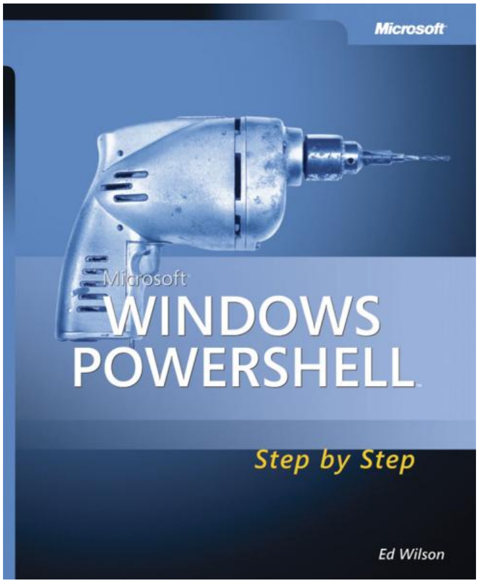


Microsoft® Windows PowerShell™ Step By Step

Ed Wilson

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/10329.aspx

9780735623958
Publication Date: May 2007

Chapter 8

Leveraging the Power of ADO

After completing this chapter, you will be able to:

- Understand the use of ADO in Windows PowerShell scripts
- Connect to Active Directory to perform a search
- Control the way data are returned
- Use compound query filters

Connecting to Active Directory with ADO

In this section, you will learn a special query technique to search Active Directory using ActiveX Data Objects (ADO). The technique is exactly the same technique you will use to search other databases. You will be able to use the results returned by that custom query to perform additional tasks. For example, you could search Active Directory for all users who don't have telephone numbers assigned to them. You could then send that list to the person in charge of maintaining the telephone numbers. Even better, you could modify the search so that it returns the user names and their managers' names. You could then take the list of users with no phone numbers that is returned and send e-mail to the managers to update the phone list in Active Directory. The functionality incorporated in your scripts is primarily limited by your imagination. The following list summarizes uses of the search technology:

- Query Active Directory for a list of computers that meet a given search criterion
- Query Active Directory for a list of users who meet a given search criterion
- Query Active Directory for a list of printers that meet a given search criterion
- Use the data returned from the preceding three queries to perform additional operations

All the scripts mentioned in this chapter can be found in the corresponding scripts folder on the CD.

Just the Steps To search Active Directory

1. Create a connection to Active Directory by using ADO.
2. Use the Open() method of the object to access Active Directory.
3. Create an ADO Command object and assign the ActiveConnection property to the Connection object.
4. Assign the query string to the CommandText property of the Command object.
5. Use the Execute() method to run the query and store the results in a RecordSet object.
6. Read information in the result set using properties of the RecordSet object.
7. Close the connection by using the Close() method of the Connection object.

The script BasicQuery.ps1 (shown later) illustrates how to search Active Directory by using ADO. Keep in mind that BasicQuery.ps1 can be used as a template script to make it easy to perform Active Directory queries using ADO.

The BasicQuery.ps1 script begins with defining the query that will be used. The string is stored in the *$strQuery* variable. When querying Active Directory using ADO, there are two ways the query can be specified. The one used here is called the *Lightweight Directory Access Protocol (LDAP) dialect*. The other means of specifying the query is called the *SQL dialect* and will be explored later in this chapter.

The LDAP dialect string is made up of four parts. Each of the parts is separated by a semicolon. If one part is left out, then the semicolon must still be present. This is actually seen in the BasicQuery.ps1 script because we do not supply a value for the filter portion. This line of code is shown here:

```
$strQuery = "<LDAP://dc=nwtraders,dc=msft>;;name;subtree"
```

Table 8-1 illustrates the LDAP dialect parts. In the BasicQuery.ps1 script, the filter is left out of the query. The base portion is used to specify the exact point of the connection into Active Directory. Here we are connecting to the root of the NwTraders.msft domain. We could connect to an organizational unit (OU) called MyTestOU by using the distinguished name, as shown here:

```
ou=myTestOU,dc=nwtraders,dc=msft
```

