Dryad and DryadLINQ

Unlocking the Power of Parallelism and Data Centers

*The goal of DryadLINQ is to make distributed computing on large computer clusters simple enough for ordinary programmers. DryadLINQ combines two important pieces of Microsoft technology: the **Dryad** distributed execution engine and the .NET Language Integrated Query (**LINQ**).*

About this brief:

We introduce key concepts in the Dryad/DryadLINQ solution,as described in the documentation provided by Microsoft Research in the Dryad/DryadLINQ package:

- *An Introduction to Dryad and DryadLINQ*
- *DryadLINQ API Reference*
- *DryadLINQ Programming Guide*
- *DryadLINQ Installation and Configuration Guide*

In this introduction:

Introduction to Dryad/DryadLINQ **2**
Dryad Basics
Dryad Jobs
DryadLINQ Basics
DryadLINQ Programming Guide **6**
DryadLINQ API Reference **11**

Learn more at:

http://research.microsoft.com/en-us/collaboration/tools/dryad.aspx

The Scientist's Challenge:

You have all the data for an important scientific problem. You just need to analyze it. With terabytes of data, you need a powerful data processing application in order to have results to talk about at the upcoming national meeting. The analysis might be straightforward in principle, but actually doing it is going to be tough.

For many important scientific investigations, efficiently analyzing large data sets is a major challenge. For example, astronomers use the Sloan Digital Sky Survey to investigate problems such as the distribution of "dark matter" around distant galaxies. The current data set—SDSS Data Release 7—covers more than a quarter of the sky and contains more than 50 TB of data representing 357 million unique objects.

Your best bet is to create a distributed application that runs on a cluster of relatively inexpensive networked PCs. However, implementing such an application is a non-trivial task: distributed applications must manage numerous threads, allocate resources across numerous individual multicore computers, handle hardware failures, and so on. Writing the code could take months, and you're a scientist, not a programming expert.

As an alternative to writing all the code yourself…

Microsoft® Dryad is a high-performance, general-purpose distributed computing engine that handles some of the most difficult aspects of cluster-based distributed computing. It's powerful: Microsoft routinely uses Dryad applications to analyze petabytes of data on clusters of thousands of computers.

But Dryad applications still aren't that easy to implement. To further simplify things, Microsoft has developed DryadLINQ, which allows developers to use an extended version of the .NET LINQ programming model and API to implement Dryad applications. DryadLINQ code is similar to what you'll see in a conventional LINQ application, and the application core is often only a few lines of code. Behind the scenes though, a DryadLINQ provider automatically converts the LINQ query into a Dryad job and executes the query as a distributed application on a cluster.

With DryadLINQ and a Dryad cluster…

Even a novice at parallel processing or cluster-based computing can implement a high-performance distributed application to efficiently analyze terabytes of data. As an example, consider one time-consuming problem: Q18 of the Sloan Digital Sky Survey, which searches the data set for possible gravitational lenses:

> Q18: Find all objects within 1' of one another that have similar colors, where the color ratios u-g, g-r, r-I are less than 0.05m. Magnitudes are logarithms, so these differences correspond to ratios.

To address this problem, Microsoft researchers used DryadLINQ to run the query on a 40-node Dryad cluster consisting of 40 off-the-shelf networked Windows®-based computers. Dryad took about an hour to install.

- The query itself is a three-way join over two input tables, one with 11 GB of data and the other with 41.8 GB.
- To perform the query, the team used Microsoft Visual Studio® and a standard Windows-based workstation to implement a DryadLINQ application that consists of approximately 100 lines of Microsoft Visual C#® code.
- The team manually distributed the data across the cluster and ran the application from the workstation. The DryadLINQ provider set up the Dryad job and ran the query on the cluster.

The results came back in under two minutes—not even enough time for a quick cup of coffee.

Microsoft® Research

Introduction to Dryad and DryadLINQ

Distributed computing is becoming an increasingly important part of application development. In particular, cluster-based distributed computing is the only practical way to analyze the large-scale data sets that are the key to addressing a variety of important problems. For example, large-scale internet services routinely analyze petabytes of search log data by using distributed applications running on clusters of thousands of computers.

A cluster-based distributed application must be implemented so that different parts of the application can execute concurrently on different computers. In general, dividing an application into concurrently executing parts is a difficult problem. There are two basic approaches:

- Task-parallel computing assigns different tasks to different processors.
- Data-parallel computing distributes the data for a task across the available computers and operates on the data concurrently.

Many important types of applications can use data-parallel computation, including data mining, image and stream processing, and scientific computations. Implementing these tasks as distributed applications on a cluster allows the application to efficiently process large volumes of data.

Data-parallel computing on a cluster of computers poses a number of challenges. For example:

- Cluster-based computations must manage thousands of threads and allocate resources across thousands of individual computers.
- Members of the cluster are commodity computers, some of which can be expected to fail during the course of the computation.
- The programming models most developers are familiar with—and that have the best tools and documentation—are designed for applications that run sequentially on a single computer, not as distributed applications on a cluster.

The solution: Dryad and DryadLINQ.

Dryad: A High-Performance Distributed-Computing Engine

Dryad is a high-performance general-purpose distributed computing engine that simplifies the task of implementing distributed applications on a cluster. The original motivation for Dryad was to efficiently execute data mining operations similar to those performed by technologies such as MapReduce or Hadoop.

