returns>
		/// <remarks>This method is useful for debugging purposes or when used
in other utilities such as LINQPad.</remarks>
		public static string UpdateBatchPreview<TEntity>( this Table<TEntity>
table, Expression<Func<TEntity, bool>> filter, Expression<Func<TEntity, TEntity>>
evaluator ) where TEntity : class
		{
			return table.UpdateBatchPreview( table.Where( filter ), evaluator
);
		}

		/// <summary>
		/// Returns the Transact SQL string representation the LINQ <see
cref="IProvider"/> command text and parameters used that would be issued to update
all entities from the collection with a single update command.
		/// </summary>
		/// <typeparam name="TEntity">Represents the object type for rows
contained in <paramref name="table"/>.</typeparam>
		/// <param name="table">Represents a table for a particular type in the
underlying database containing rows are to be updated.</param>
		/// <param name="entities">Represents the collection of items which are
to be updated in <paramref name="table"/>.</param>
		/// <param name="evaluator">A Lambda expression returning a
<typeparamref name="TEntity"/> that defines the update assignments to be performed on
each item in <paramref name="entities"/>.</param>
		/// <returns>Returns the Transact SQL string representation the LINQ
<see cref="IProvider"/> command text and parameters used that would be issued to
update all entities from the collection with a single update command.</returns>
		/// <remarks>This method is useful for debugging purposes or when LINQ
generated queries need to be passed to developers without LINQ/LINQPad.</remarks>
		public static string UpdateBatchSQL<TEntity>( this Table<TEntity> table,
IQueryable<TEntity> entities, Expression<Func<TEntity, TEntity>> evaluator ) where
TEntity : class
		{
			DbCommand update = table.GetUpdateBatchCommand<TEntity>(
entities, evaluator );
			return update.PreviewCommandText( true );
		}

		/// <summary>
		/// Returns the Transact SQL string representation the LINQ <see
cref="IProvider"/> command text and parameters used that would be issued to update
all entities from the collection with a single update command.
		/// </summary>
		/// <typeparam name="TEntity">Represents the object type for rows
contained in <paramref name="table"/>.</typeparam>
		/// <param name="table">Represents a table for a particular type in the
underlying database containing rows are to be updated.</param>
		/// <param name="filter">Represents a filter of items to be updated in
<paramref name="table"/>.</param>
		/// <param name="evaluator">A Lambda expression returning a
<typeparamref name="TEntity"/> that defines the update assignments to be performed on
each item in <paramref name="entities"/>.</param>
```

```
		/// <returns>Returns the Transact SQL string representation the LINQ 
<see cref="IProvider"/> command text and parameters used that would be issued to 
update all entities from the collection with a single update command.</returns>
		/// <remarks>This method is useful for debugging purposes or when LINQ 
generated queries need to be passed to developers without LINQ/LINQPad.</remarks>
		public static string UpdateBatchSQL<TEntity>( this Table<TEntity> table, 
Expression<Func<TEntity, bool>> filter, Expression<Func<TEntity, TEntity>> evaluator 
) where TEntity : class
		{
			return table.UpdateBatchSQL( table.Where( filter ), evaluator );
		}

		/// <summary>
		/// Immediately updates all entities in the collection with a single 
update command based on a <typeparamref name="TEntity"/> created from a Lambda 
expression.
		/// </summary>
		/// <typeparam name="TEntity">Represents the object type for rows 
contained in <paramref name="table"/>.</typeparam>
		/// <param name="table">Represents a table for a particular type in the 
underlying database containing rows are to be updated.</param>
		/// <param name="entities">Represents the collection of items which are 
to be updated in <paramref name="table"/>.</param>
		/// <param name="evaluator">A Lambda expression returning a 
<typeparamref name="TEntity"/> that defines the update assignments to be performed on 
each item in <paramref name="entities"/>.</param>
		/// <returns>The number of rows updated in the database.</returns>
		/// <remarks>
		/// <para>Similiar to stored procedures, and opposite from similiar 
InsertAllOnSubmit, rows provided in <paramref name="entities"/> will be updated 
immediately with no need to call <see cref="DataContext.SubmitChanges()"/>.</para>
		/// <para>Additionally, to improve performance, instead of creating an 
update command for each item in <paramref name="entities"/>, a single update command 
is created.</para>
		/// </remarks>
		public static int UpdateBatch<TEntity>( this Table<TEntity> table, 
IQueryable<TEntity> entities, Expression<Func<TEntity, TEntity>> evaluator ) where 
TEntity : class
		{
			DbCommand update = table.GetUpdateBatchCommand<TEntity>( 
entities, evaluator );

			var parameters = from p in update.Parameters.Cast<DbParameter>()
							 select p.Value;
			return table.Context.ExecuteCommand( update.CommandText, 
parameters.ToArray() );
		}

		/// <summary>
		/// Immediately updates all entities in the collection with a single 
update command based on a <typeparamref name="TEntity"/> created from a Lambda 
expression.
		/// </summary>
		/// <typeparam name="TEntity">Represents the object type for rows 
contained in <paramref name="table"/>.</typeparam>
```

```
		/// <param name="table">Represents a table for a particular type in the 
underlying database containing rows are to be updated.</param>
		/// <param name="filter">Represents a filter of items to be updated in 
<paramref name="table"/>.</param>
		/// <param name="evaluator">A Lambda expression returning a 
<typeparamref name="TEntity"/> that defines the update assignments to be performed on 
each item in <paramref name="entities"/>.</param>
		/// <returns>The number of rows updated in the database.</returns>
		/// <remarks>
		/// <para>Similiar to stored procedures, and opposite from similiar 
InsertAllOnSubmit, rows provided in <paramref name="entities"/> will be updated 
immediately with no need to call <see cref="DataContext.SubmitChanges()"/>.</para>
		/// <para>Additionally, to improve performance, instead of creating an 
update command for each item in <paramref name="entities"/>, a single update command 
is created.</para>
		/// </remarks>
		public static int UpdateBatch<TEntity>( this Table<TEntity> table, 
Expression<Func<TEntity, bool>> filter, Expression<Func<TEntity, TEntity>> evaluator 
) where TEntity : class
		{
			return table.UpdateBatch( table.Where( filter ), evaluator );
		}

		/// <summary>
		/// Returns the Transact SQL string representation the LINQ <see 
cref="IProvider"/> command text and parameters used that would be issued to perform 
the query's select statement.
		/// </summary>
		/// <param name="context">The DataContext to execute the batch select 
against.</param>
		/// <param name="query">Represents the SELECT query to execute.</param>
		/// <returns>Returns the Transact SQL string representation of the <see 
cref="DbCommand.CommandText"/> along with <see cref="DbCommand.Parameters"/> if 
present.</returns>
		/// <remarks>This method is useful for debugging purposes or when used 
in other utilities such as LINQPad.</remarks>
		public static string PreviewSQL( this DataContext context, IQueryable 
query )
		{
			var cmd = context.GetCommand( query );
			return cmd.PreviewCommandText( true );
		}

		/// <summary>
		/// Returns a string representation the LINQ <see cref="IProvider"/> 
command text and parameters used that would be issued to perform the query's select 
statement.
		/// </summary>
		/// <param name="context">The DataContext to execute the batch select 
against.</param>
		/// <param name="query">Represents the SELECT query to execute.</param>
		/// <returns>Returns a string representation of the <see 
cref="DbCommand.CommandText"/> along with <see cref="DbCommand.Parameters"/> if 
present.</returns>
		/// <remarks>This method is useful for debugging purposes or when used
```

```
in other utilities such as LINQPad.</remarks>
		public static string PreviewCommandText( this DataContext context,
IQueryable query )
		{
			var cmd = context.GetCommand( query );
			return cmd.PreviewCommandText( false);
		}

