acmqueue

The World According to LINQ

Big data is about more than size, and LINQ is more than up to the task.

Erik Meijer

Programmers building Web- and cloud-based applications wire together data from many different sources such as sensors, social networks, user interfaces, spreadsheets, and stock tickers. Most of this data does not fit in the closed and clean world of traditional relational databases. It is too big, unstructured, denormalized, and streaming in realtime. Presenting a unified programming model across all these disparate data models and query languages seems impossible at first. By focusing on the commonalities instead of the differences, however, most data sources will accept some form of computation to filter and transform collections of data.

Mathematicians long ago observed similarities between seemingly different mathematical structures and formalized this insight via category theory, specifically the notion of monads as a generalization of collections. Languages such as Haskell, Scala, Python, and even future versions of JavaScript have incorporated list and monad comprehensions to deal with side effects and computations over collections. The .NET languages of Visual Basic and C# adopted monads in the form of LINQ (Language-Integrated Query) as a way to bridge the gap between the worlds of objects and data. This article describes monads and LINQ as a generalization of the relational algebra and SQL used with arbitrary collections of arbitrary types, and explains why this makes LINQ a compelling basis for big data.

LINQ was introduced in C# 3.0 and Visual Basic 9 as a set of APIs and accompanying language extensions that bridge the gap between the world of programming languages and the world of databases. Despite the continuing excitement about LINQ in the external developer community, the full potential of the technology has not yet been reached. Thanks to the foundational nature of LINQ, there is still enormous potential for its mapping scenarios outside O/R (object-relational), especially in the area of big data.

The advent of big data makes it more important than ever for programmers to have a single abstraction that allows them to process, transform, compose, query, analyze, and compute across at least three different dimensions: *volume*, big or small, ranging from billions of items to a handful of results; *variety* in models, structured or unstructured, flat or nested; and *velocity*, streaming or persisted, push or pull. As a result, we see a mind-blowing number of new data models, query languages, and execution fabrics. LINQ can virtualize all these aspects behind a single abstraction.

Take, for example, Apache's Hadoop ecosystem. It comes with at least eight *external* DSLs (domain-specific languages) or APIs: a set of low-level Java interfaces for MapReduce computations; *Cascading,* a "data-processing definition language, implemented as a simple Java API"; *Flume,* a "simple and flexible architecture based on streaming data flows"; *Pig,* a "high-level language for expressing data analysis programs"; *HiveQL*, a "SQL-like language for easy data summarization, ad hoc queries, and the analysis of large data sets"; *CQL,* a "proposed language for data management in Cassandra"; *Oozie,* an XML-based "coordinator engine specialized in running workflows based on time and data triggers"; and *Avro,* a schema language for data serialization.

To create an end-to-end application, programmers need to use several of these external DSLs in addition to a general-purpose programming language such as Java to glue everything together. If data comes from an external RDBMS (relational database management system) or push-based source, then even more DSLs such as SQL or StreamBase are required. Using LINQ and C# or Visual Basic on the other hand, programmers can use *internal* DSLs to program against any shape or form of data inside a general-purpose OO (object-oriented) language that comes with tooling (Visual Studio or cross-platform solutions from Xamarin such as MonoDevelop, Mono Touch for iPhone, or Mono for Android) and an extensive collection of standard libraries (.NET Framework).

STANDARD QUERY OPERATORS AND LINQ

Assume that given a file of text—say, words.txt—you need to count the number of distinct words in that file, find the five most common ones, and visualize the result in a pie chart. If you think about this for a minute, it becomes clear that this is really an exercise in transforming collections. This is exactly the kind of task for which LINQ was designed. To keep things simple, we have implemented this example using LINQ to Objects to process the data in memory; however, with minimal modification the same code runs on LINQ to HPC (high-performance computing) over terabytes of data stored in commodity clusters.

The standard File.ReadAllText method provides the content of the file as a single giant string. You first need to chop up this string into individual words by breaking it at delimiter characters such as space, comma, period, etc. Once you have a list of words, you need to clean it up, removing all empty words. Finally, normalize all words to lowercase.