Table 8-1 LDAP Dialect Query Syntax

Base	Filter	Attributes	Search Scope
<LDAP://dc=nwtraders,dc=msft>	(objectCategory=computer)	name	subtree

When we create the filter portion of the LDAP dialect query, we specify the attribute name on the left and the value for the attribute we are looking for on the right. If I were looking for every object that had a location of Atlanta, then the filter would look like the one shown here:

```
(l=Atlanta)
```

The attribute portion of the LDAP query is a simple list of attributes you are looking for, each separated by a comma. If after you had found objects in Atlanta, you wanted to know the name and category of the objects, your attribute list would look like the following:

```
Name, objectCategory
```

The search scope is the last portion of the LDAP dialect query. There are three possible values for the search scope. The first is base. If we specify the search scope as base, then it will only return the single that was the target of the query, that is, the base portion of the query. Using base is valuable if you want to determine whether an object is present in active directory.

The second allowable value for the search scope is oneLevel. When you use the search scope of oneLevel, it will return the Child objects of the base of your query. It does not, however, perform a recursive query. If your base is an OU, then it will list the items contained in the OU. But it will not go into any child OUs and list their members. This is an effective query technique and should be considered standard practice.

The last allowable value for the search scope is subtree. Subtree begins at the base of your query and then recurses into everything under the base of your query. It is sometimes referred to in the Platform Software Development Kit (SDK) as the deep search option because it will dig deeply into all sublevels of your Active Directory hierarchy. If you target the domain root, then it will go into every OU under the domain root, and then into the child OUs, and so forth. This should be done with great care because it can generate a great deal of network traffic and a great deal of workload on the server. If you do need to perform such a query, then you should perform the query asynchronously, and use paging to break the result set into smaller chunks. This will level out the network utilization. In addition, you should try to include one attribute that is indexed. If the attributes you are interested in are replicated to the Global Catalog (GC), then you should query the GC instead of connecting to rootDSE (DSA-specific entry). These techniques will all be examined in this chapter.

After the query is defined, we need to create two objects. We will use the *New-Object* cmdlet to create these objects. The first object to create is the ADODB.Connection object. The line of code used to create the Connection object is shown here:

```
$objConnection = New-Object -comObject "ADODB.Connection"
```

This object is a com object and is contained in the variable *$objConnection*. The second object that is needed is the ADODB.Command object. The code to create the Command object is shown here:

```
objCommand = New-Object -comObject "ADODB.Command"
```

After the two objects are created, we need to open the connection into Active Directory. To open the connection, we use the Open method from the ADODB.Connection object. When we call the Open method, we need to specify the name of the provider that knows how to read the Active Directory database. For this, we will use the ADsDSOObject provider. This line of code is shown here:

```
$objConnection.Open("Provider=ADsDSOObject;")
```

After the connection into the Active Directory database has been opened, we need to associate the Command object with the Connection object. To do this, we use the ActiveConnection property of the Command object. The line of code that does this is shown here:

```
$objCommand.ActiveConnection = $objConnection
```

Now that we have an active connection into Active Directory, we can go ahead and assign the query to the command text of the Command object. To do this, we use the CommandText property of the Command object. In the BasicQuery.ps1 script, we use the following line of code to do this:

```
$objCommand.CommandText = $strQuery
```

After everything is lined up, we call the Execute method of the Command object. The Execute method will return a RecordSet object, which is stored in the *$objRecordSet* variable. This line of code is shown here:

```
$objRecordSet = $objCommand.Execute()
```

To examine individual records from the RecordSet object, we use the *do ... until* statement to walk through the collection. The script block of the *do ... until* statement is used to retrieve the Name property from the RecordSet object. To retrieve the specific property, we retrieve the Fields.Item property and specify the property we retrieved from the attributes portion of the query. We then pipeline the resulting object into the *Select-Object* cmdlet and choose both the name and the Value property. This line of code is shown here:

```
$objRecordSet.Fields.item("name") |Select-Object Name,Value
```

