How to: Use LINQ to query files and directories

Article • 04/25/2024

Many file system operations are essentially queries and are therefore well suited to the LINQ approach. These queries are nondestructive. They don't change the contents of the original files or folders. Queries shouldn't cause any side-effects. In general, any code (including queries that perform create / update / delete operations) that modifies source data should be kept separate from the code that just queries the data.

There's some complexity involved in creating a data source that accurately represents the contents of the file system and handles exceptions gracefully. The examples in this section create a snapshot collection of FileInfo objects that represents all the files under a specified root folder and all its subfolders. The actual state of each FileInfo might change in the time between when you begin and end executing a query. For example, you can create a list of FileInfo objects to use as a data source. If you try to access the `Length` property in a query, the FileInfo object tries to access the file system to update the value of `Length`. If the file no longer exists, you get a FileNotFoundException in your query, even though you aren't querying the file system directly.

How to query for files with a specified attribute or name

This example shows how to find all files that have a specified file name extension (for example ".txt") in a specified directory tree. It also shows how to return either the newest or oldest file in the tree based on the creation time. You might need to modify the first line of many of the samples whether you're running this code on either Windows, Mac, or a Linux system.

C#

```csharp
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);
var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var fileQuery = from file in fileList
                where file.Extension == ".txt"
                orderby file.Name
                select file;
```

```
// Uncomment this block to see the full query
// foreach (FileInfo fi in fileQuery)
// {
//    Console.WriteLine(fi.FullName);
// }

var newestFile = (from file in fileQuery
                  orderby file.CreationTime
                  select new { file.FullName, file.CreationTime })
                  .Last();

Console.WriteLine($"\r\nThe newest .txt file is
{newestFile.FullName}. Creation time: {newestFile.CreationTime}");
```

How to group files by extension

This example shows how LINQ can be used to perform advanced grouping and sorting operations on lists of files or folders. It also shows how to page output in the console window by using the Skip and Take methods.

The following query shows how to group the contents of a specified directory tree by the file name extension.

```C#
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

int trimLength = startFolder.Length;

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var queryGroupByExt = from file in fileList
                      group file by file.Extension.ToLower() into fi-
leGroup
                      orderby fileGroup.Count(), fileGroup.Key
                      select fileGroup;

// Iterate through the outer collection of groups.
foreach (var filegroup in queryGroupByExt.Take(5))
{
    Console.WriteLine($"Extension: {filegroup.Key}");
    var resultPage = filegroup.Take(20);

    //Execute the resultPage query
    foreach (var f in resultPage)
    {
        Console.WriteLine($"\t{f.FullName.Substring(trimLength)}");
    }
    Console.WriteLine();
}
```

The output from this program can be long, depending on the details of the local file system and what the `startFolder` is set to. To enable viewing of all results, this example shows how to page through results. A nested `foreach` loop is required because each group is enumerated separately.

How to query for the total number of bytes in a set of folders

This example shows how to retrieve the total number of bytes used by all the files in a specified folder and all its subfolders. The Sum method adds the values of all the items selected in the `select` clause. You can modify this query to retrieve the biggest or smallest file in the specified directory tree by calling the Min or Max method instead of Sum.

```C#
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

var fileList = Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories);

var fileQuery = from file in fileList
                let fileLen = new FileInfo(file).Length
                where fileLen > 0
                select fileLen;

// Cache the results to avoid multiple trips to the file system.
long[] fileLengths = fileQuery.ToArray();

// Return the size of the largest file
long largestFile = fileLengths.Max();

// Return the total number of bytes in all the files under the speci-
fied folder.
long totalBytes = fileLengths.Sum();

Console.WriteLine($"There are {totalBytes} bytes in
{fileList.Count()} files under {startFolder}");
Console.WriteLine($"The largest file is {largestFile} bytes.");
```

This example extends the previous example to do the following:

- How to retrieve the size in bytes of the largest file.
- How to retrieve the size in bytes of the smallest file.
- How to retrieve the FileInfo object largest or smallest file from one or more folders under a specified root folder.
- How to retrieve a sequence such as the 10 largest files.
- How to order files into groups based on their file size in bytes, ignoring files that are less than a specified size.

The following example contains five separate queries that show how to query and group files, depending on their file size in bytes. You can modify these examples to base the query on some other property of the FileInfo object.

