How to Transfer Millions of Files Using PowerShell

Devolutions

THIS ARTICLE WRITTEN BY THOMAS HIGGINS, DEVOPS ENGINEER, IS PART OF OUR GUEST BLOG SERIES. PLEASE CONTACT US IF YOU WOULD LIKE TO BE FEATURED ON OUR BLOG.

I was tasked with moving and archiving approximately 16.5 million files from one location, while ingesting around 4,000 new files per hour.

This job was for a single client, and my company has multiple customers with similar needs. Keeping the data organized and available has been a challenge for over a year, and we have been working toward a solution for some time now.

Background

My company takes raw data from the medical industry and converts those metrics into various types of measures, allowing the customer to gauge their performance and setup strategies for improvement. It also allows for several other options, but these core operations are what cause us to take in so many records.

Each record is typically tiny – no more than 100kb, and typically under 50kb in size. Individually, the files are not an issue, but in aggregate, the files cannot be listed before the default timeout is hit for such processes. This is regardless of OS, as the ingest point is a Linux server, and the archive server is Windows-based. In both cases, the files could not be listed using normal methods.

The Problem

This lack of listing made it difficult to determine a logical way to break down the data into usable chunks.

There was no consistent naming convention to pull from, and even if there was, there was no way to determine what that might be. It was estimated that there were tens of thousands of files per day, but prior to attacking the problem, that was just an educated guess – there was no way to know beforehand.

The Solution

Several potential solutions presented themselves, most of which involved scripting. For reasons unrelated to the direct problem, we chose to mount a separate drive to a new archive server that we had set up for other files. We would then migrate them to that new server using a script written to filter all files into separate folders based on the one piece of data we did know was consistent for all files – the last write date (modified date).

This was moving from Windows to Windows, so PowerShell was the go-to scripting language. (I might even consider it for accessing Linux if I were pulling from Linux to Windows, but that is another story.) Getting started with the script was not exactly easy. It took time and several false starts to figure out how to get the data I wanted in the necessary format.

Googling around for solutions led me down some bad paths for my particular use case, which is not uncommon when writing new scripts. It is unfortunately rare that a script you find will do exactly what you want, and even rarer that it works in your environment after changing only the path names. For example, one bad road I went down was using the C# library for optimizing the read in speed over Get-ChildItem.

While it did speed up the read times, I was unable to treat the objects in the same manner, and my lack of understanding around C# code prevented me from working through this to get the files read in faster. More on speed of execution later though. The key, for me, was when I stumbled across the LastWriteTime file property. When I found this and learned it was the property I was trying to filter with, the rest came easily

Once the basic script was finished it looked something like this:

```
$MyFile = Get-ChildItem -Path E:

foreach ($flob in $MyFile) {
    $fldate = Get-Date -Format d ($flob.lastwritetime)
    $modday = Get-Date -Format dd $fldate
    $modmonth = Get-Date -Format MM $fldate
    $folderloc = "2015" + $modmonth + $modday + "_Data"
    $TARGET = "G:\$folderloc"

    if (!(Test-Path -Path $TARGET)) {
        New-Item -ItemType directory -Path $TARGET
    }
}

cp E:\$flob $TARGET  #Using cp instead of move for testing - don't wa
```

The problem was, it took a LONG TIME. In fact, I thought the entire system froze. So to verify, I added a Write-Host line to the end of the file. This proved it took time to read in the file (minutes, in fact), but that it actually was running. So after a little more searching, I learned to tweak it to just the first x number of files using the select statement. I also learned an easy way to provide times to feed into an optimization loop (though I didn't know at the time that is what I wanted to create). So the final code at this point looked like:

```
$starttime = (Get-Date)
$MyFile = Get-ChildItem -Path E:\ | select -First 300000
$midtime = (Get-Date)

foreach ($flob in $MyFile) {
    $fldate = Get-Date -Format d ($flob.lastwritetime)
    $modday = Get-Date -Format dd $fldate
    $modmonth = Get-Date -Format MM $fldate
    $folderloc = "2015" + $modmonth + $modday + "_Data"
    $TARGET = "G:\$folderloc"

    if (!(Test-Path -Path $TARGET)) {
        New-Item -ItemType directory -Path $TARGET
    }

    mv E:\$flob $TARGET  #Use cp instead of move for testing - don't
    Write-Host "Move Complete"
}

$endtime = (Get-Date)
$splittime = $midtime-$starttime
$writesplit = $endtime-$midtime
$totaltime = $endtime-$starttime
Write-Host "Read in split time:  $splittime"
Write-Host "Write out split time: $writesplit"
Write-Host "Total elapsed time: $totaltime"
```