However, Dryad is a general purpose execution engine. It can be used to implement a wide range of other application types, including time series analysis, image processing, and a variety of scientific computations.

The Dryad engine handles some of the most difficult aspects of large scale distributed applications. In general, Dryad:

- Efficiently distributes applications across clusters of as many as thousands of commodity computers running the Windows operating system.
- Automatically schedules and distributes cluster resources.
- Monitors the cluster and automatically recovers from computer or network failures.

Dryad provides excellent performance and scalability, and can handle very large-scale data-parallel computations. Dryad has been deployed by Microsoft since 2006, and Dryad applications are used daily to analyze petabytes of data on clusters of thousands of commodity computers. However, Dryad can be used effectively even on clusters of only a few computers.

DryadLINQ: An Efficient Way to Implement Dryad Applications

Implementing a native Dryad application is typically still a complex and demanding task, so application developers won't use Dryad directly. Instead, Microsoft has simplified the process of implementing Dryad applications by creating DryadLINQ.

DryadLINQ is an abstraction layer over Dryad that provides a straightforward and efficient way to implement Dryad applications. Developers can use Microsoft Visual Studio to implement their DryadLINQ applications in any language that supports LINQ.

The DryadLINQ API and programming model is an extended version of LINQ, which is a new SQL-like query technology for Microsoft .NET-connected applications. LINQ defines a set of general-purpose operators that allow applications to declaratively express query operations such as traversal, filter, and projection in any .NET programming language.

Much of the code in a typical DryadLINQ application is similar to that used by LINQ applications. The DryadLINQ provider then translates LINQ queries into a Dryad job, executes the job on a cluster, and returns the results to the application.

With DryadLINQ, a developer does not need to know much about Dryad—or even about parallel or distributed computing—to implement a Dryad application that can efficiently analyze terabytes of data. However, developers who are familiar with these technologies can take advantage of that knowledge to optimize performance.

Dryad Basics

Figure 1 shows the essential elements of the Dryad stack.

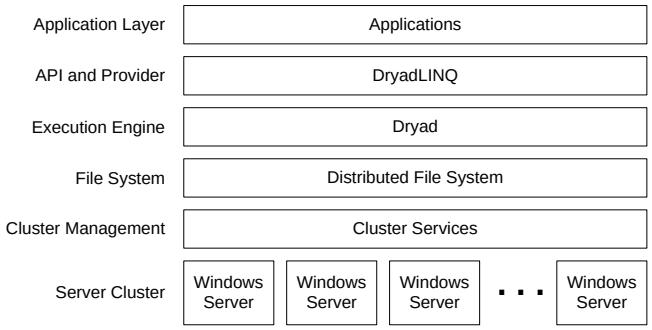

Figure 1. Dryad Architecture

Application Layer

DryadLINQ applications use the DryadLINQ API for the Dryad-based parts of the application. Applications do not have direct access to Dryad or the distributed file system. They access those components indirectly, through the DryadLINQ API.

API and Provider

The DryadLINQ provider converts LINQ queries into a Dryad job, and uses Dryad to execute the job on the cluster.

Execution Engine

The Dryad execution engine manages the execution of the Dryad job on the server cluster.

Distributed File System

Distributed file systems—such as those supported by Microsoft codename "Cosmos" or Windows Azure cloud services operating system—manage files across the cluster. Dryad does not require a distributed file system, but can use it if it is present.

Cluster Services

Cluster services manage the remote execution of processes on the cluster's computers. It also handles the cluster's name services, metadata, and so on.

Note: Dryad cluster services are specific to Dryad, and are not related to Microsoft Windows Cluster Services.

Server Cluster

Dryad applications typically run on large clusters of Windows-based computers. However, Dryad can be used with clusters of a few computers, or even on a single computer. The latter option provides limited performance, but it is quite useful for debugging and learning purposes.

Dryad Jobs

Dryad applications are hosted by a client workstation that is connected to the cluster by a network link. Much of the application code—such as the user interface—executes on the workstation. The Dryad-based parts of the application are packaged as a Dryad job, which executes on the cluster.

A Dryad job is a mechanism for efficiently executing a distributed application on a cluster. Consider a very simple data-parallel programming operation: multiply each element of an array by a constant value. Figure 2 shows a diagram of how this operation would be implemented as Dryad job.

The basic execution plan as shown on the left side of Figure 2 represents how the job would execute on a single computer. To execute this plan as a Dryad job:

1. The input data is partitioned into manageable pieces, and each partition is copied to one of the computers in the cluster.
2. A separate instance of the array processing code is dispatched to each computer in the cluster.
3. Each instance of the array processing code simultaneously processes the data partition on its computer.
4. The processed partitions returned to the client application as a data set.

The application in the preceding example consists of a single query. A Dryad application typically consists of multiple related queries. The execution plan for a Dryad job is represented by a directed acyclic graph (DAG), called a Dryad graph.

