Some sample programs written in DryadLINQ

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, Jon Currey, Frank McSherry, Kannan Achan, Christophe Poulain

May 11, 2008	**First publication**
December 3, 2009	**Revised to reflect current programming application interfaces**

Contents

0 Introduction

0.1 About this document

The goal of this document is to illustrate the use of **DryadLINQ** parallel computation framework through a set of examples. For each program we present the essential source code and a brief description. This document does not describe the installation or configuration of **DryadLINQ** or the configuration parameters which can be used to influence the compilation and execution. A non-commercial release of the **DryadLINQ** research software is available for download at http://connect.microsoft.com/DryadLINQ.

0.2 What is DryadLINQ?

DryadLINQ is a compiler which translates **LINQ** programs to distributed computations which can be run on a PC cluster. **LINQ** is an extension to .Net, launched with Visual Studio 2008, which provides declarative programming for data manipulation.

By using **DryadLINQ** the programmer does not need to have any knowledge about parallel or distributed computation (though a little knowledge can help with writing efficient programs). Thus any **LINQ** programmer turns instantly into a cluster computing programmer. FIGURE 1 shows the software stack used by DryadLINQ.

While **LINQ** extensions have been made to Visual Basic and C#, the **DryadLINQ** compiler only supports C#.

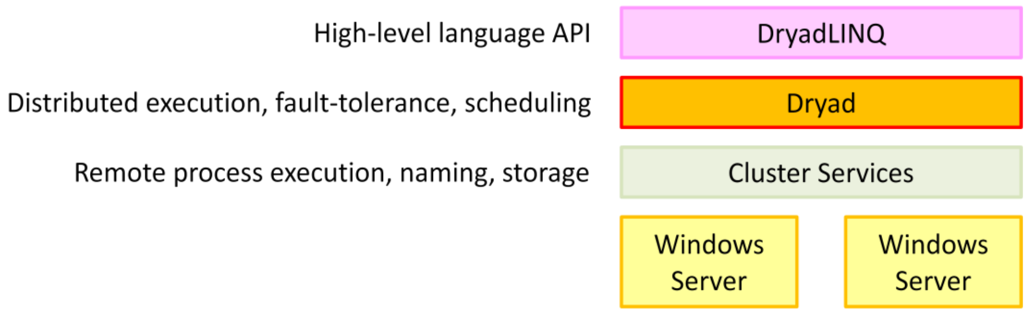
High-level language API
DryadLINQ
Distributed execution, fault-tolerance, scheduling
Dryad
Remote process execution, naming, storage
Cluster Services
Windows Server
Windows Server

Figure 1: The DryadLINQ software stack

FIGURE 2 shows the flow of execution when a program is executed by **DryadLINQ**.

1) A C# user application runs. It creates a **DryadLINQ** expression object. Because of LINQ's deferred evaluation, the actual execution of the expression does not occur yet.
2) A call within the application to `ToPartitionedTable` or to a method that requires the output data sets triggers a data-parallel execution. The expression tree is handed to **DryadLINQ**.
3) **DryadLINQ** compiles the LINQ expression tree into a distributed Dryad execution plan.
4) **DryadLINQ** invokes a custom **DryadLINQ**-specific job manager. The job manager may be executed behind a cluster firewall.
5) The job manager creates the Dryad job.
6) The Dryad job is executed on the cluster.

7) When the Dryad job completes successfully it writes the data to the output table(s).
8) The job manager process terminates, and it returns control back to **DryadLINQ**. **DryadLINQ** creates the local `PartitionedTable` objects encapsulating the outputs of the execution. These objects may be used as inputs to subsequent expressions in the user program.
9) Control returns to the user application. The iterator interface over a `PartitionedTable` allows the user to read its contents as C# objects.

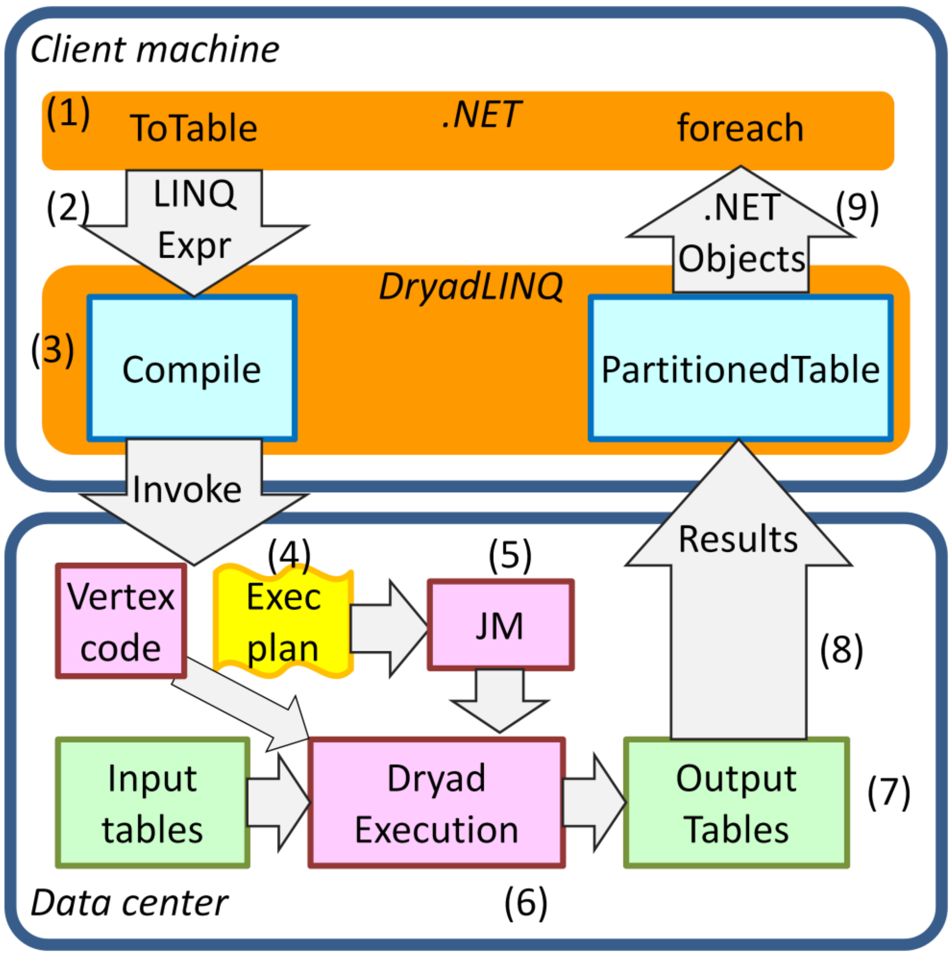

Figure 2: Architecture of the DryadLINQ system.

1 Examples

1.1 Displaying the contents of a text file

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LinqToDryad;

public static class Program
{
    static void ShowOnConsole<T>(IQueryable<T> data)
    {
       foreach (T r in data)
            Console.WriteLine("{0}", r);
    }

    static void Main(string[] args)
    {
        string uri = @"file://\\machine\directory\input.pt";

        PartitionedTable<LineRecord> table =
                      PartitionedTable.Get<LineRecord>(uri);

        ShowOnConsole(table);
        Console.ReadKey();
    }
}
```

Code with the current PartitionedTable API

The generic function `ShowOnConsole` displays the contents of an arbitrary **DryadLINQ** collection of objects of type `T`. (Each object is transformed to a string using its `ToString()` method.)

The `Main` function creates a partitioned table by calling the `Get` method of the `PartitionedTable` class. (Section 1.5 contains more detailed description of partitioned tables.) The table is a sequence of pre-defined **DryadLINQ** `LineRecord` objects. It implements both `IEnumerable and IQueryable` interfaces. Passing the table to the `ShowOnConsole` method achieves the desired result. Note that this program does not require remote execution of code. I.e., this execution only involves steps 8 and 9 from
FIGURE 2.

Earlier version of **DryadLINQ** used the notion of `DryadDataContext` and `DryadTable` to specify the input datasets. It has been replaced by `PartitionedTable` in the current version.

From now on we omit the `using` directives in the C# code.

1.2 Copying a file

```
public static class Program
{
    static void Main(string[] args)
    {
        string uri = @"file://\\machine\directory\input.pt";

        PartitionedTable<LineRecord> table =
                      PartitionedTable.Get<LineRecord>(uri);
        table.ToPartitionedTable("copy.pt");
    }
}
```

This program only goes through steps (1)-(7) from
FIGURE 2. It copies a partitioned table into another partitioned table.

1.3 Counting the records in an input file

```
public static class Program
{
    static void Main(string[] args)
    {
        string uri = @"file://\\machine\directory\input.pt";

        PartitionedTable<LineRecord> table =
                      PartitionedTable.Get<LineRecord>(uri);

        int lines = table.Count();
        Console.WriteLine("Lines: {0}", lines);
        Console.ReadKey();
    }
}
```

The execution of this program goes through all steps from
FIGURE 2.

Even though the result is a scalar value, **DryadLINQ** will first create a temporary anonymous table holding the value of the count.

1.4 fgrep

This program displays all lines matching a fixed string.

