David Chappell
& Associates

INTRODUCING LINQ TO HPC

DATA-INTENSIVE COMPUTING WITH WINDOWS HPC SERVER

DAVID CHAPPELL

JUNE 2011

SPONSORED BY MICROSOFT CORPORATION

CONTENTS

UNDERSTANDING LINQ TO HPC

The era of data-intensive computing has arrived. Applications that must work with large amounts of unstructured data are becoming common. Yet creating those applications, then having them execute in a reasonable amount of time, can be challenging. Doing this successfully requires building on a platform designed to support this class of problems.

LINQ to HPC provides this platform for Windows developers. Built on Windows HPC Server, LINQ to HPC allows creating and running applications that process large amounts of unstructured data on a cluster of commodity servers. This paper provides a big-picture look at the LINQ to HPC technology, describing what it does and how it works.

THE PROBLEM: WORKING WITH LARGE AMOUNTS OF UNSTRUCTURED DATA

Suppose you're responsible for running a large Web farm containing hundreds or thousands of servers. Each of those machines produces log files, and so the farm generates many gigabytes of new data every day. This data contains all kinds of potentially interesting information: patterns of user behavior, evidence of possible attacks, and more. You'd like to analyze these daily log files to tease out what's useful for you.

Yet processing all of this unstructured data on a single machine isn't feasible; it would take too long. Fortunately, the data can be divided into chunks, then processed in parallel on multiple computers. But doing this isn't especially simple. Writing parallel applications is a challenge for many developers, as is handling the failures that are likely to occur when the application runs across a group of machines. You know what you need to do, but doing it can be hard.

And while analyzing very large log files is a good example, it's certainly not the only important challenge in this area. Many business and scientific scenarios create large amounts of unstructured data that must be processed in a reasonable amount of time. For example, think of a medical researcher who needs to apply an image processing algorithm to a large number of MRI scans, perhaps to find common anomalies across all of them. Or suppose a business has purchased a large non-relational customer data set from a third party and needs to understand how that data compares to its existing customer information. The ultimate goal might be to combine all of this information in a traditional data warehouse, but determining how best to do this might first require a good deal of experimentation with the data.

The right kind of platform can make solving problems like these easier. But what should that platform look like? One obvious characteristic is that it must let applications and the data they work on be divided across many computers, allowing work to be done in parallel on cluster. This suggests a clear connection between data-intensive applications and the traditional world of high-performance computing (HPC).

But processing large amounts of unstructured data is different from classic HPC in an important way. HPC jobs today are most often CPU-bound, and so the goal of running on a cluster is to let them use many CPUs at once. Data-intensive jobs, by contrast, are typically I/O bound—they're limited by how fast they can read and write data. Rather than relying primarily on processing power, these jobs most often do relatively simple computations on massive amounts of information. Running data-intensive jobs on a cluster still makes sense, but the goal isn't to use lots of CPUs at once—it's to use lots of disks at once. When a single application can read from many disk spindles simultaneously, processing that data gets much faster.

It can be challenging to work out exactly how much data an application needs to work with to qualify as data-intensive. Is ten gigabytes enough? How about fifty? Rather than trying to work out a minimum size, a clearer way to think about the issue is to list three main criteria for applications that can benefit from a data-intensive approach:

- The application takes longer to run than you'd like it to, i.e., you're not happy with its current performance.
- The application is I/O bound, not CPU bound. Making the job run faster requires speeding up data access, not adding more CPU power.
- The data you're working with is unstructured; it's not stored in a relational database.

If your application currently has all three of these characteristics, it probably qualifies as a data-intensive problem.

Yet how can ordinary mortals create this kind of application? Writing software that works across many computers in parallel to process a large amount of unstructured data can be hard. What's needed is a technology that makes this easier, something that lets non-specialists create and run data-intensive applications. In the Windows world, this technology is LINQ to HPC.

THE SOLUTION: WHAT LINQ TO HPC PROVIDES

LINQ to HPC was released in a beta version at the same time as Windows HPC Server 2008 R2 Service Pack (SP) 2. The basics of how a LINQ to HPC job works are straightforward. Figure 1 shows the technology's main components.

