CHAPTER 13

PowerShell Workflows

In this chapter, we will discuss creating and executing workflows in PowerShell. Workflow Foundation is a .NET framework for defining complex processes that can span a long period of time, can be stopped and restarted where they left off, support the parallel execution of tasks, and can be packaged into reusable components. Until PowerShell 3.0, workflows could only be created in Visual Studio using C# or VB.Net. PowerShell 3.0 added support for the creation and execution of workflows. Since workflows can be executed either locally or on remote machines, we'll start by discussing PowerShell remote execution. Workflows are suspended and restarted using PowerShell job cmdlets, which were covered in the previous chapter. Here, we'll see how they apply to workflows. We will discuss using the commands `parallel`, `foreach -parallel`, `sequence`, and `inlinescript` to control workflow execution. Then, we will delve into using the cmdlets `Checkpoint-Workflow`, `Suspend-Workflow`, and `Resume-Job` to pause and resume workflow execution. Workflows can work on a single object at a time or on a collection. For example, John Doe applies for insurance coverage, and a series of steps occur until he is either insured or rejected. This pattern of usage is common to ASP.Net applications. A workflow could be used to load a series of external files into staging tables, validate the data, and, finally, load them into a data warehouse. We will cover two data-centric use cases for workflows. One is a typical data-warehouse load scenario. The other uses a workflow to speed up the extraction of data from flat files using parallel processing, similar to Hadoop.

PowerShell Remote Execution

PowerShell can execute commands on other computers either immediately or as a background job. It can even create new PowerShell sessions on other computers. If remoting is not enabled on the target machine, you must enable it. Then, you will be able to send commands to the machine across the network. You can enable remoting via a group policy or by logging into the machine and configuring it for remote execution. See the link that follows for information on handling this via a group policy:

`http://www.grouppolicy.biz/2014/05/enable-winrm-via-group-policy/`

Let's discuss manually setting up remote execution.

Configuring Remote Execution

To enable a single machine for PowerShell remoting, perform the following steps:

1. Log on to the target machine.
2. Start PowerShell as Administrator, i.e., right mouse click on the PowerShell program in Windows Accessories and select '*Run as Administrator*' as shown in Figure 13-1.

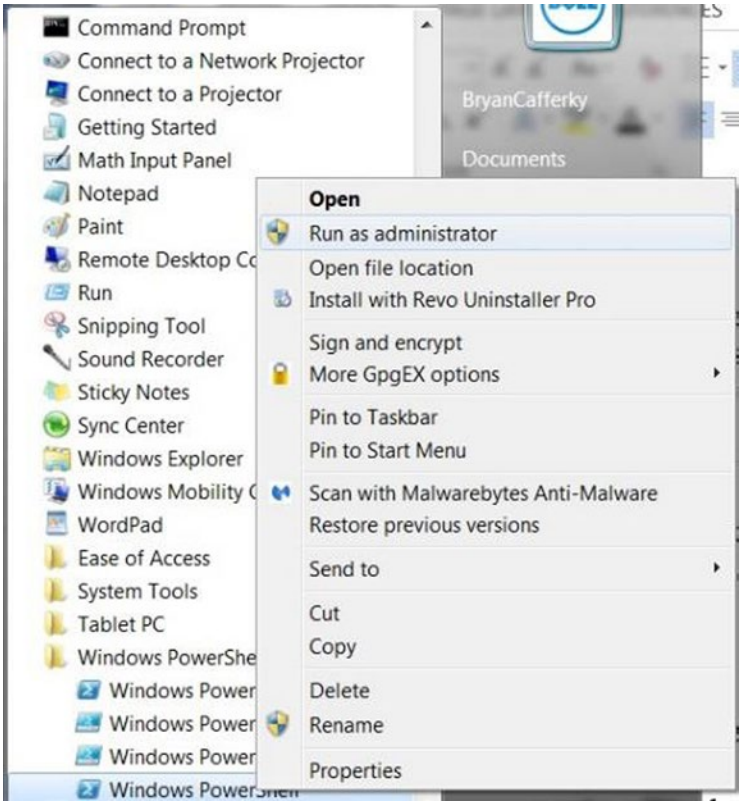

Figure 13-1. *Starting PowerShell as Administrator*

3. Enable remoting by entering the statement seen here:

```
Enable-PSRemoting -Force
```

4. The messages that follow will be seen if remoting is already enabled. Otherwise it will be enabled.

```
WinRM is already set up to receive requests on this computer.
WinRM is already set up for remote management on this computer
```

5. Test the connection by entering the statement that follows, replacing `ComputerName` with the name of the computer:

```
Test-WsMan ComputerName
```

Messages should be seen displayed to the console like the ones here:

```
wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

That's all there is to it. You can now execute PowerShell commands to the machine you just configured from other machines. Let's give it a try. From a different machine than the one you just configured, enter the statement that follows, replacing *ComputerName* with the computer you just configured and *UserName* with your network user name:

```
Invoke-Command -ComputerName ComputerName -ScriptBlock { Get-Process } -credential UserName
```

You should see the list of processes running on the remote machine displayed on your console. We'll review what `Invoke-Command` is in full detail a bit later. The point here was to get remoting set up.

Using Remote Execution with Invoke-Command

The cmdlet `Invoke-Command` executes a script file or script block either locally or on remote computers. Let's look at a simple example:

```
Invoke-Command –ScriptBlock { Get-Culture }
```

This statement just runs whatever code is in the script block, so the local culture setting should display. If we want to run the code on a remote machine, we can use the `ComputerName` parameter, as shown here:

```
Invoke-Command –ScriptBlock { Get-Process} –ComputerName machine1, machine2
```

This statement will get a list of running processes on `machine1` and `machine2`. The code executes on those machines but sends the results back to the local session. The `AsJob` parameter, which can only be used with remote execution, submits the code as a background job, as shown next. The job will show in the local job queue, but will run on the remote machines.