Using the LINQ sequence operators, you can transliterate the description from the previous paragraph directly into code:

```
var file = System.IO.File.ReadAllText("words.txt");
var words = file.Split(delimiters)
             .Where(w⇒!w.IsNullOrWhiteSpace())
             .Select(w⇒w.ToLower());
```

Instead of using the sequence operators directly, LINQ also provides a more "declarative" query comprehension syntax. Using comprehensions, you can rewrite the code as follows:

```
var words = from w in file.Split(delimiters)
            where !w.IsNullOrWhiteSpace()
            select w.ToLower();
```

Once you have converted the file into a sequence of individual words, you can find the number of occurrences of each word by first grouping the collection by each word and then counting the number of elements in each group (which contains all occurrences of that word):

```
var wordcount = from w in words
                group by w into group
                select new{ Word = group.Key, Count = group.Count() };
```

Without using query comprehension syntax, the code would look like this:

```
var wordcount = words.GroupBy(w⇒w).Select(group⇒
                    new{ Word = group.Key, Count = group.Count() };
```

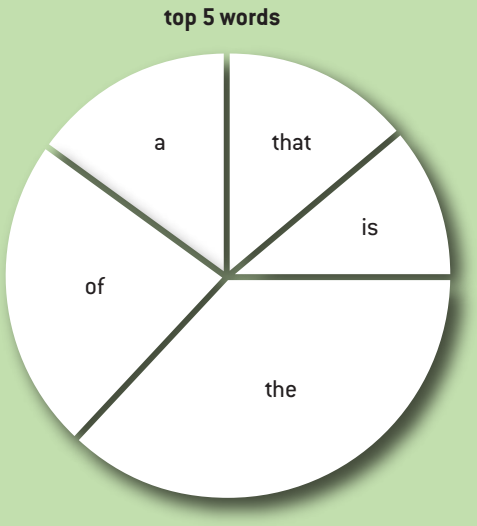
To find the five most frequent words, you can order each record by Count and take the first five elements:

```
var top5 = wc.OrderByDescending(p⇒p.Count).Take(5);
```

Now that you have a collection of the top five words in the file, you can visualize them in a pie chart, as in figure 1. A pie chart is really nothing more than a collection of slices, where each slice consists of a number that represents the proportion of the total pie and a legend that describes what the slice represents. This means that by defining the charting API to be LINQ-friendly, you can create charts by writing a query over the Google image charts API:

```
var chart = new Pie(from w in top5
                        select new Slice(w.Count){ Legend = r.Word })
              { Title = "Top 5 words" };
var image = await chart;
```

FIGURE 1 **Example of Pie Chart**



The await keyword is used in an unorthodox way to make the expensive coercion from Google.Linq.Charts.Pie into an image that requires a network round-trip explicit.

This example just scratches the surface of LINQ. It provides a library of sequence operators such as Select, Where, GroupBy,... to transform collections, and it provides syntactic sugar in the form of query comprehensions that allows programmers to write transformations over collections at a higher level of abstraction.

To truly understand the power of LINQ, however, let's take a step back and investigate its origins and mathematical foundations. Don't worry, you need knowledge of only high-school-level mathematics.

DATACENTRIC INTERPRETATION

Relational algebra, which forms the formal basis for SQL, defines a number of constants and constructors for sets of values $\{\Sigma\}$, such as the *empty set* $\varnothing \in \{\Sigma\}$; *injection* of a value into a singleton collection $\{\_\}\in\Sigma\to\{\Sigma\}$; and the *union* of two sets into a new combined set $\cup\in\{\Sigma\}\times\{\Sigma\}\to\{\Sigma\}$. There are also a number of relational operators such as *projection,* which applies a transformation to each element in a set $\pi\in(\Sigma\to\Lambda)\times\{\Sigma\}\to\{\Lambda\}$; *selection,* which selects only those elements in a set that satisfy a given property $\sigma\in(\Sigma\to B)\times\{\Sigma\}\to\{\Sigma\}$; *Cartesian product,* which pairs up all the elements of a pair of sets $X\in\{\Sigma\}\times\{\Lambda\}\to\{\Sigma\times\Lambda\}$; and *cross-apply,* which generates a secondary set of values for each element in a first set $@\in(\Sigma\to\{\Lambda\})\times\{\Sigma\}\to\{\Lambda\}$.

Figure 2 depicts the relational algebra operators using clouds to denote sets of values.