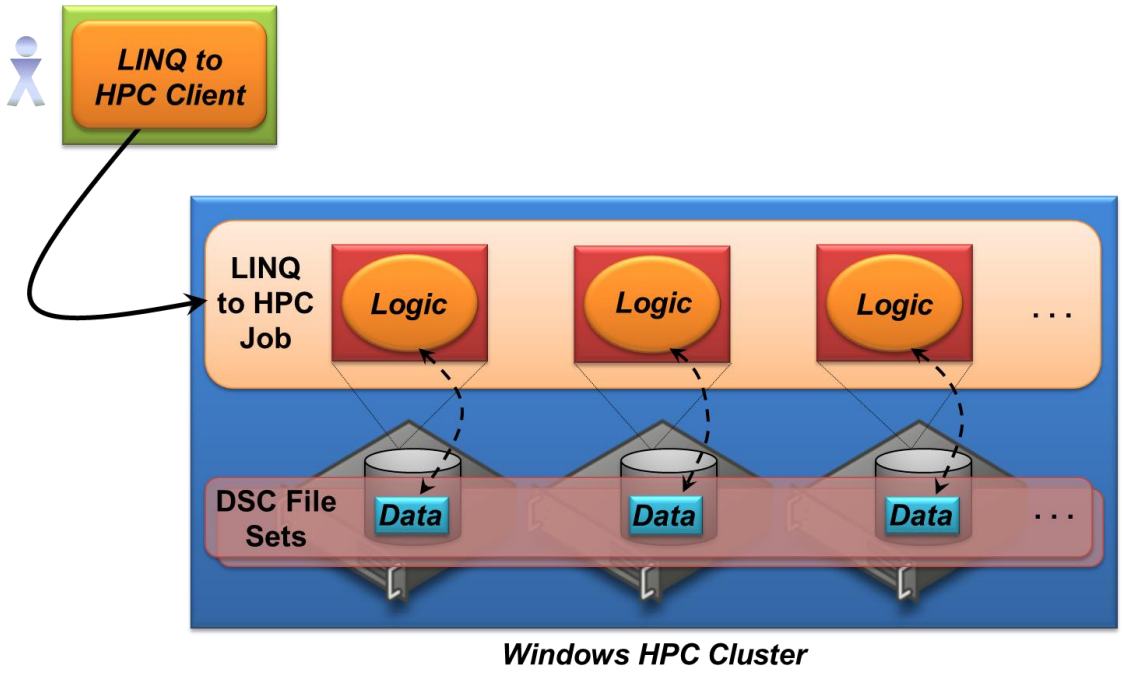

Figure 1: A LINQ to HPC client generates a LINQ to HPC job that runs on a Windows HPC cluster and processes data stored in DSC file sets.

An application built using LINQ to HPC runs on a cluster of commodity servers, each running Windows HPC Server. Figure 1 shows only three computers, but a more realistic scenario would use 30 or 300 or 3,000 machines, all potentially running a single LINQ to HPC application. The application's logic runs on the cluster as a *LINQ to HPC job*, while the data this application works on is organized into *DSC file sets*. ("DSC" stands for "Distributed Storage Catalog", a technology described later.) Like the application logic, a DSC file set's data is distributed across many machines in the cluster, as the figure shows.

It's important to realize that each instance of a LINQ to HPC job's logic typically works with data stored on the machine this instance is running on. As much as possible, LINQ to HPC runs a job's logic where the data is; it doesn't move data to the logic. This *data locality* means that each instance of a job's logic has its own unique disk and works on its own distinct part of the DSC file set, an approach that speeds up I/O-bound applications.

To use LINQ to HPC, a developer typically does two things. First, she creates a DSC file set by spreading the unstructured (i.e., non-relational) data she's working with across the disks of machines in the cluster. Next, she creates a LINQ to HPC job that processes this data. Developers don't create LINQ to HPC jobs directly, however. Instead, they write a *LINQ to HPC client*, which generates a LINQ to HPC job for them.

As its name suggests, LINQ to HPC uses a flavor of the *Language INtegrated Query (LINQ)* technology that Windows developers commonly use to access relational and other data. Like other LINQ dialects, LINQ to HPC lets a developer create a query in C# that resembles a SQL statement, complete with keywords such as **`select`**, **`from`**, and **`where`**. When a LINQ to HPC application is executed, the system generates and runs a LINQ to HPC job on the cluster. Rather than explicitly building and executing a parallel application, the developer lets LINQ to HPC handle this under the covers.

Microsoft first made LINQ to HPC available as a commercial product in 2011. The underlying technology was originally created several years earlier by Microsoft Research, however. Referred to as *Dryad*, versions of this technology have been (and still are) used internally by Microsoft to address a variety of data-intensive problems. Here are some examples:

- Dryad technology underlies Microsoft's Bing search analytics, running on a cluster of several thousand machines.
- Dryad applications are used to detect botnets by examining traffic patterns and other information.
- Dryad applications were used to create the software for Kinect, Microsoft's motion-sensing game controller for Xbox. Starting with a very large amount of motion-capture data produced by the actions of real people, Dryad applications analyzed this information to work out the best algorithms for converting human motion into software input.

Microsoft says that it uses Dryad to examine more than ten petabytes of unstructured data each day. By turning it into a commercial product with LINQ to HPC, the goal is to make this technology available to Microsoft's customers as well.

OTHER MICROSOFT TECHNOLOGIES FOR WORKING WITH LARGE AMOUNTS OF DATA

LINQ to HPC is a good fit for some kinds of problems. If you're a Windows developer trying to understand a large amount of unstructured data, it's probably the right choice. Yet suppose you're a data analyst who

needs to work with lots of relational data. You might prefer to use higher-level analysis tools—you don't want to write applications—and this data is likely stored in a database such as SQL Server. To address this situation, Microsoft provides SQL Server Parallel Data Warehouse.

Like LINQ to HPC, SQL Server Parallel Data Warehouse can be used to analyze very large datasets. Also like LINQ to HPC, applications built on this foundation can run in parallel across many machines. Yet these two technologies are quite distinct, with each one targeting a particular set of problems.

The most obvious difference is the one just mentioned: LINQ to HPC works on unstructured data stored in files, while SQL Server Parallel Data Warehouse relies on relational data stored in SQL Server. Traditional data warehouses commonly use relational storage, and when those warehouses get very large, processing their data in parallel makes sense. SQL Server Parallel Data Warehouse is the right choice for these scenarios.

Because it's based on SQL Server, this technology also lets analysts create cubes, perform data mining, and do other kinds of online analytical processing (OLAP). It's a good choice when the analyst knows what cube he wants to create—and thus already understands the data to some degree—and when the same data will be analyzed multiple times. The usual SQL Server tools are available for doing this kind of work with Parallel Data Warehouse, such as SQL Server Analysis Services (SSAS) and SQL Server Reporting Services (SSRS).