C#

```csharp
// Return the FileInfo object for the largest file
// by sorting and selecting from beginning of list
FileInfo longestFile = (from file in fileList
                        let fileInfo = new FileInfo(file)
                        where fileInfo.Length > 0
                        orderby fileInfo.Length descending
                        select fileInfo
                        ).First();

Console.WriteLine($"The largest file under {startFolder} is {longest-
File.FullName} with a length of {longestFile.Length} bytes");

//Return the FileInfo of the smallest file
FileInfo smallestFile = (from file in fileList
                         let fileInfo = new FileInfo(file)
                         where fileInfo.Length > 0
                         orderby fileInfo.Length ascending
                         select fileInfo
                        ).First();

Console.WriteLine($"The smallest file under {startFolder} is {small-
estFile.FullName} with a length of {smallestFile.Length} bytes");

//Return the FileInfos for the 10 largest files
var queryTenLargest = (from file in fileList
                       let fileInfo = new FileInfo(file)
                       let len = fileInfo.Length
                       orderby len descending
                       select fileInfo
                      ).Take(10);

Console.WriteLine($"The 10 largest files under {startFolder} are:");

foreach (var v in queryTenLargest)
{
    Console.WriteLine($"{v.FullName}: {v.Length} bytes");
}

// Group the files according to their size, leaving out
// files that are less than 200000 bytes.
var querySizeGroups = from file in fileList
                      let fileInfo = new FileInfo(file)
                      let len = fileInfo.Length
                      where len > 0
                      group fileInfo by (len / 100000) into fileGroup
                      where fileGroup.Key >= 2
                      orderby fileGroup.Key descending
                      select fileGroup;

foreach (var filegroup in querySizeGroups)
{
    Console.WriteLine($"{filegroup.Key}00000");
    foreach (var item in filegroup)
    {
        Console.WriteLine($"\t{item.Name}: {item.Length}");
    }
}
```

To return one or more complete FileInfo objects, the query first must examine each one in the data source, and then sort them by the value of their Length property. Then it can return the single one or the sequence with the greatest lengths. Use First to return the first element in a list. Use Take to return the first n number of elements. Specify a descending sort order to put the smallest elements at the start of the list.

How to query for duplicate files in a directory tree

Sometimes files that have the same name can be located in more than one folder. This example shows how to query for such duplicate file names under a specified root folder. The second example shows how to query for files whose size and LastWrite times also match.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);

IEnumerable<FileInfo> fileList = dir.GetFiles("*.*",
SearchOption.AllDirectories);
```

```
// used in WriteLine to keep the lines shorter
int charsToSkip = startFolder.Length;

// var can be used for convenience with groups.
var queryDupNames = from file in fileList
                    group file.FullName.Substring(charsToSkip) by
file.Name into fileGroup
                    where fileGroup.Count() > 1
                    select fileGroup;

foreach (var queryDup in queryDupNames.Take(20))
{
    Console.WriteLine($"Filename = {(queryDup.Key.ToString() ==
string.Empty ? "[none]" : queryDup.Key.ToString())}");

    foreach (var fileName in queryDup.Take(10))
    {
        Console.WriteLine($"\t{fileName}");
    }
}
```

The first query uses a key to determine a match. It finds files that have the same name but whose contents might be different. The second query uses a compound key to match against three properties of the FileInfo object. This query is much more likely to find files that have the same name and similar or identical content.

C#

```
    string startFolder = """C:\Program Files\dotnet\sdk""";
    // Or
    // string startFolder = "/usr/local/share/dotnet/sdk";

    // Make the lines shorter for the console display
    int charsToSkip = startFolder.Length;

    // Take a snapshot of the file system.
    DirectoryInfo dir = new DirectoryInfo(startFolder);
    IEnumerable<FileInfo> fileList = dir.GetFiles("*.*",
SearchOption.AllDirectories);

    // Note the use of a compound key. Files that match
    // all three properties belong to the same group.
    // A named type is used to enable the query to be
    // passed to another method. Anonymous types can also be used
    // for composite keys but cannot be passed across method bound-
aries
    //
    var queryDupFiles = from file in fileList
                        group file.FullName.Substring(charsToSkip) by
                        (Name: file.Name, LastWriteTime: file.Last-
WriteTime, Length: file.Length )
                        into fileGroup
```