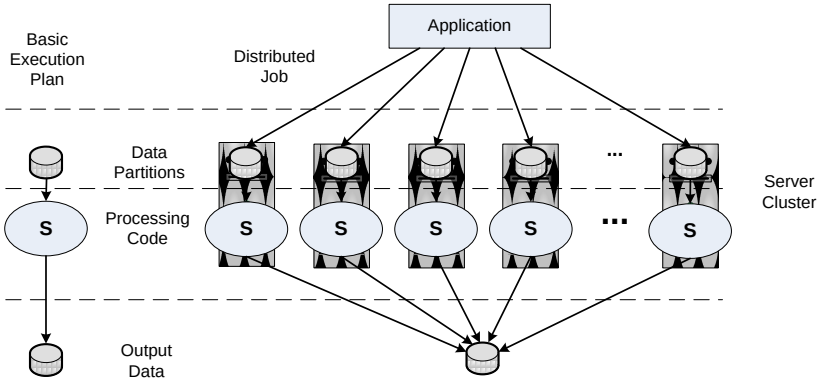

Figure 2. Simple Dryad Job

Figure 3 shows an example of a more complex Dryad graph. The basic execution plan—on the left—shows the structure of the operation as it would run on a single computer. The graph on the right shows how Dryad might implement the operation as a distributed job running on a cluster of at least six computers.

Dryad applications are implemented in stages. Each stage corresponds to one element of the basic execution plan, and handles a query or a group of related queries. A Dryad graph has several elements

Input Data Partitions

A Dryad job typically starts with a partitioned data set—one partition per computer—that provides the input for the first stage.

Vertices

Each stage consists of one or more identical vertices, typically one vertex per computer. A vertex is an instance of the data processing code for the stage, and processes the data for its computer. For DryadLINQ, each vertex includes a .NET assembly that contains LINQ processing code.

The number of vertices can vary from stage to stage. However, for stages that process data partitions, the number of vertices matches the number of partitions.

Channels

The output of a stage's vertices is passed over channels to the next stage, and becomes input for that stage's vertices. Dryad supports a variety of channels and distribution schemes to allow applications to efficiently pass data from one stage to the next.

Details such as the number of vertices in each stage, or the way that data is distributed from one stage to the next are chosen to optimize job performance.

The DryadLINQ provider automatically generates a Dryad graph for the application's queries, and serialization code for channel communication. In addition, DryadLINQ can sometimes improve job performance as the job is executing by using runtime information to dynamically modify the later stages of the graph.

DryadLINQ Basics

The DryadLINQ API supports a set of operators, which can be broken into three categories:

- Unmodified LINQ operators, such as **Select** and **Where**.
- Modified LINQ operators, such as **FirstAsQuery** or **LastAsQuery**.

  To produce more efficient distributed code, DryadLINQ supports modified versions of those LINQ operators that return scalar values, such as **First** or **Last**.
- DryadLINQ-specific operators, such as **Apply** or **Fork**.

  These operators represent operations that cannot be efficiently synthesized by composing native LINQ operators. They can substantially improve application performance in a distributed environment.

DryadLINQ also supports several attributes, which developers can use to annotate their methods to improve application performance.

How DryadLINQ Evaluates Queries

As a LINQ application progresses, LINQ simply constructs a LINQ expression to represent the operations and data. LINQ continues adding operations to the expression until the application takes an action that requires the result of the query.

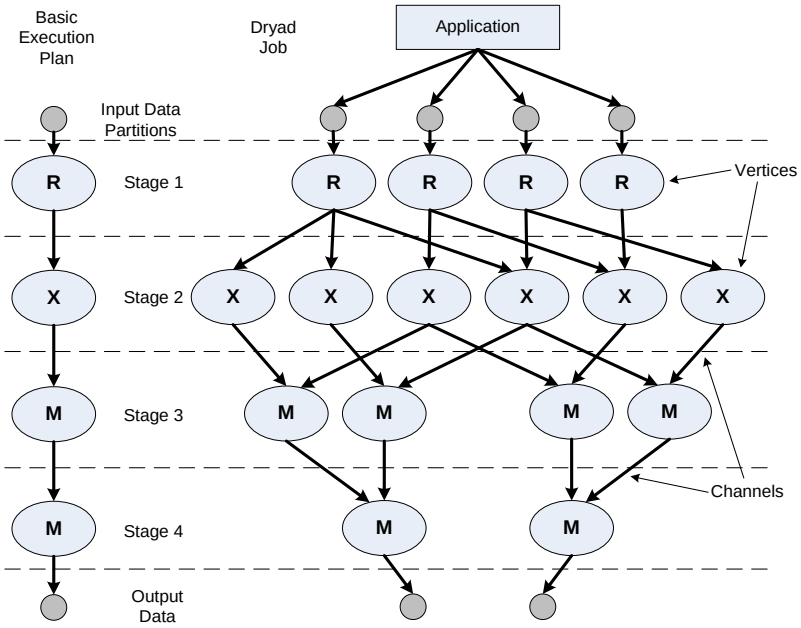

Figure 3. A Typical Dryad Graph

For example, the application calls **foreach** to enumerate the collection that represents the query's final result. LINQ then evaluates the results of the query by using the collection's **IEnumerable<T>** interface to perform the specified operations on each element of the data set.

DryadLINQ data is represented by the **IQueryable<T>** interface, which inherits from **IEnumerable<T>** and supports all of its methods. From an application perspective, collections that expose **IQueryable<T>** are used in much the same way as those that expose **IEnumerable<T>**. Applications construct queries over the data by applying LINQ operators to the collections, use **foreach** to enumerate the collections, and so on.