```
Invoke-Command –ScriptBlock { Get-Process } –ComputerName machine1 –AsJob –JobName MyJob
```

The job information displays to the console, as shown here:

```
Id     Name     PSJobTypeName   State         HasMoreData     Location             Command
--     ----     -------------   -----         -----------     --------             -------
5      MyJob    RemoteJob       Running       True            machine1             Get-Culture
```

Notice the job name is `MyJob`, because we used the `JobName` parameter.

In these statements, PowerShell uses the credential of the current session to authenticate the permissions. However, we can use the `Credential` parameter to specify the user ID and password to be used for authentication, as shown here:

```
Invoke-Command –ScriptBlock { Get-Process } –ComputerName machine1 –Credential myuserid
```

In this statement, we are running the `Get-Process` cmdlet on `machine1` using the logon `myuserid`. Using the credential parameter and passing just the user ID causes PowerShell to prompt you to enter the password. If we wanted to run this from within a scheduled job, we would need to include the password, which the `Credential` parameter does not support directly. We can include the password if we create a credential object first and pass that as the `Credential` parameter. However, storing clear-text passwords is not a best practice, so let's encrypt and store the password in a file from which we can retrieve it when needed. Recall that we have a function in the `umd_database` module to help us do this, which we can execute as shown here:

```
Import-Module umd_database

Save-UdfEncryptedCredential
```

When we run these statements, we are prompted to enter the password with the dialog box shown in Figure 13-2.

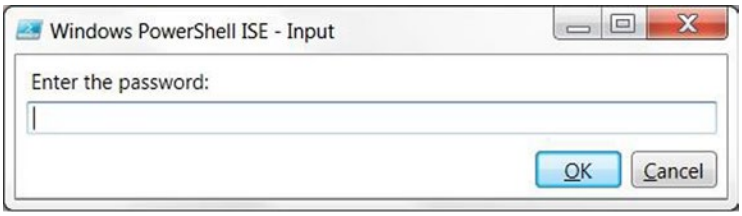

Figure 13-2. *Password prompt from function Save-UdfEncryptedCredentials*

We enter the password and click OK. Now, we are asked to specify where we want the file written, seen in the dialog box shown in Figure 13-3.

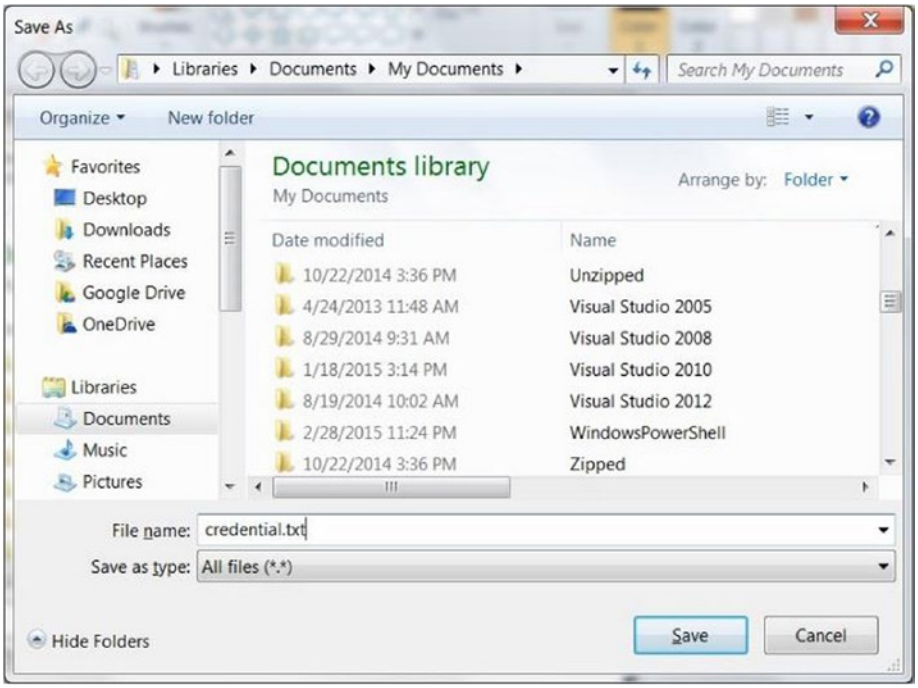

Figure 13-3. *Encrypt credential Save As dialog box*

Since the Save As dialog box defaults to the Documents folder of the current user, just enter the name of the file and click the Save button. In Figure 13-3, the file is being saved as `credential.txt`. To see the code for `Save-UdfEncryptedCredentials` with a full explanation, refer back to Chapter 7, "Accessing SQL Server," under the heading "Encrypting and Saving the Password to a File."

Now that we have the password saved to a file, let's try the `Invoke-Command` cmdlet using the `Credential` object, as shown in the code in Listing 13-1.

Listing 13-1. Using a Credential object

```
$password = Get-Content ($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\credential.txt" ) |
            Convertto-Securestring -AsPlainText -Force

$credential = New-Object System.Management.Automation.PSCredential ("someuser", $password )

Invoke-Command -ScriptBlock { Get-Culture } -Credential $credential
```

The first line in Listing 13-1 loads the encrypted password, using `Get-Content`, from the file we created. The path with file name is dynamically created using the environment variables `$env:HOMEDRIVE` and `$env:HOMEPATH` concatenated with the string `"\Documents\credential.txt"`. The result is piped into the `Convertto-Securestring` cmdlet, passing the parameters `AsPlainText` so it is a readable string and `Force` to tell PowerShell we accept the risk of outputting the string in plain text. Although there may be some risk to converting the string to text, because it is a `securestring`, we cannot see its contents. The `credential` object requires this type of format for a password. The second statement creates a .NET `credential` object with the user ID and password. The credential object is stored in variable `$credential`. Finally, the last line calls `Invoke-Command`, passing a `scriptblock` and the `credential` object we just created. Be aware that there is a

limitation with Invoke-Command. It will not accept a credential object unless PowerShell remoting has been enabled, even when running the script on the local machine. If it is not enabled, we get the error message seen here:

```
Invoke-Command : Parameter set cannot be resolved using the specified named parameters.
At line:2 char:1
+ Invoke-Command -ScriptBlock { Get-Culture } -Credential $cred
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Invoke-Command],
ParameterBindingException
    + FullyQualifiedErrorId : AmbiguousParameterSet,Microsoft.PowerShell.Commands.
InvokeCommandCommand
```

Even though we are not trying to make a remote call, the credential object is rejected with the message just shown. The link that follows explains this in more detail:

https://connect.microsoft.com/PowerShell/feedback/details/676872/invoke-command-parameter-bug-parameter-set-cannot-be-resolved

Workflows

Workflows provide a lot of power and flexibility. They are designed to support processes that have a common starting point, go through a series of stages, and have an end point. Each step in a workflow is often associated with a status. For example, an insurance claim may start with the claim submission and a status of 'Initiated'. This is followed by an insurance representative assigning the claim to an adjuster and changing the status to 'Assigned'. The adjuster reviews the claim and approves or rejects it. If approved, an award amount is assigned and a check is issued to the insured. Whether approved or rejected, the insured is notified. At the end of the process the status of the claim is either 'Rejected' or 'Paid'.

Features of workflows that make them attractive to use in development include:

- Can run processes that span long periods of time, even if the machine is rebooted
- Can be stopped and started where they left off
- Can run tasks in parallel on the local machine or across multiple machines
- Can accept parameters
- Can be nested

PowerShell makes implementing workflows easy, but it can be confusing. The workflow keyword is used to define a workflow in a way similar to defining a function, but a workflow is actually quite different. This is because the cmdlets in a workflow are translated into the Workflow Foundation language before being submitted for processing. PowerShell is not running the code—the Workflow Foundation is. Consider the code in Listing 13-2, which creates and then executes a workflow called simple.

Listing 13-2. A simple workflow example

