acmqueue The Pain of Implementing LINQ Providers

It's no easy task for NoSQL

Oren Eini, aka Ayende Rahien

I remember sitting on the edge of my seat watching the 2005 PDC (Professional Developers Conference) videos that first showed LINQ (Language Integrated Query). I wanted LINQ: it offered just about everything that I could hope for to make working with data easy.

The impetus for building queries into the language is quite simple; it is something that is used all the time; and the promise of a unified querying model is good enough, even before you add all the language goodies that were dropped on us. Being able to write in C# and have the database magically understand what I am doing? Awesome! Getting compilation errors from Visual Studio, rather than runtime errors at testing (or worse, production)? Wonderful! Getting rid of most SQL injection issues? Amazing!

Then .NET 3.5 rolled around, and we had LINQ, and I found myself in quite a pickle. There are actually two sides to the LINQ story: the consumer side and the implementer side. For the consumer, it is all sunshine and flowers, but for the implementers it is doom, gloom with just a tad of despair. And even for the consumers, past the initial love at first sight phase, they are issues that require handling. Let me focus first on the consumer side.

CONSUMING THE LINQ API

Despite appearances, LINQ isn't actually SQL in C# (or VB.NET). It is just a syntax transformation from a set of well-known keywords to well-known methods. The usage of `Select`, `Where`, and `GroupBy` has made it taste very similar to SQL, but there are drastic differences between the way LINQ works conceptually and the set-based relational logic that rules SQL databases.

Let us examine a simple LINQ query:

```
from user in users
where user.IsActive
select user;
```

We don't know the type of the `users` variable, but in general it would be either an `IEnumerable<T>` or an `IQueryable<T>`. `IEnumerable<T>` is the basic interface for everything in memory enumeration and iteration. But `IQueryable<T>`, along with the syntax transformation done by the compiler, is something else altogether. It allows implementers of `IQueryable<T>` to access the expression tree (the compiler abstract syntax tree) for the query.

Let us look at how this query actually translates. Table 1 shows the transformation depending on the type of the `users` variable.

Because `IQueryable<T>` implementers are called with the expression tree of the query, they have the chance to decide, based on that expression tree, how actually to execute that query. In many cases,

TABLE 1 Compiler transformation for IEnumerable<T> queries and IQueryable<T> queries

Our code	Compiler transformation result
<pre>IEnumerable<User> users = null; var activeUsers = from user in users where user.IsActive select user;</pre>	<pre>IEnumerable<User> users = null; var activeUsers = users.Where(delegate(User user) { return user.IsActive; });</pre>
<pre>IQueryable<User> users = null; var activeUsers = from user in users where user.IsActive select user;</pre>	<pre>IQueryable<User> users = null; var isActiveProperty = Expression.Parameter(typeof (User), "user"); var activeUsers = users.Where(Expression.Lambda<Func<User, bool>>[Expression.Property(isActiveProperty, typeof (User).GetProperty("IsActive"), new[] { isActiveProperty })]);</pre>

the IQueryable<T> implementation will translate the expression tree into the querying medium of the underlying data store (for example, SQL, XPath, etc.) and not execute the code directly.

The LINQ to Objects (querying against an IEnumerable<T>) behavior is considered to be the definitive implementation for how a LINQ query should behave. That implementation is actually executing the code in memory, and as such, has access to the entire Base Class Library and can execute any operation it wishes. That is not the case with other LINQ implementations, since it is the IQueryable<T> implementation that determines which operations are supported and which are not. The problem is, however, that beyond the most trivial of queries, it is actually quite hard to map the intent of the user to a query that makes sense on the target data store. Operations that make perfect sense when all of the data is accessible in main memory make poor choices when the data is distributed in different database tables or even different servers.

For example, consider the following nontrivial LINQ query, which finds all the incomplete late tasks for the active users on projects that they own:

```
from user in users
where user.IsActive
from task in user.Tasks
where task.Project.Owners.Contains(user) &&
    task.DueDate < DateTime.Now &&
    task.Completed == false
select new { task.Title, task.Status, user.UserName };
```

This is a very clean way of expressing the intent, and it reads quite nicely. When running on a small data set in memory, it also performs beautifully. The problem is, for the most part, we are *not* running on small data sets or in memory.