A SQL compiler translates queries expressed in the familiar SELECT-FROM-WHERE syntax into relational algebra expressions; to optimize the query it applies algebraic laws such as distribution of selection: $\sigma(p,\sigma(q,xs)) = \sigma(x\mapsto p(x)\wedge q(x),xs)$; and then translates these logical expressions into a physical query plan that is executed by the RDBMS.

For example, the SQL query SELECT Name FROM Friend WHERE Likes(Friend, Sushi) is translated into the relational algebra expression $\pi(f\Rightarrow f.Name,(\sigma(f\Rightarrow Likes(f,Sushi),Friend)$. To speed up the execution of the query, the RDBMS may use an index to quickly look up friends who like Sushi instead of doing a linear scan over the whole collection.

The cross-apply operator @ is particularly powerful since it allows for correlated subqueries where you generate a second collection for each value from a first collection and flatten the results into a single collection $@(f,\{a,...,z\})=f(a)\cup...\cup f(z)$. All other relational operators can be defined in terms of the cross-apply operator:

$$xs\ X\ ys = @(x\Rightarrow\pi(y\Rightarrow(x,y),ys),xs)$$
$$\pi(f,xs) = @(x\Rightarrow\{f(x)\},xs)$$
$$\sigma(p,xs) = @(x\Rightarrow p(x)?\{x\}:\varnothing,xs)$$

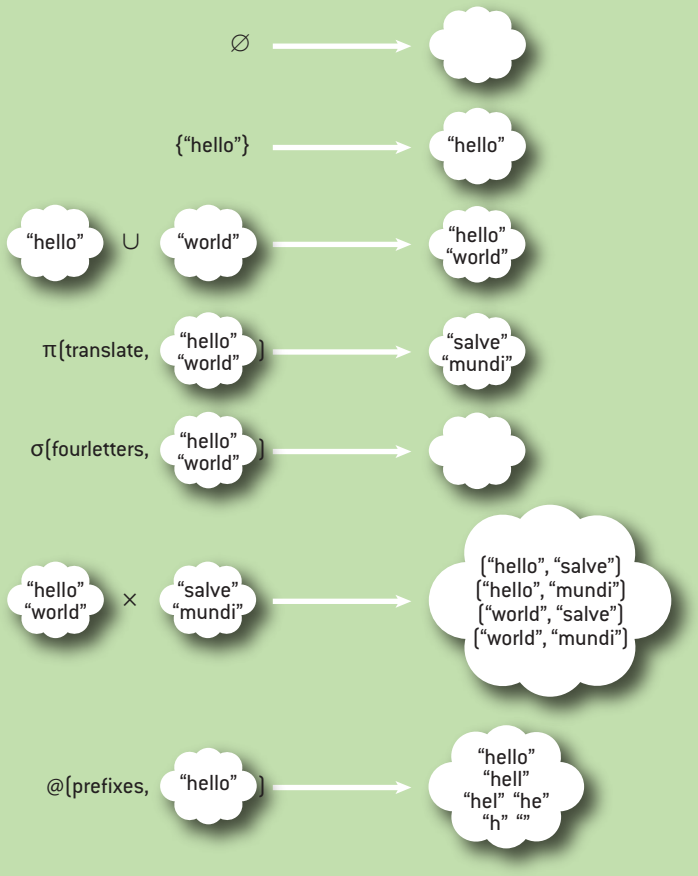
As a programmer you can easily imagine writing up a simple implementation of cross-apply: you would just iterate over the items in the input set, apply the given function, and accumulate the results into a result set. Such an implementation, however, wouldn't need its argument to be a *set* $\{\Sigma\}$; anything that we can iterate over (such as a list, array, or hash table) would suffice. Similarly, there is no reason at all that relational algebra operations should be restricted to sets of values $\{\Sigma\}$. They can be implemented based on other types of collections as well.

Perhaps surprisingly, there is also no reason that the operations passed into π, σ, and @ should be restricted to concrete *functions* Σ→Λ. In fact, you can use any representation of a function from which to determine which computation to perform. For example, in a language such as JavaScript you could simply pass a string and then use eval to turn it into executable code.

