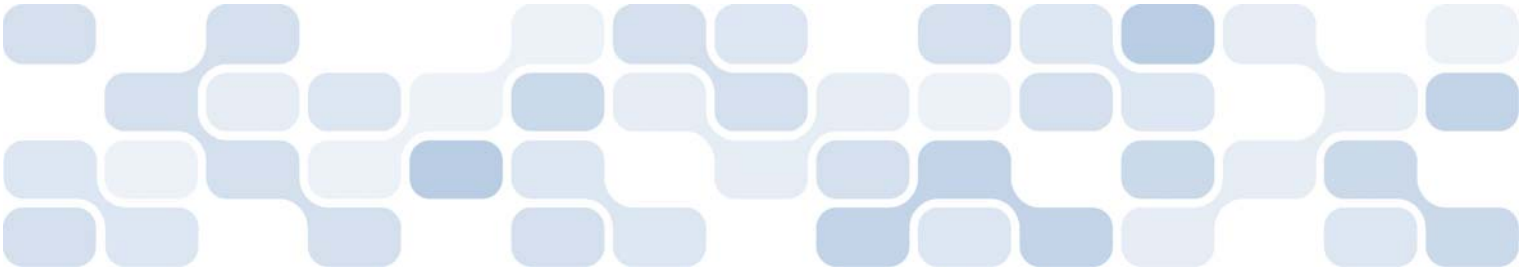
Microsoft

The LINQ Project

.NET Language Integrated Query

May 2006

Don Box, Architect, Microsoft Corporation and
Anders Hejlsberg, Technical Fellow, Microsoft Corporation



.NET Language Integrated Query

After two decades, the industry has reached a stable point in the evolution of object oriented programming technologies. Programmers now take for granted features like classes, objects, and methods. In looking at the current and next generation of technologies, it has become apparent that the next big challenge in programming technology is to reduce the complexity of accessing and integrating information that is not natively defined using OO technology. The two most common sources of non-OO information are relational databases and XML.

Rather than add relational or XML-specific features to our programming languages and runtime, with the LINQ project we have taken a more general approach and are adding general purpose query facilities to the .NET Framework that apply to all sources of information, not just relational or XML data. This facility is called .NET Language Integrated Query (LINQ).

We use the term *language integrated query* to indicate that query is an integrated feature of the developer's primary programming languages (e.g., C#, Visual Basic). Language integrated query allows *query expressions* to benefit from the rich metadata, compile-time syntax checking, static typing and IntelliSense that was previously available only to imperative code. Language integrated query also allows a single general purpose declarative query facility to be applied to all in-memory information, not just information from external sources.

.NET Language Integrated Query defines a set of general purpose *standard query operators* that allow traversal, filter, and projection operations to be expressed in a direct yet declarative way in any .NET-based programming language. The standard query operators allow queries to be applied to any `IEnumerable<T>`-based information source. LINQ allows third parties to augment the set of standard query operators with new domain-specific operators that are appropriate for the target domain or technology. More importantly, third parties are also free to replace the standard query operators with their own implementations that provide additional services such as remote evaluation, query translation, optimization, etc. By adhering to the conventions of the *LINQ pattern*, such implementations enjoy the same language integration and tool support as the standard query operators.

The extensibility of the query architecture is used in the LINQ project itself to provide implementations that work over both XML and SQL data. The query operators over XML (XLinq) use an efficient, easy-to-use in-memory XML facility to provide XPath/XQuery functionality in the host programming language. The query operators over relational data (DLinq) build on the integration of SQL-based schema definitions into the CLR type system. This integration provides strong typing over relational data while retaining the expressive power of the relational model and the performance of query evaluation directly in the underlying store.

Getting Started with Standard Query Operators

To see language integrated query at work, we'll begin with a simple C# 3.0 program that uses the standard query operators to process the contents of an array:

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

class app {
  static void Main() {
    string[] names = { "Burke", "Connor", "Frank",
                       "Everett", "Albert", "George",
                       "Harris", "David" };

    IEnumerable<string> expr = from s in names
                               where s.Length == 5
                               orderby s
                               select s.ToUpper();

    foreach (string item in expr)
      Console.WriteLine(item);
  }
}
```

If you were to compile and run this program, you'd see this as output:

```
BURKE
DAVID
FRANK
```

To understand how language integrated query works, we need to dissect the first statement of our program.

```
IEnumerable<string> expr = from s in names
                           where s.Length == 5
                           orderby s
                           select s.ToUpper();
```

The local variable `expr` is initialized with a *query expression*. A query expression operates on one or more information sources by applying one or more query operators from either the standard query operators or domain-specific operators. This expression uses three of the standard query operators: `Where`, `OrderBy`, and `Select`.

Visual Basic 9.0 supports LINQ as well. Here's the preceding statement written in Visual Basic 9.0:

```
Dim expr As IEnumerable(Of String) = From s in names _
                                     Where s.Length = 5 _
                                     Order By s _
                                     Select s.ToUpper()
```

Both the C# and Visual Basic statements shown here use *query syntax*. Like the `foreach` statement, query syntax is a convenient declarative shorthand over code you could write

manually. The statements above are semantically identical to the following explicit syntax shown in C#:

```
IEnumerable<string> expr = names
                           .Where(s => s.Length == 5)
                           .OrderBy(s => s)
                           .Select(s => s.ToUpper());
```

The arguments to the `Where`, `OrderBy`, and `Select` operators are called *lambda expressions*, which are fragments of code much like delegates. They allow the standard query operators to be defined individually as methods and strung together using dot notation. Together, these methods form the basis for an extensible query language.

Language features supporting the LINQ Project

LINQ is built entirely on general purpose language features, some of which are new to C# 3.0 and Visual Basic 9.0. Each of these features has utility on its own, yet collectively these features provide an extensible way to define queries and queryable API's. In this section we explore these language features and how they contribute to a much more direct and declarative style of queries.

Lambda Expressions and Expression Trees

Many query operators allow the user to provide a function that performs filtering, projection, or key extraction. The query facilities build on the concept of lambda expressions, which provides developers with a convenient way to write functions that can be passed as arguments for subsequent evaluation. Lambda expressions are similar to CLR delegates and must adhere to a method signature defined by a delegate type. To illustrate this, we can expand the statement above into an equivalent but more explicit form using the `Func` delegate type:

```
Func<string, bool>   filter  = s => s.Length == 5;
Func<string, string> extract = s => s;
Func<string, string> project = s => s.ToUpper();

IEnumerable<string> expr = names.Where(filter)
                                .OrderBy(extract)
                                .Select(project);
```

Lambda expressions are the natural evolution of C# 2.0's anonymous methods. For example, we could have written the previous example using anonymous methods like this:

```
Func<string, bool>   filter  = delegate (string s) {
                                   return s.Length == 5;
                               };

Func<string, string> extract = delegate (string s) {
                                   return s;
                               };

Func<string, string> project = delegate (string s) {
                                   return s.ToUpper();
                               };

IEnumerable<string> expr = names.Where(filter)
                                .OrderBy(extract)
                                .Select(project);
```

In general, the developer is free to use named methods, anonymous methods, or lambda expressions with query operators. Lambda expressions have the advantage of providing the most direct and compact syntax for authoring. More importantly, lambda expressions can be compiled as either code or data, which allows lambda expressions to be processed at runtime by optimizers, translators, and evaluators.

LINQ defines a distinguished type, `Expression<T>` (in the `System.Expressions` namespace), which indicates that an *expression tree* is desired for a given lambda expression rather than a traditional IL-based method body. Expression trees are efficient in-memory data representations of lambda expressions and make the structure of the expression transparent and explicit.

The determination of whether the compiler will emit executable IL or an expression tree is determined by how the lambda expression is used. When a lambda expression is assigned to a variable, field, or parameter whose type is a delegate, the compiler emits IL that is identical to that of an anonymous method. When a lambda expression is assigned to a variable, field, or parameter whose type is `Expression<T>`, the compiler emits an expression tree instead.

For example, consider the following two variable declarations:

```
Func<int, bool>             f = n => n < 5;
Expression<Func<int, bool>> e = n => n < 5;
```

The variable `f` is a reference to a delegate that is directly executable:

```
bool isSmall = f(2); // isSmall is now true
```

The variable `e` is a reference to an expression tree that is not directly executable:

```
bool isSmall = e(2); // compile error, expressions == data
```

Unlike delegates, which are effectively opaque code, we can interact with the expression tree just like any other data structure in our program. For example, this program:

```
Expression<Func<int, bool>> filter = n => n < 5;

BinaryExpression    body  = (BinaryExpression)filter.Body;
ParameterExpression left  = (ParameterExpression)body.Left;
ConstantExpression  right = (ConstantExpression)body.Right;

Console.WriteLine("{0} {1} {2}",
                  left.Name, body.NodeType, right.Value);
```

decomposes the expression tree at runtime and prints out the string:

```
n LT 5
```

This ability to treat expressions as data at runtime is critical to enable an ecosystem of third-party libraries that leverage the base query abstractions that are part of the platform. The DLinq data access implementation leverages this facility to translate expression trees to T-SQL statements suitable for evaluation in the store.

Extension Methods

Lambda expressions are one important piece of the query architecture. *Extension methods* are another. Extension methods combine the flexibility of "duck typing" made popular in dynamic languages with the performance and compile-time validation of statically-typed languages. With extension methods third parties may augment the public contract of a type with new methods while still allowing individual type authors to provide their own specialized implementation of those methods.

Extension methods are defined in static classes as static methods, but are marked with the `[System.Runtime.CompilerServices.Extension]` attribute in CLR metadata. Languages are encouraged to provide a direct syntax for extension methods. In C#, extension methods are indicated by the `this` modifier which must be applied to the first parameter of the extension method. Let's look at the definition of the simplest query operator, `Where`:

```
namespace System.Query {
  using System;
  using System.Collections.Generic;

  public static class Sequence {
    public static IEnumerable<T> Where<T>(
             this IEnumerable<T> source,
             Func<T, bool> predicate) {

      foreach (T item in source)
        if (predicate(item))
          yield return item;
    }
  }
}
```

The type of the first parameter of an extension method indicates what type the extension applies to. In the example above, the `Where` extension method extends the type `IEnumerable<T>`. Because `Where` is a static method, we can invoke it directly just like any other static method:

```
IEnumerable<string> expr = Sequence.Where(names,
                                          s => s.Length < 6);
```

However, what makes extension methods unique is that they can also be invoked using instance syntax:

```
IEnumerable<string> expr = names.Where(s => s.Length < 6);
```

Extension methods are resolved at compile-time based on which extension methods are in scope. When a namespace is imported with C#'s `using` statement or VB's `Import` statement, all extension methods that are defined by static classes from that namespace are brought into scope.

The standard query operators are defined as extension methods in the type `System.Query.Sequence`. When examining the standard query operators, you'll notice that all but a few of them are defined in terms of the `IEnumerable<T>` interface. This means that every `IEnumerable<T>`-compatible information source gets the standard query operators simply by adding the following using statement in C#:

```
using System.Query; // makes query operators visible
```

Users that wish to replace the standard query operators for a specific type may either (a) define their own same-named methods on the specific type with compatible signatures or (b) define new same-named extension methods that extend the specific type. Users that want to eschew the standard query operators altogether can simply not put `System.Query` into scope and write their own extension methods for `IEnumerable<T>`.

Extension methods are given the lowest priority in terms of resolution and are only used if there is no suitable match on the target type and its base types. This allows user-defined types to provide their own query operators that take precedence over the standard operators. For example, consider the custom collection shown here:

```
public class MySequence : IEnumerable<int> {
  public IEnumerator<int> GetEnumerator() {
    for (int i = 1; i <= 10; i++)
      yield return i;
  }

  IEnumerator IEnumerable.GetEnumerator() {
    return GetEnumerator();
  }

  public IEnumerable<int> Where(Func<int, bool> filter) {
    for (int i = 1; i <= 10; i++)
      if (filter(i))
        yield return i;
  }
}
```

Given this class definition, the following program:

```
MySequence s = new MySequence();
foreach (int item in s.Where(n => n > 3))
    Console.WriteLine(item);
```

will use the `MySequence.Where` implementation, not the extension method, as instance methods take precedence over extension methods.

The `OfType` operator is one of the few standard query operators that doesn't extend an `IEnumerable<T>`-based information source. Let's look at the `OfType` query operator:

```
public static IEnumerable<T> OfType<T>(this IEnumerable source) {
  foreach (object item in source)
    if (item is T)
      yield return (T)item;
}
```

`OfType` accepts not only `IEnumerable<T>`-based sources, but also sources that are written against the non-parameterized `IEnumerable` interface that was present in version 1 of the .NET Framework. The `OfType` operator allows users to apply the standard query operators to classic .NET collections like this:

```
// "classic" cannot be used directly with query operators
IEnumerable classic = new OlderCollectionType();

// "modern" can be used directly with query operators
IEnumerable<object> modern = classic.OfType<object>();
```

In this example, the variable `modern` yields the same sequence of values as does `classic`, however, its type is compatible with modern `IEnumerable<T>` code, including the standard query operators.

The `OfType` operator is also useful for newer information sources, as it allows filtering values from a source based on type. When producing the new sequence, `OfType` simply omits members of the original sequence that that are not compatible with the type argument. Consider this simple program that extracts strings from a heterogeneous array:

```
object[] vals = { 1, "Hello", true, "World", 9.1 };
IEnumerable<string> justStrings = vals.OfType<string>();
```

When we enumerate the `justStrings` variable in a `foreach` statement, we'll get a sequence of two strings "Hello" and "World".

Deferred Query Evaluation

Observant readers may have noted that the standard `Where` operator is implemented using the `yield` construct introduced in C# 2.0. This implementation technique is common for all of the standard operators that return sequences of values. The use of `yield` has an interesting benefit which is that the query is not actually evaluated until it is iterated over, either with a `foreach` statement or manually using the underlying `GetEnumerator` and `MoveNext` methods. This deferred evaluation allows queries to be kept as `IEnumerable<T>`-based values that can be evaluated multiple times, each time yielding potentially different results.

For many applications, this is exactly the behavior that is desired. For applications that want to cache the results of query evaluation, two operators, `ToList` and `ToArray`, are provided that force the immediate evaluation of the query and return either a `List<T>` or an array containing the results of the query evaluation.

To see how deferred query evaluation works consider this program that runs a simple query over an array:

```
// declare a variable containing some strings
string[] names = { "Allen", "Arthur", "Bennett" };

// declare a variable that represents a query
IEnumerable<string> ayes = names.Where(s => s[0] == 'A');

// evaluate the query
foreach (string item in ayes)
  Console.WriteLine(item);

// modify the original information source
names[0] = "Bob";

// evaluate the query again, this time no "Allen"
foreach (string item in ayes)
    Console.WriteLine(item);
```

The query is evaluated each time the variable `ayes` is iterated over. To indicate that a cached copy of the results is needed, we can simply append a `ToList` or `ToArray` operator to the query like this:

```
// declare a variable containing some strings
string[] names = { "Allen", "Arthur", "Bennett" };

// declare a variable that represents the result
// of an immediate query evaluation
string[] ayes = names.Where(s => s[0] == 'A').ToArray();

// iterate over the cached query results
foreach (string item in ayes)
    Console.WriteLine(item);

// modifying the original source has no effect on ayes
names[0] = "Bob";

// iterate over result again, which still contains "Allen"
foreach (string item in ayes)
    Console.WriteLine(item);
```

Both `ToArray` and `ToList` force immediate query evaluation. The same is true for the standard query operators that return singleton values (e.g., `First`, `ElementAt`, `Sum`, `Average`, `All`, `Any`).

The IQueryable<T> interface

The same deferred execution model is usually desired for data sources that implement the query functionality using expression trees, such as DLinq. These data sources can benefit

from implementing the `IQueryable<T>`, for which all the query operators required by the LINQ pattern are implemented using expression trees. Each `IQueryable<T>` has a representation of "the code needed to run the query" in the form of an expression tree. All the deferred query operators return a new IQueryable which augments that expression tree with a representation of a call to that query operator. Thus, when it becomes time to evaluate the query, typically because the IQueryable is enumerated, the data source can process the expression tree representing the whole query in one batch. As an example, a complicated DLinq query obtained by numerous calls to query operators, may result in only a single SQL query getting sent to the database.

The benefit for data source implementers of reusing this deferring functionality by implementing the `IQueryable<T>` interface is obvious. To the clients who write the queries, on the other hand, it is a great advantage to have a common type for remote information sources. Not only does it allow them to write polymorphic queries that can be used against different sources of data, but it also opens up the possibility for writing queries that go across domains.

Initializing Compound Values

Lambda expressions and extension methods provide us with everything we need for queries that simply filter members out of a sequence of values. Most query expressions also perform projection over those members, effectively transforming members of the original sequence into members whose value and type may differ from the original. To support writing these transforms, LINQ relies on a new construct called *object initialization expressions* to create new instances of structured types. For the rest of this document, we'll assume the following type has been defined:

```
public class Person {
  string name;
  int age;
  bool canCode;

  public string Name {
    get { return name; } set { name = value; }
  }

  public int Age {
    get { return age; } set { age = value; }
  }

  public bool CanCode {
    get { return canCode; } set { canCode = value; }
  }
}
```

Object initialization expressions allow us to easily construct values based on the public fields and properties of a type. For example, to create a new value of type `Person`, we can write this statement:

```
Person value = new Person {
    Name = "Chris Smith", Age = 31, CanCode = false
};
```

Semantically, this statement is equivalent to the following sequence of statements:

```
Person value = new Person();
value.Name = "Chris Smith";
value.Age = 31;
value.CanCode = false;
```

Object initialization expressions are an important feature for language integrated query, as they allow the construction of new structured values in contexts where only expressions are allowed (such as within lambda expressions and expression trees). For example, consider this query expression that creates a new `Person` value for each value in the input sequence:

```
IEnumerable<Person> expr = names.Select(s => new Person {
    Name = s, Age = 21, CanCode = s.Length == 5
});
```

Object initialization syntax is also convenient for initializing arrays of structured values. For example, consider this array variable that is initialized using individual object initializers:

```
static Person[] people = {
  new Person { Name="Allen Frances", Age=11, CanCode=false },
  new Person { Name="Burke Madison", Age=50, CanCode=true },
  new Person { Name="Connor Morgan", Age=59, CanCode=false },
  new Person { Name="David Charles", Age=33, CanCode=true },
  new Person { Name="Everett Frank", Age=16, CanCode=true },
};
```

Structured values and types

The LINQ project supports a data-centric programming style in which some types exist primarily to provide a static “shape” over a structured value rather than a full-blown object with both state and behavior. Taking this premise to its logical conclusion, it is often the case that all the developer cares about is the structure of the value, and the need for a named type for that shape is of little use. This leads to the introduction of *anonymous types* that allow new structures to be defined “inline” with their initialization.

In C#, the syntax for anonymous types is similar to the object initialization syntax except that the name of the type is omitted. For example, consider the following two statements:

```
object v1 = new Person {
    Name = "Chris Smith", Age = 31, CanCode = false
};

object v2 = new { // note the omission of type name
    Name = "Chris Smith", Age = 31, CanCode = false
};
```

The variables `v1` and `v2` both point to an in-memory object whose CLR type has three public properties `Name`, `Age`, and `CanCode`. The variables differ in that `v2` refers to an

instance of an *anonymous type*. In CLR terms, anonymous types are no different than any other type. What makes anonymous types special is that they have no meaningful name in your programming language – the only way to create instances of an anonymous type is using the syntax shown above.

To allow variables to refer to instances of anonymous types yet still benefit from static typing, C# introduces the *var* keyword that may be used in place of the type name for local variable declarations. For example, consider this legal C# 3.0 program:

```
var s = "Bob";
var n = 32;
var b = true;
```

The `var` keyword tells the compiler to infer the type of the variable from the static type of the expression used to initialize the variable. In this example, the types of `s`, `n`, and `b` are `string`, `int`, and `bool`, respectively. This program is identical to the following:

```
string s = "Bob";
int    n = 32;
bool   b = true;
```

The `var` keyword is a convenience for variables whose types have meaningful names, but it is a necessity for variables that refer to instances of anonymous types.

```
var value = new {
  Name = "Chris Smith", Age = 31, CanCode = false
};
```

In the example above, variable `value` is of an anonymous type whose definition is equivalent to the following pseudo-C#:

```
internal class ??? {
  string _Name;
  int    _Age;
  bool   _CanCode;

  public string Name {
    get { return _Name; } set { _Name = value; }
  }

  public int Age{
    get { return _Age; } set { _Age = value; }
  }

  public bool CanCode {
    get { return _CanCode; } set { _CanCode = value; }
  }

  public bool Equals(object obj) { … }

  public bool GetHashCode() { … }
}
```

Anonymous types cannot be shared across assembly boundaries; however, the compiler ensures that there is at most one anonymous type for a given sequence of property name/type pairs within each assembly.

Because anonymous types are often used in projections to select one or more members of an existing structured value, we can simply reference fields or properties from another value in the initialization of an anonymous type. This results in the new anonymous type getting a property whose name, type, and value are all copied from the referenced property or field.

For instance, consider this example that creates a new structured value by combining properties from other values:

```
var bob = new Person { Name = "Bob", Age = 51, CanCode = true };
var jane = new { Age = 29, FirstName = "Jane" };

var couple = new {
    Husband = new { bob.Name, bob.Age },
    Wife = new { Name = jane.FirstName, jane.Age }
};

int    ha = couple.Husband.Age; // ha == 51
string wn = couple.Wife.Name;   // wn == "Jane"
```

The referencing of fields or properties shown above is simply a convenient syntax for writing the following more explicit form:

```
var couple = new {
    Husband = new { Name = bob.Name, Age = bob.Age },
    Wife = new { Name = jane.FirstName, Age = jane.Age }
};
```

In both cases, the `couple` variable gets its own copy of the `Name` and `Age` properties from `bob` and `jane`.

Anonymous types are most often used in the `select` clause of a query. For example, consider the following query:

```
var expr = people.Select(p => new {
                p.Name, BadCoder = p.Age == 11
           });

foreach (var item in expr)
  Console.WriteLine("{0} is a {1} coder",
                    item.Name,
                    item.BadCoder ? "bad" : "good");
```

In this example, we were able to create a new projection over the `Person` type that exactly matched the shape we needed for our processing code yet still gave us the benefits of a static type.

More Standard Query Operators

On top of the basic query facilities described above, a number of operators provide useful ways of manipulating sequences and composing queries, giving the user a high degree of control over the result within the convenient framework of the standard query operators.

Sorting and Grouping

In general, the evaluation of a query expression results in a sequence of values that are produced in some order that is intrinsic in the underlying information sources. To give developers explicit control over the order in which these values are produced, standard query operators are defined for controlling the order. The most basic of these operators is the `OrderBy` operator.

The `OrderBy` and `OrderByDescending` operators can be applied to any information source and allow the user to provide a key extraction function that produces the value that is used to sort the results. `OrderBy` and `OrderByDescending` also accept an optional comparison function that can be used to impose a partial order over the keys. Let's look at a basic example:

```
string[] names = { "Burke", "Connor", "Frank", "Everett",
                   "Albert", "George", "Harris", "David" };

// unity sort
var s1 = names.OrderBy(s => s);
var s2 = names.OrderByDescending(s => s);

// sort by length
var s3 = names.OrderBy(s => s.Length);
var s4 = names.OrderByDescending(s => s.Length);
```

The first two query expressions produce new sequences that are based on sorting the members of the source based on string comparison. The second two queries produce new sequences that are based on sorting the members of the source based on the length of each string.

To allow multiple sort criteria, both `OrderBy` and `OrderByDescending` return `SortedSequence<T>` rather than the generic `IEnumerable<T>`. Two operators are defined only on `SortedSequence<T>`, namely `ThenBy` and `ThenByDescending` which apply an additional (subordinate) sort criterion. `ThenBy`/`ThenByDescending` themselves return `SortedSequence<T>`, allowing any number of `ThenBy`/`ThenByDescending` operators to be applied:

```
string[] names = { "Burke", "Connor", "Frank", "Everett",
                   "Albert", "George", "Harris", "David" };

var s1 = names.OrderBy(s => s.Length).ThenBy(s => s);
```

Evaluating the query referenced by `s1` in this example would yield the following sequence of values:

```
"Burke", "David", "Frank",
"Albert", "Connor", "George", "Harris",
"Everett"
```

In addition to the `OrderBy` family of operators, the standard query operators also include a `Reverse` operator. `Reverse` simply enumerates over a sequence and yields the same values in reverse order. Unlike `OrderBy`, `Reverse` doesn't consider the actual values

themselves in determining the order, rather it relies solely on the order the values are produced by the underlying source.

The `OrderBy` operator imposes a sort order over a sequence of values. The standard query operators also include the `GroupBy` operator, which imposes a partitioning over a sequence of values based on a key extraction function. The `GroupBy` operator returns a sequence of `IGrouping` values, one for each distinct key value that was encountered. An `IGrouping` is an `IEnumerable` that additionally contains the key that was used to extract its contents:

```
public interface IGrouping<K, T> : IEnumerable<T> {
  public K Key { get; }
}
```

The simplest application of `GroupBy` looks like this:

```
string[] names = { "Albert", "Burke", "Connor", "David",
                   "Everett", "Frank", "George", "Harris"};

// group by length
var groups = names.GroupBy(s => s.Length);

foreach (IGrouping<int, string> group in groups) {
    Console.WriteLine("Strings of length {0}", group.Key);

    foreach (string value in group)
        Console.WriteLine("  {0}", value);
}
```

When run, this program prints out the following:

```
Strings of length 6
  Albert
  Connor
  George
  Harris
Strings of length 5
  Burke
  David
  Frank
Strings of length 7
  Everett
```

A la `Select`, `GroupBy` allows you to provide a projection function that is used to populate members of the groups.

```
string[] names = { "Albert", "Burke", "Connor", "David",
                   "Everett", "Frank", "George", "Harris"};

// group by length
var groups = names.GroupBy(s => s.Length, s => s[0]);
foreach (IGrouping<int, char> group in groups) {
    Console.WriteLine("Strings of length {0}", group.Key);

    foreach (char value in group)
        Console.WriteLine("  {0}", value);
}
```

This variation prints out the following:

```
Strings of length 6
  A
  C
  G
  H
Strings of length 5
  B
  D
  F
Strings of length 7
  E
```

Note from this example that the projected type does not need to be the same as the source. In this case, we created a grouping of integers to characters from a sequence of strings.

Aggregation Operators

Several standard query operators are defined for aggregating a sequence of values into a single value. The most general aggregation operator is `Aggregate`, which is defined like this:

```
public static U Aggregate<T, U>(this IEnumerable<T> source,
                                 U seed, Func<U, T, U> func) {
  U result = seed;

  foreach (T element in source)
      result = func(result, element);

  return result;
}
```

The `Aggregate` operator makes it simple to perform a calculation over a sequence of values. `Aggregate` works by calling the lambda expression once for each member of the underlying sequence. Each time `Aggregate` calls the lambda expression, it passes both the member from the sequence and an aggregated value (the initial value is the seed parameter to `Aggregate`). The result of the lambda expression replaces the previous aggregated value, and `Aggregate` returns the final result of the lambda expression.

For example, this program uses `Aggregate` to accumulate the total character count over an array of strings:

```
string[] names = { "Albert", "Burke", "Connor", "David",
                   "Everett", "Frank", "George", "Harris"};

int count = names.Aggregate(0, (c, s) => c + s.Length);
// count == 46
```

In addition to the general purpose `Aggregate` operator, the standard query operators also include a general purpose `Count` operator and four numeric aggregation operators (`Min`, `Max`, `Sum`, and `Average`) that simplify these common aggregation operations. The numeric aggregation functions work over sequences of numeric types (e.g., `int`, `double`, `decimal`) or over sequences of arbitrary values as long as a function is provided that projects members of the sequence into a numeric type.

This program illustrates both forms of the `Sum` operator just described:

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
string[] names = { "Albert", "Burke", "Connor", "David",
                   "Everett", "Frank", "George", "Harris"};

int total1 = numbers.Sum();            // total1 == 55
int total2 = names.Sum(s => s.Length); // total2 == 46
```

Note that the second `Sum` statement is equivalent to the previous example using `Aggregate`.

Select vs. SelectMany

The `Select` operator requires the transform function to produce one value for each value in the source sequence. If your transform function returns a value that is itself a sequence, it is up to the consumer to traverse the sub-sequences manually. For example, consider this program that breaks strings into tokens using the existing `String.Split` method:

```
string[] text = { "Albert was here",
                  "Burke slept late",
                  "Connor is happy" };

var tokens = text.Select(s => s.Split(' '));

foreach (string[] line in tokens)
    foreach (string token in line)
        Console.Write("{0}.", token);
```

When run, this program prints out the following text:

```
Albert.was.here.Burke.slept.late.Connor.is.happy.
```

Ideally, we would have liked our query to have returned a coalesced sequence of tokens and not exposed the intermediate `string[]` to the consumer. To achieve this, we use the `SelectMany` operator instead of the `Select` operator. The `SelectMany` operator works similarly to the `Select` operator. It differs in that the transform function is expected to return a sequence that is then expanded by the `SelectMany` operator. Here's our program rewritten using `SelectMany`:

```
string[] text = { "Albert was here",
                  "Burke slept late",
                  "Connor is happy" };

var tokens = text.SelectMany(s => s.Split(' '));

foreach (string token in tokens)
    Console.Write("{0}.", token);
```

The use of `SelectMany` causes each intermediate sequence to be expanded as part of normal evaluation.

`SelectMany` is ideal for combining two information sources:

```
string[] names = { "Burke", "Connor", "Frank", "Everett",
                   "Albert", "George", "Harris", "David" };

var query = names.SelectMany(n =>
                    people.Where(p => n.Equals(p.Name))
                 );
```

In the lambda expression passed to `SelectMany`, the nested query applies to a different source, but has in scope the `n` parameter passed in from the outer source. Thus `people.Where` is called once for each `n`, with the resulting sequences flattened by `SelectMany` for the final output. The result is a sequence of all the people whose name appears in the `names` array.

Join Operators

In an object oriented program, objects that are related to each other will typically be linked up with object references which are easy to navigate. The same usually does not hold true for external information sources, where data entries often have no option but to "point" to each other symbolically, with IDs or other data that can uniquely identify the entity pointed to. The concept of *joins* refers to the operation of bringing the elements of a sequence together with the elements they "match up with" from another sequence.

The previous example with `SelectMany` actually does exactly that, matching up strings with people whose names are those strings. However, for this particular purpose, the SelectMany approach isn't very efficient – it will loop through all the elements of `people` for each and every element of `names`. By bringing all the information of this scenario – the two information sources and the "keys" by which they are matched up – together in one method call, the Join operator is able to do a much better job:

```
string[] names = { "Burke", "Connor", "Frank", "Everett",
                   "Albert", "George", "Harris", "David" };

var query = names.Join(people, n => n, p => p.Name, (n,p) => p);
```

This is a bit of a mouthful, but see how the pieces fit together: The `Join` method is called on the "outer" data source, `names`. The first argument is the "inner" data source, `people`. The second and third arguments are lambda expressions to extract keys from the elements of the outer and inner sources, respectively. These keys are what the `Join` method uses to match up the elements. Here we want the names themselves to match the

`Name` property of the people. The final lambda expression is then responsible for producing the elements of the resulting sequence: It is called with each pair of matching elements `n` and `p`, and is used to shape the result. In this case we choose to discard the `n` and return the `p`. The end result is the list of `Person` elements of `people` whose `Name` is in the list of `names`.