Consider for a moment the effect of such a query on a large data set, even in memory. With LINQ to Objects, most primitive operations have a O(N) cost. In the previous query, we have composite

calls that feed on one another.

What this leads to is that even when we are talking about pure in-memory models, large data sets require us to take optimized approaches. We could build an `IQueryable<T>` implementation that would use in-memory indexes for performing this query. For the most part, however, we are not writing such queries against an in-memory representation; we are writing such queries against a database. A database can handle large data sets easily; that is what it is *for*.

Using the LINQ to Entities framework, the previous query example is translated into the following SQL:

```
SELECT [Extent1].[Id] AS [Id],
       [Filter1].[Title] AS [Title],
       [Filter1].[Status] AS [Status],
       [Extent1].[UserName] AS [UserName]
FROM   [dbo].[Users] AS [Extent1]
       INNER JOIN (SELECT [Extent2].[UserId] AS [UserId],
                          [Extent3].[Title] AS [Title],
                          [Extent3].[Status] AS [Status],
                          [Extent3].[ProjectId] AS [ProjectId]
FROM     [dbo].[UsersTasks] AS [Extent2]
          INNER JOIN [dbo].[Tasks] AS [Extent3]
                ON [Extent3].[Id] = [Extent2].[TaskId]
          WHERE ( [Extent3].[DueDate] < CAST(sysdatetime() AS DATETIME2) )
                AND ( 0 = [Extent3].[Completed] )) AS [Filter1]
       ON [Extent1].[Id] = [Filter1].[UserId]
WHERE  ( [Extent1].[IsActive] = 1 )
       AND ( EXISTS (SELECT 1 AS [C1]
                     FROM   [dbo].[ProjectOwners] AS [Extent4]
                     WHERE  ( [Filter1].[ProjectId] = [Extent4].[ProjectId] )
                           AND ( [Extent4].[UserId] = [Extent1].[Id] )) ) )
```

So far, so good, but it is very easy to write LINQ queries that no provider can translate, simply because there is no mapping between whatever it is you want to do in the query and the operations provided by the underlying database. For example, consider something as simple as:

```
from user in users
where user.UserName[5] == 'A'
select user;
```

With LINQ to Objects, this query runs just as you would expect, but trying this query using LINQ to Entities would throw an error. That error would happen only at runtime, because the *code* is perfectly fine, and the compiler has no way of knowing if a specific `IQueryable<T>` implementation supports this. Or, what is actually worse, the provider will be able to translate it, but into a query that is going to perform poorly.

There is no technical reason why LINQ to Entities cannot support queries on string indices,

but consider what it would mean from the database point of view. That would be translated into something like `SUBSTRING(Username, 5,1) = 'A'`, and that would mean forcing a table scan to answer this query.

The problem, as always, is that when you have a layer of abstraction, it can easily abstract away important details. The result is that while in theory you could move your queries from one data store to another, in practice you have to be well aware of the actual behavior of the LINQ provider and the underlying database.

It isn't that LINQ generates inefficient queries, or even that it allows the generation of inefficient queries. This is simply an issue of abstraction hiding what is really happening. LINQ gives a lot of latitude to the users, and it is easy to create queries that simply cannot be executed efficiently by the underlying data store. That is mostly because there is a big gap between the semantics of querying in memory collections (which is the abstraction that LINQ uses) and querying data stores such as relational and nonrelational databases.

As a consumer of LINQ, I absolutely adore it, despite some of the issues just discussed. But *consuming* LINQ is the part that is all sunshine and roses. The task of *implementing* a LINQ provider is one of Herculean proportion and involves much use of fertilizer, sweat, and hard work.

IMPLEMENTING CAN BE PAINFUL

Since .NET 3.5 came out, I've had to implement a LINQ provider several times. I implemented the first one for NHibernate, and I implemented a few that didn't target a relational database, the most important of which is the LINQ provider for RavenDB (<http://ravendb.net>).

In each and every case, the process was extremely painful. To build a LINQ provider, you first have to understand how the compiler looks at the language, because that is all a LINQ provider is. You are given an expression tree, which is just the compiler view of a piece of code, and are told: "Make something of that and give me a result of T."