```
                              where fileGroup.Count() > 1
                              select fileGroup;

    foreach (var queryDup in queryDupFiles.Take(20))
    {
        Console.WriteLine($"Filename = {(queryDup.Key.ToString() ==
string.Empty ? "[none]" : queryDup.Key.ToString())}");

        foreach (var fileName in queryDup)
        {
            Console.WriteLine($"\t{fileName}");
        }
    }
}
```

How to query the contents of text files in a folder

This example shows how to query over all the files in a specified directory tree, open each file, and inspect its contents. This type of technique could be used to create indexes or reverse indexes of the contents of a directory tree. A simple string search is performed in this example. However, more complex types of pattern matching can be performed with a regular expression.

C#

```csharp
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

string searchTerm = "change";

var queryMatchingFiles = from file in fileList
                         where file.Extension == ".txt"
                         let fileText =
File.ReadAllText(file.FullName)
                         where fileText.Contains(searchTerm)
                         select file.FullName;

// Execute the query.
Console.WriteLine($"""The term "{searchTerm}" was found in:""");
foreach (string filename in queryMatchingFiles)
{
    Console.WriteLine(filename);
}
```

How to compare the contents of two folders

This example demonstrates three ways to compare two file listings:

- By querying for a Boolean value that specifies whether the two file lists are identical.
- By querying for the intersection to retrieve the files that are in both folders.
- By querying for the set difference to retrieve the files that are in one folder but not the other.

The techniques shown here can be adapted to compare sequences of objects of any type.

The `FileComparer` class shown here demonstrates how to use a custom comparer class together with the Standard Query Operators. The class isn't intended for use in real-world scenarios. It just uses the name and length in bytes of each file to determine whether the contents of each folder are identical or not. In a real-world scenario, you should modify this comparer to perform a more rigorous equality check.

```C#
// This implementation defines a very simple comparison
// between two FileInfo objects. It only compares the name
// of the files being compared and their length in bytes.
class FileCompare : IEqualityComparer<FileInfo>
{
    public bool Equals(FileInfo? f1, FileInfo? f2)
    {
        return (f1?.Name == f2?.Name &&
                f1?.Length == f2?.Length);
    }

    // Return a hash that reflects the comparison criteria. According
to the
    // rules for IEqualityComparer<T>, if Equals is true, then the
hash codes must
    // also be equal. Because equality as defined here is a simple
value equality, not
    // reference identity, it is possible that two or more objects
will produce the same
    // hash code.
    public int GetHashCode(FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}

public static void CompareDirectories()
{
    string pathA = """C:\Program Files\dotnet\sdk\8.0.104""";
    string pathB = """C:\Program Files\dotnet\sdk\8.0.204""";

    DirectoryInfo dir1 = new DirectoryInfo(pathA);
    DirectoryInfo dir2 = new DirectoryInfo(pathB);

    IEnumerable<FileInfo> list1 = dir1.GetFiles("*.*",
SearchOption.AllDirectories);
    IEnumerable<FileInfo> list2 = dir2.GetFiles("*.*",
SearchOption.AllDirectories);

    //A custom file comparer defined below
    FileCompare myFileCompare = new FileCompare();

    // This query determines whether the two folders contain
    // identical file lists, based on the custom file comparer
    // that is defined in the FileCompare class.
    // The query executes immediately because it returns a bool.
    bool areIdentical = list1.SequenceEqual(list2, myFileCompare);

    if (areIdentical == true)
    {
        Console.WriteLine("the two folders are the same");
    }
    else
    {
        Console.WriteLine("The two folders are not the same");
    }

    // Find the common files. It produces a sequence and doesn't
    // execute until the foreach statement.
    var queryCommonFiles = list1.Intersect(list2, myFileCompare);

    if (queryCommonFiles.Any())
    {
        Console.WriteLine($"The following files are in both folders
(total number = {queryCommonFiles.Count()}):");
        foreach (var v in queryCommonFiles.Take(10))
        {
            Console.WriteLine(v.Name); //shows which items end up in
result list
        }
    }
    else
    {
        Console.WriteLine("There are no common files in the two fold-
ers.");
    }

    // Find the set difference between the two folders.
    var queryList1Only = (from file in list1
                          select file)
                          .Except(list2, myFileCompare);

    Console.WriteLine();
    Console.WriteLine($"The following files are in list1 but not
list2 (total number = {queryList1Only.Count()}):");
    foreach (var v in queryList1Only.Take(10))
    {
        Console.WriteLine(v.FullName);
    }

    var queryList2Only = (from file in list2
                          select file)
                          .Except(list1, myFileCompare);