		/// <summary>
		/// Returns a string representation of the <see
cref="DbCommand.CommandText"/> along with <see cref="DbCommand.Parameters"/> if
present.
		/// </summary>
		/// <param name="cmd">The <see cref="DbCommand"/> to analyze.</param>
		/// <param name="forTransactSQL">Whether or not the text should be
formatted as 'logging' similiar to LINQ to SQL output, or in valid Transact SQL
syntax ready for use with a 'query analyzer' type tool.</param>
		/// <returns>Returns a string representation of the <see
cref="DbCommand.CommandText"/> along with <see cref="DbCommand.Parameters"/> if
present.</returns>
		/// <remarks>This method is useful for debugging purposes or when used
in other utilities such as LINQPad.</remarks>
		private static string PreviewCommandText( this DbCommand cmd, bool
forTransactSQL )
		{
			var output = new StringBuilder();

			if ( !forTransactSQL ) output.AppendLine( cmd.CommandText );

			foreach ( DbParameter parameter in cmd.Parameters )
			{
				int num = 0;
				int num2 = 0;
				PropertyInfo property = parameter.GetType().GetProperty(
"Precision" );
				if ( property != null )
				{
					num = (int)Convert.ChangeType( property.GetValue(
parameter, null ), typeof( int ), CultureInfo.InvariantCulture );
				}
				PropertyInfo info2 = parameter.GetType().GetProperty(
"Scale" );
				if ( info2 != null )
				{
					num2 = (int)Convert.ChangeType( info2.GetValue(
parameter, null ), typeof( int ), CultureInfo.InvariantCulture );
				}
				SqlParameter parameter2 = parameter as SqlParameter;

				if ( forTransactSQL )
				{
					output.AppendFormat( "DECLARE {0} {1}{2}; SET {0} =
{3}\r\n",
						new object[] {
							parameter.ParameterName,
                                                    ( parameter2 == null ) ?
parameter.DbType.ToString() : parameter2.SqlDbType.ToString(),
                                                    ( parameter.Size > 0 ) ? "( " +
parameter.Size.ToString( CultureInfo.CurrentCulture ) + " )" : "",
                                                    GetParameterTransactValue(
parameter, parameter2 ) } );
                          }
                          else
                          {
                                output.AppendFormat( "-- {0}: {1} {2} (Size = {3};
Prec = {4}; Scale = {5}) [{6}]\r\n", new object[] { parameter.ParameterName,
parameter.Direction, ( parameter2 == null ) ? parameter.DbType.ToString() :
parameter2.SqlDbType.ToString(), parameter.Size.ToString( CultureInfo.CurrentCulture
), num, num2, ( parameter2 == null ) ? parameter.Value : parameter2.SqlValue } );
                          }
                    }

                    if ( forTransactSQL ) output.Append( "\r\n" + cmd.CommandText );

                    return output.ToString();
             }

             private static string GetParameterTransactValue( DbParameter parameter,
SqlParameter parameter2 )
             {
                    if ( parameter2 == null ) return parameter.Value.ToString(); //
Not going to deal with NON SQL parameters.

                    switch( parameter2.SqlDbType )
                    {
                          case SqlDbType.Char:
                          case SqlDbType.Date:
                          case SqlDbType.DateTime:
                          case SqlDbType.DateTime2:
                          case SqlDbType.NChar:
                          case SqlDbType.NText:
                          case SqlDbType.NVarChar:
                          case SqlDbType.SmallDateTime:
                          case SqlDbType.Text:
                          case SqlDbType.VarChar:
                          case SqlDbType.UniqueIdentifier:
                                return string.Format( "'{0}'", parameter2.SqlValue
);

                          default:
                                return parameter2.SqlValue.ToString();
                    }
             }

             private static DbCommand GetDeleteBatchCommand<TEntity>( this
Table<TEntity> table, IQueryable<TEntity> entities ) where TEntity : class
             {
                    var deleteCommand = table.Context.GetCommand( entities );
                    deleteCommand.CommandText = string.Format( "DELETE {0}\r\n",
table.GetDbName() ) + GetBatchJoinQuery<TEntity>( table, entities );
```

```
                        return deleteCommand;
                }

                private static DbCommand GetUpdateBatchCommand<TEntity>( this
Table<TEntity> table, IQueryable<TEntity> entities, Expression<Func<TEntity,
TEntity>> evaluator ) where TEntity : class
                {
                        var updateCommand = table.Context.GetCommand( entities );

                        var setSB = new StringBuilder();
                        int memberInitCount = 1;

                        // Process the MemberInitExpression (there should only be one in
the evaluator Lambda) to convert the expression tree
                        // into a valid DbCommand.  The Visit<> method will only process
expressions of type MemberInitExpression and requires
                        // that a MemberInitExpression be returned - in our case we'll
return the same one we are passed since we are building
                        // a DbCommand and not 'really using' the evaluator Lambda.
                        evaluator.Visit<MemberInitExpression>( delegate(
MemberInitExpression expression )
                        {
                                if ( memberInitCount > 1 )
                                {
                                        throw new NotImplementedException( "Currently only
one MemberInitExpression is allowed for the evaluator parameter." );
                                }
                                memberInitCount++;

                                setSB.Append( GetDbSetStatement<TEntity>( expression,
table, updateCommand ) );

                                return expression; // just return passed in expression to
keep 'visitor' happy.
                        } );

                        // Complete the command text by concatenating bits together.
                        updateCommand.CommandText = string.Format( "UPDATE
{0}\r\n{1}\r\n\r\n{2}",

                table.GetDbName(),
        // Database table name

                setSB.ToString(),
        // SET fld = {}, fld2 = {}, ...

                GetBatchJoinQuery<TEntity>( table, entities ) );     // Subquery join
created from entities command text

                        if ( updateCommand.CommandText.IndexOf( "[arg0]" ) >= 0 ||
updateCommand.CommandText.IndexOf( "NULL AS [EMPTY]" ) >= 0 )
                        {
                                // TODO (Chris): Probably a better way to determine this
by using an visitor on the expression before the
                                //                         var selectExpression =
Expression.Call... method call (search for that) and see which funcitons
                    //                         are being used and determine if
supported by LINQ to SQL
                    throw new NotSupportedException( string.Format( "The
evaluator Expression<Func<{0},{0}>> has processing that needs to be performed once
the query is returned (i.e. string.Format()) and therefore can not be used during
batch updating.", table.GetType() ) );
                }

                return updateCommand;
            }

            private static string GetDbSetStatement<TEntity>( MemberInitExpression
memberInitExpression, Table<TEntity> table, DbCommand updateCommand ) where TEntity :
class
            {
                var entityType = typeof( TEntity );

                if ( memberInitExpression.Type != entityType )
                {
                    throw new NotImplementedException( string.Format( "The
MemberInitExpression is intializing a class of the incorrect type '{0}' and it should
be '{1}'.", memberInitExpression.Type, entityType ) );
                }

                var setSB = new StringBuilder();

                var tableName = table.GetDbName();
                var metaTable = table.Context.Mapping.GetTable( entityType );
                // Used to look up actual field names when MemberAssignment is a
constant,
                // need both the Name (matches the property name on LINQ object)
and the
                // MappedName (db field name).
                var dbCols = from mdm in metaTable.RowType.DataMembers
                             select new { mdm.MappedName, mdm.Name };

                // Walk all the expression bindings and generate SQL 'commands'
from them.  Each binding represents a property assignment
                // on the TEntity object initializer Lambda expression.
                foreach ( var binding in memberInitExpression.Bindings )
                {
                    var assignment = binding as MemberAssignment;

                    if ( binding == null )
                    {
                        throw new NotImplementedException( "All bindings
inside the MemberInitExpression are expected to be of type MemberAssignment." );
                    }

                    // TODO (Document): What is this doing?  I know it's
grabbing existing parameter to pass into Expression.Call() but explain 'why'
                    //                            I assume it has something
to do with fact we can't just access the parameters of assignment.Expression?
                    //                            Also, any concerns of
```

```
whether or not if there are two params of type entity type?
                            ParameterExpression entityParam = null;
                            assignment.Expression.Visit<ParameterExpression>(
delegate( ParameterExpression p ) { if ( p.Type == entityType ) entityParam = p;
return p; } );

                            // Get the real database field name.  binding.Member.Name
is the 'property' name of the LINQ object
                            // so I match that to the Name property of the table
mapping DataMembers.
                            string name = binding.Member.Name;
                            var dbCol = ( from c in dbCols
                                          where c.Name == name
                                          select c ).FirstOrDefault();

                            if ( dbCol == null )
                            {
                                   throw new ArgumentOutOfRangeException( name,
string.Format( "The corresponding field on the {0} table could not be found.",
tableName ) );
                            }

                            // If entityParam is NULL, then no references to other
columns on the TEntity row and need to eval 'constant' value...
                            if ( entityParam == null )
                            {
                                   // Compile and invoke the assignment expression to
obtain the contant value to add as a parameter.
                                   var constant = Expression.Lambda(
assignment.Expression, null ).Compile().DynamicInvoke();