LINQ to HPC works on raw unstructured data, and so it's not aimed at solving classic OLAP problems. Instead, developers typically create LINQ to HPC applications to do ad hoc data analysis. Rather than creating a cube that's used many times, for example, it's not uncommon for a developer to create a LINQ to HPC application, then run it only once over a specific dataset. If the goal is to examine a large log file, for instance, looking for evidence of break-ins or other problems, the file will most likely be read just once. In situations like these, investing time and money in loading the data into a conventional data warehouse probably isn't worthwhile. And because it's built on Windows HPC Server, which can support clusters containing thousands of computers, LINQ to HPC can be used for larger scenarios than SQL Server Parallel Data Warehouse.

Both LINQ to HPC and SQL Server Parallel Data Warehouse are useful for working with large amounts of data, and Windows users can choose the one that's the best fit for their problem. They can also combine the two, perhaps using LINQ to HPC for initial analysis or data cleansing, then relying on SQL Server Integration Services (SSIS) to load the data into SQL Server Parallel Data Warehouse. (This effectively makes LINQ to HPC part of the extract, transform, and load—ETL—tool chain for the data warehouse.) From there, SSAS and other tools can be used to further analyze the data. Different data problems call for different data tools, and so Microsoft supports multiple options.

EXAMINING LINQ TO HPC: A CLOSER LOOK AT THE TECHNOLOGY

As Figure 1 showed, understanding LINQ to HPC means understanding three distinct components:

- DSC file sets that contain large amounts of unstructured data.
- LINQ to HPC jobs that process DSC file sets.
- LINQ to HPC clients that generate LINQ to HPC jobs.

This section looks at all three. First, however, it's important to take a quick look at the foundation on which LINQ to HPC depends: Windows HPC Server.

THE FOUNDATION: WINDOWS HPC SERVER

If an application needs lots of compute power, lots of disk access, or both, it often makes sense to run that application on a cluster of machines. The purpose of Windows HPC Server is to help create and manage a cluster, then to run parallel applications on the computers it contains.

A LINQ to HPC application spreads its logic across multiple *compute nodes*, each of which is a server running Windows HPC Server 2008 or Windows HPC Server 2008 R2[1]. In general, an HPC application might run the same logic on every compute node, with each instance processing different data, or it might run different logic on different nodes. To assign an application's logic to compute nodes, Windows HPC Server provides the *HPC Job Scheduler*. This scheduler runs on a machine called the *head node*. Figure 1 shows how all of this looks.

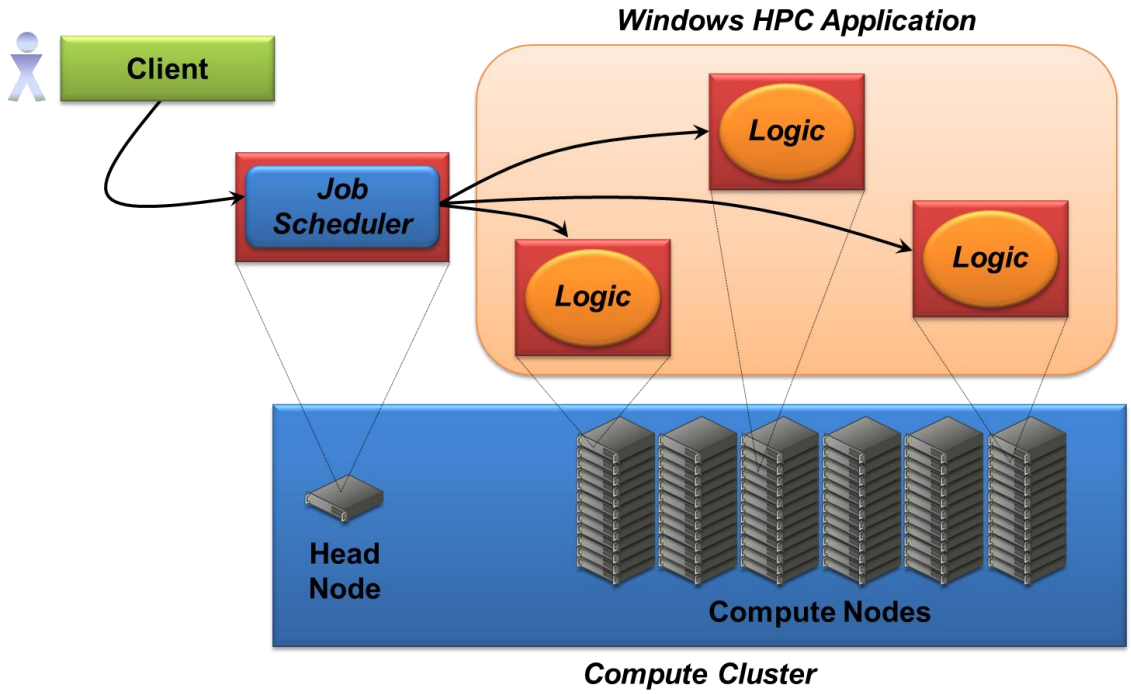

Figure 2: In Windows HPC Server, the Job Scheduler running on a head node distributes the logic of a parallel application across a group of compute nodes.

As the figure shows, both the head node and the compute nodes are part of the same compute cluster within an organization. To use these resources, a user typically submits a Windows HPC application from a client workstation to the head node. The job scheduler parcels the application's logic across the compute

[1] Windows HPC Server 2008 R2 SP2 also allows a cluster to use client workstations and Windows Azure instances as compute nodes. These options aren't supported by the LINQ to HPC beta release, however.

nodes that this application has access to. For higher reliability, an organization can also create a secondary head node that automatically takes over if the primary fails.

Assigning work to compute nodes isn't a simple task, and so the Job Scheduler is a sophisticated piece of software. Among other things, it assigns jobs to compute nodes based on their priority, allowing new high-priority jobs to jump to the head of the line. It also lets the user submitting a job specify what kind of resources the job needs, then places the job appropriately.

