LINQ-to-Datacenter[1]

Erik Meijer and Dragos Manolescu, {emeijer,dragosm}@microsoft.com

Abstract

A plethora of Cloud/fabric frameworks/substrates have emerged within the industry: S3/EC2, Bigtable/Sawzall, Hadoop/PigLatin. Typically these substrates have low-level, idiosyncratic interfaces with data- and query- models heavily influenced by legacy SQL.

The many choices translate into high pain of adoption by developers because of the high risk of making the wrong bet. The SQL-like query model translates into high pain of adoption because it doesn't appeal to developers who embraced object-oriented languages like C# or Java. The SQL-like data model is suboptimal for MapReduce computations because it is not fully compositional. This conservative approach is puzzling because recent language and tool innovations such as Language Integrated Query (LINQ) address precisely the problem of compositional programming with data in modern object-oriented languages.

The proponents of the current substrates have no incentive to come up with a general and developer-friendly abstraction that hides the idiosyncrasies of their proprietary solutions and graduates from the SQL model to a modern, object-oriented and compositional style.

We propose extending the LINQ programming model to massively-parallel, data-driven computations. LINQ provides a seamless transition path from computing on top of traditional stores like relational databases or XML to computing on the Cloud. It offers an object-oriented, compositional model that hides the idiosyncrasies of the underlying substrates. We anticipate that just as the community already built custom LINQ providers for sources such as Amazon, Flickr, or SharePoint, this model will trigger a similar convergence in the space of Cloud-based storage and computation substrates.

Introduction

The MapReduce programming model enables massively parallel processing through elementary techniques from functional programming. The brilliance behind MapReduce is that many useful data-mining queries can be expressed as the composition of a preprocessing step that parses and filters raw data to extract relevant information ("map"), followed by an aggregation phase that groups and combines the data from the first phase into the final result ("reduce").

As Microsoft, Google, and Yahoo! quickly discovered, MapReduce alone is too low-level to be productive for non-specialists. Consequently domain-specific languages such as Yahoo!'s PigLatin, Google's Sawzall,

[1] The information in this article represents our personal views and doesn't necessarily represent the current view of our employer, Microsoft Corporation.

or Microsoft's SCOPE provide higher-level programming models on top of MapReduce. The common trait across these languages is that they represent a radical departure from the current mainstream programming languages, forcing developers to learn something new. This contrasts with the philosophy of Language Integrated Query (LINQ), whose query operators are integrated within popular languages like C# or Visual Basic. This higher pain of adoption increases the risk of failure of new technologies.

Motivating Examples

The following Sawzall program reads a collection of floating point numbers and computes the number of values (count), the sum of the values (total) and the sum of the squares of all values (sum_of_squares):

```
count: table sum of int;
total: table sum of float;
sum_of_squares: table sum of float;

x: float = input;

emit count <- 1;
emit total <- x;
emit sum_of_squares <- x * x;
```

In this simple example, we recognize three uses of the basic summation aggregator (`table` *sum* `of` …). The most interesting aspect of Sawzall is the large set of unusual and novel (statistical) aggregation operators. Still, it is yet another special purpose little domain-specific language that, as always, over time will evolve into an ugly, complex real language. Domain-specific languages are a bad idea; the road to hell is paved with good intentions.

Instead of going through all the trouble of defining a new language from scratch such as Sawzall for doing MapReduce queries, we argue that LINQ with standard modern object-oriented languages like C# or Visual Basic provides a better option. For instance, here is the same query as above using LINQ query comprehensions in C# 3.0:

```
var q = from x in input
        group x by new{} into g
        select new { count          = g.Count()
                   , sum            = g.Sum()
                   , sum_of_squares = g.Sum(x=>x*x)
                   }
```

or, in Visual Basic:

```
Dim Q = Aggregate X In Input
        Into Count(),
             Sum(),
             Sum_Of_Squares=Sum(X*X)
```

It is no coincidence that the Visual Basic query is shorter that the C# one; the syntax of query comprehensions in Visual Basic 9 was designed explicitly with writing MapReduce queries in mind.