```
public static IQueryable<string>
fgrep(IQueryable<string> collection, string tosearch)
{
    return collection.Where(s => s.IndexOf(tosearch) >= 0);
}

static void Main(string[] args)
{
    string uri = @"file://\\machine\directory\input.pt";

    PartitionedTable<LineRecord> table =
                     PartitionedTable.Get<LineRecord>(uri);
    IQueryable<string> lines = table.Select(lr => lr.line);

    IQueryable<string> match = fgrep(lines, "and");
    ShowOnConsole(match);
    Console.ReadKey();
}
```

There are several notable things about this program:

- The input table of `LineRecord` objects is converted to a collection of `string`s by using the operation `lr => lr.line`
- The `fgrep` procedure uses the `Where` operator to keep in the output collection only the strings from the input collection containing `tosearch` as a substring
- The collection `match` is directly passed to the `ShowOnConsole` procedure. DryadLINQ will first create a temporary anonymous table holding the contents of the `match` collection, and then use an iterator to traverse this table.
- The plan generated by **DryadLINQ** for the program is shown in FIGURE 3 for the case where a relatively small text file has been partitioned using the method outlined at the end of Section 1.2. There is only one remote process which performs the reading, the `Select` and the `Where` because there is only one part in the partitioned file.

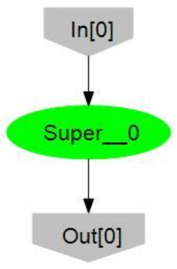

Figure 3: Plan for the fgrep program.

1.5 Partitioned Files

The `fgrep` program can be effectively parallelized for scanning a large amount of data. It is sufficient to cut the data into pieces (preserving line boundaries) and run the scan in parallel on all pieces. The hardest part to do is to describe the file pieces. For this purpose **DryadLINQ** provides a datatype `PartitionedFile`. A partitioned file on disk is composed of two parts:

1) The pieces themselves and
2) The metadata: a textual description of all the pieces of a file which has been split. FIGURE 4 shows how the metadata is organized:
    - The first line indicates the name prefix of each piece. The pieces *must* all be placed in the same directory on all the machines. In this example each file will be in the \mydata directory, and its name will have the form Piece.XXXXXXXX. Here XXXXXXXX is an 8-digit hexadecimal number.
    - The second line is the number of pieces, in this example 4.
    - Each line that follows describes a piece:
        - The piece number, in decimal.
        - The piece size in bytes.
        - Finally, a comma-separated list of machines. A piece may be replicated on several machines, for fault-tolerance.

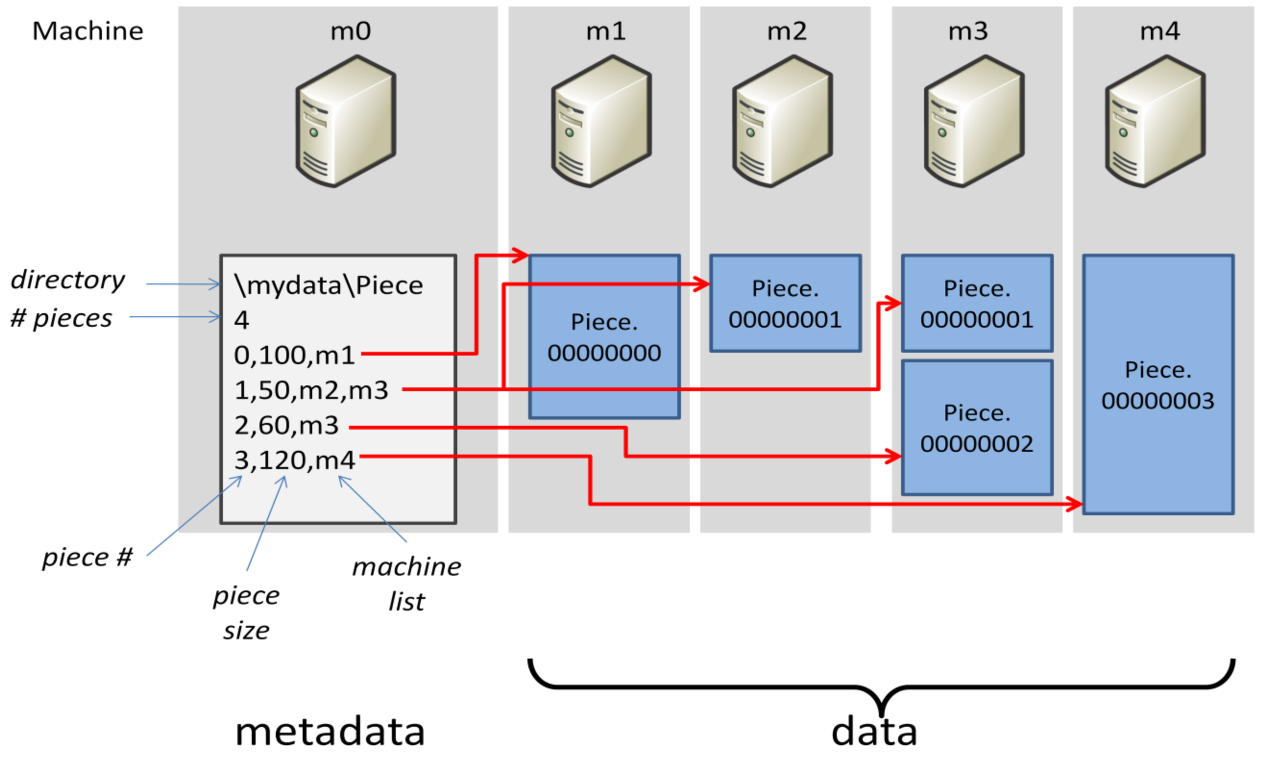

Figure 4: Partitioned File Structure

The description in FIGURE 4 corresponds to the following pieces:

- \\m1\mydata\Piece.00000000
- \\m2\mydata\Piece.00000001
- \\m3\mydata\Piece.00000001
- \\m3\mydata\Piece.00000002
- \\m4\mydata\Piece.00000003

Piece.00000001 is present on two machines.

Once you have partitioned your data in this way, you only need to make a tiny change to enable the computation to use the partitioned table:

```
static void Main(string[] args)
{
    string uri = @"file://\\machine\directory\input4.pt";

    PartitionedTable<LineRecord> table =
                     PartitionedTable.Get<LineRecord>(uri);
    IQueryable<string> lines = table.Select(lr => lr.line);

    IQueryable<string> match = Match(lines, "and");
    match.ToDryadPartitionedTable("matching.pt");
}
```

'input4.pt' is the new metadata file describing the partitioned file with four pieces. When running this job, the job will operate in parallel on all four partitions:

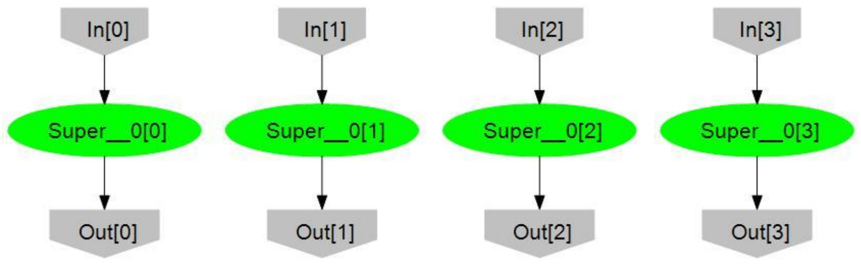

Figure 5: The program operating on a partitioned file with 4 partitions.

The throughput of this computation will be increased by a factor of 4 (assuming the cluster contains at least 4 machines). If the input partitions are on different machines, they can be read all in parallel. The file output by the program also contains four partitions; each partition will reside on the machine which computed it; the metadata file for the result "`matching.pt`" is created in a place directed by a configuration variable.

1.6 Counting elements of a partitioned file

If we apply the counting function to the partitioned file we get a typical plan for associative aggregation operators: each machine does a local count, then these counts are aggregated together by a global phase.

```
static void Main(string[] args)
{
    string uri = @"file://\\machine\directory\input4.pt";

    PartitionedTable<LineRecord> i = PartitionedTable.Get<LineRecord>(uri);

    Console.WriteLine(i.Count());
}
```

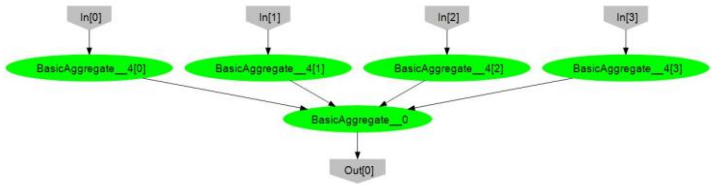

Figure 6: Plan for distributed counting.

1.7 Computing histograms

Let us assume that the input is a large text file distributed over many machines. We want to compute a histogram of the words in the web pages, and extract the top *k* words and their counts. The program uses a helper class `Pair`, which will be used to represent for each word a count.

```
public struct Pair {
    string word;
    int count;
    public Pair(string w, int c)
    {
        word = w;
        count = c;
    }
    public override string  ToString() {
         return word + ":" + count.ToString();
    }
}
```

The distributed computation will express intermediate results as collections of `Pair` objects, and these collections need to be shipped between machines. **DryadLINQ** automatically builds efficient serializers for the data structures in your program; its serialization is much less verbose than the default C# reflection-based serialization.