What you are searching for is the underlying *interface* that relational algebra implements. As long as there is a type constructor for collections M<Σ> that provides the operations that satisfy similar set-like algebraic properties as {Σ}, and a type constructor for computations Σ↠Λ that satisfies similar function-like properties as Σ→Λ, you can generalize relational algebra to the following set of operators and still be able to write SQL queries over these collections by desugaring query syntax:

$\varnothing \in M<\Sigma>$
$\{\_\} \in \Sigma \rightarrow M<\Sigma>$
$\cup \in M<\Sigma> \times M<\Sigma> \rightarrow M<\Sigma>$
$@ \in (\Sigma \twoheadrightarrow M<\Lambda>) \times M<\Sigma> \rightarrow M<\Lambda>$

FIGURE 2 **Relational Algebra Operators**



For programmers this is just separating interface from implementation; mathematicians call the resulting structures *monads*, and instead of queries they speak of *comprehensions*.

An OO language such as C# uses the canonical interface for collections IEnumerable<T> as a specific instance of the abstract collection type M<T> and uses delegates Func<Σ,Λ> to represent computations Σ→↠Λ. By doing this, you recognize the operators from relational algebra as the LINQ standard query operators as defined in the Linq.Enumerable class.

```
//projection π
IEnumerable<T> Select<S,T>(IEnumerable<S> source,
                           Func<S,T> selector)
//CROSS-APPLY @
IEnumerable<T> SelectMany<S,T>(IEnumerable<S> source,
                               Func<S,IEnumerable<T>> selector)
//selection σ
IEnumerable<T> Where<T>(IEnumerable<T> source,
                        Func<T,bool> predicate)
```

Alternatively, you can use the IQueryable<T> interface to represent the collections M<T> and expression tree Expression<Func<Σ,Λ>> to represent the computations Σ→↠Λ. In that case you recognize the relational algebra operators as the LINQ standard query operators as defined in the Linq.Queryable class. The ability to treat code as data using morphisms—or in the C# case using the Expression type and lambda expressions for code literals—is a fundamental capability that allows the program itself to manipulate, optimize, and translate queries at runtime.

Instead of SQL syntax, the C# language defines XQuery-like comprehensions of the form from-where-select. The previous SQL query example looks like this:

```
from friend in friends where friend.Likes(Sushi) select friend.Name
```

Just as in SQL, comprehensions are translated by the compiler into the underlying LINQ query algebra:

```
friends.Where(friend⇒friend.Likes(Sushi)).Select(friend⇒friend.Name)
```

Depending on the overloads of Where and Select, the lambda expressions will be interpreted as code or data. A simplified implementation of IQueryable is presented later in this article.

As already shown, monads and their incarnation in practical programming languages such as LINQ are simply a generalization of relational algebra by imagining the interface that relational algebra implements. The concepts and ideas behind LINQ should therefore be deeply familiar to both database people and programmers.

THEORY INTO PRACTICE

Unlike Haskell, which has incorporated monads and monad comprehensions in a principled way, the C# type system is not expressive enough for the mathematical signatures of the monad operators. Instead, the translation of query comprehensions is defined in a purely pattern-based way. In a

first pass, the compiler blindly desugars comprehensions, using a set of fixed rules, into regular C# method calls and then relies on standard type-based overload resolution to bind query operators to their actual implementations.

For example, the method Foo Select(Bar source, Func<Baz, Qux> selector), which does not involve any collection types, will be bound as the result of translating the comprehension

```
var foo = from baz in bar select qux
```

into the desugared expression

```
var foo = bar.Select(baz⇒qux)
```

This technique is used extensively in the example presented in the next section.

Another difference between LINQ and its monadic basis is a much larger class of query operators including grouping and aggregation, which is more SQL-like. Interestingly, the inclusion of comprehensions in C#, which was inspired by monad and list comprehensions in Haskell, has recursively inspired Haskell to add support for grouping and aggregation to its comprehensions.