To move to the next record in the recordset, we need to use the MoveNext method from the RecordSet object. This line of code is shown here:

```
$objRecordSet.MoveNext()
```

The complete BasicQuery.ps1 script is shown here:

BasicQuery.ps1

```
$strQuery = "<LDAP://dc=nwtraders,dc=msft>;;name;subtree"

$objConnection = New-Object -comObject "ADODB.Connection"
$objCommand = New-Object -comObject "ADODB.Command"
$objConnection.Open("Provider=ADsDSOObject;")
$objCommand.ActiveConnection = $objConnection
$objCommand.CommandText = $strQuery
```

```
$objRecordSet = $objCommand.Execute()

Do
{
    $objRecordSet.Fields.item("name") |Select-Object Name,Value
    $objRecordSet.MoveNext()
}
Until ($objRecordSet.eof)

$objConnection.Close()
```

Quick Check

Q. What technology is utilized to search Active Directory?

A. DO is the technology that is used to search Active Directory.

Q. Which part of the script is used to perform the query?

A. The command portion of the script is used to perform the query.

Q. How are results returned from an ADO search of Active Directory?

A. The results are returned in a recordset.

Creating More Effective Queries

The BasicQuery.ps1 script is a fairly wasteful script in that all it does is produce a list of user names and print them out. Although the script illustrates the basics of making a connection into Active Directory by using ADO, it is not exactly a paradigm of efficiency. ADO, however, is a very powerful technology, and there are many pieces of the puzzle we can use to make the script more efficient and more effective. The first thing we need to do is to understand the objects we have that we can use with ADO. These objects are listed in Table 8-2.

Table 8-2 **Objects Used to Search Active Directory**

Object	Description
Connection	An open connection to an OLE DB data source such as ADSI
Command	Defines a specific command to execute against the data source
Parameter	An optional collection for any parameters to provide to the Command object
RecordSet	A set of records from a table, a Command object, or SQL syntax A RecordSet object can be created without any underlying Connection object
Field	A single column of data in a recordset
Property	A collection of values supplied by the provider for ADO
Error	Contains details about data access errors. Refreshed when an error occurs in a single operation

When we use ADO to talk to Active Directory, we often are working with three different objects: the Connection object, the Command object, and the RecordSet object. The Command object is used to maintain the connection, pass along the query parameters, and perform such tasks as specifying the page size and search scope and executing the query. The Connection object is used to load the provider and to validate the user's credentials. By default, it utilizes the credentials of the currently logged-on user. If you need to specify alternative credentials, you can use the properties listed in Table 8-3. To do this, we need to use the Properties property of the Connection object. After we have the Connection object, and we use Properties to get to the properties, we then need to use Item to supply value for the specific property item we want to work with.

Table 8-3 Authentication Properties for the Connection Object

Property	Description
User ID	A string that identifies the user whose security context is used when performing the search. (For more information about the format of the user name string, see IADsOpenDSObject::OpenDSObject in the Platform SDK.) If the value is not specified, the default is the logged-on user or the user impersonated by the calling process.
Password	A string that specifies the password of the user identified by "User ID"
Encrypt Password	A Boolean value that specifies whether the password is encrypted. The default is False.
ADSI Flag	A set of flags from the ADS_AUTHENTICATION_ENUM enumeration. The flag specifies the binding authentication options. The default is zero.

Using Alternative Credentials

As network administration becomes more granular, with multiple domains, work groups, OUs, and similar grouping techniques, it becomes less common for everyone on the IT team to be a member of the Domain Admins group. If the script does not impersonate a user who is a member of the Domain Admins group, then it is quite likely it will need to derive permissions from some other source. One method to do this is to supply alternative credentials in the script. To do this, we need to specify certain properties of the Connection object.