The following more interesting Sawzall query (the first example in section 9 of the Sawzall paper) finds the page with the highest page rank within each domain:

```
max_pagerank_url:
    table maximum(1) [domain: string] of url: string
      weight pagerank: int;
doc: Document = input;
emit max_pagerank_url[domain(doc.url)] <- doc.url
       weight doc.pagerank;
```

According the Sawzall paper:

> *Sawzall makes it easy to express calculations like this, the program is nice and short. Even using MapReduce, the equivalent straight C++ program is about a hundred lines long.*

Actually we find the above query quite hard to comprehend. By way of comparison, here is the same query in C#/Visual Basic and LINQ:

```
var pagerank = from click in ClickInfo.Clicks
               group click by click.Uri.Host into domain
               let highestRankedWithinDomain = domain.Max(c => c.PageRank)
               select new { Host = domain.Key
                          , highestRankedWithinDomain.PageRank
                          , highestRankedWithinDomain.Uri
                          };

Dim pagerank = From C In Clicks
               Group C By C.Uri.Host
               Into Url = Max(C.PageRank, C.Url)
```

This query groups all clicks by the host domain of the `Uri` being clicked, and then for each group `g` selects the `Uri` of the page within the domain with the highest rank. The overload of the standard aggregate `Max` used here takes a function that determines how the comparison is made, and determined the resulting value by using the standard Max aggregate:

```
public static T Max<T, S>(this IEnumerable<T> src, Func<T, S> f)
{
    var m = System.Linq.Enumerable.Max<T, S>(src, f);
    var r = (from t in src
             let n = f(t)
             where n.Equals(m)
             select t).First();
    return r;
}
```

Next consider a fraud-detection query that finds all users that clicked at least X times on the same advertisement:

```
var clickFraud = from c in ClickInfo.Clicks
                 group c by new { c.UserId, c.AdId } into g
                 select new { g.Key, count = g.Count(), g } into h
                 where h.count > X
                 select new { h.Key.UserId
                            , h.Key.AdId
                            , h.count
                            , h.g
                            };
```

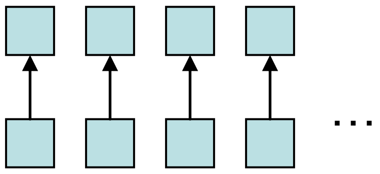
As a third and final motivating example, the next query classifies users according to the X number of times they clicked on a particular category of links. This query is hard to express in a SQL-style language.

```
var categories = from c in ClickInfo.Clicks
                 let google = c.Uri.Host.Contains("google")
                 let live = c.Uri.Host.Contains("live")
                 let microsoft = c.Uri.Host.Contains("microsoft")
                 group new { google, live, microsoft } by c.UserId into g
                 select new { UserId = g.Key
                            , google = g.Count(x => x.google) > X
                            , live = g.Count(x => x.live) > X
                            , microsoft = g.Count(x => x.microsoft) > X
                            };
```

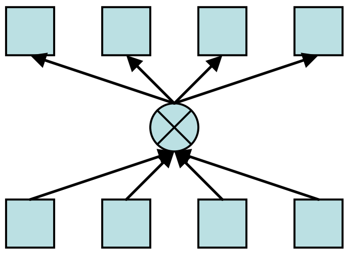
To summarize, LINQ brings all the benefits of modern, object-oriented languages such as C# and Visual Basic to express massively parallel programs. With LINQ we could provide a programming model without introducing anything new, but rather reusing the familiar .NET languages, libraries, and development tools. Research on DryadLINQ confirms that LINQ is perfectly suitable for expressing MapReduce computations. Through running LINQ on the Dryad cluster-computing infrastructure DryadLINQ provides a 1-to-1 mapping between LINQ and the data center. To further lower the pain of adoption and broaden the audience we'd like to bridge between LINQ and other storage and computation substrates, in effect extending the mapping to 1-to-many.

Execution Model

The underlying computation fabric for running MapReduce queries like the above just requires two primitive steps. The ***FanOut*** step runs a given LINQ query in parallel on all nodes in a datacenter.



The ***Repartition*** step gathers data computed by a ***FanOut*** step and scatters is across the nodes based on the new partitioning key computed by ***FanOut***. Obviously implementing this step efficiently is tricky. For instance, instead of repartitioning unprocessed data we can already pre-aggregate data on each node prior to repartitioning, thus greatly reducing the amount of data that would otherwise be sent across the network.