```
workflow simple ([string] $myparm)
{
   "Parameter is $myparm"

   Get-Date
```

```
   "Some activity"

   "Third activity"
}

simple "test"
```

We should see the following output:

```
Parameter is test

Sunday, May 17, 2015 4:03:07 PM
Some activity
Third activity
```

Judging from the code in Listing 13-2, it appears that replacing the word `function` with `workflow` is the only difference between writing a `workflow` versus a `function`. However, this is not true. In an effort to make it easy for PowerShell developers to migrate to workflows, Microsoft added a cmdlet translator to PowerShell that is invoked via the workflow keyword. Anything contained in the workflow code block is submitted to a translator, which converts the code into workflow code and submits it to the workflow engine to be processed. Let's try the code in Listing 13-3, which breaks the workflow engine, to see the limitations.

Listing 13-3. A simple workflow with a bug

```
workflow simplebroken ([string] $myparm)
{
   Write-Host "Parameter is $myparm"

   $object = New-Object PSObject

   $object | Add-Member -MemberType NoteProperty -Name MyProperty -Value 'something'

   $object.MyProperty
}

simplebroken 'test'
```

The code in Listing 13-3 does not work. In fact, we can't even get the workflow definition to be accepted without errors. The first problem is that `Write-Host` is not one of the cmdlets supported by the workflow translator, because workflows are not supposed to be interactive. Also, cmdlets that create and change objects—i.e., `New-Object` and `Add-Member`—are not supported by workflows. Just to prove the code is valid for a function, let's try the same code but change the `workflow` keyword to `function`, as shown in Listing 13-4.

Listing 13-4. A simple workflow with the bug fixed

```
function simplefunction ([string] $myparm)
{
   Write-Host "Parameter is $myparm"

   $object = New-Object PSObject
```

```
   $object | Add-Member -MemberType NoteProperty -Name MyProperty -Value 'something'

   $object.MyProperty

}

simplefunction 'test'
```

We should get the output seen here:

```
Parameter is test
Something
```

The point of Listing 13-4 is that we are limited to the cmdlets supported by the workflow translator. There *is* a way to get around these issues, and that is to use the `inlinescript` command, which will run a series of PowerShell statements together in one activity. Let's look at the code in Listing 13-5.

Listing 13-5. A simple workflow using inlinescript

```
workflow simpleinline ([string] $myparm)
{
  inlinescript
  {
     Write-Verbose "Parameter is $Using:myparm"

     $object = New-Object PSObject

     $object | Add-Member -MemberType NoteProperty -Name MyProperty -Value 'something'

     $object.MyProperty
  }
}

simpleinline 'test' –Verbose
```

Using the `inlinescript` command solved most of the problem. However, `Write-Host` is considered by PowerShell to be an interactive cmdlet, and workflows are supposed to be non-interactive processes. `Write-Verbose` is supported, though, so we use that instead. However, using `inlinescript` raised another issue. The code in the `inlinescript` block runs in separate memory from the workflow, so it cannot see the workflow parameter `$myparm`. By prefixing the variable with `$using`, as shown in Listing 13-5, the script can see the workflow parameter. We should get the output seen here:

```
VERBOSE: [localhost]:Parameter is test
Something
```

In addition to the workflow command `inlinescript`, three other workflow commands are supported in executing code, which are `parallel`, `sequence`, and `foreach –parallel`. Let's look at these now.

Parallel Execution

Parallel execution means that multiple processes run at the same time. To run activities in parallel, we use the `Parallel` command. Since each statement is a separate activity, each will run independent of each other simultaneously. Let's look at a simple example of this in Listing 13-6.

Listing 13-6. A workflow with parallel execution

```
workflow paralleltest {

 parallel
 {

 for($i=1; $i -le 10000; $i++){ "a" }

 for($i=1; $i -le 10000; $i++){ "b" }

 for($i=1; $i -le 10000; $i++){ "c" }

 for($i=1; $i -le 10000; $i++){ "d" }

 for($i=1; $i -le 10000; $i++){ "e" }

 }
}

paralleltest
```

A sample of the output is seen here:

```
e
a
b
d
c
e
a
b
```

In Listing 13-6, each `for` loop is run as a separate, parallel activity. Based on the output, we can see that the activities are running concurrently.

Recall that in Chapter 11, we developed a set of ETL scripts for Northwind. Since there are no dependencies among any of the files getting loaded, we could use a workflow to load them in parallel. Let's look at the code to do this in Listing 13-7. Note: As with cmdlet names, Microsoft recommends naming workflows using the verb-noun format, with the verb being from the approved list. The noun can be whatever the developer likes. We will use the noun prefix `udw`, which stands for user-defined workflow, so an example of a workflow name would be `Invoke-UdwMyProcess`. Remember, you need to change the configurable settings in the scripts that follow to what is available in your environment in order for the code to work.

Listing 13-7. Running the Northwind ETL as a workflow

```
Import-Module umd_northwind_etl -Force

Clear-Host

workflow Invoke-UdwNorthwindETL
{

 parallel

 {

    Invoke-UdfStateSalesTaxLoad

    Invoke-UdfOrderLoad

    Invoke-udfCustomerLoad

  }

}

Invoke-UdwNorthwindETL
```

When we run the code in Listing 13-7, we can see that the various SQL statements being executed for each load are scrambled in the console, indicating that they are running concurrently. Since these are small loads, using a workflow to load the files in parallel on the local machine actually takes longer than it would when running the loads sequentially without a workflow. Part of the reason for this is that there is some overhead created by PowerShell when it generates the workflow code to send to the workflow engine. Another reason is that we are using the same machine. If our goal is to improve performance, testing is required to determine if workflows help. To maximize performance, processes can be executed on different machines. In the case of the workflow in question, we could rewrite it to run each ETL process on a different computer. The code in Listing 13-8 does this.

Listing 13-8. Running the Northwind ETL as a workflow with parallel activities

```
Import-Module umd_northwind_etl

workflow Invoke-UdwNorthwindETL
{

 parallel

 {

   workflow Invoke-UdwStateSalesTaxLoad
   {
    Invoke-UdfStateSalesTaxLoad
   }
   Invoke-UdwStateSalesTaxLoad -PSComputerName remotepc1
```

```
   workflow Invoke-UfwOrderLoad
   {
    Invoke-UdfOrderLoad
   }
   Invoke-UfwOrderLoad  -PSComputerName remotepc2
  }

}

Invoke-UdwNorthwindETL
```

By using the `PSComputerName` parameter when we execute a workflow, the workflow runs on the specified machine. In Listing 13-8, we define the state sales tax load and order load as sub workflows. Then we just call them, specifying the computer on which they should run. For this to work, remote execution must be set up on the target machines, as explained at the beginning of this chapter. In an ETL case like this, although we are spreading the work out, the target of all three ETL processes is the same SQL Server instance, so actual performance improvements may depend on the database server workload. Also, to actually run the code on other machines, we would need to copy the files and functions to those machines as well.

Sequence

To run activities in a specific order we use the `Sequence` command. This instructs the workflow engine to complete each activity before starting the next one. Let's look at a simple example of this in Listing 13-9.

Listing 13-9. A simple workflow using sequence