CUSTOM QUERY PROVIDERS

The Yahoo weather service (http://developer.yahoo.com/weather/) allows weather forecast queries for a given location, using either metric or imperial units for the temperature. This simple service is a good way to illustrate a nonstandard implementation of the LINQ query operators that is completely specialized for this particular target and that will allow only strongly typed queries of the form

```
var request = Yahoo.WeatherService().
              Where(forecast⇒forecast.City == city).
              Where(forecast⇒forecast.Temperature.In.units);
var response = await request;
```

or equivalently using query comprehensions

```
var request = from forecast in Yahoo.WeatherService()
              where forecast.City == city
              where forecastTemperature.In.units
              select forecast;
var response = await request;
```

The implementation of the operators extracts the city and temperature unit from the query and uses them to create a REST call (http://weather.yahooapis.com/forecastrss?w=*woeid*&u=*unit*) to the Yahoo service as a result of using the await keyword to explicitly coerce the request into a response.

The technical trick in this style of custom LINQ provider is to project the capabilities of the target query language—in this case the Yahoo weather service that requires (a) a city and (b) a unit—into a

type-level state machine that guides users in "fluent" style (and supported by IntelliSense) through the possible choices they can make (figure 3).

At each transition in the state machine we collect the various parts of interest of the query—in this case, the particular city and the temperature unit. In principle, the city doesn't really need to come first, but it might be more natural for the graph to allow either type of where clause to be specified first, but with the restriction that both where clauses are required. I leave the lifting of this restriction in the state machine as an exercise for the reader.

Note that none of the types Weather, WeatherInCity, or WeatherInCityInUnits implements any of the standard collection interfaces. Instead they represent the stages in the computation of a request that will be submitted to the Yahoo Web service, for which you do not need to define an explicit container type. What also surprises many people is that neither of the two Where methods actually computes a Boolean predicate. Even stranger is that each of the three occurrences of the range variable forecast in the query has a different type.

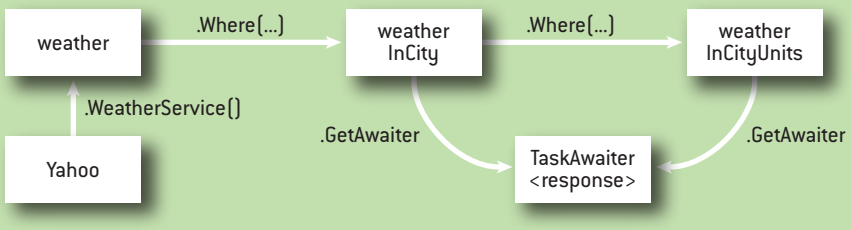
The Weather class defines a single method that picks the city specified in the query and passes it on to WeatherInCity, which is the next state in the type-based state machine:

```
WeatherInCity Where(Weather source, Func<CityPicker,string> city)
{
    return new WeatherInCity{ City = city(new CityPicker()) };
}
```

The "predicate" in the Where method is a function that takes a value of type CityPicker, which has a single property that returns the phantom class City that exists only to facilitate IntelliSense and whose equality check immediately returns the string passed to the equality operator:

```
class CityPicker { City City; }
class City
{
    static string operator == (City c, string s) { return s; }
}
```

FIGURE 3 **Yahoo Weather State Machine**



As a result of this, calling Yahoo.Weather().Where(forecast⇒forecast=="Seattle") really is just a convoluted way of creating a new WeatherInCity{ City = "Seattle" } instance using a Where method that does not take a Boolean predicate and an equality operator that returns a string.

You can use the same trickery in WeatherInCityInUnits Where(Func<UnitPicker,Unit> predicate), so that calling Where(forecast⇒forecast.Temperature.In.Celsius) on the result of the previous filter creates an instance of new WeatherInCityInUnits{ City = "Seattle", Unit = Unit.Celsius }. The techniques used here are not only useful for defining custom implementations of the LINQ operators, but also can be leveraged for building fluent interfaces in general.

Since the Yahoo service requires the city as a WOEID (where on earth ID), we need to make two service calls under the hood in order to retrieve the weather forecast. The first service call retrieves the WOEID of a requested city via http://where.yahooapis.com/v1/places.q(city)?appid=XXXX. If that successfully returns, then a second call is made to retrieve the weather forecast for that location. The calls to the Web server are performed asynchronously and both return a Task<T> (in Java you would use java.util.concurrent.Future<T> to represent the result of an asynchronous operation). Since we can consider a Task<T> as a kind of collection that contains (at most) one element, it also supports the LINQ query operators, and we have turtles all the way down; the LINQ implementation for Weather is defined using the LINQ implementation of Task<T>.

```
class YahooWeatherInCityInUnits
{
    string City; string Units; string AppID;

    TaskAwaiter<ForeCast> GetAwaiter()
    {
        var www = new WebClient();
        var response =
            from xml in www.DownloadStringTaskAsync(...City, AppID...)
            let woeid = ...fish WOEID from result...
            from rss in www.DownloadStringTaskAsync(...woeid...)
            let forecast = ...deserialize forecast from rss...
            select forecast;
        return response.GetAwaiter();
    }
}
```