In some HPC applications, the logic running on different machines needs to interact while the job runs. To allow this, Windows HPC Server supports the industry-standard *Message Passing Interface (MPI)*. Windows HPC Server also allows creating applications where the distributed logic doesn't interact while it's running, a category commonly known as *embarrassingly parallel* applications. The system also provides debuggers and profilers designed expressly for working with applications running on a cluster.

Much of what Windows HPC Server offers is focused on creating and running applications. Yet managing a cluster and the applications that run on it is a non-trivial task. For example, an administrator must create the cluster, then determine things such as which compute nodes are available for each job. An application might have a specific set assigned to it, or it might be able to draw compute nodes from a generally available pool. To make life easier for cluster administrators, Windows HPC Server includes *HPC Cluster Manager*, a graphical tool for doing these things and more.

Today, Windows HPC Server is commonly used for both MPI applications and embarrassingly parallel applications. Adding LINQ to HPC brings another option: the ability to create data-intensive applications. Because it's based on Windows and .NET technologies, LINQ to HPC is designed to be familiar to Windows developers. Windows administrators should also have a relatively easy time with the product, since its management interfaces are similar to those used by Microsoft's System Center tools. By adding support for data-intensive applications to its existing Windows-based cluster technology, Microsoft aims at expanding the range of problems these clusters can address.

STORING DATA: THE DISTRIBUTED STORAGE CATALOG

By definition, a data-intensive application works with lots of data, usually too much to fit in a single Windows file. Yet forcing developers to work with data spread across many different files is unappealing. To make life simpler, LINQ to HPC instead lets them work with large amounts of data as a single unit using the *Distributed Storage Catalog (DSC)*.

The DSC allows creating DSC file sets, each of which is a logical grouping of data that's physically stored across multiple machines in a cluster. Each component of a DSC file set—called a *DSC file* —is really just an ordinary Windows file, stored using the usual NTFS file system. To a LINQ to HPC developer, however, the entire DSC file set can be processed as a single unit. The DSC keeps track of all of the DSC files that belong to each DSC file set, as Figure 2 shows.

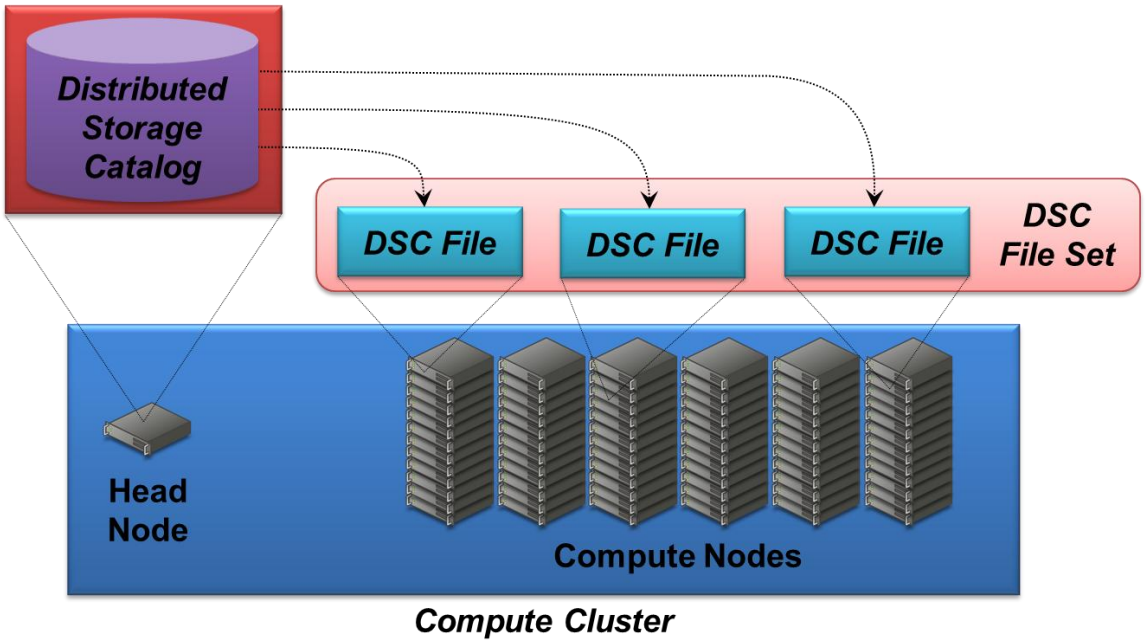

Figure 3: The Distributed Storage Catalog keeps track of where each DSC file in a DSC file set is located.

The DSC runs on the cluster's head node, and it stores information about where each DSC file in each DSC file set is located. It stores other metadata as well, such as the DSC file set's size, creation time, and more.

Developers have a few options for creating a DSC file set. One choice is to create and run a LINQ to HPC application that reads data from some other source, then copies it into DSC files on machines in the cluster. As it does this, the application makes calls to DSC via a LINQ to HPC API to tell it which DSC files comprise a particular DSC file set and where each DSC file is located. A developer can also create a DSC file set with an ordinary Windows application—using LINQ to HPC isn't required. One more option is to use a LINQ to HPC command-line tool to copy data from another location into a DSC file set.

When a developer creates a DSC file set, it's up to her to decide how big each DSC file should be—there's no fixed size. Whatever choice she makes, the DSC replicates each of the DSC files in a DSC file set to different machines in the cluster. (By default, it creates three replicas of each one, but a LINQ to HPC user can change this if desired.) The DSC then monitors the machine that each replica lives on, creating a new replica somewhere else if that machine becomes unavailable.