DryadLINQ applications are actually based on **DryadTable<T>** objects. **DryadTable<T>** exposes **IQueryable<T>**, and applications construct queries over **DryadTable<T>** objects much like any other **IQueryable<T>** collection. The distinction is that **DryadTable<T>** is a specialized type of **IQueryable<T>** collection that represents persistent data sets for DryadLINQ.

IQueryable<T> evaluation also takes place only after the application takes an action that requires the result of the query. However, the evaluation process for the two types of collection is quite different.

- **IEnumerable<T>** represents iterators.

  Typically, **IEnumerable<T>** represents a collection on the local computer. During evaluation, the object is compiled by the local .NET JIT compiler and the computation is performed locally.

- **IQueryable<T>** represents queries.

  During evaluation, DryadLINQ queries are passed to the DryadLINQ provider, which handles the computation on the cluster.

Figure 4 shows a typical DryadLINQ evaluation process.

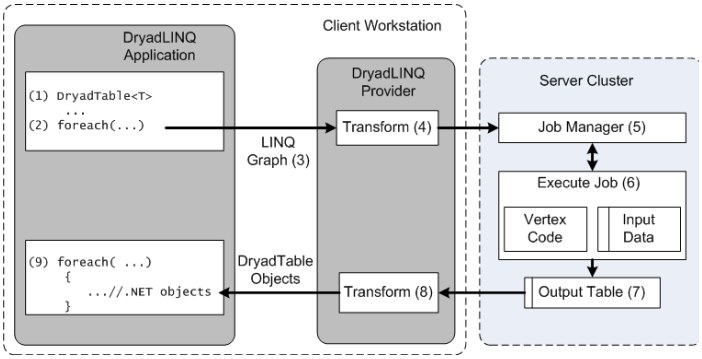

Figure 4. How DryadLINQ Works

The following procedure explains the details in Figure 6. The numbers correspond to the parenthetical numbers in Figure 6.

1. An application creates one or more **DryadTable<T>** objects to represent persistent data and defines a query by applying one or more LINQ operators to the collection.

   DryadLINQ builds a LINQ expression to represent the query, but defers evaluation.

2. The application triggers the object evaluation process by calling **foreach**.

   The **foreach** operator initiates the evaluation when it calls the object's **GetEnumerator** method. The actual enumeration doesn't take place until evaluation is complete.

3. LINQ passes the LINQ expression to the DryadLINQ provider.
4. The DryadLINQ provider:
   - Transforms the LINQ expression into sub-expressions and uses them to generate a query plan. The job manager uses the query plan to create a Dryad graph for the job.
   - Generates LINQ-to-Object data processing code for the vertices. The code for each stage is compiled to a .NET assembly, and dispatched to the cluster's computers at the appropriate stage of the operation.
   - Generates serialization code to handle channel communication between stages.
5. The DryadLINQ provider creates a job manager, which constructs a Dryad graph for the job.

   Because LINQ applications operate on datasets rather than individual items, the DryadLINQ provider has considerable flexibility in how it translates the LINQ code into an efficient Dryad graph.
6. The job manager executes the job on the cluster:
   - The job manager—in conjunction with the Dryad runtime—schedules and spawns vertices as resources are available.
   - The job manager monitors the cluster for failure and initiates recovery procedures, as required.
   - The job manager applies DryadLINQ-specific policies, as appropriate, to rewrite the later stages of the graph to optimize performance.
7. When execution completes, the job manager passes the results—and control—to the DryadLINQ provider.
8. The DryadLINQ provider transforms the output into **DryadTable<T>** objects, and passes the objects—and control—to the DryadLINQ application.
9. The **foreach** call can now enumerate the **DryadTable<T>** objects to obtain the data as .NET objects.

   The application can also use the **DryadTable<T>** objects for subsequent DryadLINQ queries.

Any subsequent queries are handled in the same way.

DryadLINQ Programming Guide

Programming models are valuable only to the extent that you understand how to use them. The DryadLINQ package includes a programmer's guide, which provides details for implementing DryadLINQ applications, including a set tutorials accompanied by complete code samples.

Listing 1 is an excerpt from "DryadLINQ Programming Guide," showing how to use DryadLINQ to implement a simple string-matching application and a MapReduce application.

A Simple DryadLINQ MatchString Application

This section implements MatchString, a DryadLINQ application that takes a single input file and runs on a single-computer Dryad configuration. Listing 1 shows the code for this version of MatchString.

IQueryable<T> and DryadTable<T>

The DryadLINQ *Match* method returns an **IQueryable<T>** collection instead of **IEnumerable<T>**.

The MatchString input data is represented by a **DryadTable<T>** object.

The Data Context

The **DryadDataContext** object represents an instance of the DryadLINQ provider. Applications initialize the object with the URI of the folder that contains the persistent input data, which is shared as \\MyComputer\DryadData\input.

DryadDataContext supports a number of useful methods. The MatchString example creates a **DryadTable<LineRecord>** object to represent the input text by passing the file name to **DryadDataContext.GetTable**. The **LineRecord** type is a DryadLINQ structure that represents a line of text.