A more powerful cousin of Join is the GroupJoin operator. GroupJoin differs from Join in the way the result shaping lambda expression is used: Instead of being invoked with each individual pair of outer and inner elements, it will be called only once for each outer element, with a sequence of all of the inner elements that match that outer element. To make that concrete:

```
string[] names = { "Burke", "Connor", "Frank", "Everett",
                   "Albert", "George", "Harris", "David" };

var query = names.GroupJoin(people, n => n, p => p.Name,
                  (n, matching) =>
                       new { Name = n, Count = matching.Count() }
);
```

This call produces a sequence of the names you started out with paired with the number of people who have that name. Thus, the `GroupJoin` operator allows you to base your results on the whole "set of matches" for an outer element.

Query syntax

C#'s existing `foreach` statement provides a declarative syntax for iteration over the .NET Framework's `IEnumerable`/`IEnumerator` methods. The `foreach` statement is strictly optional, but it has proven to be a very convenient and popular language mechanism.

Building on this precedent, *query syntax* simplifies query expressions with a declarative syntax for the most common query operators: `Where`, `Join`, `GroupJoin`, `Select`, `SelectMany`, `GroupBy`, `OrderBy`, `ThenBy`, `OrderByDescending`, and `ThenByDescending`.

Let's start by looking at the simple query we began this paper with:

```
IEnumerable<string> expr = names
                           .Where(s => s.Length == 5)
                           .OrderBy(s => s)
                           .Select(s => s.ToUpper());
```

Using query syntax we can rewrite this exact statement like this:

```
IEnumerable<string> expr = from s in names
                           where s.Length == 5
                           orderby s
                           select s.ToUpper();
```

Like C#'s `foreach` statement, query syntax expressions are more compact and easier to read, but are completely optional. Every expression that can be written in query syntax has a corresponding (albeit more verbose) syntax using dot notation.

Let's begin by looking at the basic structure of a query expression. Every syntactic query expression in C# begins with a `from` clause and ends with either a `select` or `group` clause. The initial `from` clause can be followed by zero or more `from`, `let` or `where` clauses. Additionally, any number of `join` clauses can follow immediately after a from clause. Each `from` clause is a generator that introduces an iteration variable ranging over a sequence, each `let` clause gives name to the result of an expression and each `where` clause is a filter that excludes items from the result. Every `join` clause correlates a new data source with the results of a previous `from` or `join`. The final `select` or `group` clause may be preceded by an `orderby` clause that specifies an ordering for the result:

```
query-expression ::= from-clause query-body

query-body ::=
      join-clause*
      (from-clause join-clause* | let-clause | where-clause)*
      orderby-clause?
      (select-clause | groupby-clause)
      query-continuation?

from-clause ::= from itemName in srcExpr

join-clause ::=
      join itemName in srcExpr on keyExpr equals keyExpr
      (into itemName)?

let-clause ::= let itemName = selExpr

where-clause ::= where predExpr

orderby-clause ::= orderby (keyExpr (ascending | descending)?)*

select-clause ::= select selExpr

groupby-clause ::= group selExpr by keyExpr

query-continuation ::= into itemName query-body
```

For example, consider these two query expressions:

```
var query1 = from p in people
             where p.Age > 20
             orderby p.Age descending, p.Name
             select new {
                 p.Name, Senior = p.Age > 30, p.CanCode
             };

var query2 = from p in people
             where p.Age > 20
             orderby p.Age descending, p.Name
             group new {
                p.Name, Senior = p.Age > 30, p.CanCode
             } by p.CanCode;
```

The compiler treats these query expressions as if they were written using the following explicit dot-notation:

```
var query1 = people.Where(p => p.Age > 20)
                   .OrderByDescending(p => p.Age)
                   .ThenBy(p => p.Name)
                   .Select(p => new {
                       p.Name,
                       Senior = p.Age > 30,
                       p.CanCode
                   });

var query2 = people.Where(p => p.Age > 20)
                   .OrderByDescending(p => p.Age)
                   .ThenBy(p => p.Name)
                   .GroupBy(p => p.CanCode,
                            p => new {
                                   p.Name,
                                   Senior = p.Age > 30,
                                   p.CanCode
                   });
```

Query expressions perform a mechanical translation into calls of methods with specific names. The exact query operator *implementation* that is chosen therefore depends both on the type of the variables being queried and the extension methods that are in scope.

The query expressions shown so far have only used one generator. When more than one generator is used, each subsequent generator is evaluated in the context of its predecessor. For example, consider this slight modification to our query:

```
var query = from s1 in names where s1.Length == 5
            from s2 in names where s1 == s2
            select s1 + " " + s2;
```

When run against this input array:

```
string[] names = { "Burke", "Connor", "Frank", "Everett",
                   "Albert", "George", "Harris", "David" };
```

we get the following results:

```
Burke Burke
Frank Frank
David David
```

The query expression above expands to this dot notation expression:

```
var query = names.Where(s1 => s1.Length == 5)
                 .SelectMany(s1 =>
                     names.Where(s2 => s1 == s2)
                          .Select(s2 => s1 + " " + s2)
                 );
```

Note that the use of `SelectMany` causes the inner query expression to be flattened in the outer result.

A special kind of generator is the `join` clause, which will introduce elements of another source that match up with the elements of the preceding `from` or `join` clause according to given keys. A `join` clause may yield the matching elements one by one, but if specified with an `into` clause, the matching elements will be given as a group:

```
var query = from n in names
            join p in people on n equals p.Name into matching
            select new { Name = n, Count = matching.Count() };
```

Not surprisingly, this query expands quite directly into one we have seen before:

```
var query = names.GroupJoin(people, n => n, p => p.Name,
                 (n, matching) =>
                      new { Name = n, Count = matching.Count() }
);
```

It is often useful to treat the results of one query as a generator in a subsequent query. To support this, query expressions use the `into` keyword to splice a new query expression after a select or group clause.