The problem is, there is actually very little relation between what the compiler hands to you and what the user actually wrote. The compiler view of the code is often drastically different than what you would expect.

Consider this simple LINQ query:

```
from user in users
where user.Id == 1
select user;
```

Though this is probably the very simplest LINQ query possible, the compiler translates it to something that looks drastically different and quite complex, as shown in listing 1. This is one of the more problematic aspects of writing a LINQ provider. You get something that doesn't look very much like what the user provided.

If you are used to writing compilers, this all makes perfect sense. When you have more complex queries, however, even queries that do not seem much different on the surface, the result is drastically different and far more complex. The following query produces an expression tree that goes on for five pages:

```

from user in users
where user.IsActive
from task in user.Tasks
where task.Project.Owners.Contains(user) &&
      task.DueDate < DateTime.Now &&
      task.Completed == false
select new { task.Title, task.Status, user.UserName };

```

The first query is three lines and has an expression tree that goes on for about a page. The second is five lines, and has an expression that goes on for five pages. You can see the full expression trees for both queries in the files, http://queue.acm.org/downloads/2011/Trivial_Query.html and http://queue.acm.org/downloads/2011/Non_Trivial_Query.html, respectively. For compiler writers, this just makes sense. For developers who aren't well versed in compiler theory, this is a big stumbling block.

Most mature LINQ providers actually use a two-stage process. In the first part of the process the provider runs over the expression tree, compiling its own in-memory representation of the query, as

LISTING 1 The expression tree from a simple query

1. MethodCallExpression
 - a. Method : MethodInfo : "Where"
 - b. Arguments : ReadOnlyCollection
 - i. ConstantExpression
 1. Value : Object : "LINQConsoleApplication1.User[]"
 2. NodeType : ExpressionType : "Constant"
 3. Type : Type : "EnumerableQuery"
 - ii. UnaryExpression
 1. Operand : ExpressionLambda
 - a. Expression>
 - i. Body : ExpressionEqual
 1. BinaryExpression
 - a. Left : ExpressionMemberAccess
 - i. MemberExpression
 2. Expression : ExpressionParameter
 - a. ParameterExpression
 - i. Name : String : "user"
 - ii. NodeType : ExpressionType : "Parameter"
 - iii. Type : Type : "User"
3. Member : MemberInfo : "Int32 Id"
4. NodeType : ExpressionType : "MemberAccess"

listing continues

```

5. Type : Type : "Int32"
    a. Right : ExpressionConstant
    i. ConstantExpression
6. Value : Object : "1"
7. NodeType : ExpressionType : "Constant"
8. Type : Type : "Int32"
    a. Method : MethodInfo : null
    b. Conversion : LambdaExpression : null
    c. IsLifted : Boolean : "False"
    d. IsLiftedToNull : Boolean : "False"
    e. NodeType : ExpressionType : "Equal"
    f. Type : Type : "Boolean"
    i. Parameters : ReadOnlyCollection
        1. ParameterExpression
            a. Name : String : "user"
            b. NodeType : ExpressionType : "Parameter"
            c. Type : Type : "User"
        ii. NodeType : ExpressionType : "Lambda"
        iii. Type : Type : "Func"
    1. Method : MethodInfo : null
    2. IsLifted : Boolean : "False"
    3. IsLiftedToNull : Boolean : "False"
    4. NodeType : ExpressionType : "Quote"
    5. Type : Type : "Expression>"
    a. NodeType : ExpressionType : "Call"
    b. Type : Type : "IQueryable"

```

extracted from the expression tree. The second part is performing whatever it is that the user has actually wanted to do.

While it might not surprise you, almost all of the hard work involves translating the expression tree the compiler gave us into something that actually closely resembles what the user initially provided.

The start of a better approach can be seen in the re-linq framework (<http://relinq.codeplex.com>), which takes care of most of the work of translating from the internal compiler representation and the user intent. Personally, I would have constructed an AST (abstract syntax tree) mimicking the actual query semantics from the get-go instead of providing `IQueryable<T>` with the raw compiler output. Listing 2 is an example of what the trivial query AST looks like when using the NRefactory

project (<http://wiki.sharpdevelop.net/NRefactory.ashx>) to process it. Working with something like this is drastically simpler than working with the compiler output, because we now have the relevant meaning. As it stands, we have to extract the meaning ourselves, which isn't trivial.