                                   // use the MappedName from the table mapping
DataMembers - that is field name in DB table.
                                   if ( constant == null )
                                   {
                                          setSB.AppendFormat( "[{0}] = null, ",
dbCol.MappedName );
                                   }
                                   else
                                   {
                                          // Add new parameter with massaged name to
avoid clashes.
                                          setSB.AppendFormat( "[{0}] = @p{1}, ",
dbCol.MappedName, updateCommand.Parameters.Count );
                                          updateCommand.Parameters.Add( new
SqlParameter( string.Format( "@p{0}", updateCommand.Parameters.Count ), constant ) );
                                   }
                            }
                            else
                            {
                                   // TODO (Documentation): Explain what we are doing
here again, I remember you telling me why we have to call but I can't remember now.
                                   // Wny are we calling Expression.Call and what are
we passing it?  Below comments are just 'made up' and probably wrong.
```

```
                                // Create a MethodCallExpression which represents a
'simple' select of *only* the assignment part (right hand operator) of
                                // of the MemberInitExpression.MemberAssignment so
that we can let the Linq Provider do all the 'sql syntax' generation for
                                // us.
                                //
                                // For Example: TEntity.Property1 =
TEntity.Property1 + " Hello"
                                // This selectExpression will be only dealing with
TEntity.Property1 + " Hello"
                                var selectExpression = Expression.Call(

      typeof( Queryable ),

      "Select",
                                                                              new
Type[] { entityType, assignment.Expression.Type },

                                // TODO (Documentation): How do we know there are
only 'two' parameters?  And what is Expression.Lambda
                                //                                      doing?  I
assume it's returning a type of assignment.Expression.Type to match above?


      Expression.Constant( table ),

      Expression.Lambda( assignment.Expression, entityParam ) );

                                setSB.AppendFormat( "[{0}] = {1}, ",

      dbCol.MappedName,

      GetDbSetAssignment( table, selectExpression, updateCommand, name ) );
                          }
                    }

                    var setStatements = setSB.ToString();
                    return "SET " + setStatements.Substring( 0, setStatements.Length
- 2 ); // remove ', '
             }

             /// <summary>
             /// Some LINQ Query syntax is invalid because SQL (or whomever the
provider is) can not translate it to its native language.
             /// DataContext.GetCommand() does not detect this, only
IProvider.Execute or IProvider.Compile call the necessary code to
             /// check this.  This function invokes the IProvider.Compile to make
sure the provider can translate the expression.
             /// </summary>
             /// <remarks>
             /// An example of a LINQ query that previously 'worked' in the *Batch
methods but needs to throw an exception is something
             /// like the following:
             ///
             /// var pay =
             ///         from h in HistoryData
             ///         where h.his.Groups.gName == "Ochsner" && h.hisType ==
"pay"
             ///         select h;
             ///
             /// HistoryData.UpdateBatchPreview( pay, h => new HistoryData { hisIndex
= ( int.Parse( h.hisIndex ) - 1 ).ToString() } ).Dump();
             ///
             /// The int.Parse is not valid and needs to throw an exception like:
             ///
             ///         Could not translate expression '(Parse(p.hisIndex) -
1).ToString()' into SQL and could not treat it as a local expression.
             ///
             ///    Unfortunately, the IProvider.Compile is internal and I need to
use Reflection to call it (ugh).  I've several e-mails sent into
             ///    MS LINQ team members and am waiting for a response and will
correct/improve code as soon as possible.
             /// </remarks>
             private static void ValidateExpression( ITable table, Expression
expression )
             {
                   var context = table.Context;
                   PropertyInfo providerProperty = context.GetType().GetProperty(
"Provider", BindingFlags.Instance | BindingFlags.NonPublic );
                   var provider = providerProperty.GetValue( context, null );
                   var compileMI = provider.GetType().GetMethod(
"System.Data.Linq.Provider.IProvider.Compile", BindingFlags.Instance |
BindingFlags.NonPublic );

                   // Simply compile the expression to see if it will work.
                   compileMI.Invoke( provider, new object[] { expression } );
             }

             private static string GetDbSetAssignment( ITable table,
MethodCallExpression selectExpression, DbCommand updateCommand, string bindingName )
             {
                   ValidateExpression( table, selectExpression );

                   // Convert the selectExpression into an IQueryable query so that
I can get the CommandText
                   var selectQuery = ( table as IQueryable ).Provider.CreateQuery(
selectExpression );

                   // Get the DbCommand so I can grab relavent parts of CommandText
to construct a field
                   // assignment and based on the 'current TEntity row'.
Additionally need to massage parameter
                   // names from temporary command when adding to the final update
command.
                   var selectCmd = table.Context.GetCommand( selectQuery );
                   var selectStmt = selectCmd.CommandText;
                   selectStmt = selectStmt.Substring( 7,
                          // Remove 'SELECT ' from front ( 7 )
                                                              selectStmt.IndexOf(
"\r\nFROM " ) - 7 )       // Return only the selection field expression
```

```
                                                            .Replace( "[t0].", "" )
                                        // Remove table alias from the select
                                                            .Replace( " AS [value]",
"" )                                // If the select is not a direct field (constant or
expression), remove the field alias
                                                            .Replace( "@p", "@p" +
bindingName );                      // Replace parameter name so doesn't conflict with
existing ones.

                    foreach ( var selectParam in
selectCmd.Parameters.Cast<DbParameter>() )
                    {
                          var paramName = string.Format( "@p{0}",
updateCommand.Parameters.Count );

                          // DataContext.ExecuteCommand ultimately just takes a
object array of parameters and names them p0-N.
                          // So I need to now do replaces on the massaged value to
get it in proper format.
                          selectStmt = selectStmt.Replace(

      selectParam.ParameterName.Replace( "@p", "@p" + bindingName ),
                                                            paramName );

                          updateCommand.Parameters.Add( new SqlParameter( paramName,
selectParam.Value ) );
                    }

                    return selectStmt;
             }

             private static string GetBatchJoinQuery<TEntity>( Table<TEntity> table,
IQueryable<TEntity> entities ) where TEntity : class
             {
                    var metaTable = table.Context.Mapping.GetTable( typeof( TEntity )
);

                    var keys = from mdm in metaTable.RowType.DataMembers
                                where mdm.IsPrimaryKey
                                select new { mdm.MappedName };

                    var joinSB = new StringBuilder();
                    var subSelectSB = new StringBuilder();

                    foreach ( var key in keys )
                    {
                          joinSB.AppendFormat( "j0.[{0}] = j1.[{0}] AND ",
key.MappedName );
                          // For now, always assuming table is aliased as t0.
Should probably improve at some point.
                          // Just writing a smaller sub-select so it doesn't get all
the columns of data, but instead
                          // only the primary key fields used for joining.
                          subSelectSB.AppendFormat( "[t0].[{0}], ", key.MappedName
);
```

```
                    }

                    var selectCommand = table.Context.GetCommand( entities );
                    var select = selectCommand.CommandText;

                    var join = joinSB.ToString();

                    if ( join == "" )
                    {
                          throw new MissingPrimaryKeyException( string.Format( "{0}
does not have a primary key defined.  Batch updating/deleting can not be used for
tables without a primary key.", metaTable.TableName ) );
                    }

                    join = join.Substring( 0, join.Length - 5 );
                                                // Remove last ' AND '
                    #region - Better ExpressionTree Handling Needed -
                    /*

                    Below is a sample query where the let statement was used to
simply the 'where clause'.  However, it produced an extra level
                    in the query.

                    var manage =
                          from u in User
                          join g in Groups on u.User_Group_id equals g.gKey into
groups
                          from g in groups.DefaultIfEmpty()
                          let correctGroup = groupsToManage.Contains( g.gName ) || (
groupsToManage.Contains( "_GLOBAL" ) && g.gKey == null )
                          where correctGroup && ( users.Contains(
u.User_Authenticate_id ) || userEmails.Contains( u.User_Email ) ) ||
userKeys.Contains( u.User_id )
                          select u;

                    Produces this SQL:
                    SELECT [t2].[User_id] AS [uKey], [t2].[User_Authenticate_id] AS
[uAuthID], [t2].[User_Email] AS [uEmail], [t2].[User_Pin] AS [uPin],
[t2].[User_Active] AS [uActive], [t2].[uAdminAuthID], [t2].[uFailureCount]
                    FROM (
                          SELECT [t0].[User_id], [t0].[User_Authenticate_id],
[t0].[User_Email], [t0].[User_Pin], [t0].[User_Active], [t0].[uFailureCount],
[t0].[uAdminAuthID],
                                (CASE
                                      WHEN [t1].[gName] IN (@p0) THEN 1
                                      WHEN NOT ([t1].[gName] IN (@p0)) THEN 0
                                      ELSE NULL
                                 END) AS [value]
                          FROM [User] AS [t0]
                          LEFT OUTER JOIN [Groups] AS [t1] ON [t0].[User_Group_id] =
([t1].[gKey])
                          ) AS [t2]
                    WHERE (([t2].[value] = 1) AND (([t2].[User_Authenticate_id] IN
(@p1)) OR ([t2].[User_Email] IN (@p2)))) OR ([t2].[User_id] IN (@p3))
```