I strongly recommend any long running process to include those times so you can get actual runtime values at each major point of the code as well as an overall time. For me, that was huge. I started at 100,000 files, and moved my way up to 3 million files in a move. I kept track of the times and found very similar results from 100,000 to 300,000 files with results slowing as it grew to 1 million files at a run. Still, the times were generally acceptable. At 100,000 files, the transfer took about 35 minutes. At 300,000 about 80 minutes. At 500,000, it took exactly 2.5 hours. At 1 million files, 5.5 hours.

As a side note, I found it difficult to track how far along the script was, so I added a counter to the script and put it in the Write-Host line to tell me which file was moved last.
Figuring the time drop-off would be consistent, I let the script run overnight (around 16 hours). I figured I had 16 hours to kill and 2 million files would likely take only 12 hours. So I thought I'd shoot for 3 million, assuming it would finish about 3 hours into my work. (Around 19 hours expected runtime.) The script ran successfully throughout the night, but the time was far longer than expected. When I came in the next morning, after over 16 hours of runtime, only 1.7 million files had moved. At this point, I decided to break the script and start optimizing it. I was going to use that as a feedback loop, but decided the coding to do so for variable-sized files like this wasn't worth the effort. I basically "eyeballed" it using the times listed above and decided somewhere around 300,000 files was the sweet spot for this particular work.

So then, all that was left to do was to loop through the program until all the files were moved. To make time tracking easier, you can output to file or use the tee command to output to both a file and console (my recommendation). The end code looks like this:

```
for ($i=0; $i -lt 40; $i++) {
    $starttime = (Get-Date)
    $MyFile = Get-ChildItem -Path E:\ | select -First 300000
    $midtime = (Get-Date)
    $fileno = 0

    foreach ($flob in $MyFile) {
        $fldate = Get-Date -Format d ($flob.lastwritetime)
        $modday = Get-Date -Format dd $fldate
        $modmonth = Get-Date -Format MM $fldate
        $folderloc = "2015" + $modmonth + $modday + "_Data"
        $TARGET = "G:\$folderloc"

        if (!(Test-Path -Path $TARGET)) {
            New-Item -ItemType directory -Path $TARGET
        }

        $fileno++
        mv E:\$flob $TARGET  #Use cp instead of move for testing - d
        Write-Host "Run $i, File $fileno Move Complete"
    }

    $endtime = (Get-Date)
    $splittime = $midtime-$starttime
    $writesplit = $endtime-$midtime
    $totaltime = $endtime-$starttime
    Write-Output "************************************"  | Tee-Obje
    Write-Output "Run $i Times"  | Tee-Object -FilePath c:\outfile.t
    Write-Output "-------------------------------------------------
    Write-Output "Read in split time:  $splittime"  | Tee-Object -Fi
    Write-Output "Write out split time: $writesplit" | Tee-Object -F
    Write-Output "Total elapsed time: $totaltime" | Tee-Object -File
}
```

Using these few lines of code, I was able to move over 11 million files filtered into one of about 200 folders based on when the file was last modified in just over 30 hours (without having to create the directory structure before it started). As mentioned earlier, this is likely not the most optimized code.

Reading in the files, for example, can be optimized using C# libraries. (Actually, the whole process probably could be.) Using streams, I could pick off files as they were listed instead of requiring they be read into memory first. The problem with these two is I don't know how to do it, and the research and time required to make this happen was not worth the investment. I needed a solution right away, and this solution worked far faster than it would have taken me to optimize it for the fastest performance possible.

Are there better approaches out there? Maybe. Are there faster ones? Most likely. Are they readily available for free? Not that I could find. Hopefully you are better at finding them than I am; but if not, I can tell you this worked for me.