Query Operators

Match selects those lines that contain the search string by applying the standard LINQ **Select** and **Where** operators to the input data collection. *Match* then returns an **IQueryable<T>** collection to the caller that represents the selected lines.

In practice, DryadLINQ applications use regular LINQ syntax and lambda expressions for most standard operations, and implement custom delegates in the same way as standard LINQ delegates.

The compiler recognizes the difference and produces the appropriate code. For example, the **Where** operator specifies which lines are to be selected, and LINQ-to-objects and DryadLINQ versions of the query would both use the same lambda expression:

```
.Where(s => (s.IndexOf(toSearch)) >= 0)
```

However, the expression resolves quite differently at compile time for the two applications.

LINQ-to-Objects uses an **IEnumerable<T>** version of **Where**:

```
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, bool> filter
)
```

The *filter* argument corresponds to the lambda expression, and compiles to a **Func<T, bool>** delegate that handles the operation on the local system.

Listing 1. A DryadLINQ String-Matching Application

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

class Program
{
  public static IQueryable<string> Match(string directory,
                                         string fileName,
                                         string toSearch)
  {
    DryadDataContext ddc = new DryadDataContext("file://" + directory);
    DryadTable<LineRecord> table = ddc.GetTable<LineRecord>(fileName);

    return table.Select(s => s.line)
                .Where(s => s.IndexOf(toSearch) >= 0);
  }

  static void Main(string[] args)
  {
    IQueryable<string> results = Match(@"\\MyComputer\DryadData\input", "TestFile.txt", "here");
    foreach (string s in results)
      Console.WriteLine(s.ToString());
  }
}
```

DryadLINQ uses an **IQueryable<T>** version of **Where**:

```
public static IQueryable<T> Where<T>(
    this IQueryable<T> source,
    Expression<Func<T, bool>> filter
)
```

The *filter* argument is the same lambda expression, but it compiles to an **Expression<Func<T, bool>>** object, which represents the lambda expression as an expression tree, and is passed to the DryadLINQ provider for evaluation.

The DryadLINQ Evaluation Process

Main calls **foreach** to enumerate and print the selected lines. The DryadLINQ provider then executes the shaded code in the *Match* method as a Dryad job. After evaluation is complete, *Main* regains control and the **foreach** loop prints the results.

DryadLinqConfig.xml

The configuration file, DryadLinqConfig.xml, defines how a DryadLINQ application executes, and includes most of the information that distinguishes between single-computer and cluster-based applications. The file must be with the executable file in the project's output folder.

The example in Listing 2 is a slightly edited version of the DryadLinqConfig.xml file that was used to run MatchString. It is based on a computer named *MyComputerName*, which is configured for single-computer Dryad operation. To use this example on your system, simply modify the computer name, domain name, and file paths, as appropriate.

The settings are as follows:

DryadLinqRoot
The root folder of the DryadLINQ installation for this example is c:\DryadLinq_release.

ClusterName
This example uses a single-computer configuration, so the cluster name is just the computer name: MyComputerName.

DryadOutputDir
This example directs Dryad to write its temporary output files to c:\dryaddata\output on the client workstation.

APEnvironmentPath and APConfigPath
This example was run on the workstation that installed Dryad, so Autopilot.ini and Cosmos.ini are in c:\dryad_release\clusters\MyComputerName\client and c:\dryad_release\clusters\MyComputerName\client\bin, respectively.

LocalArch
The example ran on an x86 client workstation, so this element is set to "i386".

LocalFlavor
This example references a debug build of the Dryad job manager.

PartitionUncDir
The folder path for output files is c:\dryaddata\output, so this property is set to dryaddata\output.

Cluster
Because this example uses a single-computer configuration, the Cluster attribute values are similar to those used for the local settings:

- name: The client workstation's computer name
- host: "localhost"
- domain: The client workstation's domain name
- arch: The client workstation's architecture
- flavor: Debug

LinqToDryad.dll

All DryadLINQ projects must reference LinqToDryad.dll, which contains the DryadLINQ provider implementation and exposes the DryadLINQ class library. The DLL is located in the DryadLINQ installation's lib folder—typically c:\dryadlinq_release\lib.

The DryadLINQ class library is in the **LinqToDryad** namespace, so most DryadLINQ applications simplify their code by including a **using LinqToDryad** directive.

Listing 2. A DryadLINQ Configuration File

```
<DryadLinqConfig>
  <DryadLinqRoot>c:\DryadLinq_release</DryadLinqRoot>
  <ClusterName>MyComputerName</ClusterName>
  <DryadOutputDir> file://c:\dryaddata\output </DryadOutputDir>
  <APEnvironmentPath>c:\dryad_release\clusters\MyComputerName\client</APEnvironmentPath>
  <APConfigPath>c:\dryad_release\clusters\MyComputerName\client\bin</APConfigPath>
  <LocalArch> i386 </LocalArch>
  <LocalFlavor> debug </LocalFlavor>
  <PartitionUncDir>dryaddata\output</PartitionUncDir>