The `into` keyword is especially useful for post-processing the results of a `group by` clause. For example, consider this program:

```
var query = from item in names
            orderby item
            group item by item.Length into lengthGroups
            orderby lengthGroups.Key descending
            select lengthGroups;

foreach (var group in query) {
    Console.WriteLine("Strings of length {0}", group.Key);

    foreach (var val in group.Group)
        Console.WriteLine("  {0}", val);
}
```

This program outputs the following:

```
Strings of length 7
  Everett
Strings of length 6
  Albert
  Connor
  George
  Harris
Strings of length 5
  Burke
  David
  Frank
```

This section described how C# implements query expressions. Other languages may elect to support additional query operators with explicit syntax, or not to have query expressions at all.

It is important to note that the query syntax is by no means hard-wired to the standard query operators. It is a purely syntactic feature that applies to anything which fulfills the *LINQ pattern* by implementing underlying methods with the appropriate names and signatures. The standard query operators described above do so by using extension methods to augment the `IEnumerable<T>` interface. Developers may exploit the query syntax on any type they wish, as long as they make sure that it adheres to the LINQ

pattern, either by direct implementation of the necessary methods or by adding them as extension methods.

This extensibility is exploited in the LINQ project itself by the provision of two *LINQ-enabled* API's, namely DLinq, which implements the LINQ pattern for SQL-based data access, and XLinq which allows LINQ queries over XML data. Both of these are described in the following sections.

DLinq: SQL Integration

.NET Language Integrated Query can be used to query relational data stores without leaving the syntax or compile-time environment of the local programming language. This facility, code-named DLinq, takes advantage of the integration of SQL schema information into CLR metadata. This integration compiles SQL table and view definitions into CLR types that can be accessed from any language.

DLinq defines two core attributes, `[Table]` and `[Column]`, which indicate which CLR types and properties correspond to external SQL data. The `[Table]` attribute can be applied to a class and associates the CLR type with a named SQL table or view. The `[Column]` attribute can be applied to any field or property and associates the member with a named SQL column. Both attributes are parameterized to allow SQL-specific metadata to be retained. For example, consider this simple SQL schema definition:

```
create table People (
    Name nvarchar(32) primary key not null,
    Age int not null,
    CanCode bit not null
)

create table Orders (
    OrderID nvarchar(32) primary key not null,
    Customer nvarchar(32) not null,
    Amount int
)
```

The CLR equivalent looks like this:

```
[Table(Name="People")]
public class Person {
  [Column(DbType="nvarchar(32) not null", Id=true)]
  public string Name;

  [Column]
  public int Age;

  [Column]
  public bool CanCode;
}

[Table(Name="Orders")]
public class Order {
  [Column(DbType="nvarchar(32) not null", Id=true)]
  public string OrderID;

  [Column(DbType="nvarchar(32) not null")]
  public string Customer;

  [Column]
  public int? Amount;
}
```

Note from this example that nullable columns map to nullable types in the CLR (nullable types first appeared in version 2 of the .NET Framework), and that for SQL types that don't have a 1:1 correspondence with a CLR type (e.g., `nvarchar`, `char`, `text`), the original SQL type is retained in the CLR metadata.

To issue a query against a relational store, the DLinq implementation of the LINQ pattern translates the query from its expression tree form into a SQL expression and ADO.NET `DbCommand` object suitable for remote evaluation. For example, consider this simple query:

```
// establish a query context over ADO.NET sql connection
DataContext context = new DataContext(
     "Initial Catalog=petdb;Integrated Security=sspi");

// grab variables that represent the remote tables that
// correspond to the Person and Order CLR types
Table<Person> custs = context.GetTable<Person>();
Table<Order> orders   = context.GetTable<Order>();

// build the query
var query =
    from c in custs
    from o in orders
    where o.Customer == c.Name
    select new {
        c.Name, o.OrderID, o.Amount, c.Age
    };

// execute the query
foreach (var item in query)
    Console.WriteLine("{0} {1} {2} {3}",
        item.Name, item.OrderID, item.Amount, item.Age);
```

The `DataContext` type provides a lightweight translator that does the work of translating the standard query operators to SQL. `DataContext` uses the existing ADO.NET `IDbConnection` for accessing the store and can be initialized with either an established ADO.NET connection object or a connection string that can be used to create one.

The `GetTable` method provides `IEnumerable`-compatible variables that can be used in query expressions to represent the remote table or view. Calls to `GetTable` do not cause any interaction with the database – rather they represent *the potential* to interact with the remote table or view using query expressions. In our example above, the query does not get transmitted to the store until the program iterates over the query expression, in this case using the `foreach` statement in C#. When the program first iterates over the query, the `DataContext` machinery translates the expression tree into the following SQL statement that is sent to the store:

```
SELECT [t0].[Age], [t1].[Amount],
       [t0].[Name], [t1].[OrderID]
FROM [Customers] AS [t0], [Orders] AS [t1]
WHERE [t1].[Customer] = [t0].[Name]
```

It's important to note that by building query capability directly into the local programming language, developers get the full power of the relational model without having to statically bake the relationships into the CLR type. That stated, full blown object/relational mapping can also take advantage of this core query capability for users that want that functionality. DLinq provides object-relational mapping functionality with which the developer can define and navigate relationships between objects. You can refer to `Orders` as a property of the `Customer` class using mapping, so that you do not need explicit joins to tie the two together. External mapping files allow the mapping to be separated from the object model for richer mapping capabilities.

XLinq: XML Integration

.NET Language Integrated Query for XML (XLinq) allows XML data to be queried using the standard query operators as well as tree-specific operators that provide XPath-like navigation through descendants, ancestors, and siblings. It provides an efficient in-memory representation for XML that integrates with the existing `System.Xml` reader/writer infrastructure and is easier to use than W3C DOM. There are three types that do most of the work of integrating XML with queries: `XName`, `XElement` and `XAttribute`.

`XName` provides an easy to use way to deal with the namespace-qualified identifiers (QNames) used as both element and attribute names. `XName` handles the efficient atomization of identifiers transparently and allows either symbols or plain strings to be used wherever a QName is needed.