```
public static IQueryable<Pair> Histogram(IQueryable<string> input, int k)
{
    IQueryable<string> words = input.SelectMany(x => x.Split(' '));
    IQueryable<IGrouping<string, string>> groups = words.GroupBy(x => x);
    IQueryable<Pair> counts = groups.Select(x => new Pair(x.Key, x.Count()));
    IQueryable<Pair> ordered = counts.OrderByDescending(x => x.count);
    IQueryable<Pair> top = ordered.Take(k);
    return top;
}
```

Calling `ShowOnConsole` on the result of this function will display the output. TABLE 1 shows a sample execution for k=3.

Operator	Output
table	"A line of words of wisdom"
SelectMany	["A", "line", "of", "words", "of", "wisdom"]
GroupBy	[["A"], ["line"], ["of", "of"], ["words"], ["wisdom"]]
Select	[{"A", 1}, {"line", 1}, {"of", 2}, {"words", 1}, {"wisdom", 1}]
OrderByDescending	[{"of", 2}, {"A", 1}, {"line", 1}, {"words", 1}, {"wisdom", 1}]
Take(3)	[{"of", 2}, {"A", 1}, {"line", 1}]

Table 1: Sample execution of Histogram

Let's dissect this program.

- The `SelectMany` method transforms a scalar into an `IEnumerable.` In our case, we use it to transform a `string` representing a line into an `IEnumerable<string>` containing all the words on the line.
- The `GroupBy` has a key selector lambda-expression as argument (which returns the "key" associated to each input). The result is a set of bags (called `IGrouping`), where all elements in a bag have the same "key". We use the identity function for the argument of the `GroupBy`, and thus all identical words are grouped together.
- Each group is summarized with a pair containing just the representative word (`x.Key`) and the count of elements in the group.
- The pairs are sorted descending on their `Count` value (`OrderBy`).
- Finally, the `Take`() method just selects the first k elements of the result.

The generated plan is actually quite nifty:

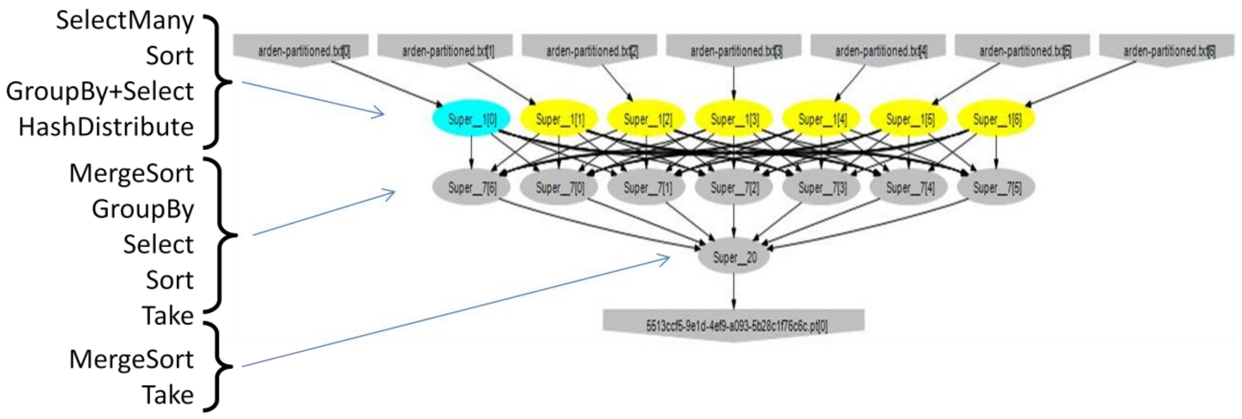

Figure 7: Distributed Histogram plan generated by DryadLINQ.

There are seven input partitions, reading seven files. The `GroupBy` is computed in a distributed way, using a pattern called *hash-partitioning*. The computation is divided into three stages. In the first stage, we apply a local `GroupBy`+Select, and hash partition the result based on the hash function applied to the word; in this way, each machine computes a local `GroupBy` just for a subset of the words. (Note that all identical words will end up on the same machine). The optimizer inserts and `OrderBy` and `Take` at the level of each partition. The final stage can thus just use merge-sort to combine the ordered streams, and then apply `Take` to the result. Currently the final `Take` operator requires the whole data to be present on a single machine; this explains why the data was merged and the output file has a single partition.

Note that it would be perfectly acceptable to omit the types of all temporary variables in the program. I.e., the following program works just fine, and is equivalent to the previous one:

```
var words = input.SelectMany(x => x.Split(' '));
var groups = words.GroupBy(x => x);
var counts = groups.Select(x => new Pair(x.Key, x.Count()));
var ordered = counts.OrderByDescending(x => x.Count);
var top = ordered.Take(k);
```

1.8 Reductions (Aggregations)

The following program computes the sum of all values in a text file. It first parses the file as a set of floating-point values, one per line.

```
static double Add(double x, double y) { return x + y; }

IQueryable<double> numbers = table.Select(x => Double.Parse(x.line));
double total = numbers.Aggregate((x, y) => Add(x, y));
```

The distributed computation which aggregates an input with two partitions looks as follows:

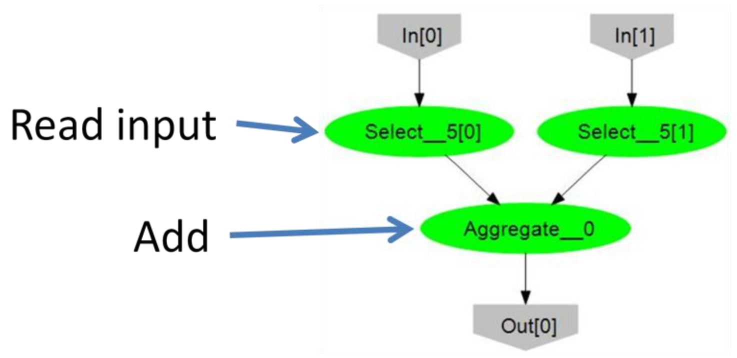

Figure 8: Aggregation is done after collecting all inputs in a single vertex.

The *Aggregate* vertex collects all the data from the two input readers and sums it up.
However, if the aggregating function is associative, a much better parallel computation plan is possible by aggregating each partition of the data separately and then combining the results at the end. By adding an `[Associative]` annotation to a function, you can enable **DryadLINQ** to generate a much better plan.

```
[Associative]
static double Add(double x, double y) { return x + y; }
…
double total = numbers.Aggregate((x, y) => Add(x, y));
```

The generated plan looks much better:

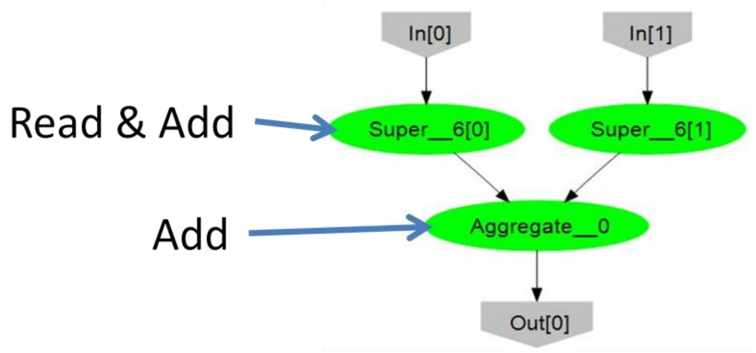

Figure 9: Aggregation of associative function is done using multiple machines.

Each machine does pipelined reading followed by local aggregation on its own data, and then a global stage combines the partial results.

1.9 Apply

DryadLINQ takes one `IQueryable` object and transforms it into a distributed network of processes (vertices). Each of the vertices manipulates only a partition of the data. In the generated code, each vertex executes an independent **LINQ** program. ***The inputs and outputs to each vertex are all*** ***`IEnumerable` objects***. Thus each vertex takes automatically advantage of the lazy evaluation and pipelining provided by the iterator model.

DryadLINQ extends **LINQ** with several powerful operators. The `Apply` operator is a new addition. It corresponds roughly to `Select`: it has a delegate argument, which produces the output by transforming the input. Unlike `Select`, the input to the `Apply` delegate is the whole input stream, and the output is *a complete stream*.

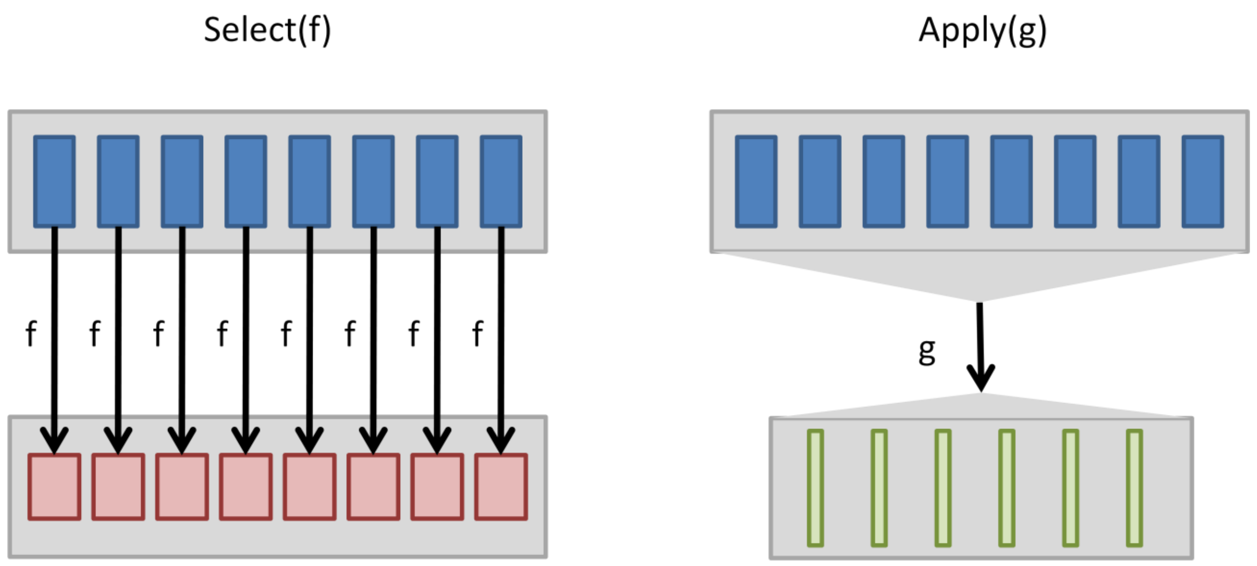

Figure 10: The `Select` function argument receives each element individually, while the one of `Apply` receives the whole stream.

In other words, in FIGURE 10 the type of `f` is `Expression<Func<T,S>>`, while the type of `g` is `Expression<Func<IEnumerable<T>,IEnumerable<S>>>`.

There exists a binary version of `Apply`, which operates on two input streams:

```
public static IQueryable<T3>
Apply<T1, T2, T3>(this IQueryable<T1> source1,
                  IQueryable<T2> source2,
                  Expression<Func<IEnumerable<T1>,
                                  IEnumerable<T2>,
                                  IEnumerable<T3>>> procFunc);
```

Unfortunately, there is no binary version of `Select`. But we can build one using `Apply`. For example, here is how to implement a binary `Select`-like operator which adds the corresponding numbers in two streams (the two streams must have the same length). First, we write the per-vertex transformation, which operates on `IEnumerable` inputs:

```
[Homomorphic]
public static IEnumerable<int>
addeach(IEnumerable<int> left, IEnumerable<int> right)
{
    IEnumerator<int> left_enu = left.GetEnumerator();
    IEnumerator<int> right_enu = right.GetEnumerator();
    while (true)
    {
        bool more_left = left_enu.MoveNext();
        bool more_right = right_enu.MoveNext();
        if (more_left != more_right)
        {
            throw new Exception("Streams with different lengths");
        }
        if (!more_left) yield break;  // both are finished

        int l = left_enu.Current;
        int r = right_enu.Current;

        int q = l + r;
        yield return q;
    }
}
```

The `addeach` function is hopefully obvious: it iterates over two streams in parallel using two iterators (it uses the `MoveNext()` and `Current` stream operators rather than `foreach`).

To create the `IQueryable` version of addition it is just enough to invoke `addeach` on the two inputs:

```
public static IQueryable<int>
AddStreams(IQueryable<int> left, IQueryable<int> right)
{
    return left.Apply<int, int, int>(right, (x,y) => addeach(x,y));
}
```

(It is surprisingly harder to write a generic `Select`, which takes an arbitrary lambda-expression; this is covered in Section 1.15.)

If the `[Homomorphic]` annotation is left out the `Apply` operator first merges all input partitions, as shown in FIGURE 11. With the `[Homomorphic]` annotation the plan performs additions on all vertices in parallel, as shown in FIGURE 12.

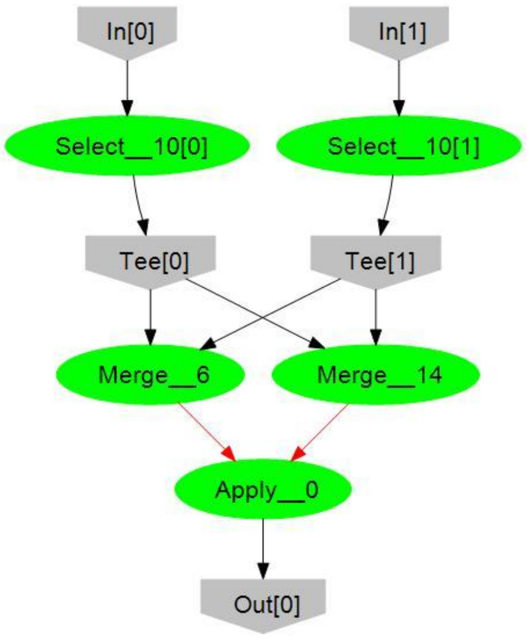

Figure 11: Plan for the pairwise addition when the addeach function is not annotated with [Homomorphic].

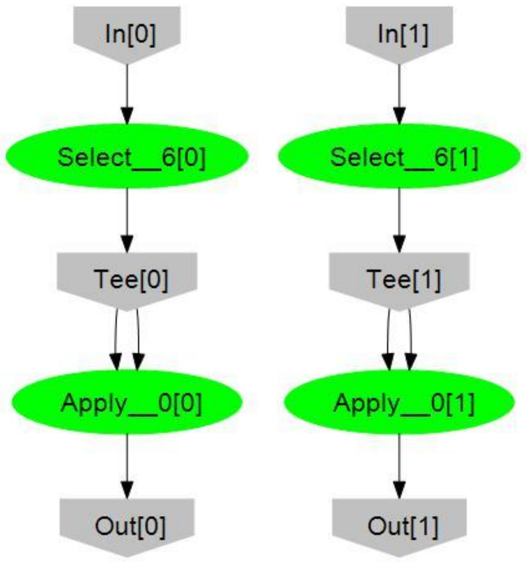

Figure 12: Plan when using the [Homomorphic] annotation.

1.10 Join

This program computes the set of lines in a text file which start with a keyword. The keywords are listed in a second input file.

```
PartitionedTable<LineRecord> table =
     PartitionedTable.Get<LineRecord>(metadata);
PartitionedTable<LineRecord> keywords =
     PartitionedTable.Get<LineRecord>(keys);
IQueryable<LineRecord> matches =
     table.Join(keywords,
                l1 => l1.line.Split(' ').First(),    /* first key */
                l2 => l2.line,                        /* second key */
                (l1, l2) => l1);                      /* keep first line */
```

The matching is done of the first word in each line of table and an entire line of the keywords file. The result is composed from the lines from the first file that match.

The plan uses hash-partitioning to distribute the work:

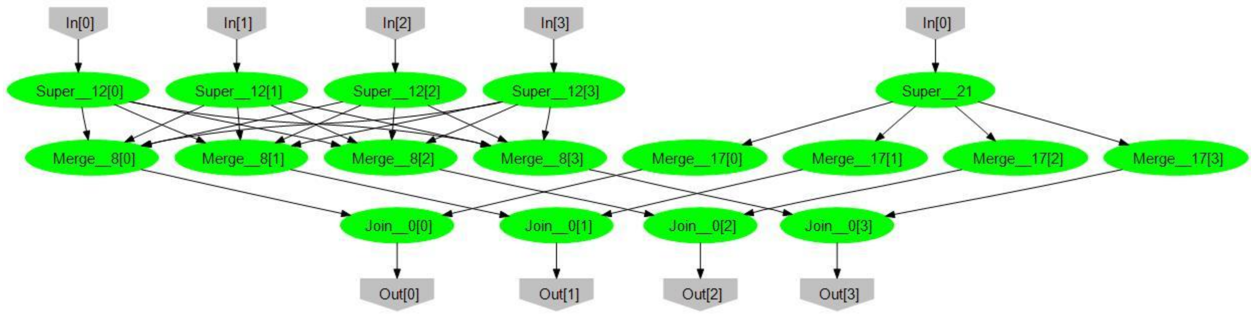

Figure 13: Join computed using hash-partitioning.

Both input tables are redistributed to four partitions which are combined separately.

1.11 Computing multiple outputs

Here we show how to create multiple output tables in a single query. The program first filters non-empty lines from the first table, then applies two joins similar to the previous one.

```
IQueryable<LineRecord> filtered =
            table.Where(l => l.line.Split(' ').Length > 0);

IQueryable<LineRecord> matches1 = filtered.Join(keywords,
            l1 => l1.line.Split(' ').First(),
            l2 => l2.line, (l1, l2) => l1);
IQueryable<LineRecord> matches2 = filtered.Join(keywords,
            l1 => l1.line.Split(' ').Last(),
            l2 => l2.line, (l1, l2) => l1);

IQueryable<LineRecord> m1 = matches1.ToPartitionedTableLazy("first");
IQueryable<LineRecord> m2 = matches2.ToPartitionedTableLazy("last");
DryadLinq.Materialize(m1, m2);
```

The `Materialize` call will invoke a single query which computes both results. Notice how the two queries share the filtering subquery.

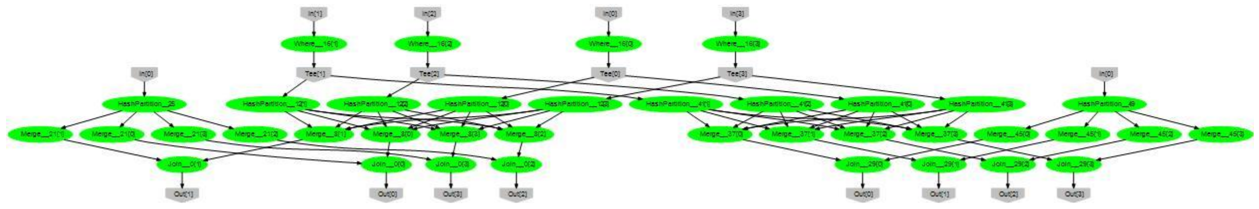
Figure 14: Materializing multiple results.

1.12 Statistics in DryadLINQ

In this section we write a program to manipulate more complex C# data structures. We tackle a statistics application: given a very large set of high-dimensional sparse vectors, compute their mean and variance.

$$\mu = \frac{\sum v_i}{n}, \sigma = \sqrt{\frac{\sum(v_i - \mu)^2}{n}}$$

We won't delve into the implementation of the sparse vectors; any implementation with the following interface would do:

```
public class SparseVector
{
    public SparseVector();
    public SparseVector(string line); /* read from text file */
    public double this[uint index] { get; set; }
    [Associative]
    public SparseVector Add(SparseVector r);
    public SparseVector Subtract(SparseVector r);
    public SparseVector Square(); // elementwise square
    public SparseVector SqRoot(); // elementwise square root
    public SparseVector Divide(double scalar);
}
```

The statistics program will ship around objects of type (collection of) `SparseVector`.

We first tackle computing the average. This is done by summing-up all the values and dividing the result by the count of values. Both the sum and count are built-in aggregations. A naïve attempt to do this fails:

```
// this program is not good enough
public static SparseVector
ComputeStatistics(this IQueryable<SparseVector> v)
{
    SparseVector sum = v.Aggregate( (x, y) => x.Add(y));
    int count = v.Count();
    SparseVector average = sum.Divide((double)count);

    IQueryable<SparseVector> normalized =
              v.Select(x => (x.Subtract(average).Square()));
    SparseVector sum = normalized.Aggregate((x, y) => x.Add(y));
    Sum = sum.Divide(count);
    sum = sum.SqRoot();
    return sum;
}
```

This code does indeed compute the average and standard deviation of all the `SparseVectors` in `v`. However, `average` is a `SparseVector`, and not an `IQueryable`. This means that there will be *three* queries executed: one to compute the sum, a second to compute the count , and a third the standard deviation. In between, the count and average values are shipped back and forth to the C# program.

In order to blend the computation in a single big query we have to perform a few changes:

1) First, we have to use special **DryadLINQ** extensions for `Aggregate` and `Count` which return `IQueryable`s and not values: `AggregateAsQuery` and `CountAsQuery`. These two operators return an `IQueryable` which will always contain a single element when evaluated.
2) The `average` computation becomes much more involved, since we can no longer perform simple arithmetic between the `sum` and `count`. We need to use `Apply` to manipulate them. The `Apply` operation needs to be spelled out:

```
[Homomorphic]
public static IEnumerable<SparseVector>
Scale(IEnumerable<SparseVector> left, IEnumerable<int> right)
   // left and right should contain a single value
{
   SparseVector l = left.Single();
   int coef = right.Single();
   yield return l.Divide((double)coef);
}

public static IQueryable<SparseVector>
Average(this IQueryable<SparseVector> v, IQueryable<int> count)
{
   IQueryable<SparseVector> sum =
           v.AggregateAsQuery<SparseVector>((x, y) => x.Add(y));
   IQueryable<SparseVector> average =
           sum.Apply(count, (x, y) => Scale(x, y));
   return average;
}
```

The `Single()` method returns the unique element of a stream.
We have factored out the `count` computation, since it will be reused.

3) The standard deviation involves a computation between a big stream (the input vector), and a singleton stream (the `average`, which is subtracted from each element of the vector). This can be done with another instance of `Apply`, using the following code:

```
[Homomorphic(Left = true)]
public static
IEnumerable<SparseVector> stddev(IEnumerable<SparseVector> left,
                                 IEnumerable<SparseVector> average)
    // average is a single value, left is a vector
{
    SparseVector avg = average.Single();
    foreach (SparseVector l in left)
    {
        SparseVector tmp = l.Subtract(avg);
        SparseVector tmp_sq = tmp.Square();
        yield return tmp_sq;
    }
}

public static
IQueryable<SparseVector> StdDev(this IQueryable<SparseVector> v,
                                IQueryable<SparseVector> average,
                                IQueryable<int> count)
{
    IQueryable<SparseVector> normalized =
              v.Apply(average, (x, y) => stddev(x, y));
    IQueryable<SparseVector> sum =
              normalized.AggregateAsQuery<SparseVector>((x, y) => x.Add(y));
    IQueryable<SparseVector> scaled =
              sum.Apply(count, (x, y) => Scale(x, y));
    IQueryable<SparseVector> result =
              scaled.Apply(x => x.SqRoot());
    return result;
}
```

We know that the first `Apply` operation (the normalization) can be performed in parallel, but how do we express this fact? In other words, `stddev` can handle in parallel all elements of the `left` input, but it needs to see the whole `right` input (which is a single element stream anyway). This can be described using a special variant of the `Homomorphic` attribute for the `stddev` argument:

```
[Homomorphic(Left = true)]
public static IEnumerable<SparseVector>
stddev(IEnumerable<SparseVector> left, IEnumerable<SparseVector> average)
```

This means that the function is distributive in its left argument, but not in the right one.

```
public static
IQueryable<SparseVector> ReadVectors(string tableUri)
{
   PartitionedTable<LineRecord> table =
         PartitionedTable.Get<LineRecord>(tableUri);
   return table.Select(s => new SparseVector(s.line));
}

public static void
ComputeStatistics(this IQueryable<SparseVector> v)
{
    IQueryable<int> count = v.CountAsQuery();
    IQueryable<SparseVector> average = v.Average(count);
    IQueryable<SparseVector> dev = v.StdDev(average, count);

    IQueryable<SparseVector> a = average.ToPartitionedTableLazy("average");
    IQueryable<SparseVector> d = dev.ToPartitionedTableLazy("stddev");
    DryadLinq.Materialize(a, d);
}
```

This query will generate two tables when executed. The query plan for an input with two partitions looks pretty good:

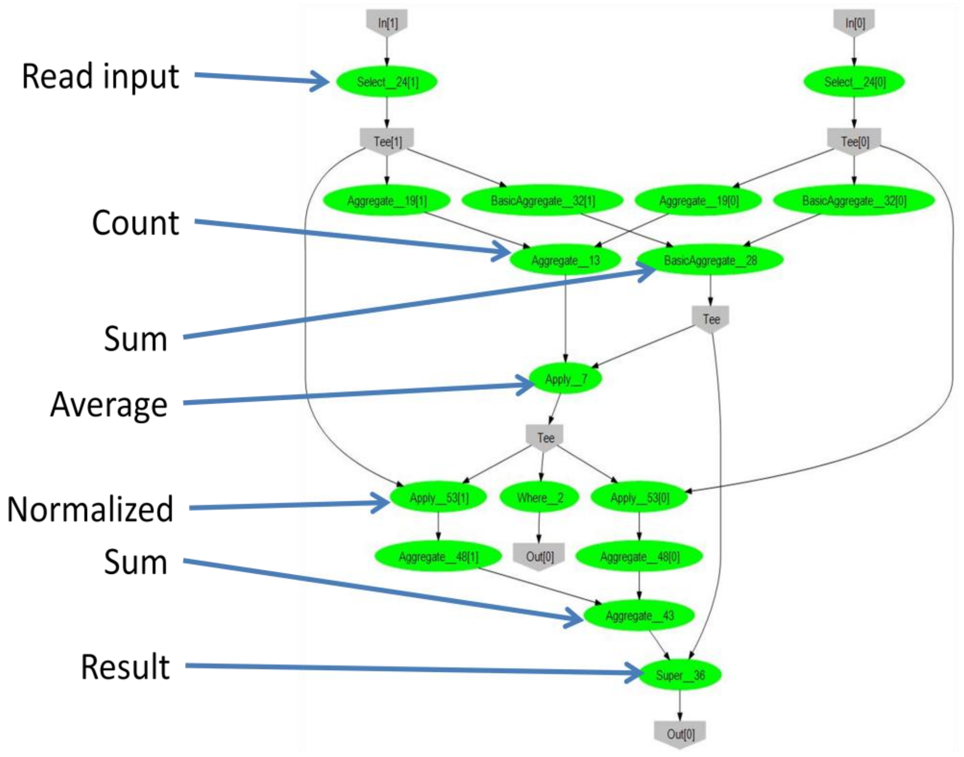

Figure 15: Plan for the statistics computation.

This program can be written in a simpler fashion using the Large Vector library; we revisit it again in Section 2.1.

1.13 Writing Custom Serializers

DryadLINQ writes binary data in the output tables, using the same binary serialization routines that are used to ship data between vertices. As a consequence, data written by a **DryadLINQ** query can be read by another query without any special preparations; the two programs should just specify the same type for the table contents:

```
PartitionedTable<T> output = result.ToPartitionedTable(histogramUri);

[. . .]

PartitionedTable<T> input =  PartitionedTable.Get<T>(histogramUri);
```

For class `MyRecord` you can control the way it is represented on the wire by endowing it with the following two methods:

```
public struct MyRecord
{
    public static MyRecord Read(DryadBinaryReader rd);
    public static void Write(DryadBinaryWriter wr, MyRecord rec);
}
```

For example, to write the `SparseVectors` of the previous section in text form you can either:

- Add an extra `Select` computation stage before the output:

```
result.Select(x => new LineRecord(x.ToString())).ToPartitionedTable(destUri);
```

- Add a `Write` method to the `SparseVector` class:

```
public static void Write(DryadBinaryWriter wr, SparseVector vec)
{
     wr.Write(new LineRecord(vec.ToString()));
}
```

1.14 TeraSort

The following code shows the complete implementation of a program that can be used for sorting data in the format of the Terasort benchmark. The data is described by 100-byte character records, with a 10-byte key. Most of the complexity of the code is in defining a C# data structure which can read, write and compare data in this format. The actual sorting code is exactly one line: an invocation of `OrderBy`.

