

# LINQ 2.0: Democratizing the Cloud

*Erik Meijer*

Microsoft

[emeijer@microsoft.com](mailto:emeijer@microsoft.com)

## The Past

Distributed applications are often structured into layers of tiers, typically three, the presentation or client tier, the middle or business tier and the data tier. Each of these tiers has its own data model. The presentation tier usually deals with semi-structured data as XML, or these days, as JSON. The middle tier usually deals with business entities defined in some object-oriented language. Finally, the data tier usually deals with tabular data stored in a relational database. Before we can even start thinking about concurrency and the distributed aspects of the problem, we must first solve the impedance mismatch between these three disparate data models.

One school of thought is to pick one data model as the universal one and map all other data models into it. The most popular option is to view all data as XML and then use XQuery as the glue language. Others try to come up with a new data model that encompasses all previously know data models. We do not believe that this universal data model approach is the right solution to the impedance mismatch problem. Instead of trying to unify at the data model level, a better approach is to unify at the level of algebraic operations that can be defined *the same way* over each data model. This allows us to define a single query language that can be used to query and transform any data model. All the data model need to do is to implement a small set of standard query operators, and each data model can do so in a way natural to itself.

Currently, the various data models and their query languages are tightly coupled. Our proposal boils down to separating data models from their query languages and leveraging the fundamental similarities in the operational semantics of each query language. To make this more concrete, let us look at some simple example queries in each data model. Each query will select the `name` and `age` of all customers that live in Seattle. In SQL the query looks something like:

```
Select Name, Age
From Customers
Where City = "Seattle"
```

In XQuery, the query would look like.

```
from $c in customers/customer
where $c/city == "Seattle"
return <result>{$c/Name}{$c/Age}</result>
```

Finally, in an object-oriented language one would write something along the lines of:

```
class Result { string Name; int Age; }
var r = new List<Result>();
foreach(var c in Customers)
    if(c.City == "Seattle")r.Add(new Result{c.Name, c.Age});
```

In each case, the query essentially iterates over some form of collection, filters out those elements from that collection that do not satisfy a given condition, and finally create a result collection by applying a transformation to the value pulled from the original collection.

Mathematicians recognized this “design pattern” many decades ago, and say that each data model defines a *monoid*, or more generally, a *monad*, and each query is an example of a *monad comprehension*.

Using monads and comprehensions to query arbitrary data sources was first introduced in the functional language Haskell, where they are primarily used to deal with imperative side-effecting computation. Many people wonder what the connection is between side-effects and collections, but if you think about it, it is not far-fetched to consider a whole side-effecting computation that returns say strings, as a some kind of collection that contains strings that yields a new string each time the computation is executed. The idea of using monads and comprehensions to bridge the impedance mismatch between data models is also

the basis of the *Language Integrated Query* (LINQ) technology developed by Microsoft.

## The Present

The LINQ framework defines a basic pattern of so-called *Standard Sequence Operators*, which are monadic primitives for filtering, transforming, joining, grouping, and aggregating over *arbitrary* collections of *arbitrary* types:

```
static class Sequence {
    static IEnumerable<T> Where<T>
        (this IEnumerable<T> s, Func<T, bool> p) { }
    static IEnumerable<S> Select<T, S>
        (this IEnumerable<T> s, Func<T, S> f) { }
    static IEnumerable<S> SelectMany<T, S>
        (this IEnumerable<T> s, Func<T, IEnumerable<S>> f) { }
    static IEnumerable<V> Join<T, U, K, V>
        (this IEnumerable<T> o, IEnumerable<U> i,
         Func<T, K> f, Func<U, K> g, Func<T, U, V> h) { }
    static IOrderedSequence<T> OrderBy<T, K>
        (this IEnumerable<T> s, Func<T, K> f) { }
    static IEnumerable<IGrouping<K, T>> GroupBy<T, K>
        (this IEnumerable<T> s, Func<T, K> f) { }
}
```

LINQ-enabled languages such as C# 3.0 and Visual Basic 9 support convenient high-level query comprehension syntax that the compiler translates into the underlying primitive standard sequence operators. Our previous query can then be written in C# as follows (note that the query looks exactly the same no matter whether we are querying an in-memory collection or a remote database):

```
from C in Customers
where c.City == "Seattle"
select new{ c.Name, c.Age }
```

Visual Basic 9 supports XML literals and axis members, so if the data source or result is an XML document, we can write the query in Visual Basic as follows:

```
From C in Customers.<customer>
Where C.@City = "Seattle"
```

```
Select <result>
    <%= C.<Name> %>
    <%= C.<Phone> %>
</result>
```

Both these queries are translated by the compiler into complex expressions in terms of the standard sequence operators.

While LINQ has essentially solved the impedance mismatch, much work remains to solve our original challenge of simplifying the development of data-intensive distributed applications. This is what we are set out to solve in the next couple of years in what we call LINQ 2.0.