```
                    If I put the entire where in one line...
                    where  ( groupsToManage.Contains( g.gName ) || (
groupsToManage.Contains( "_GLOBAL" ) && g.gKey == null ) ) &&
                                 ( users.Contains( u.User_Authenticate_id ) ||
userEmails.Contains( u.User_Email ) ) || userKeys.Contains ( u.User_id )

                    I get this SQL:
                    SELECT [t0].[User_id] AS [uKey], [t0].[User_Authenticate_id] AS
[uAuthID], [t0].[User_Email] AS [uEmail], [t0].[User_Pin] AS [uPin],
[t0].[User_Active] AS [uActive], [t0].[uAdminAuthID], [t0].[uFailureCount]
                    FROM [User] AS [t0]
                    LEFT OUTER JOIN [Groups] AS [t1] ON [t0].[User_Group_id] =
([t1].[gKey])
                    WHERE (([t1].[gName] IN (@p0)) AND (([t0].[User_Authenticate_id]
IN (@p1)) OR ([t0].[User_Email] IN (@p2)))) OR ([t0].[User_id] IN (@p3))

                    The second 'cleaner' SQL worked with my original 'string parsing'
of simply looking for [t0] and stripping everything before it
                    to get rid of the SELECT and any 'TOP' clause if present.  But
the first SQL introduced a layer which caused [t2] to be used.  So
                    I have to do a bit different string parsing.  There is probably a
more efficient way to examine the ExpressionTree and figure out
                    if something like this is going to happen.  I will explore it
later.
                    */
                    #endregion
                    var endSelect = select.IndexOf( "[t" );
                                                                 // Get 'SELECT ' and any TOP
clause if present
                    var selectClause = select.Substring( 0, endSelect );
                    var selectTableNameStart = endSelect + 1;
                                                                 // Get the table name LINQ to
SQL used in query generation
                    var selectTableName = select.Substring( selectTableNameStart,
                                                    // because I have to replace [t0] with it in
the subSelectSB
                                                                                 select.IndexOf(
"]", selectTableNameStart ) - ( selectTableNameStart ) );

                    // TODO (Chris): I think instead of searching for ORDER BY in the
entire select statement, I should examine the ExpressionTree and see
                    // if the *outer* select (in case there are nested subselects)
has an orderby clause applied to it.
                    var needsTopClause = selectClause.IndexOf( " TOP " ) < 0 &&
select.IndexOf( "\r\nORDER BY " ) > 0;

                    var subSelect = selectClause
                                                                 + ( needsTopClause ? "TOP 100
PERCENT " : "" )                                                       // If order by in original
select without TOP clause, need TOP
                                                                 + subSelectSB.ToString()
                                                                        // Append just the
primary keys.
                                                                                       .Replace(
"[t0]", string.Format( "[{0}]", selectTableName ) );
```

```
                    subSelect = subSelect.Substring( 0, subSelect.Length - 2 );
                                                             // Remove last ', '

                    subSelect += select.Substring( select.IndexOf( "\r\nFROM " ) );
// Create a sub SELECT that *only* includes the primary key fields

                    var batchJoin = String.Format( "FROM {0} AS j0 INNER JOIN
(\r\n\r\n{1}\r\n\r\n) AS j1 ON ({2})\r\n", table.GetDbName(), subSelect, join );
                    return batchJoin;
            }

            private static string GetDbName<TEntity>( this Table<TEntity> table )
where TEntity : class
            {
                    var entityType = typeof( TEntity );
                    var metaTable = table.Context.Mapping.GetTable( entityType );
                    var tableName = metaTable.TableName;

                    if ( !tableName.StartsWith( "[" ) )
                    {
                            string[] parts = tableName.Split( '.' );
                            tableName = string.Format( "[{0}]", string.Join( "].[",
parts ) );
                    }

                    return tableName;
            }

            private static string GetLog( this DataContext context )
            {
                    PropertyInfo providerProperty = context.GetType().GetProperty(
"Provider", BindingFlags.Instance | BindingFlags.NonPublic );
                    var provider = providerProperty.GetValue( context, null );
                    Type providerType = provider.GetType();

                    PropertyInfo modeProperty = providerType.GetProperty( "Mode",
BindingFlags.Instance | BindingFlags.NonPublic );
                    FieldInfo servicesField = providerType.GetField( "services",
BindingFlags.Instance | BindingFlags.NonPublic );
                    object services = servicesField != null ? servicesField.GetValue(
provider ) : null;
                    PropertyInfo modelProperty = services != null ?
services.GetType().GetProperty( "Model", BindingFlags.Instance |
BindingFlags.NonPublic | BindingFlags.Public | BindingFlags.GetProperty ) : null;

                    return string.Format( "-- Context: {0}({1}) Model: {2} Build:
{3}\r\n",
                                                    providerType.Name,
                                                    modeProperty != null ?
modeProperty.GetValue( provider, null ) : "unknown",
                                                    modelProperty != null ?
modelProperty.GetValue( services, null ).GetType().Name : "unknown",
                                                    "3.5.21022.8" );
            }
            /// <summary>
            /// Returns a list of changed items inside the Context before being
applied to the data store.
            /// </summary>
            /// <param name="context">The DataContext to interogate for pending
changes.</param>
            /// <returns>Returns a list of changed items inside the Context before
being applied to the data store.</returns>
            /// <remarks>Based on Ryan Haney's code at
http://dotnetslackers.com/articles/csharp/GettingChangedEntitiesFromLINQToSQLDataCont
ext.aspx.  Note that this code relies on reflection of private fields and
members.</remarks>
            public static List<ChangedItems<TEntity>> GetChangedItems<TEntity>( this
DataContext context )
            {
                  // create a dictionary of type TItem for return to caller
                  List<ChangedItems<TEntity>> changedItems = new
List<ChangedItems<TEntity>>();

                  PropertyInfo providerProperty = context.GetType().GetProperty(
"Provider", BindingFlags.Instance | BindingFlags.NonPublic );
                  var provider = providerProperty.GetValue( context, null );
                  Type providerType = provider.GetType();

                  // use reflection to get changed items from data context
                  object services = providerType.GetField( "services",
                    BindingFlags.NonPublic |
                    BindingFlags.Instance |
                    BindingFlags.GetField ).GetValue( provider );

                  object tracker = services.GetType().GetField( "tracker",
                    BindingFlags.NonPublic |
                    BindingFlags.Instance |
                    BindingFlags.GetField ).GetValue( services );

                  System.Collections.IDictionary trackerItems =
                    (System.Collections.IDictionary)tracker.GetType().GetField(
"items",
                    BindingFlags.NonPublic |
                    BindingFlags.Instance |
                    BindingFlags.GetField ).GetValue( tracker );

                  // iterate through each item in context, adding
                  // only those that are of type TItem to the changedItems
dictionary
                  foreach ( System.Collections.DictionaryEntry entry in
trackerItems )
                  {
                        object original = entry.Value.GetType().GetField(
"original",
                                                           BindingFlags.NonPublic |
                                                           BindingFlags.Instance |
                                                           BindingFlags.GetField
).GetValue( entry.Value );
```

```
                            if ( entry.Key is TEntity && original is TEntity )
                            {
                                   changedItems.Add(
                                     new ChangedItems<TEntity>( (TEntity)entry.Key,
(TEntity)original )
                                   );
                            }
                     }
                     return changedItems;
              }
       }

       public class ChangedItems<TEntity>
       {
              public ChangedItems( TEntity Current, TEntity Original )
              {
                     this.Current = Current;
                     this.Original = Original;
              }
              public TEntity Current { get; set; }
              public TEntity Original { get; set; }
       }
}
```

Appendix B: Benchmark Results

```
12:02:50 AM 1. SortVerifyEquivalence: True
12:02:50 AM SortTraditional
71
71
75
71
72
12:03:06 AM SortLINQ
70
72
70
69
77