Once a DSC file set exists, a LINQ to HPC application can read from it and append new data to it. The application can't modify the DSC file set's existing data, however. Most data-intensive applications read an entire DSC file set, typically from beginning to end. Given this, DSC file sets are designed to be written once, then read many times. Allowing random-access reads and writes, as a relational database does when supporting transactional applications, wouldn't make sense. As with more traditional data warehouses, the intent is to analyze data, not update it.

Note that DSC isn't an entirely new file system dedicated to storing large amounts of unstructured data. Instead, it's more like a specialized directory that lets a LINQ to HPC job treat a group of files as a single unit. The goal is to let the job's logic access data directly using standard Windows file access

mechanisms—there's no extra layer of code to go through—and thus make access to that data as fast as possible.

PROCESSING DATA: LINQ TO HPC JOBS

Once the data a developer needs is available in a DSC file set, he can create LINQ to HPC applications that process this data. Executing a LINQ to HPC client generates a LINQ to HPC job to carry out the application's work. A developer doesn't need to care about how these jobs run; he just writes the client, and the correct LINQ to HPC job is created and executed on his behalf. Still, it's useful to have some sense of what's happening under the covers. Figure 4 illustrates what happens when a LINQ to HPC job executes.

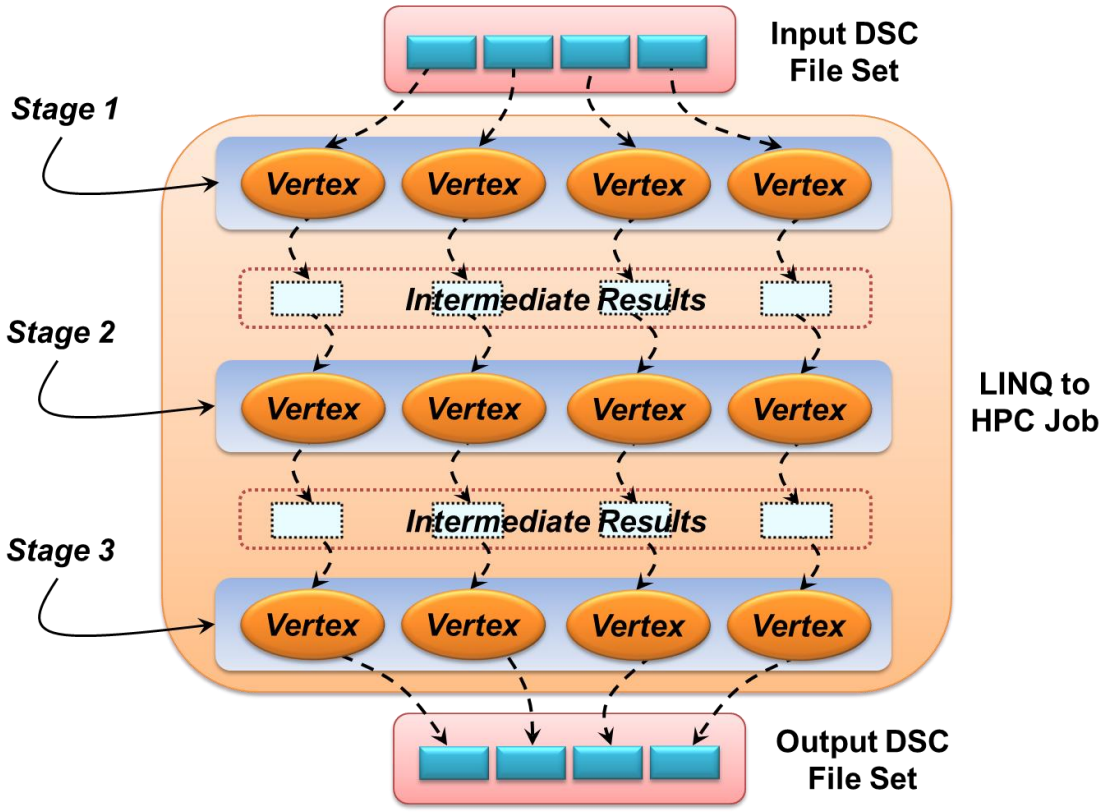

Figure 4: A LINQ to HPC job consists of one or more stages, each of which can read data, process that data using some number of vertices, then write results.

As Figure 4 suggests, the execution pattern of a LINQ to HPC job is a directed acyclic graph[2]. Each *vertex* of the graph—the orange ovals—is logic running on a computer in the cluster that carries out some part of the job. The job's vertices are grouped into *stages*, each of which performs a particular subset of the job's work. Here, the first stage reads and processes data from an input DSC file set, then writes new files

[2] This generalized tree-like structure is the source of the technology's original name: A dryad is a tree nymph in Greek mythology.

containing intermediate results. (LINQ to HPC is smart enough to automatically delete the intermediate results when they're no longer needed.) These intermediate files are read by the vertices in the second stage, which perform more processing and write more intermediate files. These results are in turn read and processed by the vertices in the third stage, which write the job's final results into an output DSC file set.

Different LINQ to HPC jobs can have different numbers of stages, each performing a specific kind of processing. And although it's not shown in the figure, different stages can run different numbers of vertex instances. Some stages might use 100 vertices to carry out their work, while others require only half that many.

To understand what happens when a LINQ to HPC job executes, it's useful to start by illustrating all of the technology's main components and where each one runs. Figure 5 shows a more complete picture.