  <Cluster name="MyComputerName"
           host=" localhost "
           domain="MyDomainName"
           arch=" i386 "
           flavor="debug " />
</DryadLinqConfig>
```

Run the DryadLINQ MatchString Application

To compile and run the DryadLINQ MatchString application

1. Create a new Visual C#® Console Application project.
2. Replace the code in Program.cs with the example code.
3. Add a reference to System.Data.Linq.
4. Add a reference LinqToDryad.dll.
5. Create a DryadLinqConfig.xml file for the project, and place it in the project folder.
6. Add DryadLinqConfig.xml to the project and set the file's **Copy to Output Directory** property to "Copy always".
7. Put a copy of TestFile.txt from the LINQ version of MatchString in the folder that is associated with the data context, c:\DryadData\input.
8. Build the application and press Ctrl F5 to run it.

   If you are running the application on a cluster, make sure that the cluster is running before pressing Ctrl F5.

The example in Listing 3 is an edited version of the output.

The first few lines of output show the execution plan. Although the query includes two operators, **Select** and **Where**, there is only one stage, whose vertex is named Super__0. Vertex names are usually based on the associated operation. However, the DryadLINQ optimizer often determines that it is more efficient to run multiple operations in the same vertex—**Select** and **Where** in this case. Vertices that run multiple operations are named Super_*XX.*

As the program runs, the console window displays a series of job manager messages, which track the progress of the job and often contain useful debugging information. For simplicity, the example shows only the last few messages.

The application's output—the highlighted text—appears after the job manager's messages have ceased. As discussed earlier, the **foreach** loop does not enumerate the result collection and print the results until the job manager has completed the evaluation and shut itself down.

How to Implement a Simple MapReduce Application

MapReduce is an effective programming model for processing large amounts of data in parallel. A standard example used to illustrate the basic principles is counting the instances of each word in a document. A MapReduce operation has two basic stages:

1. The Map stage maps each element of input data to a key, and groups the data by key.

   The word counting example starts with a collection of words, and uses the word itself as the key. By grouping words with the same key, each group contains all instances of the associated word.

2. The Reduce stage "reduces" the group associated with each key to a single value.

   The word counting example reduces each group of words to the number of words in the group.

The operation returns a collection of key-value pairs; the keys used by Map and the associated values produced by Reduce.

DryadLINQ provides a simple and straightforward way to implement MapReduce operations. This section is a walk-through of Histogram, which implements the canonical MapReduce example; counting word frequency in a text file.

Histogram has two primary components:

- A Pair structure, which serves as a data container.
- A BuildHistogram method, which counts word frequency and returns the top five words.

Listing 3. Output from the DryadLINQ Match Application

```
Query 0 Output: file://C:\\dryaddata\\output\43ae715e-78f3-4f00-9267-3dd2e3bda725\Output.
DryadLinq0.dll was built successfully.
Input:
        [Table: file://c:\dryaddata\input\TestFile.txt]
Super__0:
        Select(s => s.line)
        Where(s => (s.IndexOf(_) >= _))

...

Cross data statistics: Total 170 Cross-machine 0 Cross-pod 170
Cross data statistics: Total 170 Cross-machine 0 Cross-pod 170
TestFile.txt[0](1),(),(nowhere),0s
Completed uninitialise cosmos
Super__0(1),(TestFile.txt[0].0),(MyComputer),1.9375s
0f75bfb3-00a3-44c2-939f-febf495cc6b6[0](0),(Super__0.0),(nowhere),0s
Job running time: 10.36 seconds
Total running time in vertices successful/failed: 1.94s/0.00s
Average job parallelism: 0.19
XmlExecHost - Finished running the app
Application completed successfully.
Some text here and there
over there and back to now
```

The Pair Structure

Pair has two properties:

- Word is a string that holds a word or key.
- Count is an int that holds the word count.

The structure also overrides **ToString** to simplify printing the results. The following example shows the *Pair* implementation.

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

The Histogram Application

Histogram is a console application that consists of the *BuildHistogram* method and a brief *Main* method that invokes *BuildHistogram* and prints the results. For simplicity, the input data for this example is in a single file on the client workstation. However, for large data sets, the input data would be partitioned and distributed across the cluster. The source code for Histogram is shown in Listing 4.

The BuildHistogram method:

1. Creates a **DryadTable<LineRecord>** object, *inputTable*, to represent the lines of input text.

   For partitioned data, use **GetPartitionedTable<T>** instead of **GetTable<T>** and pass the method a metadata file.

2. Applies the **SelectMany** operator to *inputTable* to transform the collection of lines into collection of words.
   - The **String.Split** method breaks the line into an array of words.
   - **SelectMany** concatenates the collections created by **Split** into an **IQueryable<string>** collection named *words* that represents all the words in the file.
3. Performs the Map part of the operation by applying **GroupBy** to the *words* object.

   The **GroupBy** operation groups elements with the same key, which is defined by the selector delegate. This creates a higher order collection, whose elements are groups. In this case, the delegate is an identity function, so the key is the word itself and the operation creates a *groups* collection that consists of groups of identical words.
4. Performs the Reduce part of the operation by applying **Select** to *groups*

   This operation reduces the groups of words from Step 3 to an **IQueryable<Pair>** collection named *counts* that represents the unique words in the file and how many instances there are of each word. Each key value in *groups* represents a unique word, so **Select** creates one *Pair* object for each unique word. **IGrouping**.**Count** returns the number of items in the group, so each *Pair* object's *Count* member is set to the number of instances of the word.
5. Applies **OrderByDescending** to *counts*.

   This operation sorts the input collection in descending order of frequency and creates an ordered collection named *ordered*.
6. Applies **Take** to *ordered* to create an **IQueryable<Pair>** collection named *top*, which contains the 100 most common words in the input file, and their frequency.

Listing 4. A DryadLINQ MapReduce Application