12:03:21 AM 2. SortComplexVerifyEquivalence: True
12:03:21 AM SortComplexTraditional
82
80
80
80
82
12:03:37 AM SortComplexLINQ
82
81
81
79
86

12:03:52 AM 3. DistinctVerifyEquivalence: True
12:03:52 AM DistinctTraditional
67
68
70
67
66
12:04:08 AM DistinctLINQ
69
68
```

```
66
66
65

12:04:23 AM 4. ExceptVerifyEquivalence: True
12:04:23 AM ExceptTraditional
142
140
142
141
142
12:04:39 AM ExceptLINQ
132
131
134
131
133

12:04:55 AM 5. WhereVerifyEquivalence: True
12:04:55 AM WhereTraditional
373
381
376
390
377
12:05:12 AM WhereLINQ
375
384
369
373
382

12:05:29 AM 6. AllVerifyEquivalence: True
12:05:29 AM AllTraditional
374
389
368
380
382
12:05:46 AM AllLINQ
374
390
370
370
378
```

```
12:06:03 AM 7. AnyVerifyEquivalence: True
12:06:03 AM AnyTraditional
380
379
374
374
379
12:06:20 AM AnyLINQ
390
371
384
373
385

12:06:37 AM 8. SelectVerifyEquivalence: True
12:06:37 AM SelectTraditional
374
389
373
376
396
12:06:54 AM SelectLINQ
379
385
369
369
388

12:07:20 AM 9. SelectManyVerifyEquivalence: True
12:07:20 AM SelectManyTraditional
3828
3981
3985
4022
3981
12:07:55 AM SelectManyLINQ
6773
6717
6821
6710
6994

12:08:45 AM 10. SkipVerifyEquivalence: True
12:08:45 AM SkipTraditional
385
376
```

```
371
365
387
12:09:01 AM SkipLINQ
372
383
372
374
383

12:09:19 AM 11. SkipWhileVerifyEquivalence: True
12:09:19 AM SkipWhileTraditional
383
369
371
382
371
12:09:36 AM SkipWhileLINQ
382
373
374
388
375

12:09:53 AM 12. TakeVerifyEquivalence: True
12:09:53 AM TakeTraditional
381
370
375
374
382
12:10:10 AM TakeLINQ
382
373
370
376
373

12:10:27 AM 13. TakeWhileVerifyEquivalence: True
12:10:27 AM TakeWhileTraditional
369
390
372
367
386
12:10:44 AM TakeWhileLINQ
```

```
384
390
379
378
370

12:11:01 AM 14. MaxVerifyEquivalence: True
12:11:01 AM MaxTraditional
72
73
73
73
70
12:11:16 AM MaxLINQ
73
71
71
73
73

12:11:32 AM 15. MinVerifyEquivalence: True
12:11:32 AM MinTraditional
74
73
70
72
70
12:11:47 AM MinLINQ
72
72
71
71
71

12:12:03 AM 16. CountVerifyEquivalence: True
12:12:03 AM CountTraditional
72
70
70
70
71
12:12:18 AM CountLINQ
71
72
70
71
```

```
70

12:12:34 AM 17. AverageVerifyEquivalence: True
12:12:34 AM AverageTraditional
383
379
382
370
370
12:12:51 AM AverageLINQ
383
376
383
386
379

12:13:08 AM 18. JoinVerifyEquivalence: True
12:13:08 AM JoinTraditional
384
407
390
387
400
12:13:25 AM JoinLINQ
402
385
411
383
385

12:13:43 AM 19. SequenceEqualTrueVerifyEquivalence: True
12:13:43 AM SequenceEqualTrueTraditional
383
390
374
378
388
12:14:00 AM SequenceEqualTrueLINQ
384
387
373
375
380

12:14:17 AM 20. SequenceEqualVerifyEquivalence: True
12:14:17 AM SequenceEqualTraditional
```

```
762
759
760
757
740
12:14:36 AM SequenceEqualLINQ
765
737
746
789
768

12:14:55 AM 21. ElementAtVerifyEquivalence: True
12:14:55 AM ElementAtTraditional
384
375
384
486
379
12:15:12 AM ElementAtLINQ
387
372
372
380
371

12:15:30 AM 22. ToArrayVerifyEquivalence: True
12:15:30 AM ToArrayTraditional
379
375
372
371
380
12:15:46 AM ToArrayLINQ
381
376
380
377
386

12:16:04 AM 23. ToDictionaryVerifyEquivalence: True
12:16:04 AM ToDictionary
385
377
396
371
```

```
387
12:16:21 AM ToDictionaryLINQ
375
387
374
374
391

12:16:38 AM 24. ToListVerifyEquivalence: True
12:16:38 AM ToList
14
13
13
13
12
12:16:53 AM ToListLINQ
12
12
14
13
12

12:17:08 AM 25. ConcatenateVerifyEquivalence: True
12:17:08 AM ConcatenateTraditional
390
371
379
395
383
12:17:25 AM ConcatenateLINQ
399
378
371
387
374

12:17:42 AM 26. AggregateVerifyEquivalence: True
12:17:42 AM AggregateTraditional
395
380
389
386
380
12:17:59 AM AggregateLINQ
382
386
```

```
378
382
400

*** Warmup complete: 5 iterations ***


12:18:17 AM 1. SortVerifyEquivalence: True
12:18:17 AM SortTraditional
75
69
69
73
71
71
71
69
69
69
69
70
72
72
69
69
12:19:06 AM SortLINQ
71
69
71
70
69
69
69
73
71
74
73
71
70
71
69
69

12:19:55 AM 2. SortComplexVerifyEquivalence: True
12:19:55 AM SortComplexTraditional
82
```

```
78
79
81
82
78
78
80
83
79
81
78
79
79
81
82
12:20:44 AM SortComplexLINQ
79
79
81
81
79
83
80
82
79
79
79
79
82
81
80
79

12:21:34 AM 3. DistinctVerifyEquivalence: True
12:21:34 AM DistinctTraditional
67
68
66
69
68
66
66
66
67
67
66
```

```
66
66
66
67
66
12:22:23 AM DistinctLINQ
69
66
66
68
67
66
65
67
68
69
66
66
65
65
65
66

12:23:12 AM 4. ExceptVerifyEquivalence: True
12:23:12 AM ExceptTraditional
141
138
142
142
141
144
140
139
146
142
140
139
140
140
140
142
12:24:02 AM ExceptLINQ
135
132
131
131
```

```
132
136
131
131
135
133
135
133
133
132
143
136

12:24:53 AM 5. WhereVerifyEquivalence: True
12:24:53 AM WhereTraditional
380
375
391
373
371
393
392
376
389
371
373
395
379
370
390
371
12:25:47 AM WhereLINQ
377
374
380
371
385
380
370
390
377
374
388
369
395
377
```

```
379
372

12:26:41 AM 6. AllVerifyEquivalence: True
12:26:41 AM AllTraditional
381
370
401
382
386
381
383
381
369
389
370
390
373
372
391
380
12:27:36 AM AllLINQ
372
389
374
374
374
381
370
370
385
373
375
386
391
372
378
389

12:28:30 AM 7. AnyVerifyEquivalence: True
12:28:30 AM AnyTraditional
372
387
373
371
379
```

```
374
387
367
390
388
378
383
369
387
369
372
12:29:24 AM AnyLINQ
373
384
371
369
387
373
370
382
376
373
381
380
370
378
385
373

12:30:19 AM 8. SelectVerifyEquivalence: True
12:30:19 AM SelectTraditional
380
400
375
380
374
375
388
377
379
386
384
389
382
390
386
```

```
390
12:31:13 AM SelectLINQ
386
374
385
371
388
378
384
389
388
371
385
372
380
389
376
376

12:32:16 AM 9. SelectManyVerifyEquivalence: True
12:32:16 AM SelectManyTraditional
3855
3958
4010
4045
3964
3916
4012
4137
3878
3884
3966
3962
3833
3895
4016
3969
12:34:07 AM SelectManyLINQ
6797
6882
6790
6922
6782
6688
6830
6991
```

```
6864
6700
6645
6846
6603
6761
6843
6886

12:36:44 AM 10. SkipVerifyEquivalence: True
12:36:45 AM SkipTraditional
370
385
376
384
386
380
376
385
389
374
374
384
370
370
386
384
12:37:39 AM SkipLINQ
386
377
373
371
387
373
397
372
375
378
377
378
385
380
372
386