Just the Steps **To create a connection in Active Directory using alternative credentials**

1. Create the ADODB.Connection object
2. Use the Provider property to specify the ADsDSOObject provider
3. Use Item to supply a value for the properties "User ID" and "Password"
4. Open the connection while supplying a value for the name of the connection

The technique outlined in the using alternative credentials step-by-step exercise is shown here. This code is from the QueryComputersUseCredentials.ps1 script, which is developed in the querying active directory using alternative credentials procedure.

```
$objConnection = New-Object -comObject "ADODB.Connection"
$objConnection.provider = "ADsDSOObject"
$objConnection.properties.item("user ID") = $strUser
$objConnection.properties.item("Password") = $strPwd
$objConnection.open("modifiedConnection")
```

Querying Active Directory using alternative credentials

1. Open the QueryComputers.ps1 script in Notepad or in your favorite Windows PowerShell script editor and save it as *yourname*QueryComputersUseCredentials.ps1.
2. On the first noncommented line, define a new variable called *$strBase*, and use it to assign the LDAP connection string. This variable is used to define the base of the query into Active Directory. For this example, we will connect to the root of the NwTraders.msft domain. To do this, the string is enclosed in angle brackets and begins with the moniker LDAP. The base string is shown here:

   ```
   "<LDAP://dc=nwtraders,dc=msft>"
   ```

   The new line of code is shown here:

   ```
   $strBase = "<LDAP://dc=nwtraders,dc=msft>"
   ```

3. Create a new variable called *$strFilter*. This will be used to hold the filter portion of our LDAP syntax query. Assign a string that specifies the objectCategory attribute when it is equal to the value of computer. This is shown here:

   ```
   $strFilter = "(objectCategory=computer)"
   ```

4. Create a new variable called *$strAttributes* and assign the string of name to it. This variable will be used to hold the attributes to search on in Active Directory. This line of code is shown here:

   ```
   $strAttributes = "name"
   ```

5. Create a variable called *$strScope*. This variable will be used to hold the search scope parameter of our LDAP syntax query. Assign the value of subtree to it. This line of code is shown here:

   ```
   $strScope = "subtree"
   ```

6. Modify the *$strQuery* line of code so that it uses the four variables we created:

   ```
   $strQuery = "<LDAP://dc=nwtraders,dc=msft>;;name;subtree"
   ```

   The advantage of this is that each of the four parameters that are specified for the LDAP syntax query can easily be modified by simply changing the value of the variable. This preserves the integrity of the worker section of the script. The order of the four parameters is base, filter, attributes, and scope. Thus, the revised value to assign to the *$strQuery* variable is shown here:

```
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
```

7. Create a new variable called *$strUser* and assign the string “LondonAdmin” to it. This is the name of the useraccount to use to make the connection to Active Directory. This line of code is shown here:

```
$strUser = "LondonAdmin"
```

8. Create a new variable called *$strPassword* and assign the string Password1 to it. This is the password that will be used when connecting into the NwTraders.msft domain by using the LondonAdmin account. This is shown here:

```
$strPwd = "Password1"
```

9. Between *the $objConnection = New-Object -comObject "ADODB.Connection"* command and the *$objCommand = New-Object -comObject "ADODB.Command"* command, insert four blank lines. This space will be used for rearranging the code and for inserting new properties on the Connection object. The revised code is shown here:

```
$objConnection = New-Object -comObject "ADODB.Connection"




$objCommand = New-Object -comObject "ADODB.Command"
```

10. Move the $objConnection.provider = "ADsDSOObject;" line of code from its position below the $objCommand = New-Object –comObject “ADODB.Command" line of code to below the line of code that creates the Connection object. After you have the code moved, remove the trailing semicolon because it is not needed. This revised code is shown here:

```
$objConnection = New-Object -comObject "ADODB.Connection"
$objConnection.provider = "ADsDSOObject"
```

11. Use the Item method of the properties collection of the Connection object to assign the value contained in the *$strUser* variable to the “User ID” property. This line of code is shown here:

```
$objConnection.properties.item("user ID") = $strUser
```

12. Use the Item method of the properties collection of the Connection object to assign the value contained in the *$strPassword* variable to the “Password” property. This line of code is shown here:

```
$objConnection.properties.item("Password") = $strPwd
```

13. The last line of code we need to modify from the old script is the $objConnection.Open("Provider=ADsDSOObject;") line. Because we needed to move the provider string up earlier in the code to enable us to modify the properties, we have already

specified the provider. So, now we only need to open the connection. When we open the connection, we give it a name "modifiedConnection" that we would be able to use later on in the script if we so desired. The revised line of code is shown here:

```
$objConnection.open("modifiedConnection")
```

14. Save and run your script. If it does not perform as expected, compare it with the QueryComputersUseCredentials.ps1 script.
15. This concludes the querying Active Directory using alternative credentials procedure.