There are two lessons to take from this. The first is that giving developers the raw compiler output as an expression tree is a fairly poor choice from the point of view of the developers needing to build LINQ providers on top of those. Even fairly minimal work (such as seen in listing 2) would have made creating LINQ providers drastically easier. It isn't that I think that you could make it work for all cases; I believe that you would still have this complexity in some cases, if only because we need to support arbitrarily complex inputs.

The vast majority of cases, however, falls into fairly predictable patterns, and solving this problem at the compiler/framework level would have made the cases of complex expression trees rare rather than everyday occurrences. That is why I strongly encourage the use of third-party libraries, such as the re-linq project, that take on a lot of the work of putting the common transformations into a more reasonable format.

There are other issues in implementing a proper LINQ provider:

- Different languages provide different expression trees for the same logical code. For example, equality is handled differently in C# and VB, depending on whether you are comparing to a string or not.
- Different versions of the compiler can emit different expression trees for the same queries.
- There is absolutely *no* documentation about expression trees that helps to build a LINQ provider.

LISTING 2 Trivial query AST, as generated by NRefactory

- FromClause=QueryExpressionFromClause
 - Identifier=user
 - InExpression
 - IdentifierExpression Identifier=users
 - MiddleClauses
 - QueryExpressionWhereClause Condition
 - BinaryOperatorExpression
 - Left
 - MemberReferenceExpression
 - TargetObject Identifier=user
 - MemberName=Id
 - Op=Equality
 - Right
 - PrimitiveExpression
 - Value=1
 - SelectOrGroupClause
 - QueryExpressionSelectClause Projection
 - IdentifierExpression Identifier=user

- Your input is basically any legal expression in C# or VB.Net, which means that the option space you have to deal with is huge. You can throw an exception if you don't understand the query, of course, but that means that users will discover unsupported scenarios only at runtime, defeating much of the point of having queries integrated into the language.

To make matters worse, users often come up with the most complex queries and expect them to work. Because of the reasons outlined above, implementing a LINQ provider is mostly a matter of implementing it one query at a time. The reason that you have to approach building a LINQ provider in this way is that you cannot really treat many operations as black boxes where given *X* output *Y*. For example, consider the case of the `Count()` operation. It can be applied to the query globally, to subqueries in the query, to a collection property as part of the projection or processing the query, or inside the group by clause. Each case needs to be handled differently. Furthermore, the user usually comes up with queries that may require re-architecting the way you are doing things.

LINQ TO NOSQL

So far, I have talked mostly about the problems inherent in implementing any LINQ provider, but this article is particularly focused on building a LINQ provider for a NoSQL data store.

One of the problems with building such a LINQ provider is that quite a bit of the LINQ API makes absolutely no sense outside of a relational database. For example, how do you handle grouping in a document database? Or ordering in a key/value store? You can certainly do either one (maybe not very efficiently, but you can). The LINQ API makes it seem as though it is just as easy to handle as if it were all in memory. There are ways to mark off parts of the API as unusable for a particular provider, but it is an opt-out approach and requires quite a bit of repetitive coding.

Now, given the intent in building LINQ (allow querying over anything), there really isn't a good way *not* to have this problem. Different data stores have different semantics, and it is fairly hard to consider all of those semantics in a generic API. There are ways provided to do so, and that is sufficient, if annoying.

Before we get to marking parts of the LINQ API as outside the scope of our providers, we still have to get to the first order of business: How do you handle *querying*? We will touch on that topic in the next section.

As I mentioned, I had the chance to build a few LINQ providers. Among those, RavenDB and RavenLight are two recent implementations that provide two very similar yet drastically different approaches to implementing a LINQ provider.

THE RAVENLIGHT LINQ PROVIDER

RavenLight is an embedded document database meant to run inside Silverlight applications. In other words, it is running completely on the client side with no server-side component whatsoever (it can replicate to and from a RavenDB instance, but that is beside the point for this article). Figure 1 shows a typical Silverlight application using RavenLight.

When building RavenLight's LINQ provider, we made the following assumptions:

- All the data is available locally.
- The total amount of the data in the data store is likely to be small (up to tens of thousands of items, not millions or more).
- Most Silverlight queries will be done in response to a user action.