12:38:33 AM 11. SkipWhileVerifyEquivalence: True
```

```
12:38:33 AM SkipWhileTraditional
374
398
385
374
396
373
375
378
374
373
393
370
379
384
396
377
12:39:27 AM SkipWhileLINQ
383
376
371
395
375
383
373
373
380
382
377
388
376
373
381
377

12:40:22 AM 12. TakeVerifyEquivalence: True
12:40:22 AM TakeTraditional
380
378
381
378
388
383
375
403
369
```

```
371
375
376
371
372
378
392
12:41:16 AM TakeLINQ
385
379
371
382
372
369
382
380
375
401
376
384
371
398
377
373

12:42:10 AM 13. TakeWhileVerifyEquivalence: True
12:42:10 AM TakeWhileTraditional
377
371
398
371
373
385
369
374
380
372
391
370
395
393
374
372
12:43:04 AM TakeWhileLINQ
385
375
```

```
378
374
391
371
390
373
375
377
385
381
386
377
371
381

12:43:59 AM 14. MaxVerifyEquivalence: True
12:43:59 AM MaxTraditional
72
78
73
71
72
73
70
70
70
70
70
70
73
72
71
70
12:44:48 AM MaxLINQ
72
73
70
70
71
70
72
70
81
71
70
70
```

```
71
73
72
73

12:45:37 AM 15. MinVerifyEquivalence: True
12:45:37 AM MinTraditional
71
71
70
72
70
71
70
70
70
70
70
71
71
73
72
70
12:46:26 AM MinLINQ
71
77
72
71
70
70
72
72
71
71
72
74
72
71
71
72

12:47:15 AM 16. CountVerifyEquivalence: True
12:47:16 AM CountTraditional
76
72
70
```

```
70
70
70
72
72
71
71
70
71
72
73
72
70
12:48:05 AM CountLINQ
71
70
70
73
72
71
70
72
72
72
70
70
72
71
70
71

12:48:54 AM 17. AverageVerifyEquivalence: True
12:48:54 AM AverageTraditional
374
383
384
371
385
372
370
388
373
376
382
373
402
```

```
369
497
441
12:49:49 AM AverageLINQ
385
376
376
381
373
373
390
383
379
381
380
369
382
375
381
376

12:50:43 AM 18. JoinVerifyEquivalence: True
12:50:43 AM JoinTraditional
384
401
388
384
396
392
383
406
390
388
419
388
390
402
387
393
12:51:37 AM JoinLINQ
394
382
384
384
385
394
```

```
390
388
394
383
390
381
401
380
389
402

12:52:32 AM 19. SequenceEqualTrueVerifyEquivalence: True
12:52:32 AM SequenceEqualTrueTraditional
386
369
375
390
387
373
372
385
370
376
386
381
404
374
380
372
12:53:26 AM SequenceEqualTrueLINQ
369
383
376
370
388
369
392
375
372
391
380
375
371
375
385
377
```

```

12:54:21 AM 20. SequenceEqualVerifyEquivalence: True
12:54:21 AM SequenceEqualTraditional
758
770
755
747
743
789
793
741
753
745
767
758
757
763
855
749
12:55:21 AM SequenceEqualLINQ
761
753
766
748
756
766
763
744
755
759
742
757
755
765
760
756

12:56:22 AM 21. ElementAtVerifyEquivalence: True
12:56:22 AM ElementAtTraditional
382
376
390
372
372
385
372
```

```
383
376
371
385
386
370
398
376
374
12:57:16 AM ElementAtLINQ
373
381
378
373
381
378
371
380
387
370
380
391
376
377
370
385

12:58:10 AM 22. ToArrayVerifyEquivalence: True
12:58:10 AM ToArrayTraditional
384
372
370
369
382
372
375
384
386
373
386
392
379
372
384
375
12:59:05 AM ToArrayLINQ
```

```
382
373
378
376
390
376
376
392
373
373
376
374
371
384
382
376

12:59:59 AM 23. ToDictionaryVerifyEquivalence: True
12:59:59 AM ToDictionaryTraditional
373
382
370
374
377
370
377
379
392
376
370
386
376
372
382
376
1:00:53 AM ToDictionaryLINQ
376
379
386
379
384
376
382
376
378
373
```

```
379
374
386
378
385
373

1:01:47 AM 24. ToListVerifyEquivalence: True
1:01:47 AM ToListTraditional
13
15
13
14
12
12
14
12
12
13
12
12
13
12
12
12
1:02:35 AM ToListLINQ
12
12
12
13
12
12
14
12
12
14
13
12
12
13
12
12

1:03:24 AM 25. ConcatenateVerifyEquivalence: True
1:03:24 AM ConcatenateTraditional
370
```

```
387
390
380
385
388
391
381
380
385
385
392
383
377
378
380
1:04:18 AM ConcatenateLINQ
385
383
373
400
383
379
376
375
383
386
380
380
381
386
368
374

1:05:13 AM 26. AggregateVerifyEquivalence: True
1:05:13 AM AggregateTraditional
379
453
382
380
384
380
406
379
387
385
379
```

```
376
385
392
380
381
1:06:07 AM AggregateLINQ
396
391
375
402
384
379
393
378
399
378
382
379
400
382
400
383

*** Benchmark complete: 16 iterations ***


1:07:01 AM 1. SelectVerifyEquivalence: True
1:07:01 AM SelectTraditional
71
69
71
69
69
1:07:17 AM SelectLINQ
74
72
72
73
74

1:07:32 AM 2. CountVerifyEquivalence: True
1:07:32 AM CountTraditional
49
50
53
49
```

```
49
1:07:48 AM CountLINQ
55
53
55
53
55

1:08:03 AM 3. WhereVerifyEquivalence: True
1:08:03 AM WhereTraditional
64
63
64
61
61
1:08:18 AM WhereLINQ
20
19
19
21
19

1:08:33 AM 4. WhereNoMatchVerifyEquivalence: True
1:08:33 AM WhereNoMatchTraditional
5
5
5
7
5
1:08:48 AM WhereNoMatchLINQ
12
10
10
10
10

1:09:04 AM 5. JoinVerifyEquivalence: True
1:09:04 AM JoinTraditional
14
12
12
12
12
1:09:19 AM JoinLINQ
21
18
```

```
18
18
19

1:09:34 AM 6. OrderVerifyEquivalence: True
1:09:34 AM OrderTraditional
73
70
72
70
70
1:09:49 AM OrderLINQ
77
77
75
76
74

1:10:05 AM 7. GroupVerifyEquivalence: True
1:10:05 AM GroupTraditional
66
68
68
68
66
1:10:20 AM GroupLINQ
813
651
643
807
651

1:10:39 AM 8. SumVerifyEquivalence: True
1:10:39 AM SumTraditional
280
283
285
282
283
1:10:55 AM SumLINQ
303
289
289
290
293
```

```
1:11:12 AM 9. InsertVerifyEquivalence: True
1:11:12 AM InsertTraditional
293
299
296
341
288
1:11:28 AM InsertLINQ
1227
1211
1221
1195
1170

1:12:02 AM 10. UpdateVerifyEquivalence: True
1:12:02 AM UpdateTraditional
68
118
73
120
77
1:12:18 AM UpdateLINQ
12594
13103
13027
13126
13058

1:13:54 AM 11. DeleteVerifyEquivalence: True
1:13:54 AM DeleteTraditional
28
29
27
28
29
1:14:15 AM DeleteLINQ
1161
1156
1161
1161
1171

*** Warmup complete: 5 iterations ***


1:14:42 AM 1. SelectVerifyEquivalence: True
```

```
1:14:42 AM SelectTraditional
69
69
72
69
71
69
69
71
71
69
71
70
74
71
69
70
1:15:31 AM SelectLINQ
74
73
74
72
72
72
73
72
77
80
75
73
74
74
73
72

1:16:20 AM 2. CountVerifyEquivalence: True
1:16:20 AM CountTraditional
51
49
49
49
51
49
49
49
52
```

```
50
58
49
49
49
50
49
1:17:09 AM CountLINQ
54
53
53
56
53
54
54
54
55
54
55
55
57
54
55
55

1:17:58 AM 3. WhereVerifyEquivalence: True
1:17:58 AM WhereTraditional
62
64
62
65
64
61
64
64
62
62
64
64
62
62
64
63
1:18:47 AM WhereLINQ
20
19
```