```
public class Program
{
  public static IQueryable<Pair> BuildHistogram(
                      string directory,
                      string fileName,
                      int k)
  {
    DryadDataContext ddc = new DryadDataContext("file://" + directory);
    DryadTable<LineRecord> inputTable = ddc.GetTable<LineRecord>(fileName);

    IQueryable<string> words = inputTable.SelectMany(x => x.line.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;
  }
  static void Main(string[] args)
  {
    IQueryable<Pair> results = BuildHistogram(@"c:\DryadData\input",
                                        "TestFile.txt",
                                        100);
    foreach (Pair words in results)
      Console.WriteLine(words.ToString());
  }
}
```

Main then uses the *Pair* object's **ToString** implementation to print the top one hundred words, and their frequency.

The core of the MapReduce operation is thus implemented with two lines of code—the highlighted lines in the example—which correspond to steps 3 and 4 in the preceding list.

How DryadLINQ Executes Histogram on a Cluster

Although DryadLINQ constructs the Dryad graph for Histogram, it is instructive to examine the graph and see how a moderately complex sequence of queries executes on the cluster. In particular, an understanding of how the job executes on the cluster is essential for debugging the application.

The example in Listing 5 shows the execution plan, which has been edited slightly for readability.

Note: This plan is optimized for a particular cluster and a particular configuration. If the application is run in a different environment, DryadLINQ might produce a different graph.

Figure 5 shows the graph that is associated with this plan—assuming four input partitions for convenience.

Listing 5. Output from the Histogram Application

```
Query 0 Output: file://\\Computer01\DryadData\output\5513ccf5-9e1d-4ef9-a093-5b28c1f76c6c.pt
DryadLinq0.dll was build successfully.
Input:
       [Table: file://\\Computer01\DryadData\Input\TestFile.txt
Super__1:
       SelectMany(l => l.line.Split(new [] {_}))
       DryadSort(w => w)
       DryadOrderedGroupBy(w => w,(k__0, g) => new KeyValuePair<String,Int32>(k__0,
            g.count()))
       DryadHashPartition(e => e.Key, e => e.Key)
Super__7:
       DryadMergeSort()
       DryadOrderedGroupBy(e => e.Key,e => e.Value, (k__0, g__1) =>
            newKeyValuePair<String,Int32>(k__0, g__1.Sum()))
       Select(g => g.Count())
       DryadSort(c => c)
       Take(100)
Super_20:
       DryadMergeSort()
       Take(100)
```

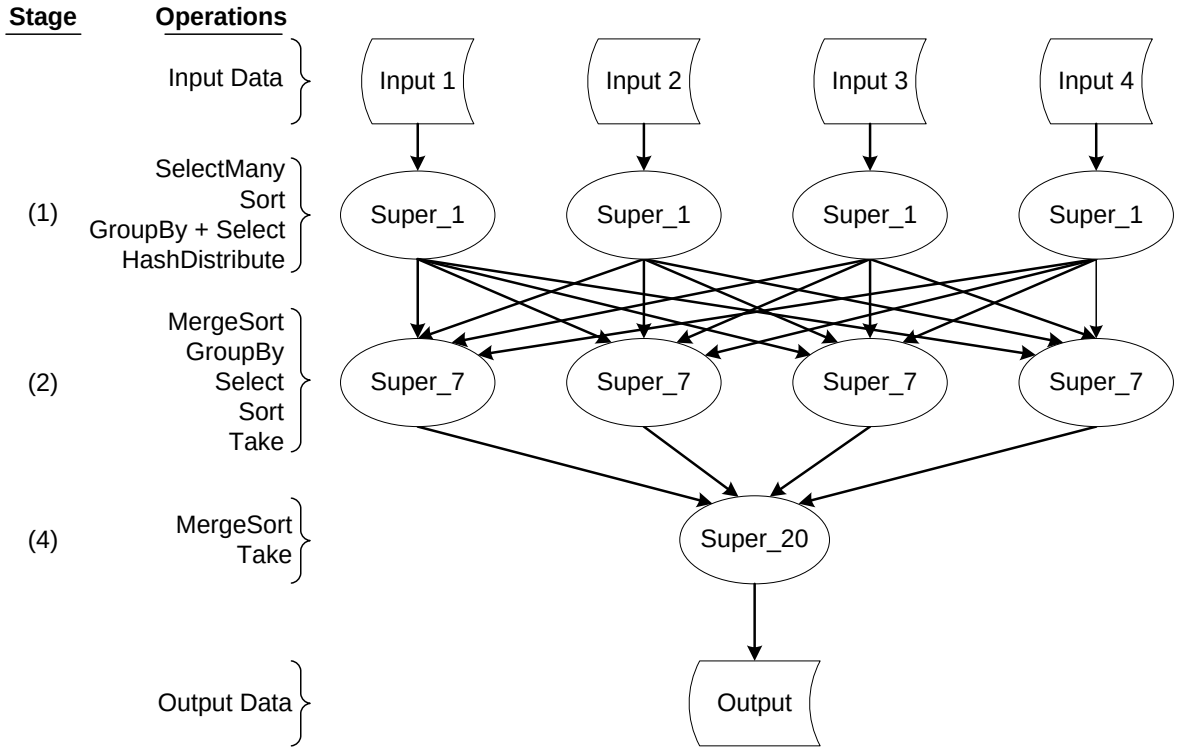

Figure 5. Dryad Graph for Histogram

The graph has three stages:

Stage 1

Each Super_1 vertex processes its input partition into an **IQueryable<string>** collection that represents the individual words.

The DryadLINQ optimizer divides the **GroupBy** query into two operations. The **GroupBy** operation in this stage runs separately on each input partition, and reduces each group of identical words in the partition to key-value pair; the word and the count. If the input partition contains 10 instances of "the", the data that is passed to Stage 2 consists of just {"the", 10}. This approach can substantially reduce the amount of data that must be passed to Stage 2.

The final operation in the stage hash-partitions the data to be sent to Stage 2. This operation calculates a hash value for each key by using the .NET **GetHashCode** method. **HashPartition** then uses the hash values to distribute the key-value pairs across the vertices of the next stage. All keys with the same hash value—which represent identical words, in this case—go to the same vertex.

Stage 2

After merging the data that it receives from the vertices of Stage 1, each Super_7 vertex performs the final part of the **GroupBy** operation, which groups identical words across the entire data set. Hash partitioning the data from Stage 1 improves performance by guaranteeing that all instances of a word are sent to the same vertex, so the **GroupBy** operation doesn't have to spend time transferring data between vertices.

To limit the amount of data that must be passed to Stage 3, the vertex then orders the collection and performs a **Take** operation, which sends the partition's top 100 unique words and their frequency to Stage 3.

Because Stage 3 selects the top 100 words from the entire data set, there is no point in sending it more than 100 words from any individual partition.

Stage3

The final stage consists of a single vertex, Super_20, which merges and orders the data from Stage 2 and performs a **Take** operation that yields the top 100 words in the data set and their frequency. The graph uses a single output partition for this stage because **Take** works only on a single computer. The output from Stage 2 must therefore be merged into a single ordered collection before **Take** can pick the top 100 words for the entire data set.

In this case, the decision to hash-partition the output from Stage 1 was made by DryadLINQ. However, developers can explicitly direct DryadLINQ to partition data by applying the **HashPartition** or **RangePartition** operators.

DryadLINQ API Reference: An Example

The DryadLINQ package includes an API reference that covers all the classes used by applications. The following is an excerpt from the DryadDataContext class reference, and shows two of the more useful methods: **GetPartitionedTable<T>** and **GetTable<T>**, which are used to create **DryadTable<T>** objects from persistent data sources.

DryadDataContext Class

This class represents DryadLINQ providers.