```
public struct TeraRecord : IComparable<TeraRecord>
{
    public const int RecordSize = 100;
    public const int KeySize = 10;
    public byte[] content;
    public int CompareTo(TeraRecord rec)
    {
        for (int i = 0; i < KeySize; i++)
        {
            int cmp = this.content[i] - rec.content[i];
            if (cmp != 0) return cmp;
        }
        return 0;
    }
    public static TeraRecord Read(DryadBinaryReader rd)
    {
        TeraRecord rec;
        rec.content = rd.ReadBytes(RecordSize);
        return rec;
    }
    public static void Write(DryadBinaryWriter wr, TeraRecord rec)
    {
        wr.WriteBytes(rec.content);
    }
}

class Terasort
{
    public static void Main(string[] args)
    {
        PartitionedTable<TeraRecord> records =
            PartitionedTable.Get<TeraRecord>(args[0]);
        var q = records.OrderBy(x => x);
        q.ToPartitionedTable(args[1]);
    }
}
```

1.15 A Generic Pairwise Select

In this section we show how to build higher-order query operations by writing a generic `Select` operator on `IQueryable` objects which operates on two inputs at once, pairwise. The argument is a binary function `mapper`.

We start by writing a helper function which manipulates `Expression` objects. Given a function f(x,y,z) and a constant value c, it will construct a function g(y,z) = f(c,y,z). But this has to be done using `Expression` objects, and not delegates:

```
public static Expression<Func<T1, T2, T3>>
Closure_cvv<T0, T1, T2, T3>(
        Func<T0, T1, T2, T3> function,
        Expression firstArg)  // type should be T0
// build a closure from a function of 3 arguments,
// first one is constant (cvv)
{
    ParameterExpression xparam = Expression.Parameter(typeof(T1), "xparam");
    ParameterExpression yparam = Expression.Parameter(typeof(T2), "yparam");
    Expression fun = Expression.Constant(function);
    Expression body = Expression.Invoke(fun, firstArg, xparam, yparam);
    Type resultType =
         typeof(Func<,,>).MakeGenericType(typeof(T1), typeof(T2), body.Type);
    LambdaExpression result = Expression.Lambda(resultType,
                                                body,
                                                xparam,
                                                yparam);
    return (Expression<Func<T1, T2, T3>>)result;
}
```

To apply a binary function to two streams, we first scan the two input streams in parallel, and create pairs of objects. Then pass each pair to a function which breaks each pair into two components and applies the binary function on components. For this purpose we build a helper generic `Pair` class with the following signature:

```
[Serializable]
public struct Pair<T1, T2> : IEquatable<Pair<T1, T2>>
{
    public T1 First { get; set }
    public T2 Second { get; set }
    public Pair(T1 f, T2 s);
    public static Pair<T1, T2> MakePair(T1 f, T2 s);
    public bool Equals(Pair<T1, T2> other);
    public override bool Equals(object obj);
    public override int GetHashCode();
}
```

Next we generalize the `addeach` function we defined in Section 1.9 to operate with an arbitrary mapper:

```
[Homomorphic]
public static IEnumerable<T3>
Pointwise<T1, T2, T3>(Func<T1, T2, T3> mapper,
                      IEnumerable<T1> x,
                      IEnumerable<T2> y)
// apply mapper pointwise
{
    IEnumerator<T1> xenu = x.GetEnumerator();
    IEnumerator<T2> yenu = y.GetEnumerator();
    while (true)
    {
        bool morex = xenu.MoveNext();
        bool morey = yenu.MoveNext();
        if (morex != morey)
        {
            throw new Exception("Non-isomorphic collections");
        }
        if (!morex) yield break;

        T1 s = xenu.Current;
        T2 r = yenu.Current;

        T3 q = mapper(s, r);
        yield return q;
    }
}
```

Then final pairwise `Select` does three things:

- creates a function closure called `pairmaker` which takes the `Pointwise` function and uses a function creating pairs as the first argument.
- Creates a function `pop` (from “pairwise operation”) that breaks the pairs away and invokes a simple mapper (equivalent to `(pair) => mapper(pair.First, pair.Second)`).
- Invokes `pairmaker` on the two input streams to create a stream of pairs
- Invokes `pop` using the regular `Select` operation on the stream of pairs.

```
public static IQueryable<T3>
Select<T1, T2, T3>(this IQueryable<T1> input0,
                   IQueryable<T2> input1,
                   Expression<Func<T1, T2, T3>> mapper)
{
      // first create pairs of elements using Apply
      Expression<Func<T1, T2, Pair<T1, T2>>>
           makepairs = (x, y) => Pair<T1, T2>.MakePair(x, y);
      Expression<Func<IEnumerable<T1>,
                      IEnumerable<T2>,
                      IEnumerable<Pair<T1,T2>>>>
           pairmaker =
           Closure_cvv<Func<T1, T2, Pair<T1,T2>>,
                       IEnumerable<T1>,
                       IEnumerable<T2>,
                       IEnumerable<Pair<T1,T2>>>(
               Pointwise, makepairs);

      // tag pairmaker as homomorphic and stateless
      HomomorphicAttribute h = new HomomorphicAttribute();
      AttributeSystem.Add(pairmaker, h);

      ResourceAttribute a = new ResourceAttribute();
      a.IsStateful = false;
      AttributeSystem.Add(pairmaker, a);

      // create a stream of pairs
      IQueryable<Pair<T1, T2>> pairs =
            input0.Apply<T1, T2, Pair<T1, T2>>(input1, pairmaker);

      // create a new mapper which operates on pairs
      ParameterExpression p12 =
          Expression.Parameter(typeof(Pair<T1, T2>),
                               "tmparg");
      Expression p1 = Expression.Property(p12, "First");
      Expression p2 = Expression.Property(p12, "Second");
      Expression body = Expression.Invoke(mapper, p1, p2);
      Expression<Func<Pair<T1, T2>, T3>> pop =
             Expression.Lambda<Func<Pair<T1, T2>, T3>>(body, p12);

      // invoke the mapper on all pairs created
      IQueryable<T3> result = pairs.Select(pop);
      return result;
}
```

1.16 PageRank

Pagerank is a popular approach to scoring web pages based off of a random walk that traverses the links of the web graph. A random surfer is started at a uniformly random page, and in each time step chooses a random outgoing link to follow, or with lower probability (usually 0.15) resets to a uniformly random page. The pagerank of a page is then defined as its stationary probability under this random walk: after an arbitrarily large number of steps, what is the probability that the surfer finds itself at the page in question.

The simplest and most common approach to pageranking keeps the pageranks for each web page, and iteratively updates these scores using power iteration. Given a vector of scores (probabilities), the scores after one step are determined by having each page distribute its score among its outgoing links, typically uniformly, and updating the scores of each page to be the total score received from incoming links. Additionally, we scale each score down by a factor of 0.85, and add 0.15/n to each score, to emulate the reset to uniformly random pages.

PR(u) = sum_v PR(v) weight(u,v) * 0.85 + 0.15/n

Given a list of edges in the graph, perhaps as a list of <source, target> pairs, we can perform this update efficiently by using a `Join` of the current scores with the sources of the pairs, outputting a list of <target, score> pairs that we can then group using a `GroupBy` on the first field. The scores in the group can then be accumulated, forming the score for the next round. Ignoring for now the matter of scaling by 0.85, dividing by degree, or adding 0.15, the following method does just that:

```
public IQueryable<Rank>
PageRank(IQueryable<Edge> edges, IQueryable<Rank> ranks)
{
   return edges.Join(ranks,
                      edge => edge.source,
                      rank => rank.source,
                      (edge, rank) => new Rank(edge.target, rank.value)).
           GroupBy(rank => rank.source).
           Select(group => new Rank(group.Key,
                                    group.Select(rank => rank.value).Sum()));
}
```

This approach is not the most efficient, for several reasons. First, it reads edge data as pairs, when surely it would be more efficient to group the pairs by their source, and emit the several new ranks for outgoing links from the page at once. At the least, this will save a factor of two in the description of the input. Additional metadata about the page, including its out-degree (eventually useful to normalize the outgoing scores) is also immediately available. The following method uses a list of `LinkRecord` rather than `Edge`, has a slightly more complicated `Join` method to output the list of scores for each page, and needs to use a `SelectMany` to merge the list of ranks before grouping.