    Console.WriteLine();
    Console.WriteLine($"The following files are in list2 but not
list1 (total number = {queryList2Only.Count()}:");
    foreach (var v in queryList2Only.Take(10))
    {
        Console.WriteLine(v.FullName);
    }
}
```

How to reorder the fields of a delimited file

A comma-separated value (CSV) file is a text file that is often used to store spreadsheet data or other tabular data represented by rows and columns. By using the Split method to separate the fields, it's easy to query and manipulate CSV files using LINQ. In fact, the same technique can be used to reorder the parts of any structured line of text; it isn't limited to CSV files.

In the following example, assume that the three columns represent students' "family name," "first name", and "ID." The fields are in alphabetical order based on the students' family names. The query produces a new sequence in which the ID column appears first, followed by a second column that combines the student's first name and family name. The lines are reordered according to the ID field. The results are saved into a new file and the original data isn't modified. The following text shows the contents of the *spreadsheet1.csv* file used in the following example:

txt

```
Adams,Terry,120
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Cesar,114
Garcia,Debra,115
```

```
Garcia,Hugo,118
Mortensen,Sven,113
O'Donnell,Claire,112
Omelchenko,Svetlana,111
Tucker,Lance,119
Tucker,Michael,122
Zabokritski,Eugene,121
```

The following code reads the source file and rearranges each column in the CSV file to rearrange the order of the columns:

```C#
string[] lines = File.ReadAllLines("spreadsheet1.csv");

// Create the query. Put field 2 first, then
// reverse and combine fields 0 and 1 from the old field
IEnumerable<string> query = from line in lines
                            let fields = line.Split(',')
                            orderby fields[2]
                            select $"{fields[2]}, {fields[1]}
{fields[0]}";

File.WriteAllLines("spreadsheet2.csv", query.ToArray());

/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
*/
```

How to split a file into many files by using groups

This example shows one way to merge the contents of two files and then create a set of new files that organize the data in a new way. The query uses the contents of two files. The following text shows the contents of the first file, *names1.txt*:

txt

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

The second file, *names2.txt*, contains a different set of names, some of which are in common with the first set:

txt

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

The following code queries both files, takes the union of both files, then writes a new file for each group, defined by the first letter of the family name:

C#

```csharp
string[] fileA = File.ReadAllLines("names1.txt");
string[] fileB = File.ReadAllLines("names2.txt");

// Concatenate and remove duplicate names
var mergeQuery = fileA.Union(fileB);

// Group the names by the first letter in the last name.
var groupQuery = from name in mergeQuery
                 let n = name.Split(',')[0]
                 group name by n[0] into g
                 orderby g.Key
                 select g;

foreach (var g in groupQuery)
{
    string fileName = $"testFile_{g.Key}.txt";

    Console.WriteLine(g.Key);
    using StreamWriter sw = new StreamWriter(fileName);
    foreach (var item in g)
    {
        sw.WriteLine(item);
        // Output to console for example purposes.
        Console.WriteLine($"   {item}");
    }
}
/* Output:
    A
       Aw, Kam Foo
    B
       Bankov, Peter
       Beebe, Ann
    E
       El Yassir, Mehdi
    G
       Garcia, Hugo
       Guy, Wey Yuan
       Garcia, Debra
       Gilchrist, Beth
       Giakoumakis, Leo
    H
       Holm, Michael
    L
       Liu, Jinghao
    M
       Myrcha, Jacek
       McLin, Nkenge
    N
       Noriega, Fabricio
    P
       Potra, Cristina
    T
       Toyoshima, Tim
 */
```

How to join content from dissimilar files

This example shows how to join data from two comma-delimited files that share a common value that is used as a matching key. This technique can be useful if you have to combine data from two spreadsheets, or from a spreadsheet and from a file that has another format, into a new file. You can modify the example to work with any kind of structured text.

The following text shows the contents of *scores.csv*. The file represents spreadsheet data. Column 1 is the student's ID, and columns 2 through 5 are test scores.

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

The following text shows the contents of *names.csv*. The file represents a spreadsheet that contains the student's family name, first name, and student ID.

txt

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

Join content from dissimilar files that contain related information. File *names.csv* contains the student name plus an ID number. File *scores.csv* contains the ID and a set of four test scores. The following query joins the scores to the student names by using ID as a matching key. The code is shown in the following example:

C#