Modifying Search Parameters

A number of search options are available to the network administrator. The use of these search options will have an extremely large impact on the performance of your queries against Active Directory. It is imperative, therefore, that you learn to use the following options. Obviously, not all options need to be specified in each situation. In fact, in many situations, the defaults will perform just fine. However, if a query is taking a long time to complete, or you seem to be flooding the network with unexpected traffic, you might want to take a look at the Search properties in Table 8-4.

Table 8-4 ADO Search Properties for Command Object

Property	Description
Asynchronous	A Boolean value that specifies whether the search is synchronous or asynchronous. The default is False (synchronous). A synchronous search blocks until the server returns the entire result (or for a paged search, the entire page). An asynchronous search blocks until one row of the search results is available, or until the time specified by the Timeout property elapses.
Cache Results	A Boolean value that specifies whether the result should be cached on the client side. The default is True; ADSI caches the resultset. Turning off this option might be desirable for large resultsets.
Chase Referrals	A value from ADS CHASE_REFERRALS_ENUM that specifies how the search chases referrals. The default is ADS_CHASE_REFERRALS EXTERNAL = 0x40. To set ADS_CHASE_REFERRALS_NEVER, set to 0.
Column Names Only	A Boolean value that indicates that the search should retrieve only the name of attributes to which values have been assigned. The default is False.
Deref (dereference) Aliases	A Boolean value that specifies whether aliases of found objects are resolved. The default is False.
PageSize	An integer value that turns on paging and specifies the maximum number of objects to return in a resultset. The default is no page size, which means that after 1000 items have been delivered from Active Directory, that is it. To turn on paging, you must supply a value for page size, and it must be less than the SizeLimit property. (For more information, see PageSize in the Platform SDK, which is available online from *http://msdn2.microsoft.com/*.)

Table 8-4 ADO Search Properties for Command Object

Property	Description
SearchScope	A value from the ADS_SCOPEENUM enumeration that specifies the search scope. The default is ADS_SCOPE_SUBTREE.
SizeLimit	An integer value that specifies the size limit for the search. For Active Directory, the size limit specifies the maximum number of returned objects. The server stops searching once the size limit is reached and returns the results accumulated up to that point. The default is No Limit.
Sort on	A string that specifies a comma-separated list of attributes to use as sort keys. This property works only for directory servers that support the LDAP control for server-side sorting. Active Directory supports the sort control, but this control can affect server performance, particularly when the resultset is large. Be aware that Active Directory supports only a single sort key. The default is No Sorting.
TimeLimit	An integer value that specifies the time limit, in seconds, for the search. When the time limit is reached, the server stops searching and returns the results accumulated to that point. The default is No Time Limit.
Timeout	An integer value that specifies the client-side timeout value, in seconds. This value indicates the time the client waits for results from the server before quitting the search. The default is No Timeout.

In the previous section, when we used alternative credentials in the script, we specified various properties on the Connection object. We will use the same type of procedure to modify search parameters, but this time we will specify values for various properties on the Command object. As an example, suppose we wanted to perform an asychronous query of Active Directory. We would need to supply a value of true for the asynchronous property. The technique is exactly the same as supplying alternative credentials: create the object, and use the Item method to specify a value for the appropriate property. This piece of code, taken from the AsynchronousQueryComputers.ps1 script, is shown here:

```
$objCommand = New-Object -comObject "ADODB.Command"
$objCommand.ActiveConnection = $objConnection
$objCommand.Properties.item("Asynchronous") = $blnTrue
```

> **Important** When specifying properties for the Command object, ensure you have the activeConnection associated with a valid Connection object before making the assignment. Otherwise, you will get an error stating the property is not valid—which can be very misleading.

