# Parsing ETL traces yourself, part 3: The TraceProcessor

devblogs.microsoft.com/oldnewthing/20201211-00

December 11, 2020

Raymond Chen

When you're processing ETL traces, you sometimes need to bring out the big guns. And in this case, the big guns are called .NET TraceProcessing. This is the same class used by the Windows Performance Analyzer to power its data analysis,[1] so you are getting exactly what WPA has.[2]

The basic idea behind using the `TraceProcessor` is to operate in four steps:

1. Create a `TraceProcessor` that parses your ETL file.
2. Tell the `TraceProcessor` which data tables you intend to use.
3. Process the trace. This populates the tables you selected.
4. Analyze the data in the tables.

Here's a simplified version of an actual program which I use to analyze disk and file traces. It builds a comma-separated-values table that categories disk and file I/O based on the issuing process, the type of I/O, and the file being accessed. For each file, it totals up the number of bytes as well as the time spent performing the I/O. I use this to compare I/O across builds looking for patterns and regressions. (Be careful how you interpret the data, however. See my discussion a few weeks ago on analyzing disk and file I/O performance with ETW traces.)

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Microsoft.Windows.EventTracing;
using Microsoft.Windows.EventTracing.Disk;
using Microsoft.Windows.EventTracing.Processes;

class FileOperation
{
    public IProcess IssuingProcess;
    public string Operation;
    public string Path;
    public long Size;
    public decimal Duration;
}
```

We start with some `using` statements to reduce typing, and then declare a class that we will use to record file operations. In the TraceProcessor world, a process is represented by an `IProcess`.

```
class Program
{
    public static void Main(string[] args)
    {
        var etlFileName = args[0];

        var diskTrace = Path.GetFileNameWithoutExtension(etlFileName) + "-disk.csv";
        var fileTrace = Path.GetFileNameWithoutExtension(etlFileName) + "-file.csv";

        using (ITraceProcessor trace = TraceProcessor.Create(etlFileName))
        {
            var pendingDisk = trace.UseDiskIOData();
            var pendingFile = trace.UseFileIOData();

            trace.Process();

            ProcessDisk(pendingDisk.Result, diskTrace);
            ProcessFile(pendingFile.Result, fileTrace);
        }
    }
```

We take a single file name on the command line and use that to derive the names of the output files, one for disk statistics, and one for file statistics.

And then we go through the standard four-step process:

1. Create a `TraceProcessor` object to parse the file.
2. Say that we want the DiskIO and FileIO data.
3. Process the trace, which fills the tables.
4. Process the tables.

In our case, the two tables are processed separately. In more complex scenarios, you may want to correlate data across tables.

Okay, so let's process the disk trace.

```
static void ProcessDisk(IDiskActivityDataSource data, string outputFileName)
{
    long totalBytes = 0;
    using (var writer = new StreamWriter(outputFileName))
    {

writer.WriteLine($"ImageName,Pid,Type,FileName,Size,ServiceDuration,IODuration");

        var query = data.Activity
            .Where(a => (a.Source == DiskIOSource.Original || a.Source ==
DiskIOSource.Prefetch) &&
                  a.IOType != DiskIOType.Flush)
            .GroupBy(a => new { a.IssuingProcess, a.IOType, a.Path })
            .OrderBy(g => g.Key.IssuingProcess.Id)
            .ThenBy(g => g.Key.Path)
            .ThenBy(g => g.Key.IOType);
        foreach (var g in query)
        {
            var k = g.Key;
            var p = k.IssuingProcess;
            var bytes = g.Sum(a => a.Size.Bytes);
            totalBytes += bytes;
            var serviceDuration = g.Sum(a =>
a.DiskServiceDuration.TotalMilliseconds);
            var ioDuration = g.Sum(a => a.IODuration.TotalMilliseconds);
            writer.WriteLine($"{p.ImageName},{p.Id},{k.IOType},\"{k.Path}\",{bytes},
{serviceDuration},{ioDuration}");
        }
    }
    System.Console.WriteLine($"Disk bytes: {totalBytes}");
}
```

We start by creating a `totalBytes` variable which accumulates the total number of bytes read from or written to disk.

The first real work we do is to create the output file and print our CSV header.

Next, we walk through the data in the table. We look only at Original and Prefetch sources, ignoring VolSnap. We also ignore Flush operations, leaving just Read and Write. We want to accumulate values by process, type, and file name, so we use that as our grouping key. To make the output look less ragged, we sort by process ID, path, and I/O type. (The real program also uses the process command line to distinguish multiple runs of the same executable, but I removed that from this version for simplicity.)

For each group, we tally up the size of the I/O in bytes, as well as the service duration and I/O duration, and print one row of the output.

And at the end, we print a summary to the screen, just for fun.

The code here looks pretty anticlimactic. The real work is done by the TraceProcessor to gather all the data for us, and by LINQ to organize the data via grouping and sorting into a form we want.

Getting the file I/O data follows a similar pattern, but it's a little more annoying because we need to merge the Read and Write data ourselves.

```
static void ProcessFile(IFileActivityDataSource data, string outputFileName)
{
    var operations = new List<FileOperation>();

    var readQuery = data.ReadFileActivity
        .GroupBy(a => new { a.IssuingProcess, a.Path });
    foreach (var g in readQuery) {
        var k = g.Key;
        operations.Add(new FileOperation {
            IssuingProcess = k.IssuingProcess,
            Operation = "Read",
            Path = k.Path,
            Size = g.Sum(a => a.ActualSize.Bytes);
            Duration = g.Sum(a => a.StopTime.TotalMilliseconds -
a.StartTime.TotalMilliseconds)
        });
    }

    var writeQuery = data.WriteFileActivity
        .GroupBy(a => new { a.IssuingProcess, a.Path });
    foreach (var g in writeQuery) {
        var k = g.Key;
        operations.Add(new FileOperation {
            IssuingProcess = k.IssuingProcess,
            Operation = "Write",
            Path = k.Path,
            Size = g.Sum(a => a.ActualSize.Bytes);
            Duration = g.Sum(a => a.StopTime.TotalMilliseconds -
a.StartTime.TotalMilliseconds)
        });
    }

    long totalBytes = 0;
    using (var writer = new StreamWriter(outputFileName)) {

writer.WriteLine($"ImageName,Pid,Type,FileName,Size,ServiceDuration,IODuration");
        var query = operations
            .OrderBy(o => o.IssuingProcess.Id)
            .ThenBy(o => o.Path)
            .ThenBy(o => o.Operation);
        foreach (var o in query) {
            var p = o.IssuingProcess;
            totalBytes += o.Size;
            writer.WriteLine($"{p.ImageName},{p.Id},{o.Operation},\"{o.Path}\",
{o.Size},{o.Duration}");
        }
    }
    System.Console.WriteLine($"File bytes: {totalBytes}");
}
```

We start by creating a list of `FileOperation` objects that represent a file operation (either Read or Write).

The first loop collects the Read data. It groups them by process and path, and creates one `FileOperation` for each such group, accumulating the number of bytes read ( `ActualSize` , as opposed to the number of bytes requested `RequestedSize` ) and also accumulating the time spent in the file I/O. These entries are recorded with the operation `Read` .

The second loop does exactly the same thing, just with the Write data.

The final loop generates the output: We sort the operations by process ID, path, and operation, and print one line of output for each result.

And for fun, we print a grand total.

[1] It's also what powers Bruce Dawson's analysis scripts.

[2] Internally, these classes are known as the *WPA Data Layer*, emphasizing that this is the same data parser that powers WPA.

Raymond Chen

**Follow**