FIGURE 1

A Typical Application Using RavenLight

I'll add a tautological statement as well: the broader the support for the LINQ API, the better it is.

With RavenLight, we actually decided to skip the bother of implementing a LINQ provider and just use LINQ to Objects.

How does this work? Because all the data is available locally and the amount of data stored is small, we are actually going to run queries directly against the data store. What this means in practice is that we can do something like this:

```
public class RavenLight
{
    // other stuff

    public IQueryable<T> Query<T>()
    {
        return ReadAll<T>().AsQueryable();
    }

    private IEnumerable<T> ReadAll<T>()
    {
        var fullName = typeof(T).FullName;
        foreach (var item in datastore["ByType"]
            .SkipTo(fullName)
            .TakeWhile(x => x.Type == fullName))
        {
            yield return item.Deserialize<T>();
        }
    }
}
```

As you can see, we aren't actually implementing anything here. The `ReadAll<T>()` method simply

does a first-level filtering based on the requested type, and that is it. I simplified the implementation drastically to make the point, obviously, but this is how it works at a conceptual level. Presto, the entire LINQ API is supported, because we are actually returning `IEnumerable<T>` under the covers.

This approach has some limitations, but we actually left the option open to sit down at a later date and implement a LINQ provider that doesn't perform a linear scan operation all the time. That said, when N is small, $O(N)$ approach is actually quite reasonable and efficient.

For that matter, if we want to get a quick speed boost we can probably do this:

```
public IQueryable<T> Query<T>()
{
    return ReadAll<T>().AsQueryable().AsParallel();
}
```

This option is efficient in terms of both developer time and actual usage. Remember, in a Silverlight application, we are literally running at the client desktop. We don't have to worry much about performance beyond ensuring that we can respond to the query in a time shorter than the human response time. That gives us *tens of milliseconds* at least, and that amount of time means that we can "waste" a lot of computing power.

This approach, however, is suitable only if you are actually able to meet all of the requirements that I have specified. For the most part, I think you'll find that is not the case.

Far more often, I see people turning to one of two options: LINQ serialization and full-blown LINQ providers. Serializing LINQ queries over the wire assumes that you have something on the other hand that can process them efficiently. Usually, this isn't the case, and even if it is, real world data set sizes would make this a very inefficient choice.

QUERYING NOSQL DATA STORES

A relational database allows you to make just about any query that you want. A NoSQL data store will generally put limits on your querying capability. For example, you may be able to query only on specific fields (predetermined) or in a specific way.

With a NoSQL solution such as key/value store (Memcached, Redis, etc.), you can query only by ID. With a BigTable data store, your querying options are limited to querying by key or key range, etc.

In many cases, it literally doesn't pay to try to provide a LINQ provider abstraction on top of highly limited querying capabilities. If we are using something like Redis (key/value store, only querying option is by ID), there really isn't much point in trying to create a LINQ provider. It would serve only to create the illusion that we can support additional query operations.

With RavenDB, the querying capability allows us to perform most queries (property equal to value, range queries on properties, etc.) without much trouble, which means that it makes a lot of sense to support a LINQ provider. That said, there are actually many operations that we cannot support. For example, a `GroupBy` operation is not something that can usually be defined on the fly.

Building the LINQ provider for RavenDB wasn't an easy task; in fact, we budgeted more time to it than to building the actual database. Yes, it took less time to build the database and the querying mechanism than to build the LINQ provider for that same database.

TABLE 2 RavenDB LINQ provider results

LINQ Query	RavenDB Syntax
where user.Name == "Ayende"	Name:Ayende
where user.Age > 18	Age:[18 TO NULL]

RavenDB allows you to issue queries such as those shown in table 2.

As you can imagine, the task of the LINQ provider is to take the LINQ query expressions and emit the RavenDB syntax. That is more or less what we have done. Building a trivial LINQ query provider is hard but not too hard.

Processing a single expression is actually quite easy once you have the framework in place to support it. It gets somewhat harder with compounded expressions, but it is still manageable.