```
public class DryadDataContext
```

Members

The class contains the following members. For overloaded methods, the link is to the first overload. Property members are indicated by shading.

DeletePartitionedTable	GetPartitionedTable	PathCombine
DeleteTable	GetTable	Source
ExtractDir	IsCosmosStream	CosmosUriPrefix
ExtractFile	IsNTFSFile	FileUriPrefix

…

GetPartitionedTable<T> Method (string)

Creates a partitioned **DryadTable<T>** object from a specified data source.

```
public DryadTable<T> GetPartitionedTable<T>(
                        string tableName
)
```

Types

T

The type of data in the collection.

Parameters

tableName

The table name, for stored tables. To create a new table from a set of files, set this parameter to the name of the metadata file.

Return Value

Returns a **DryadTable<T>** object that represents the data.

Remarks

The metadata file contains the names and location of the partition files, and how they are to be distributed to the cluster's computers. The file contains three sections:

1. The first line is set to the name of the folder that contains the files.
2. The second line is set to the number of partitions.
3. The remainder of the file consists of one line per partition, and describes how the partitions are to be distributed among the computers on the cluster.

Each line consists of three or more elements—separated by commas—in the following order:

- The partition number.
- The partition size, in bytes.
- The name of the computer that the partition is to be placed on.

For fault tolerance, it is sometimes useful to place the same partition on multiple computers. In that case, the final element on the line is a comma-separated list of the computer names that the partition file is to be placed on.

For example, assume that the Match example runs on a cluster with four worker computers, m1, m2, m3, and m4. The input text is broken into four partition files of 100,000 bytes each, as follows:

- MatchData.00000000 goes on m1.
- MatchData.00000001 goes on m2 and m3.
- MatchData.00000002 goes on m3.
- MatchData.00000003 goes on m4

All the partition files for this example are in the \DryadLINQData folder. The metadata file is named Match_Meta.txt, and contains the following text:

```
\DryadLINQData
4
0,1000000,m1
1,1000000,m2,m3
2,1000000,m3
3,1000000,m4
```

…

GetTable<T> (string) Method

Creates a **DryadTable<T>** object from a specified data source.

```
public DryadTable<T> GetTable<T>(
                        string fileName
)
```

Types

T

The type of data in the collection.

Parameters

fileName

The table name, for stored tables. To create a new table, set this parameter to the name of the data file.

Return Value

Returns a **DryadTable<T>** object that contains the data.

…