```
19
19
19
21
19
19
21
19
19
20
19
19
19
19

1:19:35 AM 4. WhereNoMatchVerifyEquivalence: True
1:19:35 AM WhereNoMatchTraditional
5
5
5
5
7
6
5
5
5
5
7
5
5
5
5
5
1:20:23 AM WhereNoMatchLINQ
11
10
10
10
10
10
10
10
10
10
12
11
```

```
10
10
10
10

1:21:12 AM 5. JoinVerifyEquivalence: True
1:21:12 AM JoinTraditional
14
12
12
12
12
12
12
12
13
12
12
12
12
12
12
12
1:22:00 AM JoinLINQ
20
18
18
18
20
20
17
18
18
19
19
18
18
18
19
18

1:22:48 AM 6. OrderVerifyEquivalence: True
1:22:48 AM OrderTraditional
71
75
72
```

```
71
73
70
72
70
71
71
71
70
72
72
73
72
1:23:38 AM OrderLINQ
75
74
76
76
74
74
76
75
77
78
74
76
74
75
75
75

1:24:27 AM 7. GroupVerifyEquivalence: True
1:24:27 AM GroupTraditional
69
67
68
68
68
68
69
66
68
68
66
68
70
```

```
68
67
66
1:25:16 AM GroupLINQ
650
1025
646
657
653
649
659
811
647
643
647
802
814
645
959
639

1:26:16 AM 8. SumVerifyEquivalence: True
1:26:16 AM SumTraditional
289
285
283
282
285
280
285
297
304
283
282
283
284
281
286
285
1:27:08 AM SumLINQ
288
287
290
287
288
285
```

```
288
293
288
291
302
288
293
289
287
287

1:28:01 AM 9. InsertVerifyEquivalence: True
1:28:01 AM InsertTraditional
304
298
289
392
307
296
288
301
287
261
293
302
289
245
293
296
1:28:54 AM InsertLINQ
1227
1218
1209
1167
1150
1191
1186
1238
1155
1177
1169
1214
1236
1181
1222
1186
```

```

1:30:42 AM 10. UpdateVerifyEquivalence: True
1:30:42 AM UpdateTraditional
133
301
146
303
155
301
178
299
156
322
178
301
186
299
166
301
1:31:34 AM UpdateLINQ
42245
42579
42454
42427
41688
42538
42483
42578
41683
42416
42513
42687
42508
42725
42332
42366

1:44:24 AM 11. DeleteVerifyEquivalence: True
1:44:24 AM DeleteTraditional
27
29
31
29
29
29
29
```

```
29
35
28
31
28
29
29
35
28
1:45:33 AM DeleteLINQ
1154
1171
1150
1181
1171
1154
1154
1166
1164
1174
1151
1156
1150
1171
1165
1180

*** Benchmark complete: 16 iterations ***

1:46:58 AM SelectLINQ
1:46:58 AM 0
79
69
68
70
71
1:47:14 AM CountLINQ
1:47:14 AM 1
52
51
50
51
51
1:47:29 AM WhereLINQ
1:47:29 AM 0
17
```

```
14
15
14
14
1:47:44 AM WhereNoMatchLINQ
1:47:44 AM 0
8
6
6
6
6
1:47:59 AM JoinLINQ
1:47:59 AM 2
15
9
9
9
9
1:48:14 AM OrderLINQ
1:48:14 AM 0
75
71
70
70
70
1:48:30 AM GroupLINQ
1:48:30 AM 0
800
637
783
791
941
1:48:49 AM SumLINQ
1:48:49 AM 1
290
284
288
281
288
1:49:05 AM InsertLINQ
1231
1226
1210
1171
1229
1:49:26 AM UpdateLINQ
```

```
327
106
200
106
202
1:49:42 AM DeleteLINQ
147
27
61
25
26

*** Warmup complete: 5 iterations ***

1:50:10 AM SelectLINQ
1:50:10 AM 0
71
71
68
68
75
70
68
68
68
68
68
68
69
68
68
68
1:50:59 AM CountLINQ
1:50:59 AM 0
51
52
51
50
49
52
49
52
49
49
49
51
```

```
49
51
50
50
1:51:48 AM WhereLINQ
1:51:48 AM 0
17
15
14
14
14
14
14
14
14
14
14
15
16
15
14
15
1:52:36 AM WhereNoMatchLINQ
1:52:36 AM 0
10
6
6
6
6
6
6
6
6
6
6
6
6
6
6
6
1:53:24 AM JoinLINQ
1:53:24 AM 1
15
11
9
9
```

```
10
9
9
9
9
9
9
9
10
9
9
9
1:54:13 AM OrderLINQ
1:54:13 AM 0
74
70
72
72
71
70
70
70
70
73
72
70
74
73
70
70
1:55:02 AM GroupLINQ
1:55:02 AM 0
643
786
797
636
645
645
635
849
641
647
633
644
631
791
```

```
797
786
1:56:01 AM SumLINQ
1:56:01 AM 0
290
292
283
291
285
283
282
282
284
281
283
285
298
283
290
283
1:56:54 AM InsertLINQ
1211
1221
1304
1249
1179
1233
1280
1225
1226
1217
1218
1213
1224
1217
1219
1251
1:58:01 AM UpdateLINQ
379
393
332
398
329
402
335
392
```

```
337
426
340
443
332
443
334
446
1:58:56 AM DeleteLINQ
168
27
25
26
26
168
25
25
27
26
168
26
27
26
26
165

*** Benchmark complete: 16 iterations ***


2:00:25 AM 1. OrderByVerifyEquivalence: True
2:00:25 AM OrderByTraditional
145
143
153
148
151
2:00:41 AM OrderByLINQ
41
37
46
38
37

2:00:56 AM 2. WhereVerifyEquivalence: True
2:00:56 AM WhereTraditional
19
```

```
35
19
18
21
2:01:11 AM WhereLINQ
18
19
17
18
18

2:01:26 AM 3. FullnameListVerifyEquivalence: True
2:01:26 AM FullnameListTraditional
72
21
19
27
21
2:01:41 AM FullnameListLINQ
13
13
15
12
18

2:01:56 AM 4. CountVerifyEquivalence: True
2:01:57 AM CountTraditional
14
14
14
16
14
2:02:12 AM CountLINQ
12
11
15
12
11

2:02:27 AM 5. AverageVerifyEquivalence: True
2:02:27 AM AverageTraditional
15
15
17
14
15
```

```
2:02:42 AM AverageLINQ
13
12
17
12
13

2:02:57 AM 6. RemoveVerifyEquivalence: True
2:02:57 AM RemoveTraditional
17
22
18
23
16
2:03:12 AM RemoveLINQ
13
13
19
13
13

2:03:27 AM 7. UpdateVerifyEquivalence: True
2:03:27 AM UpdateTraditional
18
24
18
20
23
2:03:42 AM UpdateLINQ
18
21
16
76
16

2:03:58 AM 8. GenerateVerifyEquivalence: True
2:03:58 AM GenerateTraditional
404
394
388
422
393
2:04:15 AM GenerateLINQ
403
394
408
```

```
406
392

2:04:32 AM 9. ElementAtVerifyEquivalence: True
2:04:32 AM ElementAtTraditional
14
15
14
14
14
2:04:47 AM ElementAtLINQ
12
11
17
11
11

2:05:02 AM 10. LastVerifyEquivalence: True
2:05:02 AM LastTraditional
14
14
16
14
14
2:05:18 AM LastLINQ
11
12
16
11
12

2:05:33 AM 11. TakeVerifyEquivalence: True
2:05:33 AM TakeTraditional
33
17
23
26
38
2:05:48 AM TakeLINQ
32
26
26
36
15

2:06:03 AM 12. SkipWhileVerifyEquivalence: True
```

```
2:06:03 AM SkipWhileTraditional
37
36
35
42
42
2:06:18 AM SkipWhileLINQ
37
32
33
38
37

2:06:34 AM 13. GroupByAndToDictionaryVerifyEquivalence: True
2:06:34 AM GroupByAndToDictionaryTraditional
26
28
26
26
26
2:06:49 AM GroupByAndToDictionaryLINQ
13
12
16
12
12

*** Warmup complete: 5 iterations ***


2:07:04 AM 1. OrderByVerifyEquivalence: True
2:07:04 AM OrderByTraditional
146
144
149
147
154
148
143
152
144
153
148
147
147
150
```

```
149
142
2:07:54 AM OrderByLINQ
43
37
36
47
36
36
47
37
35
52
39
37
47
37
38
45

2:08:43 AM 2. WhereVerifyEquivalence: True
2:08:43 AM WhereTraditional
19
20
19
18
22
20
19
19
23
24
19
22
20
23
20
22
2:09:32 AM WhereLINQ
19
18
19
21
18
18
15
```