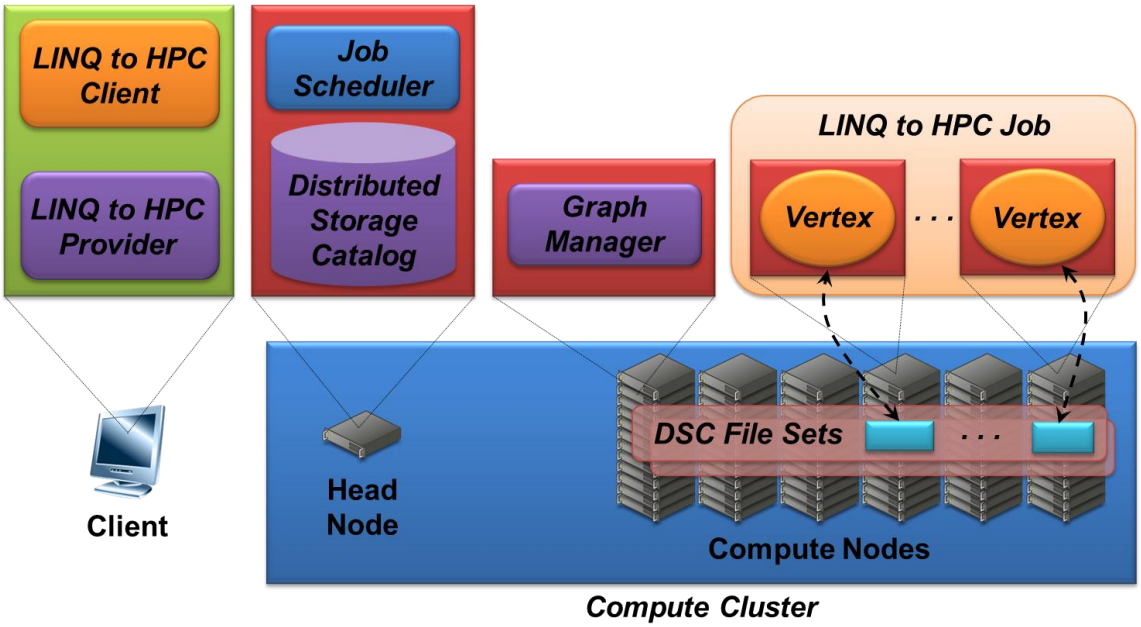

Figure 5: The various components of LINQ to HPC are spread across the client workstation, the cluster's head node, and the cluster's compute nodes.

A LINQ to HPC client relies on a *LINQ to HPC Provider* running on the user's workstation. As described earlier, the Job Scheduler and the Distributed Storage Catalog run on the compute cluster's head node, while each vertex in a LINQ to HPC job typically runs on the same machine as the DSC file it works on. Finally, a LINC to HPC component called the *Graph Manager* runs on another machine in the cluster, and it's responsible for creating the job's vertices and monitoring their execution.

All of these components work together when a LINQ to HPC application is run. Figure 6 gives a more complete view of this process.

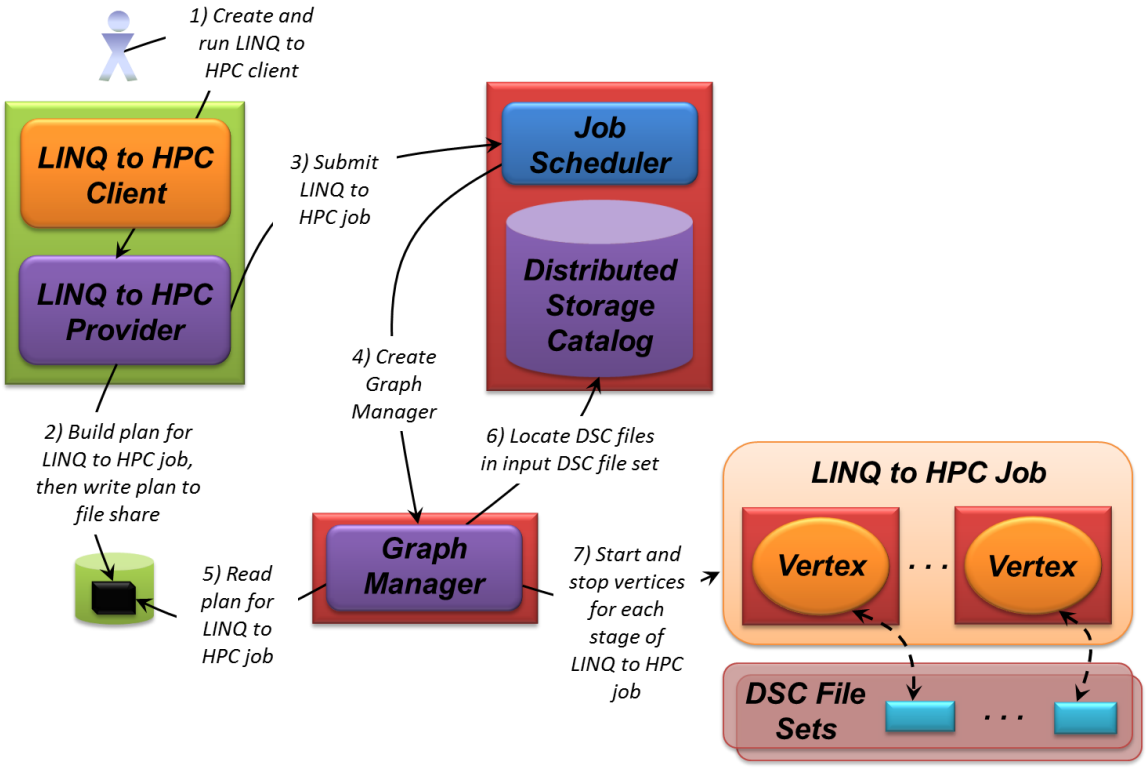

Figure 6: All of the LINQ to HPC components work together to run a LINQ to HPC application.

