# Mixing LINQ Providers and LINQ to Objects

By Bill Wagner

September 2010

Some code I was writing for Azure gave me an excellent opportunity to discuss the differences between IEnumerable<T> and IQueryable<T> and explain many of the terms you'll see in literature about LINQ providers. Understanding these issues will help you leverage both types to the fullest.

Almost every .NET developer should be familiar with IEnumerable<T>: It's the interface that enumerates all the elements in a sequence. LINQ to Objects is built on IEnumerable<T>. Any sequence that implements IEnumerable<T> can act as a LINQ data source.

When a data source provides its own native query language, developers may create a *LINQ provider* to leverage that language. A LINQ provider translates LINQ queries into specific API calls against a data source. It enables you to write LINQ queries in C#, and have those queries executed based on the native language of the data source. For example, the LINQ to SQL provider translates LINQ queries into T-SQL. Working with IQueryable<T> feels just like working with IEnumerable<T>: You enumerate a sequence or elements, applying filters, projections, or other transformations to the elements in the sequence. It looks the same, but this behavior is implemented by using different libraries.

I was writing queries against Azure Table Storage, using the StorageClient assembly. The TableServiceContext class gives you an access to a LINQ provider for the Azure Table Storage through the CreateQuery() method. One of the queries in my Azure application sorted entities by date. The query looked something like this:

```
var records = this.CreateQuery<EntityType>()
   .Where(item => item.PartitionKey == region)
   .AsEnumerable()
   .OrderByDescending(item => item.DateUpdated);
```

Notice the AsEnumerable() call in the middle of the query above. Why do you suppose that was added to this query? Equally important, why do you suppose it was added at that location in the query? The answer lies in the design of Azure table storage, the TableServiceContext implementation of IQueryable<T>.

There is a major difference between IEnumerable<T> and IQueryable<T>. An implementation built on IQueryable<T> produces expression trees that translate the query's intent into a native language for a specific data source. In my sample above, the TableServiceContext will examine the query and generate necessary REST POST or GET requests against the table source.

That's the key to this discussion: An implementation of IQueryable<T> translates an expression tree to a native representation of the same logic for its particular source type. (In most cases, this implementation should provide better performance than the LINQ to Objects implementation for that data source.) In the case of Azure Table Storage, the implementation understands queries that include **Where** clauses using the partition key or the row key. Many methods that are a part of the query syntax pattern are not implemented in the TableServiceContext class. All the unsupported methods in your query (like the **OrderBy** in my example above) will throw exceptions.

This is common behavior with most IQueryable implementation. Some handle more cases than others, but with all IQueryable implementations, you'll find that not all C# concepts may be translated accurately to other languages. Even though LINQ to SQL and Entity Framework are powerful implementations of IQueryable, neither can translate arbitrary C# methods into SQL statements. Queries containing calls to methods you've written will usually generate exceptions.

In every case when you use a LINQ provider, you'll need to learn that provider's limitations. Whatever provider you use, you'll need to understand the general techniques for mixing IEnumerable with IQueryable.

Going back to the example I showed at the beginning of this article, now it's clear why the AsEnumerable() call was added: Without it, the query always throws an exception, and never returns any results:

```
// This query always throws an exception
// when run against azure table storage.
var records = this.CreateQuery<EntityType>()
   .Where(item => item.PartitionKey == region)
   .OrderByDescending(item => item.DateUpdated);
```

Now, let's drill into some of the details on why to use AsEnumerable(), and why AsEnumerable() goes where I put it in the query. In general, the answer is easy: you want to leverage the LINQ provider to minimize network traffic, and thereby improve performance. Of course, the standard disclaimer now applies: like all performance recommendations, these are general guidelines, and you should measure your specific scenario.

IQueryable implementations produce expression trees that enable the LINQ provider to examine the expression and implement it in the most efficient manner for the data source. In the case of the Azure Table Storage, the storage API supports **Where** clauses that test for equality on the **PartitionKey** and the **RowID**. (Entity Framework and other LINQ providers support richer queries). The Azure Table Storage API supports this limited set of queries because that is the only filtering supported by the underlying REST API. This IQueryable implementation forces you to structure your queries so that you properly leverage the underlying system.

The general rule is to place any query clauses that are implemented by the LINQ provider first. Follow those clauses with the AsEnumerable() call, and then finish with those clauses that are not implemented by the LINQ provider. For example, this variation of my first example would increase network traffic between the VMs in the Azure fabric. Instead of transmitting only those entities that match a given region, you'll transfer every entity, and filter the entities on the client side.

```
// Inefficient: every entity is returned
// from storage, and the entire table
// is examined using LINQ to Objects.
```

```
var records = this.CreateQuery<EntityType>()
   .AsEnumerable()
   .Where(item => item.PartitionKey == region)
   .OrderByDescending(item => item.DateUpdated);
```

The Azure Table Storage is optimized for very large tables, so the Azure team expects that your Azure Tables will have very large numbers of records. Transferring an entire table between machines in the Azure cloud would be a very expensive operation. You want to avoid transferring all those records when you really only want a small subset. You would have similar problems if you sprinkled your Entity Framework queries with AsEnumerable() clauses. That would mean you would be retrieving far too much data from your database. You'd be filtering that data using LINQ to Objects instead of leveraging the database engine.

That answers both questions I posed when I showed this sample. The final step is to explain the different ways to force your query logic to use LINQ to Objects instead of a LINQ provider. I prefer using AsEnumerable(). However, you can also force LINQ to Objects by using ToList() or ToArray(). The following two statements will produce the same results, although they produce them in very different ways:

```
// evaluate query and use list storage:
var records = this.CreateQuery<EntityType>()
   .Where(item => item.PartitionKey == region)
   .ToList()
   .OrderByDescending(item => item.DateUpdated);
// evaluate query and use array storage:
var records = this.CreateQuery<EntityType>()
   .Where(item => item.PartitionKey == region)
   .ToArray()
   .OrderByDescending(item => item.DateUpdated);
```

I prefer the call to AsEnumerable() because its intent is more clear. AsEnumerable() casts an IQueryable<T> to an IEnumerable<T>, while ToList() and ToArray() evaluate the query and store the results in a temporary storage location. Depending on the memory storage needs, AsEnumerable() is likely faster than ToList() or ToArray(). However, that's a micro optimization that needs to be measured. Using AsEnumerable() makes it clear that you want to execute the remaining query operations using LINQ to Objects.

There's one last API to mention, and one last reason to prefer AsEnumerable(): There is also an AsQueryable() method in LINQ. AsQueryable() converts an IEnumerable<T> to an IQueryable<T>. If the source already implements IQueryable, you will work with that IQueryable implementation. However, if the sequence is not an IQueryable implementation, AsQueryable() returns an IQueryable implementation that works against in-memory sequences and this is exactly what happens when the source is a List<T> or an Array type.

In other words, AsQueryable() reverses the effect of AsEnumerable() so you can work with a specific LINQ provider again. If you used ToList() or ToArray() to store the result sequence in a List<T> or an array, you can only work with that sequence using LINQ to Objects.

Remember the general guidance for working with LINQ providers.  Whenever your provider implements IQueryable, you want to leverage the IQueryableimplementation for everything you can. However, not every LINQ provider will implement the entire LINQ query pattern.  Many times (like in the case of Azure Table Storage) this is by design to highlight the design goals of the storage API. That means you should put the query clauses implemented by the LINQ provider first, follow those with a call to AsEnumerable(), and finish the query with any other logic that must be implemented using LINQ to Objects.