```
20
17
14
23
15
20
18
17
18

2:10:20 AM 3. FullnameListVerifyEquivalence: True
2:10:20 AM FullnameListTraditional
19
21
19
25
21
19
23
23
19
23
27
19
23
19
19
23
2:11:08 AM FullnameListLINQ
12
12
18
12
17
12
12
18
12
20
12
12
17
12
20
12
```

```
2:11:57 AM 4. CountVerifyEquivalence: True
2:11:57 AM CountTraditional
14
14
14
16
13
13
25
14
14
16
13
14
16
13
13
17
2:12:45 AM CountLINQ
11
11
15
11
13
18
12
12
11
11
19
12
12
15
12
11

2:13:33 AM 5. AverageVerifyEquivalence: True
2:13:33 AM AverageTraditional
15
15
17
15
15
25
15
15
```

```
19
15
15
17
15
18
15
15
2:14:22 AM AverageLINQ
12
13
16
12
13
18
12
14
12
13
20
12
12
16
12
12

2:15:10 AM 6. RemoveVerifyEquivalence: True
2:15:10 AM RemoveTraditional
16
25
17
22
16
22
16
23
16
21
17
20
16
21
16
17
2:15:58 AM RemoveLINQ
13
```

```
13
18
13
13
18
13
13
13
13
18
13
15
18
13
20

2:16:47 AM 7. UpdateVerifyEquivalence: True
2:16:47 AM UpdateTraditional
18
23
18
19
23
18
20
23
19
20
23
18
25
23
18
23
2:17:35 AM UpdateLINQ
16
21
16
21
16
21
16
25
16
21
16
```

```
23
17
20
22
16

2:18:24 AM 8. GenerateVerifyEquivalence: True
2:18:24 AM GenerateTraditional
409
418
412
415
399
399
406
411
412
394
433
402
395
407
417
391
2:19:18 AM GenerateLINQ
400
399
429
391
416
396
395
420
399
404
407
409
391
413
408
398

2:20:13 AM 9. ElementAtVerifyEquivalence: True
2:20:13 AM ElementAtTraditional
14
14
```

```
14
14
14
14
14
13
16
13
14
16
14
14
22
14
2:21:01 AM ElementAtLINQ
11
11
15
11
11
17
12
16
11
11
19
11
12
11
11
19

2:21:50 AM 10. LastVerifyEquivalence: True
2:21:50 AM LastTraditional
14
14
16
14
14
23
16
14
16
14
16
14
```

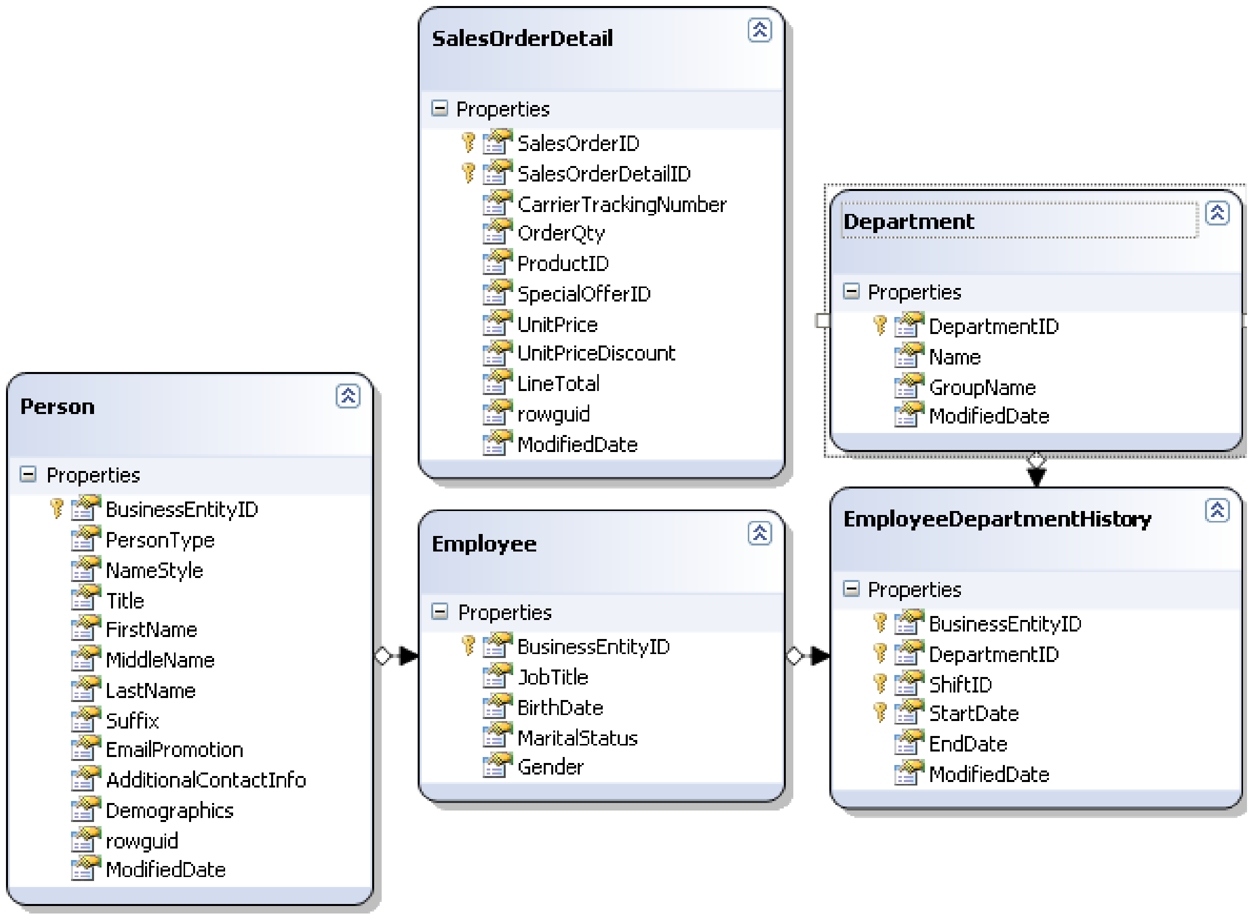
```
14
21
14
14
2:22:38 AM LastLINQ
11
13
15
12
12
17
11
16
11
12
18
11
13
12
11
18

2:23:26 AM 11. TakeVerifyEquivalence: True
2:23:26 AM TakeTraditional
35
27
20
25
27
35
33
18
19
38
28
24
26
29
25
20
2:24:15 AM TakeLINQ
18
37
24
17
31
```

```
16
32
29
17
26
22
33
35
20
16
27

2:25:03 AM 12. SkipWhileVerifyEquivalence: True
2:25:03 AM SkipWhileTraditional
38
37
43
43
36
40
42
40
42
40
43
37
36
38
37
37
2:25:52 AM SkipWhileLINQ
38
31
41
32
33
31
43
35
32
31
44
33
32
31
41
```

```
31

2:26:41 AM 13. GroupByAndToDictionaryVerifyEquivalence: True
2:26:41 AM GroupByAndToDictionaryTraditional
26
28
26
26
28
26
26
27
39
26
26
26
26
26
27
33
2:27:29 AM GroupByAndToDictionaryLINQ
12
12
15
12
12
12
12
17
12
12
12
12
17
12
17
12

*** Benchmark complete: 16 iterations ***
```

Appendix C: Generated Classes Mapped to AdventureWorks for LINQ to SQL

The following diagram represents the classes generated by the LINQ to SQL visual designer when working with specific AdventureWorks tables. These are the mapped classes used by the code in Appendix A.


SalesOrderDetail
Properties
SalesOrderID
SalesOrderDetailID
CarrierTrackingNumber
OrderQty
ProductID
SpecialOfferID
UnitPrice
UnitPriceDiscount
LineTotal
rowguid
ModifiedDate
Department
Properties
DepartmentID
Name
GroupName
ModifiedDate
Person
Properties
BusinessEntityID
PersonType
NameStyle
Title
FirstName
MiddleName
LastName
Suffix
EmailPromotion
AdditionalContactInfo
Demographics
rowguid
ModifiedDate
Employee
Properties
BusinessEntityID
JobTitle
BirthDate
MaritalStatus
Gender
EmployeeDepartmentHistory
Properties
BusinessEntityID
DepartmentID
ShiftID
StartDate
EndDate
ModifiedDate