The process begins when a developer runs a LINQ to HPC client (step 1). The LINQ to HPC Provider examines the LINQ queries in the client to construct an execution plan for a LINQ to HPC job, then writes this plan to a file share (step 2). The plan contains a description of the job's graph along with the names of its input and output DSC file sets. The provider then starts a new LINQ to HPC job by contacting the Job Scheduler (step 3). The Job Scheduler creates an instance of the Graph Manager (step 4), which reads the plan for the job created earlier by the LINQ to HPC Provider (step 5).

The Graph Manager now knows what this LINQ to HPC job will look like. It doesn't yet know exactly where to run the job, however. Since each of the vertices in the job's first stage should execute on the computer that holds the DSC file that vertex works on, the Graph Manager contacts the Distributed Storage Catalog to find a replica of each DSC file in the input DSC file set (step 6). The Graph Manager then starts the vertices for the job's first stage on those computers, starting and stopping new vertices as the job progresses through its stages (step 7).

The number of vertices the Graph Manager creates at each stage depends on how many DSC files are in the DSC file set that stage is reading. Assuming enough compute nodes are available, each DSC file in that DSC file set will be assigned its own vertex, and so the number of running vertices for a particular stage will equal the number of DSC files in the DSC file set read by that stage.

One challenge of running a job across multiple machines is dealing with failure: What happens when a machine crashes in the middle of running a LINQ to HPC job? To handle this, the Graph Manager monitors the execution of each vertex. If a machine running a vertex goes down, the Graph Manager starts another instance of that vertex on another machine in the cluster that holds a replica of the same DSC file. The

new vertex starts over from the beginning of the stage—it doesn't pick up where the failed vertex left off. Since DSC file sets are read-only, however, the DSC file it's working on hasn't changed.

As a LINQ to HPC job executes, vertices in different stages might run on different compute nodes in the cluster, since intermediate results might be stored in various places. Throughout the process, however, the basic principle remains the same: The Graph Manager strives to run each vertex on the same machine that holds the data it's processing. This strong commitment to data locality is at the heart of how LINQ to HPC enables high-performance processing of data.

CREATING DATA-INTENSIVE APPLICATIONS: LINQ TO HPC CLIENTS

Having a sense of what's going on when a LINQ to HPC job runs is useful. From a developer's point of view, though, what really matters is what a LINQ to HPC client looks like. Accordingly, it's worth taking a closer look at this approach to creating data-intensive applications.

Why Use Queries?

To understand a LINQ to HPC client, it's useful to first understand the basics of LINQ itself. As the "Q" in its name indicates, LINQ is a way to express queries. But rather than use an external language to do this, such as SQL, LINQ extends .NET programming languages to support queries. For example, here's a simple example of a LINQ query against a relational database expressed in C#:

```
var exampleQuery =
      from s in Students
      where s.Gender == "M"
      select s.Name;
```

This query returns the names of all male students as a group of strings stored in the variable **exampleQuery**. A LINQ query isn't actually executed until its results are needed, however, so to run the query and print out its results, a program might do this:

```
foreach (string name in exampleQuery) {
      Console.WriteLine(name);
}
```

The query's syntax is reminiscent of SQL. Yet it's important to understand that despite this similarity, the LINQ query shown above isn't an embedded SQL statement. Instead, it's pure C#, part of the language itself. This means that the query can use other program variables, be accessed in a debugger, and more. The name of this technology is "Language-Integrated Query" for a reason: The statements for querying diverse kinds of data are integrated directly into the programming language.

LINQ is commonly used to access relational data, as this example suggests. This isn't the only choice, however. Microsoft also provides support for using LINQ to access XML data, .NET objects, and other kinds of information. In fact, the technology was designed from the start to allow plugging in a variety of different providers.

Given this, extending LINQ to work with large amounts of unstructured data isn't such a stretch—the LINQ to HPC Provider fits nicely into the LINQ architecture. Yet creating a distributed application that processes data can seem quite different from writing a query. Writing a LINQ to HPC job directly (an option that's

not available to Microsoft's customers) certainly isn't the same thing as writing a query—it's a distributed application. Still, think about what a query does: It reads some data, performs some processing on it, then writes other data. In the simple LINQ query above, for instance, the application reads the Students table, filters its contents, then displays the results. From this perspective, it's entirely reasonable to model a LINQ to HPC application as the execution of a query.

In fact, the LINQ to HPC Provider acts much like a query processor in a database system. It reads whatever LINQ queries the developer has created, then generates a kind of query plan, expressed as a LINQ to HPC job. Exactly what stages this job contains and what each of those stages does is determined by the LINQ to HPC Provider based on what the query requires.

Taking this approach makes life significantly simpler for developers. (In fact, anybody creating a Windows HPC application that maps well to LINQ to HPC might find this technology worthwhile, even if their problem doesn't involve a large amount of data.) Using LINQ to HPC requires thinking about problems in terms of queries, however, and so it's worth looking a little more deeply into this idea.

An Example LINQ to HPC Client

The easiest way to get a feeling for the way LINQ is used here is to walk through a simple example. Suppose you've got an unstructured data set that contains a list of all names given to newborn babies in every American state for the last 40 years. Suppose further that you'd like to create a list, sorted by year, of the names given only to babies born in California between 2005 and 2010. Here's a simple C# application that does this using LINQ to HPC:

```
using System;
using Microsoft.Hpc.Linq;

public static class BabyNames {

   internal class BabyInfo {
      public string Name { get; set; }
      public string State { get; set; }
      public int Year { get; set; }
   }

   public static void Main(){

      HpcLinqConfiguration config = new
         HpcLinqConfiguration("ExampleCluster");

      HpcLinqContext context = new HpcLinqContext(config);

      var inputBabies = context.FromDsc<BabyInfo>("babyNames");


      var outputBabies = from b in inputBabies
                         where b.State == "CA"
                         && b.Year >= 2005
                         && b.Year <= 2010
                         orderby b.Year
                         select b;
```

```
        outputBabies.ToDsc("sorted-CA-babyNames-2005-2010").Submit();
    }
}
```

After the opening **using** statements, the application declares a **BabyInfo** class to hold the data it will work with. Once this is done, its **Main** method creates a **config** object that reads LINQ to HPC configuration information for the cluster. It then uses this **config** object to create a **context** object and invokes the **context** object's **FromDsc** method. This method identifies the input DSC file set, here called **babyNames**, that the LINQ to HPC job will use. It uses a generic parameter, the **BabyInfo** class defined earlier, to define the format of data read from this DSC file set.