Note that you should specify a page size. In Windows Server 2003, Active Directory is limited to returning 1000 objects from the results of a query when no page size is specified. The PageSize property tells Active Directory how many objects to return at a time. When this property is specified, there is no limit on the number of returned objects Active Directory can provide. If you specify a size limit, the page size must be smaller. The exception would be if you want an unlimited size limit of 0, then obviously the PageSize property would be larger than the value of 0. In the SizeLimitQueryUsers.ps1 script, after creating a Command object, associating the connection with the activeConnection property, we use the Item method of the properties collection to specify a size limit of 4. When the script is run, it only returns four users. The applicable portion of the code is shown here:

```
$objCommand = New-Object -comObject "ADODB.Command"
$objCommand.ActiveConnection = $objConnection
$objCommand.Properties.item("Size Limit") = 4
```

Controlling script execution

1. Open the QueryComputersUseCredentials.ps1 script, and save it as *yourname*QueryTimeOut.ps1.
2. Edit the query filter contained in the string that is assigned to the variable *$strFilter* so that the filter will return items when the ObjectCategory is equal to User. The revised line of code is shown here:

   ```
   $strFilter = "(objectCategory=User)"
   ```

3. Delete the line that creates the *$strUser* variable and assigns the LondonAdmin User to it.
4. Delete the line that creates the *$strPwd* variable and assigns the string Password1 to it.
5. Delete the two lines of code that assign the value contained in the *$strUser* variable to the User ID property, and the one that assigns the value contained in the *$strPwd* variable to the Password property of the Connection object. These two lines of code are shown commented-out here:

   ```
   #$objConnection.properties.item("user ID") = $strUser
   #$objConnection.properties.item("Password") = $strPwd
   ```

6. Inside the parentheses of the Open command that opens the Connection object to Active Directory, delete the Reference string that is contained inside it. We are able to delete this string because we did not use it to refer to the connection later in the script. The modified line of code is shown here:

   ```
   $objConnection.open()
   ```

7. Under the line of code that assigns the Connection object that is contained in the *$objConnection* variable to the ActiveConnection property of the Command object, we want to add the value of 1 to the TimeLimit property of the Command object. To do this, use the property name TimeLimit, and use the Item method to assign it to the properties collection of the Command object. The line of code that does this is shown here:

   ```
   $objCommand.properties.item("Time Limit")=1
   ```

8. Save and run your script. If it does not produce the desired results, compare it with QueryTimeOut.ps1.
9. This concludes the controlling script execution procedure.

Searching for Specific Types of Objects

One of the best ways to improve the performance of Active Directory searches is to limit the scope of the search operation. Fortunately, searching for a specific type of object is one of the easiest tasks to perform. For example, to perform a task on a group of computers, limit your search to the computer class of objects. To work with only groups, users, computers, or printers, specify the objectClass or the objectCategory attribute in the search filter. The objectCategory attribute is a single value that specifies the class from which the object in Active Directory is derived. In other words, users are derived from an objectCategory called *users*. All the properties you looked at in Chapter 7, "Working with Active Directory," when we were creating objects in Active Directory are contained in a template called an *objectCategory*. When you create a new user, Active Directory does a lookup to find out what properties the user class contains. Then it copies all those properties onto the new user you just created. In this way, all users have the same properties available to them.

Just the Steps **To limit the Active Directory search**

1. Create a connection to Active Directory by using ADO.
2. Use the Open method of the object to access Active Directory.
3. Create an ADO Command object, and assign the ActiveConnection property to the Connection object.
4. Assign the query string to the CommandText property of the Command object.
5. In the query string, specify the objectCategory of the target query.
6. Choose specific fields of data to return in response to the query.
7. Use the Execute method to run the query and store the results in a RecordSet object.
8. Read information in the result set using properties of the RecordSet object.
9. Close the connection by using the Close method of the Connection object.