XML elements and attributes are represented using `XElement` and `XAttribute` respectively. `XElement` and `XAttribute` support normal construction syntax, allowing developers to write XML expressions using a natural syntax:

```
var e = new XElement("Person",
                     new XAttribute("CanCode", true),
                     new XElement("Name", "Loren David"),
                     new XElement("Age", 31));

var s = e.ToString();
```

This corresponds to the following XML:

```
<Person CanCode="true">
  <Name>Loren David</Name>
  <Age>31</Age>
</Person>
```

Notice that no DOM-based factory pattern was needed to create the XML expression, and that the `ToString` implementation yielded the textual XML. XML elements can also be constructed from an existing `XmlReader` or from a string literal:

```
var e2 = XElement.Load(xmlReader);
var e1 = XElement.Parse(
@"<Person CanCode='true'>
  <Name>Loren David</Name>
  <Age>31</Age>
</Person>");
```

`XElement` also supports emitting XML using the existing `XmlWriter` type.

`XElement` dovetails with the query operators, allowing developers to write queries against non-XML information and produce XML results by constructing `XElements` in the body of a select clause:

```
var query = from p in people
            where p.CanCode
            select new XElement("Person",
                                new XAttribute("Age", p.Age),
                                p.Name);
```

This query returns a sequence of `XElements`. To allow `XElements` to be built out of the result of this kind of query, the `XElement` constructor allows sequences of elements to be passed as arguments directly:

```
var x = new XElement("People",
                  from p in people
                  where p.CanCode
                  select
                    new XElement("Person",
                                 new XAttribute("Age", p.Age),
                                 p.Name));
```

This XML expression results in the following XML:

```
<People>
  <Person Age="11">Allen Frances</Person>
  <Person Age="59">Connor Morgan</Person>
</People>
```

The statement above has a direct translation to Visual Basic. However, Visual Basic 9.0 also supports the use of XML literals, which allow query expressions to be expressed using a declarative XML syntax directly from Visual Basic. The previous example could be constructed with the Visual Basic statement:

```
 Dim x = _
        <People>
            <%= From p In people __
                Select <Person Age=<%= p.Age %>>p.Name</Person> _
                Where p.CanCode _
            %>
        </People>
```

The examples so far have shown how to *construct* new XML values using language integrated query. The `XElement` and `XAttribute` types also simplify the *extraction* of information from XML structures. `XElement` provides accessor methods that allow query expressions to be applied to the traditional XPath axes. For example, the following query extracts just the names from the `XElement` shown above:

```
IEnumerable<string> justNames =
    from e in x.Descendants("Person")
    select e.Value;

//justNames = ["Allen Frances", "Connor Morgan"]
```

To extract structured values from the XML, we simply use an object initializer expression in our select clause:

```
IEnumerable<Person> persons =
    from e in x.Descendants("Person")
    select new Person {
        Name = e.Value,
        Age = (int)e.Attribute("Age")
    };
```

Note that both `XAttribute` and `XElement` support explicit conversions to extract the text value as a primitive type. To deal with missing data, we can simply cast to a nullable type:

```
IEnumerable<Person> persons =
    from e in x.Descendants("Person")
    select new Person {
        Name = e.Value,
        Age = (int?)e.Attribute("Age") ?? 21
    };
```

In this case, we use a default value of 21 when the Age attribute is missing.

Visual Basic 9.0 provides direct language support for the Elements, Attribute, and Descendants accessor methods of XElement, allowing XML-based data to be accessed using a more compact and direct syntax called Xml axis properties. We can use this functionality to write the preceding C# statement like this:

```
Dim persons = _
      From e In x...<Person> _
      Select new Person { _
          .Name = e.Value, _
          .Age = e.@Age.Value ?? 21 _
      }
```

In Visual Basic `x...<Person>` gets all items in the Descendants collection of `x` with the name `Person`, while the expression `e.@Age` finds all the `XAttributes` with the name `Age`. The `Value` property gets the first attribute in the collection and calls the `Value` property on that attribute.

Summary

.NET Language Integrated Query adds query capabilities to the CLR and the languages that target it. The query facility builds on lambda expressions and expression trees to allow predicates, projections, and key extraction expressions to be used as opaque executable code or as transparent in-memory data suitable for downstream processing or translation. The standard query operators defined by the LINQ project work over any `IEnumerable<T>`-based information source, and are integrated with ADO.NET (DLinq) and `System.Xml` (XLinq) to allow relational and XML data to gain the benefits of language integrated query.

Standard Query Operators in a Nutshell

`Where`	Restriction operator based on predicate function
`Select/SelectMany`	Projection operators based on selector function
`Take/Skip/ TakeWhile/SkipWhile`	Partitioning operators based on position or predicate function
`Join/GroupJoin`	Join operators based on key selector functions
`Concat`	Concatenation operator
`OrderBy/ThenBy/ OrderByDescending/ ThenByDescending`	Sorting operators sorting in ascending or descending order based on optional key selector and comparer functions
`Reverse`	Sorting operator reversing the order of a sequence
`GroupBy`	Grouping operator based on optional key selector and comparer functions
`Distinct`	Set operator removing duplicates
`Union/Intersect`	Set operators returning set union or intersection
`Except`	Set operator returning set difference
`ToSequence`	Conversion operator to `IEnumerable<T>`
`ToArray/ToList`	Conversion operator to array or `List<T>`
`ToDictionary/ToLookup`	Conversion operators to `Dictionary<K,T>` or `Lookup<K,T>` (multi-dictionary) based on key selector function
`OfType/Cast`	Conversion operators to `IEnumerable<T>` based on filtering by or conversion to type argument
`EqualAll`	Equality operator checking pairwise element equality
`First/FirstOrDefault/ Last/LastOrDefault/ Single/SingleOrDefault`	Element operators returning initial/final/only element based on optional predicate function
`ElementAt/ ElementAtOrDefault`	Element operators returning element based on position
`DefaultIfEmpty`	Element operator replacing empty sequence with default-valued singleton sequence
`Range`	Generation operator returning numbers in a range
`Repeat`	Generation operator returning multiple occurrences of a given value

`Empty`	Generation operator returning an empty sequence
`Any/All`	Quantifier checking for existential or universal satisfaction of predicate function
`Contains`	Quantifier checking for presence of a given element
`Count/LongCount`	Aggregate operators counting elements based on optional predicate function
`Sum/Min/Max/Average`	Aggregate operators based on optional selector functions
`Aggregate`	Aggregate operator accumulating multiple values based on accumulation function and optional seed