Though this is an extremely small and limited example, it clearly illustrates many of the techniques used to create real-world LINQ providers such as LINQ to Objects, LINQ to SharePoint, LINQ to Active Directory, LINQ to Twitter, LINQ to Netflix, and many more.

GENERIC QUERY PROVIDERS

The weather service query provider example is structured as an internal DSL. While this provides a great user experience with maximum static typing, it allows little room for reusing the actual implementation of the provider. It is custom built for the particular target top-to-bottom. At the other end of the spectrum we can create a completely generic query provider that records a complete query "as is," using a little bit of meta-programming magic.

In C# a lambda expression such as x⇒x>4711 can be converted into either a delegate—say, of type Func<int,int>—or into an expression tree of type Expression<Func<int,int>>, which treats the code of a lambda expression as data. In Lisp or Scheme one would use *syntactic quoting* to treat code as data. In C# lambda expressions in combination with the type expected by the context provide a *type-based quoting* mechanism.

The class Queryable implements LINQ standard query operators that take expression trees as arguments and return an Expression representation of their selves, very much like a macro recorder:

```
class Queryable<T>
{
    Expression This { get; set; }

    Queryable(){ This = Expression.Constant(this); }

    Queryable<S> Select<S>(Expression<Func<T,S>> f)
    {
        return new Q<S>
        {
            This = Expression.Call(This,"Select",new[]{typeof(S)}, f)
        };
    }
}
```

For example, given a value xs of type Queryable<int>, the call xs.Select(x⇒x>4711) causes the lambda expression to be converted into an expression tree (shown in bold), and then returns an expression tree that represents the call itself **xs.Select(x⇒x>4711)**. Now it is up to the specific query provider (such as LINQ to SQL, Entity Framework, LINQ to HPC) to translate the resulting expression tree and compile it into the target query language.

The IQueryable-based implementation that ships with the .NET Framework uses the same scheme as the simplified example code just shown, except that it is interface based, and it therefore relies on a second interface IQueryProvider to supply a factory for creating instances of IQueryable.

The advantage of a generic query provider is that you can offer general services such as query optimization, which implement rewrite rules such as xs.Union(ys).Where(p) = xs.Where(p).Union(ys.Where(p)) that can be reused across many LINQ providers.

LINQ-FRIENDLY APIS

All examples so far have dealt with implementing particular LINQ providers. An orthogonal aspect of LINQ is APIs that leverage particular LINQ implementations, often LINQ to Objects. For example, LINQ to XML is an API for manipulating XML documents that has been designed specifically with LINQ in mind, which eliminates the need for a DSL such as XQuery or XPath to query and transform XML.

The Google Chart API is a Web service that lets you dynamically create attractive-looking charts, using a simple URI (Uniform Resource identifier) scheme. The URI syntax for Google charts,

however, is not very sequence friendly. For example, the URI for the earlier sample pie chart looks like this:

```
http://chart.apis.google.com/chart
    ?cht=p3&chtt=Top+5+words&chs=500x200
    &chd=t:21,12,7,7,6
    &chl=the|of|a|that|is
```

The problem is that the specification for the labels (chl=the|of|a|that|is) and the specification for the data set (chd=t:21,12,7,7,6) of the chart are given in two separate collections. On the other hand, to generate a pie chart using a query, you want a single collection of pairs that specify both the value and the label for each slice as in from w in top5 select new Slice(w.Count){ Legend = r.Word }.