The problem really begins to hit when you start considering how to (or even if you should) support operations that are not really provided by the underlying data store. For example, joining and grouping are two operations that RavenDB doesn't do, but they are native parts of LINQ (and are actually part of the language). There are simpler examples. One of the features of RavenDB is that it performs absolutely no computation during queries; all the data for the query is already prepared. That means that we can achieve very good querying speeds.

The issue is that the user can specify a statement such as:

```
from user in users
where user.Id % 2 == 0
select user.Name;
```

Again, this is mostly an issue of determining how much you are going to support. Obviously, we can't send that query to RavenDB, which has no built-in support for math operation during queries. You can add that support to the client portion of the API, of course, but that means that your provider implementation has to become more and more sophisticated, and it already took more time than just building the database.

Because of those issues, we have decided that we are *not* going to try to support all of the options available in the LINQ API. Instead, we are going to define a common set of operations that we will support, and anything outside that isn't going to work.

For example, consider a query for a user in a particular role:

```
var q = from user in users
        from role in user.Roles
        where role == "Admins"
        select user;
```

We can't really support this query; it is too complex for us to understand easily (and it opens up quite a lot of possibilities that we would have to support). Figuring out that role came from user is actually pretty complex, because of the way this is handled internally. You have to keep track of aliases, transparent identifiers, renames, handle multiple nested statements, etc.

We still wanted to support this, however, so we provided an alternative way of doing that:

```
var q = from user in users
        where user.Roles.Any(role => role == "Admin")
        select user;
```

The reason this is so much easier than the previous query is simple. We have all of the information for processing the expression directly in the node. In the previous query, we would have to spend quite a bit of effort just figuring out from where we got `role`. What we would get is actually something called transparent identifier, and we would have to track down from where it originated.

From the user perspective, there isn't much of a difference, but there is a huge difference in the amount of complexity involved in understanding this statement compared with the previous one.

We have made similar decisions in other places, and we have thoroughly documented them. Users get to use LINQ, with all the benefits associated with it (strong typing and intelligence among the most important), but we don't have to tackle the entire complexity involved in trying to support anything that can be written as a LINQ query.

SUMMARY

As mentioned, it actually took more time to build the LINQ provider for RavenDB than it took to build RavenDB itself. Indeed, by far the most complicated piece of code in RavenDB even now is the LINQ provider implementation.

Nevertheless, when talking about the .NET framework, if you are creating a database or a library to access a database, you pretty much have no choice but to provide a LINQ implementation. Even after saying that, I would recommend evaluating carefully whether you really need a LINQ provider or, to be more exact, whether a LINQ provider implementation would be worthwhile.

If you are connecting to a key/value store or something else that has limited querying options, you probably shouldn't bother. Having a LINQ query available will only hint at things you can do that you actually cannot. It is easier not to have one than to explain why this or that operation is not supported.

If you do decide to implement a LINQ provider against a NoSQL database, I would strongly encourage you to choose a conventional way to define each supported operation. You will be much better off if your users' input can be made to follow specific patterns.

Another option is to selectively disable parts of the LINQ API for your LINQ provider (by creating your own LINQ methods and marking them with the `[Obsolete]` attribute).

Finally, there are libraries such as `re-linq` (<http://relinq.codeplex.com/>) that can help build LINQ providers. `Re-linq` in particular takes care of doing most of the work of going from a LINQ expression tree to something far more palatable to work with.

As I said in the beginning of this article, I really like LINQ, but only from the consumer side. Writing LINQ providers is not a task for the faint of heart and shouldn't be attempted unless you know what you are actually doing.

You might be better off with a limited LINQ implementation instead. Consider creating a provider that supports very few operations, maybe even one that supports only equality tests. That can be quite impressive, and it isn't that hard to do.

It is when you want to support more that the complexity starts to pile up, so think carefully about

precisely what you want to do, the costs associated with each option, and the benefits that will be gained.

Users *love* LINQ, after all...

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

OREN EINI has more than 15 years of experience in the development world, concentrating on the Microsoft and .NET ecosystem. His main focus is on architecture and best practices that promote quality software and zero-friction development. Using his pseudonym, Ayende Rahien, Eini is a frequent blogger at <http://www.ayende.com/Blog/>. He is the founder of Hibernating Rhinos LTD, an Israeli-based company offering friction-free data-management solutions.

© 2011 ACM 1542-7730/11/0600 \$10.00