## The Future

Query comprehensions in both C# and Visual Basic, and deep XML integration in Visual Basic make it really easy to query over basically any source of data. One painful problem that is yet unsolved is the “last mile” of data programming namely the mapping between data models at the edges, in particular the mapping between relational data and objects.

Currently, most Object-Relational frameworks (for concreteness, we use LINQ-to-SQL below) support only monolithic mapping between tables, columns, and relationships on the one hand and classes, fields, and (collection-valued) navigation properties on the other hand:

```
[Table(Name="Customers")]
public class Customer {
    [Column(IsPrimaryKey=true)]
    public string CustomerID;
    ...
    private EntitySet<Order> _Orders;
    [Association(Storage="_Orders", OtherKey="CustomerID")]
    public EntitySet<Order> Orders {
        get { return this._Orders; }
        set { this._Orders.Assign(value); }
    }
}
```

```

[Table(Name="Orders")]
public class Order {
    [Column(IsPrimaryKey=true)]
    public int OrderID;
    [Column(IsForeignKey=true)]
    public string CustomerID;
    ...
    private EntityRef<Customer> _Customer;
    [Association(Storage="_Customer", ThisKey="CustomerID")]
    public Customer Customer {
        get { return this._Customer.Entity; }
        set { this._Customer.Entity = value; }
    }
}

```

This style of mapping, no matter if it uses inline attributes or an out-of-band XML file to describe the mapping, is non-compositional in many respects. Because relational databases relate tables via foreign-key relationships, the source table (`Customers`) does not need to anticipate a-priori the relationship with the target table (`Orders`). In fact, it is possible to define an unbounded number of tables that form a one-to-one or one-to-many relationship with the given `Customers` table. Unfortunately, when mapping tables to classes we must decide upfront which relationships we want to record in the source type, thus breaking composability.

Another aspect in which traditional O/R mapping is non-compositional is that the mapping is defined between tables and classes. As a result it immediately becomes impossible to compose two mappings in sequence. Relational views on the other hand are composable, that is, it is usually possible to define a second (updatable) view on top of a first (updatable) view, and the result is indistinguishable from any other (updatable) view.

Just as we provided deep support for XML in Visual Basic, in LINQ 2.0 we hope to directly support relationships and updatable views at the programming-language level. In that case, we only need a very thin layer of non-programmable default mapping at the edge between the relation and object world and allow

programmers to express everything else in their own favourite LINQ-enabled programming language. The result is that just as LINQ 1.0 side-stepped the impedance mismatch “problem” with something better (monads and monad comprehensions), LINQ 2.0 will sidestep the mapping “problem” with something better (composable and programmable mapping).

The other open problems we intend to attack in LINQ 2.0 are distribution and concurrency. In other words, the new application design patterns for the Cloud. In an increasingly distributed and service-oriented world, users want the same experience on any device, for example they want to read the same email from their mobile phone, on their PC, or using a Web browser. Moreover, they want to continue using their applications when they are offline and synchronize when they come online again. Consequently, service providers want broad reach. They do not want their services to be unusable from certain platforms. Finally, for developers this means that we want to make irreversible decisions about our distributed applications at *the last possible responsible moment*, in other words, we do not want to build completely different applications for each client target platform. Sometimes, we may even want to delay partitioning of functionality until the very last moment when we know the capabilities of the client device.

With LINQ 2.0 we attempt to stretch the current single-tier client applications design patterns to cover the Cloud. Programmers start by designing a single-tier application and iteratively partition their code into a distributed multi-tier application by successive refactorings. This is not automagical repartitioning, just good tooling for programmer-specified repartitioning. Moreover, deployment of these applications should be completely friction-free and should late-bind to both the client and server execution platforms, that is, if at all possible running an application should require no downloads or installs except for the application code itself.

Technically, the tier-splitting refactoring is based on the reverse application of the expansion theorem from process algebra. Usually, this theorem describes the semantics of parallel composition of two parallel programs as the arbitrary interleaving of a single sequential program. Instead, we take a sequential program and split it into two parallel programs that will run across tiers. The transformation also inserts the appropriate synchronization and security logic, defined using join patterns, to ensure that the parallel execution of the new programs does not allow execution sequences that violate invariants of the original sequential program. As a result, we are able to provide a fundamental solution to avoid the dreaded “back-button” problem.

On both the client and the server, we do not assume the existence of a native .NET runtime. Instead we will target any available existing runtime. In particular we anticipate that the client is a browser that supports JavaScript, and we have implemented a complete deep embedding of MSIL into JavaScript. On the server side we anticipate large clusters of commodity hardware, and we have implemented a version of LINQ that can leverage the extreme data parallelism offered by such horizontally partitioned data.

Finally, we need to make it extremely easy for programmers to provide synchronization and replication of their transient and persistent program state for occasionally-connected and collaborative scenarios. Unfortunately, a standard solution does not exist for this that works across browsers and platforms.

## Conclusion

To summarize, LINQ 1.0 provided the necessary (but not sufficient) technology for radically simplifying distributed data-intensive applications. With LINQ 2.0 we use will take things to the next level by stretching the standard .NET programming model to cover the Cloud.