In other words, to make the Google Chart API sequence friendly, you must transpose a collection of pairs M<SxT> into a pair of collections M<S>xM<T>. Functional programmers immediately recognize this as an instance of the function Unzip∈(R→S x R→T x M<R>)→M<S>xM<T>. Unzip can convert a chart that contains a sequence of slices into the URI format required by the Google Chart API by formatting the various collections using the separators prescribed by the chart service:

```
string CompileToUri()
{
    var tt = Title.UrlEncode();
    var p = Slices.Unzip(
            slices⇒slices.Select(slice⇒slice.Legend).SeparatedBy("|"),
            slices⇒slices.Select(slice⇒slice.Value).SeparatedBy(","));

    return string.Format(@"http://chart.apis.google.com/chart
            ?cht=p3&chtt={0}&chl=t:{1}&chd={2}", tt, p.First, p.Second);
}
```

Here are some convenience constructors for the types Pie and Slice, and a GetAwaiter method on Pie that triggers the compilation of the sequence to a URI and makes a Web request to Google:

```
class Pie
{
    IEnumerable<Slice> Slices; string Title;
    Chart(IEnumerable<Slice> slices){ Slices = slices; }
    TaskAwaiter<Image> GetAwaiter(){ ...CompileToUri()... }
}

class Slice
{
    int Value; string Legend;
    Slice(int Value){ Value = value;}
}
```

Now you can create pie charts (and all other Google chart types) by writing LINQ queries to generate the data set in a natural way, and then await the image of the chart to come back from the call to http://chart.apis.google.com/chart:

```
var slices = from w in top5
             select new Slice(w.Count){ Legend = r.Word };
var image = await new Pie(slices){ Title = "Top 5 words" };
```

CONCLUSION

Big data is not just about size. It is also about diversity of data, both in terms of data model (primary key/foreign key versus key/value), as well as consumption pattern (pull versus push), among many other dimensions. This article argues that LINQ is a promising basis for big data. LINQ is both a generalization of relational algebra and has deep roots in category theory—in particular, monads.

With LINQ, queries expressed in C#, Visual Basic, or JavaScript can be captured either as code or as expression trees. Either representation can then be rewritten and optimized and subsequently compiled at runtime. We have also shown how to implement custom LINQ providers that can run in memory and over SQL and CoSQL databases, and we have presented LINQ-friendly APIs over Web services. It is also possible to expose streaming data so as to implement the LINQ standard query operators, resulting in a single abstraction that allows developers to query over all three dimensions of big data.

ACKNOWLEDGMENTS

Many thanks to the Cloud Programmability and Detroit team members Savas Parastatidis, Gert Drapers, Aaron Lahman, Bart de Smet, and Wes Dyer for all the hard work in building the infrastructure and prototypes for all flavors of LINQ and coSQL; to Rene Bouw, Brian Beckman, and Terry Coatta for helping to improve the readability of this article; and to Dave Campbell and Satya Nadella for providing the necessary push to actually write it.

RELATED READING

1. C# Query Expression Translation Cheat Sheet; http://bartdesmet.net/blogs/bart/archive/2008/08/30/c-3-0-query-expression-translation-cheat-sheet.aspx
2. Comprehensions with Order by and Group by; http://research.microsoft.com/en-us/um/people/simonpj/papers/list-comp/index.htm
3. Expressions; http://www.cs.cmu.edu/Groups/AI/html/r4rs/r4rs_6.html#SEC28
4. Google Chart API; http://code.google.com/apis/chart/image/
5. Hadoop; http://hadoop.apache.org/
6. JavaScript; https://developer.mozilla.org/en/JavaScript/Guide/Predefined_Core_Objects#Array_comprehensions
7. LINQ; http://msdn.microsoft.com/en-us/netframework/aa904594
8. LINQ to HPC; http://blogs.technet.com/b/windowshpc/archive/2011/07/07/announcing-linq-to-hpc-beta-2.aspx
9. Monads; http://en.wikipedia.org/wiki/Monad_%28functional_programming%29

10. Parallel Programming with .NET; http://blogs.msdn.com/b/pfxteam/archive/2010/04/04/9990343.aspx
11. Python; http://www.python.org/dev/peps/pep-0289/
12. Rx (Reactive Extensions); http://msdn.microsoft.com/en-us/data/gg577609
13. Scala; http://www.scala-lang.org/node/111
14. Xamarin; http://xamarin.com/

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ERIK MEIJER (emeijer@microsoft.com) has been working on "Democratizing the Cloud" for the past 15 years. He is perhaps best known for his work on the Haskell language and his contributions to LINQ and Rx (Reactive Framework).