```
public static IQueryable<Rank>
PageRank(IQueryable<LinkRecord> pages, IQueryable<Rank> ranks)
{
    return pages.Join(ranks,
                      page => page.url,
                      rank => rank.source,
                      (page, rank) =>
                        page.links.Select(dest=>new Rank(dest, rank.value))).
             SelectMany(list => list).
             GroupBy(rank => rank.source).
             Select(group => new Rank(group.Key,
                                      group.Select(rank=>rank.value).Sum()));
}
```

We can be even cleverer with a bit of work and observe that most of the links on the web tend to point to the same host/domain as their source. If we group the `LinkRecord`s by the host of their source, then many of the Ranks that are produced will be shared within the host, and can be pre-aggregated before being sent to a global `GroupBy`. This can substantially reduce the amount of data that needs to be shipped around the network, resulting in substantial performance improvements. One implementation of this idea is as follows:

```
// pagerank that exploits structure in the links
public static IQueryable<HostRanks>
PageRank(IQueryable<IGrouping<string, LinkRecord>> hosts,
         IQueryable<HostRanks> hranks)
{
    return
       hosts.Join(hranks,
                  host => host.Key,
                  hrank => hrank.host,
                  (host, hrank) =>
                     host.Join(hrank.ranks,
                               page => page.url,
                               rank => rank.url,
                               (page, rank) =>
                                   page.links.Select(
                                      dest => new Rank(dest, rank.))).
                           SelectMany(list => list).
                           GroupBy(score => score.url).
                           Select(group =>
                              new Rank(group.Key,
                                       group.Select(x => x.val).Sum()))).
             SelectMany(list => list).
             HashPartition(rank => Hostname(rank.source), HashParts).
             GroupBy(rank => Hostname(rank.source)).
             Select(group =>
                new HostRanks(group.Key,
                              group.GroupBy(rank =>rank.source).
                                    Select(group2 =>
                                       new Rank(group2.Key,
                                                group2.Select(rank =>
                                                     rank.value).Sum()))).
             AssumeHashPartition(host => host.host);
}
```

Notice that the innards of the `Join` function contain another `Join`, and in fact contains code that looks very much like a small version of the previous code example. Indeed this is what is happening, with the host aggregating scores as much as possible before returning its list of outgoing values. Once produced, this list is flattened (with `SelectMany`) and grouped by host, rather than page, so that we can repeat the process. Each group (by host) has its list of scores aggregated by page, and the process is ready to continue.

The two functions `HashPartition` and `AssumeHashPartition` are present to direct the layout of data. Rather than scatter the data arbitrarily, it helps substantially to co-locate the scores associated with a host to the graph structure associated with the host. Assuming that the hosts input is also `HashPartition`ed by host into `HashParts` parts, doing the same with the vector of scores will ensure that when next we perform the `Join`, no data will need to move across the network. In fact, the only traffic that occurs is due to our explicit `HashPartition` statement, which will move only those scores that are not presently in the correct part (recall that many of the scores from a host remain within that host, and consequently need not be moved across the network).

1.17 Q18 from SkyServer

For a definition of this query see http://www.sdss.jhu.edu/SQL/SQLQueries.html. This is the most time-consuming query of a set of queries for mining astronomical data. We elide some constructor code to abbreviate the example.

```
public class PhotoObjAll
{
    public long objId;
    public bool mode;
    public float u, g, r, i, z;
}

public class Neighbor
{
    public long objId, neighborObjId;
}

public class PhotoObjNeighbor
{
    public long objId, neighborObjId;
    public bool mode;
    public float u, g, r, i, z;
    public PhotoObjNeighbor();
    public PhotoObjNeighbor(PhotoObjAll p, Neighbor n);
}

public class PhotoObjNeighborAll
{
    public PhotoObjAll l;
    public PhotoObjNeighbor p;

    public PhotoObjNeighborAll() { }
    public PhotoObjNeighborAll(PhotoObjAll l, PhotoObjNeighbor p);
}
```

```
public class SkyServer
{
    public static void Query18() {
        PartitionedTable<PhotoObjAll> photoObjAll =
            Partitioned.Get<PhotoObjAll>("ugriz-u9.txt");
        PartitionedTable<Neighbor> neighbors =
            Partitioned.Get<Neighbor>("neighbor-u9.txt");

        var j1 = from p in photoObjAll
                 join n in neighbors on p.objId equals n.objId
                 select new PhotoObjNeighbor(p, n);
        var w1 = from pn in j1
                 where pn.objId < pn.neighborObjId && pn.mode
                 select pn;
        var j2 = from l in photoObjAll
                 join pn in w1 on l.objId equals pn.neighborObjId
                 select new PhotoObjNeighborAll(l, pn);
        var w2 = from lp in j2
                 where lp.l.mode
                   && Math.Abs((lp.p.u-lp.p.g)-(lp.l.u-lp.l.g)) < 0.05
                   && Math.Abs((lp.p.g-lp.p.r)-(lp.l.g-lp.l.r)) < 0.05
                   && Math.Abs((lp.p.r-lp.p.i)-(lp.l.r-lp.l.i)) < 0.05
                   && Math.Abs((lp.p.i-lp.p.z)-(lp.l.i-lp.l.z)) < 0.05
                 select lp.p.objId;
        var q = w2.Distinct();
        q.ToPartitionedTable(“result.pt”);
    }
}
```

2 Using the Large Vector Library

Using the techniques in Section 1.15 we have built a generic simple library geared towards mathematical operations operating on large partitioned collections of typed objects. The library contains two main datatypes: `PartitionedVector<T>`, and `Scalar<T>`. A `Scalar<T>` always contains a single value of type `T`. Here are the signatures of these classes:

```
public class Scalar<T> : IQueryable<T>
{
    public Scalar(IQueryable<T> q);
    public Scalar(T v);
    public T Value { get; }
    public Scalar<T1> Map<T1>(Expression<Func<T,T1>> func);
    public Scalar<T2>
        Map<T1,T2>(Scalar<T1> input,
                   Expression<Func<T,T1,T2>> func);
    public Scalar<T3>
        Map<T1,T2,T3>(Scalar<T1> input1,
                      Scalar<T2> input2,
                      Expression<Func<T,T1,T2,T3>> func);
    public Scalar<T4>
        Map<T1,T2,T3,T4>(Scalar<T1> input1,
                         Scalar<T2> input2,
                         Scalar<T3> input3,
                         Expression<Func<T,T1,T2,T3,T4>> func);
}
```

```
public class PartitionedVector<T> : IQueryable<T>
{
    public PartitionedVector(IQueryable<T> q);
    public DryarTable<T> Materialize(string partitionedfilename);
    public T Value { get; }
    public Scalar<UInt64> Count();
    public PartitionedVector<T1> Map<T1>(Expression<Func<T,T1>> func);
    public PartitionedVector<T2>
        Map<T1,T2>(PartitionedVector<T1> input1,
                   Expression<Func<T,T1,T2>> func);
    public PartitionedVector<T2>
        Map<T1,T2>(Scalar<T1> input1,
                   Expression<Func<T,T1,T2>> func);
    public PartitionedVector<T2>
        Map<T1,T2>(T1 input1,
                   Expression<Func<T,T1,T2>> func);
    public Scalar<T> Reduce(Expression<T,T,T> reducer,
                            T seed);
    public Scalar<T> Reduce(Expression<T,T,T> reducer,
                            Scalar<T> seed);
}
```

We have also built a linear algebra library which provides a `DoubleVector`, `SparseVector` and `DoubleMatrix` class. For increased performance, these classes are not generic, they only handle double values. These classes contain the expected assortment of algebraic operations.

We have also predefined the following classes:

```
public class Vectors : PartitionedVector<DoubleVector>;
public class Matrices : PartitionedVector<DoubleMatrix>;
public class OneVector : Scalar<DoubleVector>;
public class OneMatrix : Scalar<DoubleMatrix>;
```

All the `DoubleVector` elements of a `Vectors` object have to have the same dimension; the same is true for all the `DoubleMatrix` elements in a `Matrices` object. We provide some convenient methods for `Vectors` and `Matrices.` Here are some examples:

```
public class Vectors : PartitionedVector<DoubleVector>
{
     public Vectors(IQueryable<DoubleVector> v, uint dimension);
     public uint Dimension { get; }
     static DoubleVector Zero();  // with proper dimension

     public OneVector Sum() {
         return this.Reduce( (x,y) => x.Add(y), this.Zero() );
     }
     public Vectors Add(Vectors other) {
         return new Vectors(this.Map(other, (a,b) => a.Add(b),
                            Dimension);
     }
     public Vectors Subtract(Vectors other) {
         return new Vectors(this.Map(other, (a,b) => a.Subtract(b),
                            Dimension);
     }
    public static OneVector Mean(Scalar<UInt64 count> count) {
        Expression<Func<DoubleVector, UInt64, DoubleVector>> scale =
            (v, s) => v.ScalarDivide(s);
        OneVector sum = this.Sum();
        OneVector result = sum.Map(count, scale);
        return result;
     }
     public Vectors Map(Func<DoubleVector, DoubleVector> map,
                        uint dim);
     // etc.
}
```

2.1 Statistics Revisited

Here we rewrite the program from Section 1.12 using the large vector library we have just described. To compute the mean and standard deviation of a `Vectors v` we can write:

```
public static void Statistics(Vectors v)
{
    Scalar<UInt64> count = v.Count();
    OneVector mean = v.Mean(count);
    Vectors vnorm = v.Map(mean, (a,b) => SqDiff(a,b), v.Dimension);
    OneVector sum = vnorm.Sum();
    OneVector variance = sum.Map(count, (a,b) => a.ScalarDivide(b));
    OneVector sigma = variance.Map( x => x.SqRoot() );
    // materialize(mean, sigma) not shown
}
```

2.2 Linear Regression

We are given a set of N-dimensional input vectors $x_t \in \mathcal{R}^N$ and a set of M-dimensional output vectors $y_t \in \mathcal{R}^M$. We want to compute a linear transformation A which maps each x_t into the corresponding y_t minimizing the square root error: $Ax_t = y_t$. Fortunately this problem has an analytical solution: $A = (\sum_t x_t \times y_t^T)(\sum_t x_t \times x_t^T)^{-1}$. We assume that M and N are relatively small.

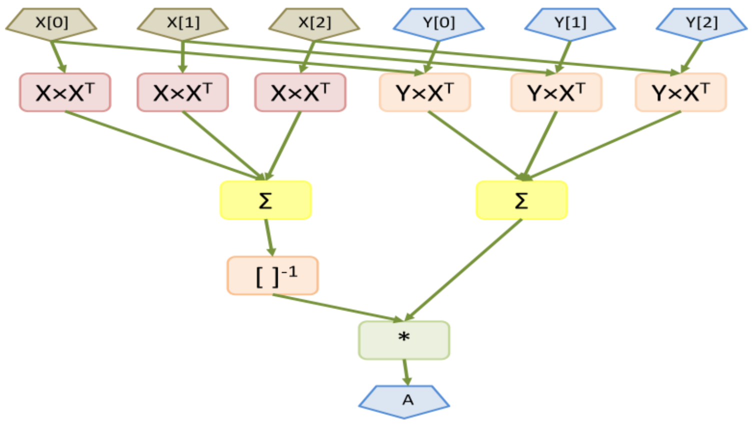

Figure 16: Linear Regression Plan

The main assumption is that the input vectors x and y are partitioned in the same way. Then we can write the computation as follows, by using the classes we have described:

```
Vectors x = input(0);
Vectors y = input(1);
Matrices xx = x.PairwiseOuterProduct(x);
OneMatrix xxs = xx.Sum();
Matrices yx = y.PairwiseOuterProduct(x);
OneMatrix yxs = yx.Sum();
OneMatrix xxinv = xxs.Map(a => a.Inverse());
OneMatrix A = yxs.Map(xxinv, (a, b) => a.Multiply(b));
```

2.3 Expectation Maximization (Mixture of Gaussians)

This is a typical iterative algorithm. We describe here a slightly simpler version, which always performs a fixed number of iterations. It is straightforward to adapt this algorithm to a dynamic version, by using multiple queries. The statistics are stored in an object MixtureModel, which we do not describe. The main E-M loop is as follows:

```
Vectors x = input(0);
MixtureModel initial = new MixtureModel(dimension, clusters);
Scalar<MixtureModel> mm = null;
Scalar<UInt64> T = x.Count();

for (uint iteration=0; iteration < max_iterations; iteration++) {

    // E step is a method of the MixtureModel
    Vectors e;
    if (iteration == 0)
        e = x.Map(initial, (inp, mm) => mm.Estep(inp), clusters);
    else
        e = x.Map(mm, (inp, mm) => mm.Estep(inp), clusters);

    // M Step
    OneVector pi = e.Sum();
    Matrices xe = x.PairwiseOuterProduct(e);
    OneMatrix mu = xe.Sum();
    Mu = new OneMatrix(mu.Map(pi, (X,Y)=>X.DivideColumnwise(Y)));

    Matrices sqdiff = x.Map(mu,
                            (X,Y) => X.squaredDiff(Y),
                            clusters,   // matrix size
                            dimension);
    Matrices sq_e = sqdiff.Map(e,
                              (p, sqd) => sqd.MultiplyColumnwise(p),
                              sqdiff.Columns,
                              sqdiff.Rows);
    Matrices sigma = sq_e.Sum();
    sigma = new OneMatrix(sigma.Map(pi,
                                    (X,Y) => X.DivideColumnwise(Y)));
    pi = new OneVector(pi.Map(p => p.ScalarDivide(T)));
    m = new Scalar<MixtureModel>(
        pi.Map(mu, sigma, (p,m,s) => new MixtureModel(p,m,s)));
}
```

The plan generated for three partitions and three iterations looks as follows:

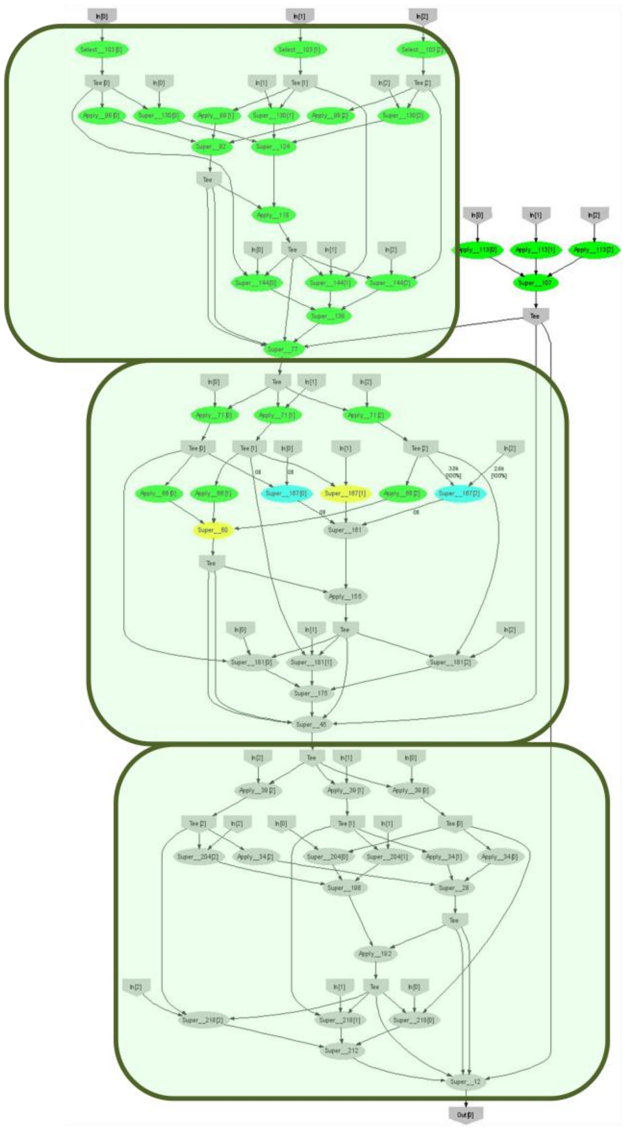
Figure 17: E-M plan for 3 iterations and 3 input partitions.

Note how the current value of the mixture model m is used to concatenate the queries corresponding to different iterations.

2.4 Principal Component Analysis

Principal Component Analysis (PCA) is a technique used for reducing the dimensionality of data. Given a high-dimensional data set $X = \{x_i\}, 0 \leq i < T, x_i \in \Re^n$, PCA involves projecting the input data onto a hyperplane whose basis vectors align with directions of high variance in the input dataset.
To achieve this, we first compute the mean μ and covariance matrix $Cov(X) = \frac{1}{T}\sum_i (x_i - \mu)(x_i - \mu)^T$ of the high dimensional input data. The basis vectors of the projection hyperplane are the top K eigen vectors of the covariance matrix (ordered by their eigen values). K is provided by the user and in general is chosen to reflect on the fraction of total variance necessary to explain the input data. K is also the dimension of the resulting low dimensional data. The rows of the projection matrix Λ correspond the

top K eigen vectors of $Cov(X)$ in order. The projection Y of X to the K-dimensional subspace proceeds as: $y_i = \Lambda(x_i - \mu)$.
Our implementation uses reduction for the count, mean and covariance computation. The computation of Λ is done on a single machine with singular value decomposition. The final projection and computation of y_i is done using a map (vector-scalar) stage. The whole implementation requires 60 lines of code. The complete body of the main algorithm, requiring 7 statements, is the following:

```
Scalar<UInt64> count = x.Count();
OneVector mx = x.Mean(count);
Matrices prods = x.Map(mx,
                      (a, b) => Cov(a, b),
                      x.Dimension, x.Dimension);
OneMatrix sum = prods.Sum();
OneMatrix cov = sum.Map(count, (a,b) => ReduceDegFreedom(a, b));
OneMatrix proj = cov.Map(
      a => ProjectionMatrix(a, reduced_dimension, xdimension));
Vectors xreduced = x.Map(proj,
                        (a, b) => b.Multiply(a),
                        reduced_dimension);
```

2.5 Image Processing

We show the sketch of a code fragment which loads image data into a `PartitionedVector` from a set of directories, specified as text lines in an input table. This is part of an algorithm for image summarization.

```
public class ImgData { /* not shown */ }

static IEnumerable<ImgData>
loadFile(FileInfo file, uint patchSize, uint overlap)
{
    Bitmap img = Image.FromFile(file.FullName);
    /* split the bitmap into pieces and yield return each piece */
}

public static IEnumerable<FileInfo>
listDirectory(string directory)
{
    System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(directory);
    System.IO.FileInfo[] filelist = dir.GetFiles("*.jpg");
    return filelist.AsEnumerable();
}

static IEnumerable<ImgData>
loadDirectory(string directory, uint patchsize, uint overlap)
{
    IEnumerable<FileInfo> files = listDirectory(directory);
    foreach (FileInfo f in files) {
       IEnumerable<ImgData> explodedFile = loadFile(f, patchsize, overlap);
       foreach (ImgData d in explodedFile)
           yield return d;
    }
}

static public IEnumerable<ImgData>
loadDirectory(LineRecord line, uint patchsize, uint overlap)
{
    string content = line.line;
    int comment = content.IndexOf('#');
    if (comment >= 0)
        content = content.Remove(comment);
    if (content.Length == 0) yield break;
    Console.WriteLine("Reading directory {0}", content);
    IEnumerable<ImgData> dircontents =
         loadDirectory(line.line, patchsize, overlap).AsQueryable<ImgData>();
    foreach (ImgData i in dircontents)
        yield return i;
}

public static
PartitionedVector<ImgData> ReadData(string dir, string input)
{
    PartitionedTable<LineRecord> inputtable =
        PartitionedTable.Get<LineRecord>(input);
    return new PartitionedVector<ImgData>(
        inputtable.SelectMany<LineRecord, ImgData>(x =>
            loadDirectory(x, s_patchSize, s_overlap));
}
```