Next, the application constructs a LINQ query to process the baby name data in the input DSC file set. Here, the query filters the data in the input DSC file set by state and year, then orders the result by year. As before, this query isn't executed until its results are needed. Calling **Submit** on **outputBabies** does this, causing the LINQ to HPC Provider to generate a LINQ to HPC job. The name of the output DSC file set to which the application's results should be written is specified in the call to **ToDsc**.

When the LINQ to HPC job generated by this client runs, it will contain two stages, each running multiple vertices in parallel across multiple machines in the cluster. Figure 7 shows a slightly simplified picture of how this would look if the data in the DSC file set **babyNames** was stored in four DSC files.

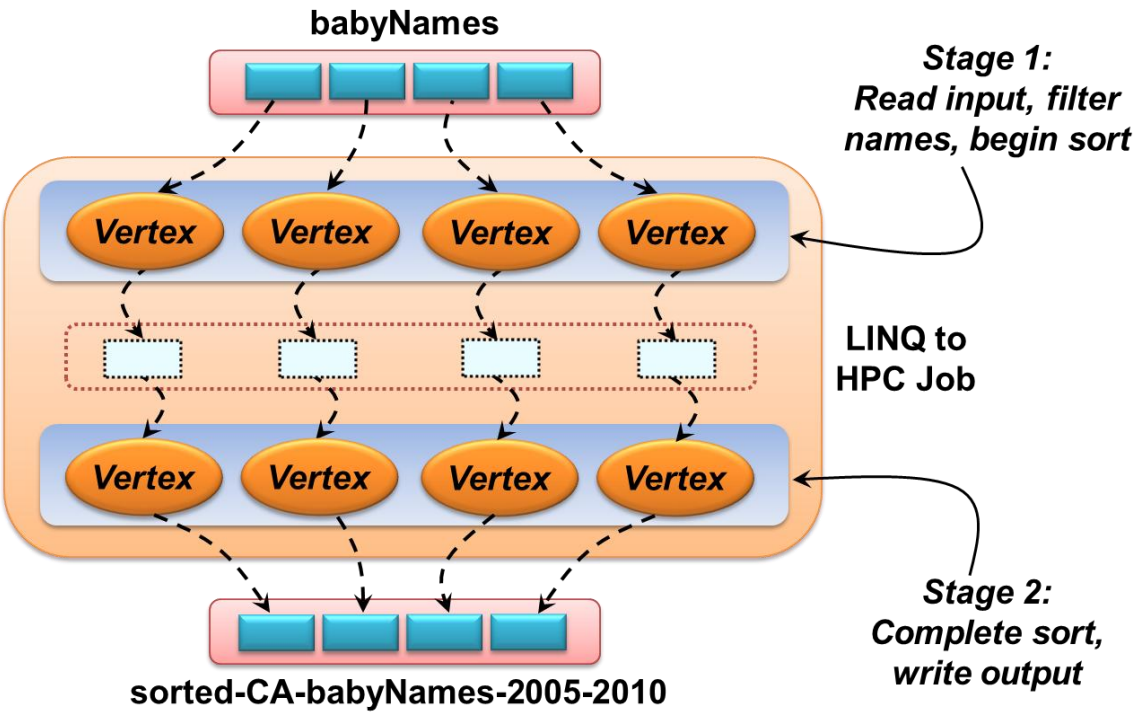

Figure 7: The Baby Names LINQ to HPC client creates a LINQ to HPC job with two stages.

As the figure shows, the vertices in the first stage read the baby name data from the input DSC file set, filter this data and begin sorting it, then write out intermediate results. The vertices in the second stage read these intermediate results, finish the sort, then write the output DSC file set. As usual, the intermediate results are deleted when they're no longer needed.

While this example illustrates the basics, LINQ to HPC can do more. For example, a developer might use multiple LINQ to HPC queries in the same application—he's not restricted to just one. The LINQ to HPC Provider is smart enough to generate a single LINQ to HPC job to perform the work done by all of them. But whatever options a developer chooses, the result is the same: A LINQ to HPC client generates a LINQ to HPC job that processes DSC file sets on a Windows HPC cluster.

CONCLUSION

The world is awash in data. The goal of LINQ to HPC is to help Windows developers—and the people who pay them—get more value from this data. Even though LINQ to HPC is new as a commercial offering, its roots inside Microsoft stretch back several years—it's not a wholly new technology. And while LINQ to HPC only runs on-premises today, Microsoft has announced plans to also make it available on Windows Azure, the company's public cloud platform.

As more scientific and business endeavors discover the value of large data sets, we can expect the demand for applications that process this data to keep growing. As more developers try to create software to meet this demand, the value of a platform designed for data-intensive applications will become increasingly evident. Given these two realities, LINQ to HPC is surely worth a look for anybody who needs to work with large amounts of unstructured information on Windows.

ABOUT THE AUTHOR

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps people around the world understand, use, and make better decisions about new technologies.