```
workflow Invoke-UdwSimpleSequence ([string] $myparm)
{
  sequence
  {
    "First"
    "Second"
    "Third"
  }
}

Invoke-UdwSimpleSequence
```

We should see the following output on the console:

```
First
Second
Third
```

In Listing 13-9, each activity will be executed and completed before the subsequent activity begins. In the code, we just write to the console. Note: For some reason, when we write to the console without explicitly stating the `Write-Host` cmdlet, it works.

Foreach –Parallel

Sometimes it is useful to start an activity by iterating over a collection. An example of this is shown in Listing 13-10.

Listing 13-10. A simple workflow using ForEach -Parallel

```
workflow Invoke-UdwForeachparallel
{

$collection = "one","two","three"

  foreach -parallel ($item in $collection)
  {
   "Length is: " + $item.Length
  }
}

Invoke-UdwForeachparallel
```

We should see the following output:

```
Length is: 5
Length is: 3
Length is: 3
```

In the workflow in Listing 13-10, the `foreach –parallel` command iterates over the array `$collection`. For each item in the array, an activity is executed in parallel. Interestingly, if we want to run activities across multiple computers, we don't need to use the `foreach –parallel` command. PowerShell automatically runs tasks in parallel on any computers passed via the workflow parameter `PSComputerName`. Let's look at the code that follows:

```
workflow Invoke-UdwRunRemote ($computerlist)
{
     Get-Process -PSComputerName $computerlist
}

Invoke-UdwRunRemote -computerlist ("remote1", "remote2")
```

In this workflow, we have the parameter defined as `$computerlist`, which is meant to receive a list of computer names. The cmdlet `Get-Process` uses the `PSComputerName` to run the cmdlet on the specified remote machines in parallel. Note: We could just as easily define an `inlinescript` containing complex tasks so as to run on each machine. The point here is that using workflows with remote parallel execution provides a powerful means to improve performance by scaling out.

Of course, the idea is to use these workflow commands together to meet our requirements. We can nest them or arrange them to fit our needs. Consider Listing 13-11, which uses all three code-execution commands.

Listing 13-11. Putting it all together; using all three workflow commands

```
workflow Invoke-UdwWorkflowCommand {

 sequence
 {
  "1"
  "2"
  "3"
 }

 parallel
 {

   for($i=1; $i -le 100; $i++){ "a" }

   for($i=1; $i -le 100; $i++){ "b" }

   for($i=1; $i -le 100; $i++){ "c" }

 }

  $collection = "one","two","three"

  foreach -parallel ($item in $collection)
  {
   "Length is: " + $item.Length
  }

}

Invoke-UdwWorkflowCommand
```

In Listing 13-11, a `sequence` command first runs three activities sequentially. Then, three `for` loops are executed in parallel. Finally, an array of three strings is created and used by a `foreach -parallel` command to run a separate activity for each array element. The ability to execute code in parallel or sequentially as needed provides good control and allows us to adapt as requirements change.

ETL Workflow

PowerShell workflows offer a lot of uses, but they also pose some challenges. In particular, the automated stopping and resuming of workflows requires special handling. When a workflow is submitted as a job, it runs in the background. If it suspends itself, the only way to restart it is to restart the job. Even if the workflow is submitted using foreground execution, suspending the workflow automatically creates a PowerShell job in the queue. To resume the workflow, we need to know which job to resume. This means we will need to store details about the suspended job.

Let's consider a potential use case. Northwind loads orders and employees to staging tables independent of each other. However, the data warehouse has an Order dimension that uses both order data and related sales employee data. Sometimes the employee data from the human resources system lags behind the arrival of related orders. We need to create a workflow that will load the order data, check the employee staging table for the related employee data, and only load the Order dimension after the employee

data has been loaded. We will use file-system events to start and resume the workflow. The overall flow chart is shown in Figure 13-4.

- Order file created event starts the workflow.
- Load the Order file.
- Are all employees on orders on the Employee table?
- If yes, load the Order dimension.
- If no, notify business and suspend the workflow.
- Employee file created event restarts the workflow.
- Load Employees and load the Order dimension.

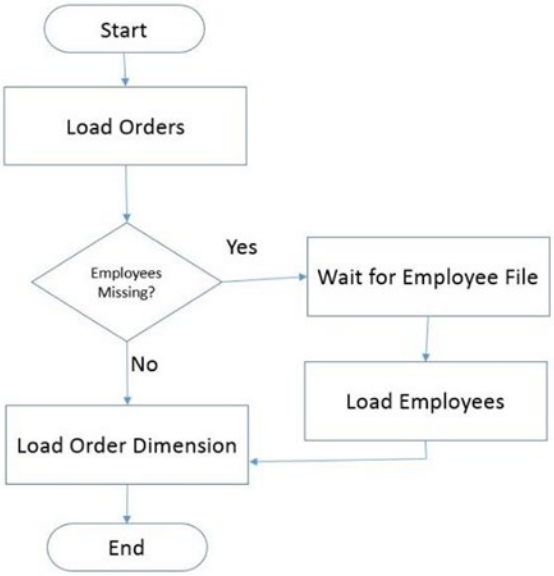

Figure 13-4. *Order load ETL workflow*

In Figure 13-4, we can see that the workflow starts when a file-created event for the Order file is fired. The workflow loads the Order file to a SQL Server staging table. Then, we match the Order rows to the staging Employees table. If there are no unmatched rows, the Order dimension is loaded. If there are orders with no matching employee, we need to suspend the workflow until the Employee file arrives, which we do by registering for a file-created event that waits for the employee data. When the employee data arrives, the event is fired that runs the code to resume the workflow. Note: When we suspend a workflow, we need to persist the data that contains the state of the workflow. This is done with the `Checkpoint-Workflow` command. Then, to suspend the workflow, we use the `Suspend-Workflow` command. To restart a workflow, we do not reference the workflow. When the workflow is suspended, a job is created and stored in the PowerShell job queue. To resume the workflow, we need to use the `Resume-Job` cmdlet.

The Workflow Code

This section has a lot of code, but fear not! We will walk through it step by step. As has been emphasized throughout this book, a good practice is to write encapsulated, reusable functions as much as possible. The code in this section does this extensively. This allows us to read the code at a high level and be able to grasp what it does. However, it also means we will delay covering the underlying functions a bit. The code is layered, so we need to review it by rolling the layers back, starting at the highest level. The functions in this section can be found in the module `umd_workflow`. You will need to modify configurable settings, such as the SQL Server instance name, to match your environment. The module `umd_workflow` contains all the code for all workflows that follow.

Let's look at the workflow code:

```
Import-module umd_northwind_etl
Import-Module umd_etl_functions

[psobject] $global:SqlServer1 = New-Object psobject
New-UdfConnection ([ref]$SqlServer1)

$global:SqlServer1.ConnectionType = 'ADO'
$global:SqlServer1.DatabaseType = 'SqlServer'
$global:SqlServer1.Server = '(local)'
$global:SqlServer1.DatabaseName = 'Development'
$global:SqlServer1.UseCredential = 'N'
$global:SqlServer1.SetAuthenticationType('Integrated')
$global:SqlServer1.BuildConnectionString()

Workflow Invoke-UdwOrderLoad
{
 param([string] $sourceidentifier, [string] $sqlserver, [string] $databasename )

 sequence
 {
    Write-Verbose $sourceidentifier
    Invoke-UdfOrderLoad

    $missing_emps = Get-UdfMissingEmployee | Where-Object -Property Employee_ID -ne $null

    if ($missing_emps.Count -gt 0)
    {
     Write-Verbose "Workflow Being Suspended."