```csharp
string[] names = File.ReadAllLines(@"names.csv");
string[] scores = File.ReadAllLines(@"scores.csv");

var scoreQuery = from name in names
                  let nameFields = name.Split(',')
                  from id in scores
                  let scoreFields = id.Split(',')
                  where Convert.ToInt32(nameFields[2]) ==
Convert.ToInt32(scoreFields[0])
                  select $"{nameFields[0]},{scoreFields[1]},{score-
Fields[2]},{scoreFields[3]},{scoreFields[4]}";
```

```
Console.WriteLine("\r\nMerge two spreadsheets:");
foreach (string item in scoreQuery)
{
    Console.WriteLine(item);
}
Console.WriteLine("{0} total names in list", scoreQuery.Count());
/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
 */
```

How to compute column values in a CSV text file

This example shows how to perform aggregate computations such as Sum, Average, Min, and Max on the columns of a .csv file. The example principles that are shown here can be applied to other types of structured text.

The following text shows the contents of *scores.csv*. Assume that the first column represents a student ID, and subsequent columns represent scores from four exams.

txt

```txt
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

The following text shows how to use the Split method to convert each line of text into an array. Each array element represents a column. Finally, the text in each column is converted to its numeric representation.

```C#
public class SumColumns
{
    public static void SumCSVColumns(string fileName)
    {
        string[] lines = File.ReadAllLines(fileName);

        // Specifies the column to compute.
        int exam = 3;

        // Spreadsheet format:
        // Student ID    Exam#1  Exam#2  Exam#3  Exam#4
        // 111,          97,     92,     81,     60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
    {
        Console.WriteLine("Single Column Query:");

        // Parameter examNum specifies the column to
        // run the calculations on. This value could be
        // passed in dynamically at run time.

        // Variable columnQuery is an IEnumerable<int>.
        // The following query performs two steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) convert the element at position examNum to an int
        //    and select it.
        var columnQuery = from line in strs
                          let elements = line.Split(',')
                          select Convert.ToInt32(elements[examNum]);

        // Execute the query and cache the results to improve
        // performance. This is helpful only with very large files.
        var results = columnQuery.ToList();

        // Perform aggregate calculations Average, Max, and
        // Min on the column specified by examNum.
        double average = results.Average();
        int max = results.Max();
        int min = results.Min();
        Console.WriteLine($"Exam #{examNum}: Average:{average:##.##}
High Score:{max} Low Score:{min}");
    }

    static void MultiColumns(IEnumerable<string> strs)
    {
        Console.WriteLine("Multi Column Query:");

        // Create a query, multiColQuery. Explicit typing is used
        // to make clear that, when executed, multiColQuery produces
        // nested sequences. However, you get the same results by
        // using 'var'.

        // The multiColQuery query performs the following steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) use Skip to skip the "Student ID" column, and store the
        //    rest of the row in scores.
        // 3) convert each score in the current row from a string to
        //    an int, and select that entire sequence as one row
        //    in the results.
        var multiColQuery = from line in strs
                            let elements = line.Split(',')
                            let scores = elements.Skip(1)
                            select (from str in scores
                                    select Convert.ToInt32(str));

        // Execute the query and cache the results to improve
        // performance.
        // ToArray could be used instead of ToList.
        var results = multiColQuery.ToList();

        // Find out how many columns you have in results.
        int columnCount = results[0].Count();

        // Perform aggregate calculations Average, Max, and
        // Min on each column.
        // Perform one iteration of the loop for each column
        // of scores.
        // You can use a for loop instead of a foreach loop
        // because you already executed the multiColQuery
        // query by calling ToList.
        for (int column = 0; column < columnCount; column++)
        {
            var results2 = from row in results
                           select row.ElementAt(column);
            double average = results2.Average();
            int max = results2.Max();
            int min = results2.Min();

            // Add one to column because the first exam is Exam #1,
            // not Exam #0.
            Console.WriteLine($"Exam #{column + 1} Average:
{average:##.##} High Score: {max} Low Score: {min}");
```

```
        }
    }
}
/* Output:
    Single Column Query:
    Exam #4: Average:76.92 High Score:94 Low Score:39

    Multi Column Query:
    Exam #1 Average: 86.08 High Score: 99 Low Score: 35
    Exam #2 Average: 86.42 High Score: 94 Low Score: 72
    Exam #3 Average: 84.75 High Score: 91 Low Score: 65
    Exam #4 Average: 76.92 High Score: 94 Low Score: 39
 */
```

If your file is a tab-separated file, just update the argument in the `Split` method to `\t`.