With ***FanOut*** and ***Repartition*** we can execute almost all LINQ queries. Our LINQ-to-Datacenter implementation rewrites the input query given as standard LINQ expression trees into a target tree wherein each ***FanOut*** node has a query fragment (in LINQ form) that it will run on all nodes. These feed into ***Repartition*** query fragments. To make this more concrete, the above fraud-detection query could be compiled into something like the following:

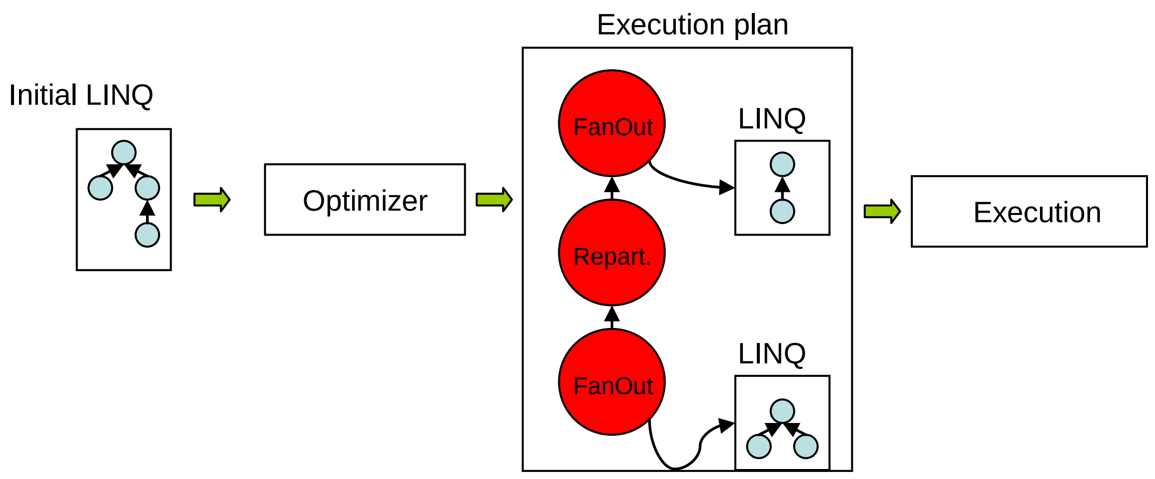
```
FanOutQuery: Table(Temp_2)
 .GroupBy(t =>new Anon9`2(UserId = t.C1, AdId = t.C2))
 .Select(g => new Anon1`2(Key = g.Key, cnt = g.Sum(r => r.C3)))
 .Where(c1 => (c1.cnt > 7))
 .Select(c1 => new Anona`3(UserId = c1.Key.UserId, AdId =
      c1.Key.AdId, cnt = c1.cnt))

      Repartition: Temp_2 = Temp_1 on C1

             FanOutStep: Temp_1 = SelectInto(Table(Clicks)
               .GroupBy(kp => new Anon9`2(UserId = kp.UserId,
                        AdId = kp.AdId), ep => ep)
               .Select(g => new Temp_1() {C1 = g.Key.UserId,
                       C2 = g.Key.AdId, C3 = g.Count()}))
```

The query compiler has the opportunity to leverage the wealth of query optimization from the database literature. For instance, the above plan partitions the aggregation between the two ***FanOut*** steps to significantly minimize the amount of data that needs to be transported between nodes during the ***Repartition*** step.

Query execution corresponds to a workflow manager that maintains a steady parallel load on a standard datacenter cluster. Leaf LINQ fragments in the ***FanOut*** steps use any standard LINQ implementation such as LINQ to SQL, LINQ to entities, LINQ to objects, or even PLINQ. The latter scenario leverages both coarse-grained parallelism in the data center and the fine-grained parallelism of each many-core CPU. In our prototype implementation based on LINQ to SQL the only change needed is an additional extension method for bulk insertions of intermediate values.



Summary

We strongly believe that LINQ is best way to expose a MapReduce style programming model over large clusters of commodity hardware for mainstream programmers. Using this approach we make very few assumptions about the underlying Cloud storage and computation fabric. Hence, just like traditional LINQ where the community developed custom LINQ providers for many sources (e.g., Entity Data Model, SQL, XML, flickr, Amazon, Active Directory), the approach described here could bridge to Cloud fabrics such as those provided by Microsoft, Yahoo!, Google, Amazon, Facebook, etc.