    Send-UdfOrderLoadEmail -subject "ETL Job Order Load - Suspended"  '
    -body "The ETL Job: Order Load has been suspended because the employee
     file has not been loaded. Please send the employee file as soon
     as possible."

     Checkpoint-Workflow
     Suspend-Workflow
     Write-Verbose "Workflow Resumed."
     Invoke-UdfEmployeeLoad
     Send-UdfOrderLoadEmail -subject "ETL Job Order Load"  '
                              -body "The ETL Job: Order Load has ended."
}
 Else
 {
  Write-Verbose "finish"

  Invoke-UdfSQLQuery -sqlserver '(local)' -sqldatabase 'development' '
  -sqlquery "update [dbo].[WorkFlowLog] set Status = 'complete' '
  where WorkflowName = 'OrderLoad' and Status = 'suspended';"
```

```
  Send-UdfOrderLoadEmail -subject "ETL Job Order Load"  '
                          -body "The ETL Job: Order Load has ended."
 }
}
}
```

In this code, we start by importing the ETL modules needed, which we discussed in prior chapters, using the statements copied here:

```
Import-module umd_northwind_etl
Import-Module umd_etl_functions
```

Then, we need to create a connection object, which the code copied here does:

```
[psobject] $global:SqlServer1 = New-Object psobject
New-UdfConnection ([ref]$SqlServer1)

$global:SqlServer1.ConnectionType = 'ADO'
$global:SqlServer1.DatabaseType = 'SqlServer'
$global:SqlServer1.Server = '(local)'
$global:SqlServer1.DatabaseName = 'Development'
$global:SqlServer1.UseCredential = 'N'
$global:SqlServer1.SetAuthenticationType('Integrated')
$global:SqlServer1.BuildConnectionString()
```

In this code, we are using the `New-UdfConnection` function we created in Chapter 7. To do this, the first line creates an empty `PSObject` variable. This is passed into the `New-UdfConnection` function, which attaches the methods and function we'll need in order to connect to a data source. The lines after this just assign the SQL Server database connection properties and finally call the `BuildConnectionString` method so as to generate the connection string. Notice that we are using the `$global` prefix, which defines the variable scope as global so that all code in the session can access it. We are assigning these values directly to make the code simpler to read, but in practice, we would want to use the configuration approach discussed in Chapter 10 to define these values.

Then, we start the workflow definition as shown here:

```
Workflow Invoke-UfwOrderLoad
{
 param([string] $sourceidentifier, [string] $sqlserver, [string] $databasename )
```

In the workflow definition we can see three parameters. `$sourceidentifier` is going to be obtained from the code that gets fired by the create file event, which we will discuss later. `$sqlserver` is the SQL Server instance, and `$databasename` is the database name. Now, let's look at the first few lines of the workflow, copied here:

```
Sequence
{
Write-Verbose $sourceidentifier
Invoke-UdfOrderLoad
```

The first line tells the workflow engine to sequentially run the activities defined in the braces. Due to the dependencies, the statements must be executed in order. The `Write-Verbose` statement simply displays the parameter `$sourceidentifier`. Recall that `Write-Host` is not supported, so we use `Write-Verbose` instead. Then the function `Invoke-UdfOrderLoad` is called. We saw this function in Chapter 11. It will load the orders file into the SQL Server table `dbo.Orders`. It displays a lot of `Write-Verbose` messages that show the SQL statements used to load the data.

The line copied here will test whether there are any orders without a matching employee:

```
$missing_emps = Get-UdfMissingEmployee | Where-Object -Property Employee_ID -ne $null
```

The function `Get-UdfMissingEmployee` will return a record set of any orders with no matching employee row. We will look at the function in detail later. The function results are piped into the `Where-Object` cmdlet. The function `Get-UdfMissingEmployee` displays some informational messages that get returned in the pipe, so we use the `Where-Object` cmdlet to filter the pipe down to the missing employee result set. By filtering on `Employee_ID`, only pipe rows with that property are returned. Now, let's look at the code that will suspend the workflow if necessary:

```
if ($missing_emps.Count -gt 0)
{
    Write-Verbose "Workflow Being Suspended."

    Send-UdfOrderLoadEmail -subject "ETL Job Order Load - Suspended"  '
    -body "The ETL Job: Order Load has been suspended because the employee
     file has not been loaded. Please send the employee file as soon as possible."

    Checkpoint-Workflow
    Suspend-Workflow
    Write-Verbose "Workflow Resumed."
    Invoke-UdfEmployeeLoad
    Send-UdfOrderLoadEmail -subject "ETL Job Order Load"  '
                           -body "The ETL Job: Order Load has ended."
}
```

In this code, the first line tests whether there are any missing employee rows. To be honest, I did not think the workflow engine would let me retain an object from one activity to the next like `$missing_emps`, but for some reason it did. The interaction between PowerShell and the workflow engine can be a bit mysterious. I am guessing this is because Microsoft tried to make the PowerShell interface to workflows work as expected wherever possible. The best way to find out if something will work is to try it. So in this case, if there are missing employees, i.e., if `$missing_emps.Count` is greater than zero, a message that the workflow is being suspended is displayed, an email is sent notifying an interested party, the workflow state is saved using the `Checkpoint-Workflow` statement, and the workflow is suspended with the `Suspend-Workflow` statement. The statements after that are not executed. Rather, the workflow is created as a job in suspended state. The statements after that will not be executed until the job is resumed with the `Resume-Job` cmdlet. When that happens, the workflow will resume at the line `Write-Verbose "Workflow Resumed."`

That's all there is to the workflow. The challenge is knowing which job to resume when the Employee file comes in. Let's look at the code that calls the workflow.

Calling the Workflow

The workflow is surprisingly simple. There's just one problem—when the workflow suspends itself, how do we know which job to resume? The same workflow can be submitted more than once, and the workflow engine tracks the instances separately. However, these instances are not readily visible to PowerShell. Instead, we need to capture the job information when the job is suspended. If we run the workflow in the foreground, there is no way to capture the job information when the workflow suspends itself. The `Suspend-Workflow` statement does not send anything to the pipeline. Instead, we'll submit the workflow as a job at the beginning and capture the job information, as shown in the following code:

```
function Invoke-UdfWorkflow ()
{
<#  Define the mapping... #>
   $mappingset = @()  # Create a collection object.
   $mappingset += (Add-UdfMapping "WorkflowName" "WorkflowName" "'" $false)
   $mappingset += (Add-UdfMapping "ID" "JobID" "" $false)
   $mappingset += (Add-UdfMapping "Name" "JobName" "'" $false)
   $mappingset += (Add-UdfMapping "Location" "Location" "'" $false)
   $mappingset += (Add-UdfMapping "Command" "Command" "'" $false)

    Try
      {
      Invoke-UfwOrderLoad  -sourceidentifier 'Order' -sqlserver $global:SqlServer1.Server '
                           -databasename $global:SqlServer1.Databasename '
                           -AsJob -JobName OrderLoad | Invoke-UdfWorkflowLogTransformation |
                           Select-Object -Property WorkflowName, ID, Name, Location, Command |
                           Invoke-UdfSQLDML -Mapping $mappingset  -DmlOperation "Insert"  '
                           -Destinationtablename "dbo.WorkFlowLog" '
                           -Connection $global:SqlServer1 -Verbose
       }
       Catch
       {
       "Error:  $error"
       }
}
```

The first set of lines creates a mapping set. In Chapter 11, we defined an ETL framework that included the use of mapping sets. This will be used to write the workflow job information to a SQL Server table. We are mapping the workflow name, the job ID, the job name, the job location (i.e., the machine), and the `Workflow` command (i.e., code). Then a `Try` block is executed. At first glance, it may look like there are multiple statements in the `Try` block, but notice the tick mark at the end of each line. Recall that the ' character continues a statement onto the next line. So, the code in the `Try` block is one long statement. This is because we need to keep the pipeline all the way through a number of cmdlets and functions.

The first statement is running the workflow `Invoke-UfwOrderLoad`, passing the values `'Order'` as `sourceidentifier`, the global object property `$global:SqlServer1.Server` as `sqlserver`, and the global object property `$global:SqlServer1.Databasename` as `databasename`. The `AsJob` parameter tells PowerShell to run the workflow as a background job, and the `JobName` assigns the name `OrderLoad` to the job. Since this is our only chance to capture the job information, we pipe the output into a custom function `Invoke-UdfWorkflowLogTransformation`, which transforms the pipeline as needed to write it to the SQL Server table. This is piped into the `Select-Object` cmdlet, which extracts just the properties we need and pipes them into the custom function `Invoke-UdfSQLDML`. This writes the pipeline to the SQL Server table

dbo.WorkFlowLog. Recall that in Chapter 11, we used Invoke-UdfSQLDML to write pipeline data to a SQL Server destination. We are leveraging our previously created functions for ETL so as to capture the workflow job information. If this chain of statements encounters a terminating error, the Catch block is executed.

In this example, we see the function Invoke-UdfWorkflowLogTransformation being called. The function just adds a single, hard-coded property to the pipeline, i.e., OrderLoad, which is the name for the workflow we want stored in the SQL Server table. Let's look at the code now:

```
function  Invoke-UdfWorkflowLogTransformation
{
 [CmdletBinding()]
        param (
                 [Parameter(ValueFromPipeline=$True)]$pipein = "default"
               )
    process
    {

       $pipein | Add-Member -MemberType NoteProperty -Name "WorkflowName"    '
                            -Value "OrderLoad"
       Return $pipein
    }
}
```

As we've seen previously, this function defines the pipeline as a parameter named $pipein with the parameter attribute ValueFromPipeline=$True. Then, we just add a property to it using the Add-Member cmdlet. The Return statement returns the pipeline variable to the caller. Note: Do not use the Passthru parameter with the Add-Member cmdlet or you will get duplicate pipelines.

Starting the workflow is easy. Restarting it is a bit more challenging. However, since we have a log that stored the job details of the job that started the workflow, we can use that to resume the job. Let's look at the code that does this:

```
function Resume-UdfWorkflow ()
{
    Try
    {
      $job = Invoke-UdfSQLQuery -sqlserver $global:SqlServer1.Server '
             -sqldatabase 'development' '
             -sqlquery "select JobID from [dbo].[WorkFlowLog] where WorkflowName = 'OrderLoad'
              and Status = 'suspended';" '

      Resume-Job -id $job.JobID -Wait

      Invoke-UdfSQLQuery -sqlserver $global:SqlServer1.Server '
                         -sqldatabase$global:SqlServer1.DatabaseName '
      -sqlquery "update [dbo].[WorkFlowLog] set Status = 'complete' where WorkflowName =
       'OrderLoad' and Status = 'suspended';" '

    }
    Catch
    {
      "Error:  $error"
    }
}
```

In this function, the first statement in the `Try` block calls the function `Invoke-UdfSQLQuery` to execute a SQL query that will retrieve the job information of the suspended job. The results are stored in `$job`. We'll look at the code for that function a bit later. We use the `JobID` property of `$job` to resume the workflow, i.e., `Resume-Job –id $job.JobID –Wait`. The `Wait` parameter tells PowerShell to stop and wait for the job to finish before proceeding to the next line. After this, we call `Invoke-UdfSQLQuery` to update the status of the row in the workflow log table with a status of `'complete'`.

We've seen the code that starts the workflow, and we've seen the code that resumes the workflow. But how will these be called? We want it to be completely automated, so we'll use the .NET `FileSystemWatcher` to trap when the Order file is created in the expected folder. When this happens, the workflow will be started. We'll use another `FileSystemWatcher` to trap when an Employee file is created in the expected folder in order to resume the workflow.

Let's look at the function to register a trap to the order file-create event:

```
function Register-UdfOrderFileCreateEvent(
[string] $source, [string] $filter, [string]$sourceidentifier)
{
   try
   {
     $filesystemwatcher = New-Object IO.FileSystemWatcher $source, $filter -Property
     @{IncludeSubdirectories = $false; NotifyFilter = [IO.NotifyFilters]'FileName, LastWrite'}
     Register-ObjectEvent $filesystemwatcher Created -SourceIdentifier $sourceidentifier '
                          -Action { Invoke-UdfWorkflow }
    }
   catch
    {
      "Error registering file create event."
    }
}
```

The function takes the parameters `$source`, which is the path to the folder we want to monitor, `$filter`, which is the file-name filter to watch for, and `$sourceidentifier`, which is just a name for the event. The first line in the `try` block creates an instance of the `FileSystemWatcher`, with `$source` as the folder to watch and `$filter`, along with the file name to watch for. The statement after this registers the event, which means it assigns an action to be taken when the event fires. The `Action` parameter specifies the function `Invoke-UdfWorkflow` to start the workflow. What this all means is that when an `Order` file is created in the target folder, the function `Invoke-UdfWorkflow` will execute.

The function to register an event trap for the employee file creation is as follows:

```
function Register-UdfEmployeeFileCreateEvent([string] $source, [string] $filter,
$sourceidentifier)
{
   try
   {
     $filesystemwatcher = New-Object IO.FileSystemWatcher $source, $filter -Property
     @{IncludeSubdirectories = $false; NotifyFilter = [IO.NotifyFilters]'FileName, LastWrite'}

     Register-ObjectEvent $filesystemwatcher Created -SourceIdentifier $sourceidentifier '
                          -Action { Resume-UdfWorkflow }
    }
```

```
   catch
    {
      "Error registring file create event."
    }
}
```

This function takes the parameters `$source`, which is the path to the folder we want to monitor, `$filter`, which is the file-name filter to watch for, and `$sourceidentifier`, which is just a name for the event. One thing to bear in mind about using the File System Watcher to trap events is that the event registration is lost when the PowerShell session ends. The first line in the `try` block creates an instance of the `FileSystemWatcher`, with `$source` as the folder to watch and `$filter`, along with the file name to watch for. The statement after this registers the event, which means it assigns the action to be taken when the event fires. The `Action` parameter specifies the function `Resume-UdfWorkflow`, which will resume the job with the suspended workflow. If the code in the `try` block fails, the `catch` block will display the error message.

Register the Events

Using the functions just reviewed, we register to trap the file-creation events with the code in Listing 13-12.

Listing 13-12. Register the file-create events

```
Import-Module umd_workflow

$global:rootfilepath = $env:HOMEDRIVE + $env:HOMEPATH + '\documents\'

Register-UdfOrderFileCreateEvent $global:rootfilepath "Orders.xml"  -sourceidentifier 'orders'

Register-UdfEmployeeFileCreateEvent $global:rootfilepath "Employees.txt" '
                                    -sourceidentifier 'employees'
```

The first statement in Listing 13-12 defines the path where the files will be created as a global variable named `$global:rootfilepath`. This will be the Documents folder for the current user. Then, the event trap is registered for the creation of the `orders.xml` file. This is followed by the event-trap registration for the Employee file.

Once the code in Listing 13-12 is executed, the function `Invoke-UdfWorkflow` will start when the `Orders.xml` file is created in the user's Documents folder. This starts the workflow, and if the Employee file has not already been loaded, i.e., if there are orders with employee IDs that are not on the Employees table, the workflow will suspend itself. When the `Employees.txt` file is created in the Documents folder, the function `Resume-UdfWorkflow` will be called, which looks up the job ID of the suspended workflow and resumes it right after the point where the workflow was suspended.

Sending Mail

It's often useful to send email notifications to relevant employees telling them when jobs have started, finished, or failed. The function that follows is used to send an email as needed during the order-load process:

```
function Send-UdfOrderLoadEmail ([string]$subject, [string]$body)
{
   $credin = Get-Content ($global:rootfilepath + 'bryan256') | convertto-securestring
   $credential = New-Object System.Management.Automation.PSCredential '
                 ("bryan@msn.com",  $credin)
```

```
   Send-MailMessage -smtpServer smtp.live.com -Credential $credential '
   -from 'bryan@msn.com'  -to 'bryan@msn.com' -subject "$subject" '
   -Body "$body" -Usessl -Port 587
}
```

The first line in the function's body retrieves a previously encrypted password from a file. The line after this uses the retrieved password with the email account to create a credential object. Then, the `Send-MailMessage` cmdlet is executed with parameters for the SMTP server, the credential object, the from email address, the to email address, the subject, and body text. The `Usessl` parameter secures the email, and `Port` specifies what port address is to be used.

Workflow Creation Steps

When we put all the previous code together, we have a complete workflow to support the order load. The workflow steps for creating a new order are as follows:

Workflow Start: An Order file arrives in the Documents folder.

1. The `FileSystemWatcher` Created File event fires, executing the workflow `Invoke-UdwOrderLoad`.
2. `Invoke-UdwOrderLoad` loads the Orders file into the SQL Server Orders table.
3. The workflow runs a query to find any employees on the orders that are missing on the Employees table.
4. If there are any missing employees:
   - The workflow is suspended.
   - Send email notification.
5. If there are no missing employees:
   - Load the table `DimOrderEmployee`.
   - Send email notification.

Workflow Resumes: An employee file arrives in the Documents folder.

1. Load the file into the Employees table.
2. Load the table `DimOrderEmployee`.
3. Send email notification.

In the module, the Send Email steps are commented out because they will fail until you enter valid configuration and credentials for them. Once you have done this, just remove the comment tags. Assuming you have all the modules loaded and the file-creation events have been registered, you can test the workflow. Copy the `Orders.xml` to your Documents folder, which should fire the workflow to load it. Then, copy `Employees.txt` to your Documents folder, which should resume the workflow loading the Employee file and the `DimOrderEmployee` dimension table.

A Poor Man's Hadoop

When I first studied the capabilities of workflows, it occurred to me that they provide much of the functionality that the open source, big-data tool, Hadoop, provides. I thought it would be interesting to see if we could create a Hadoop-like function using PowerShell workflows. Hadoop's most impressive feature is the speed at which it can extract data from large files. Hadoop does this by splitting up the work among multiple machines, called nodes. The data is duplicated across the nodes by something called the Hadoop File System. If we had a number of large files that we needed to extract data from, we could copy one or more files to a number of machines and use remote workflows to have the files processed in parallel, just like Hadoop does. Best of all, it is a lot less work to set up, and everything can be done in PowerShell. The functions in this section can be found in the module `umd_workflow`. You will need to modify configurable settings like email smtp server name to match your environment.

Setting Up the Poor Man's Hadoop Workflow

Imagine we have a number of large text files, such as web logs. We want to search for rows that have a specified string and extract them to another file, which we could load into SQL Server for querying. We will have a workflow run that executes each search for each file in parallel. To try this out, we need some text files to work with. Let's use the code in Listing 13-13 to generate these files.

Listing 13-13. Script to generate test data

```
$filepath = $env:HOMEDRIVE + $env:HOMEPATH + "\Documents\"

function New-UdfFile ([string]$fullfilepath)
{
  for ($i=1; $i -le 1000; $i++)
  {
    Get-ChildItem | Out-File ($fullfilepath) -Append
  }
}
```

The function in Listing 13-13 takes the full path with file name as the parameter `$fullfilepath`. To generate data, a `for` loop iteratively calls `Get-ChildItem` to get a file list, which is piped into `Out-File`, where it is written to the `filepath` parameter. Now, let's call the function twice to create two files to test with, as shown here:

```
New-UdfFile ($filepath + "logfile1.txt")
New-UdfFile ($filepath + "logfile2.txt")
```

These function calls are to run the function `New-UdfFile` so as to create some test files for us. The first call will create a file named `logfile1.txt` and the second call will create a file called `logfile2.txt`. Admittedly, these files are not real log files, but they generate text files well enough to test our search workflow.

To run multiple searches in parallel, we need to use a workflow. The workflow that follows will call our function to search the files:

```
workflow Search-UdwFile ([string]$filepath, $filelist, $searchstring)
{
   foreach -parallel ($file in $filelist)
   {
     Write-Verbose 'Processing searches...'
```

```
    inlinescript
      {
        function Search-UdfSingleFile ([string]$searchstring,[string] $outfilepath)
          {
            begin
            {
             "" > $outfilepath  # Clear out file contents
            }
            process
            {
             if ($_ -like "*$searchstring*")
             {
              $_ >> $outfilepath
             }
            }
          }

        $sourcefile = $using:filepath + $using:file ;
        $outfile = $using:filepath + "out_" + $using:file ;
        Get-Content $sourcefile  |
        Search-UdfSingleFile -searchstring  $using:searchstring -outfilepath $outfile
      }
  }
}
```

The `workflow` keyword tells PowerShell that what follows is a workflow, which means everything will be translated into workflow code and submitted to the workflow engine. Three parameters are supported: `$filepath`, which is the folder where the files are stored, `$filelist`, which is an array of file names to be searched, and `$searchstring`, which is the string to be searched for. We want to run a separate process for each file in the list, i.e., run them in parallel. Therefore, we see the `foreach -parallel` command, which will spawn a separate activity for each file. `Write-Verbose` is just there to confirm the workflow is running.

To perform the search, we need to use a lot of PowerShell code not supported by workflows. Therefore, we use the workflow command `inlinescript` to submit code as an activity. The workflow process is separate from the regular PowerShell process, so we include the search function definition within the inline script. Note: We could also import the function if it were in a module. To maximize speed and minimize memory requirements, the function takes the source file as pipeline input, i.e., we will pipe the source file into the function.

The function takes two parameters: `$searchstring`, which is the string we want to look for, and `$outfilepath`, which is the path and file name where the matching rows will be written to. Recall that a `begin` block will execute once, before the pipeline starts to be received. We are clearing out the output file here by piping a zero-length string to the file. The character > means to overwrite the file. The `process` block will execute for every row in the pipeline. Notice we did not declare the pipeline as a parameter. We don't need to declare the pipeline, but if it is not declared as a parameter, we need to use the system default name, `$_`, to access each iteration. The statement `"if ($_ -like "*$searchstring*")"` will test every row as it comes in for a string matching the `$searchstring` parameter. If a match is found, the row is appended to the output file specified in the `$outfilepath` parameter. Recall that >> means to append to the file. Note: When we want to search a string for a character pattern, we use the `like` operator. If we want to search an array to see if it contains something, we use the `contains` operator.

After the function definition, the script builds the path to the source file using the workflow parameters. For the `inlinescript` to access the workflow parameters, we need to prefix the parameters with `$using`. This is because the `inlinescript` runs in a separate process. Then, the output file path is stored in `$outfile`. Finally `Get-Content` is called to pipe the contents of the source file into the `Search-UdfSingleFile` function. The result file where matches are written is the input file name, prefixed with `out_`.

Calling the Poor Man's Hadoop Workflow

We've discussed the workflow code and embedded search function. Now, let's review the code in Listing 13-14 that will call the workflow.

Listing 13-14. Calling the workflow

```
Import-Module umd_workflow  # To get the functions we will use  below

$filelist = "logcombined.txt", "logcombined2.txt", "logcombined3.txt", "logcombined4.txt"

Clear-Host  # Just to clear the console.

$starttime = Get-Date
Search-UdwFile -filepath $filepath -filelist $filelist -searchstring "*txt*"
$endtime = get-date
"Execution time: " + ($endtime - $starttime)
```

The first line in Listing 13-14 assigns the list of file names to search to `$filelist`—this is a string array. Then, `Clear-Host` clears the console so we can see the output more easily. Since the idea behind using a workflow is to improve performance, there is code to store the start time and end time and then calculate the duration of the run. The cmdlet `Get-Date` returns the current date/time. So, we are storing the start time before running the workflow. Then, the workflow is called, passing the path to where the files are stored as `$filepath`, the list of files to be searched as `$filelist`, and the string to search for as `*txt*`. Running parallel processes on a single machine may hit a performance limit at some point. Workflows can be executed remotely; i.e., we can submit them to multiple machines to execute in parallel. To run multiple instances of the file-search workflow, we just need to modify the execution code, as shown in Listing 13-15.

Listing 13-15. Calling the workflow to run remotely

```
Import-Module umd_workflow  # To get the functions we will use below

$filelist1 = "logcombined.txt", "logcombined2.txt"
$filelist2 = "logcombined3.txt", "logcombined4.txt"

Search-UdfFile -filepath $filepath -filelist $filelist1 -searchstring "*txt*" '
                 -PSComputerName remote1 -AsJob -JobName 'remote1search'

Search-UdfFile -filepath $filepath -filelist $filelist2 -searchstring "*txt*" '
                 -PSComputerName remote2 -AsJob -JobName 'remote2search'
```

In order for us to search files on other machines, we need to copy the files we want searched to the remote machines. Once that is done, we can just run the code, and it will submit the workflows to machines `remote1` and `remote2` to be executed in a background job. Note the special line-continuation character, `` ` ``, which allows a statement to span multiple lines. The workflow `Search-UdfFile` is executed, passing the path to the source files as `$filepath` and the list of files to be searched as `$filelist1` in the first execution and `$filelist2` in the second. The `$searchstring` parameter defines the character string to search for. `PSComputerName` is the workflow parameter that tells PowerShell where to run the workflow. `AsJob` makes the workflow run as a background job on the remote machine, and `JobName` names the job. Although the job runs remotely, the local job queue tracks it for us just like a local job.

Summary

In this chapter, we discussed creating and executing PowerShell workflows. Workflows can support complex processes that can span a long period of time, can be stopped and restarted while retaining state, can support parallel execution, and can be packaged into reusable components. PowerShell 3.0 introduced support for creating workflows, which previously could only be created in Visual Studio using C# or VB .NET. We discussed how workflows can be executed either locally or on remote machines. We discussed using the workflow commands `parallel`, `foreach -parallel`, `sequence`, and `inlinescript` to control workflow execution. Then, we covered using the cmdlets `Checkpoint-Workflow`, `Suspend-Workflow`, and `Resume-Job` to pause and resume workflow execution. Workflows can work on a single object at a time or on a collection. In database development, we tend to work with sets of data. We covered two data-centric examples of using workflows. One was in a typical data-warehouse load scenario. The other was using a workflow to speed up the extraction of data from flat files using parallel processing, similar to Hadoop. Workflows should be reserved for solutions that require their capabilities. Often, simpler approaches can be used. However, when you need them, workflows offer amazing power.