In the QueryComputers.ps1 script, you use ADO to query Active Directory with the goal of returning a recordset containing selected properties from all the computers with accounts in the directory.

To make the script easier to edit, we abstracted each of the four parts of the LDAP dialect query into a separate variable. The *$strBase* variable in the QueryComputers.ps1 script is used to hold the base of the ADO query. The base is used to determine where the script will make its connection into Active Directory. The line of code that does this in the QueryComputers.ps1 script is shown here:

```
$strBase = "<LDAP://dc=nwtraders,dc=msft>"
```

The filter is used to remove the type of objects that are returned by the ADO query. In the QueryComputers.ps1 script, we filter on the value of the objectCategory attribute when it has a value of computer. This line of code is shown here:

```
$strFilter = "(objectCategory=computer)"
```

The attributes to be selected from the query are specified in the *$strAttributes* variable. In the QueryComputers.ps1 script, we choose only the Name attribute. This line of code is shown here:

```
$strAttributes = "name"
```

The search scope determines how deep the query will go. There are three possible values for this: base, oneLevel, and subtree. Base searches only at the level where the script connects. OneLevel tells ADO to go one level below where the *$strbase* connection is made. Subtree is probably the most commonly used and tells ADO to make a recursive query through Active Directory. This is the kind of query we do in QueryComputers.ps1. This line of code is shown here:

```
$strScope = "subtree"
```

The $strQuery is used to hold the query used to query from Active Directory. When it is abstracted into variables, it becomes easy to modify. The revised code is shown here:

```
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
```

The complete QueryComputers.ps1 is shown here:

QueryComputers.ps1

```
$strBase = "<LDAP://dc=nwtraders,dc=msft>"
$strFilter = "(objectCategory=computer)"
$strAttributes = "name"
$strScope = "subtree"
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"

$objConnection = New-Object -comObject "ADODB.Connection"
$objCommand = New-Object -comObject "ADODB.Command"
$objConnection.Open("Provider=ADsDSOObject;")
$objCommand.ActiveConnection = $objConnection
$objCommand.CommandText = $strQuery
$objRecordSet = $objCommand.Execute()

Do
{
    $objRecordSet.Fields.item("name") |Select-Object Name,Value
    $objRecordSet.MoveNext()
}
Until ($objRecordSet.eof)

$objConnection.Close()
```

Querying multiple attributes

1. Open Notepad or your favorite Windows PowerShell editor.
2. Open QueryComputers.ps1, and save it as *yourname*QueryComputersByName.ps1.
3. Edit the $strFilter line so that it includes the additional attribute name. To do this using the LDAP dialect, we will need to first add an extra set of parentheses around the entire filter expression. This is shown here:

```
$strFilter = "((objectCategory=computer))"
```

4. Between the first set of double parentheses, we will add the ampersand character (&), which will tell the LDAP dialect search filter we want both of the attributes we are getting ready to supply. This is shown here:

```
$strFilter = "(&(objectCategory=computer))"
```

5. At end of the first search filter expression, we want to add a second expression. We want to search by both Computer Type objects and usernames. This modified line of code is shown here:

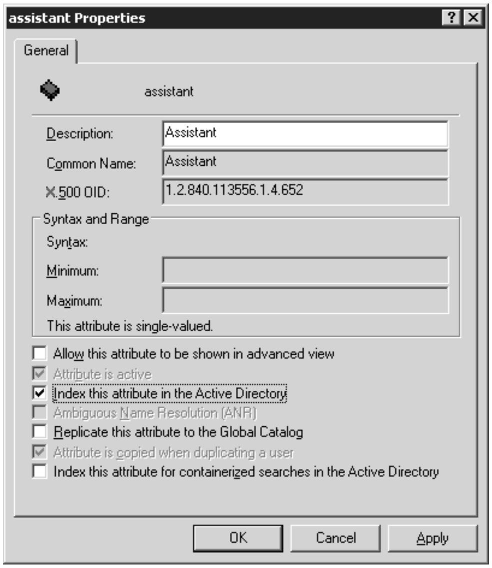
```
$strFilter = "(&(objectCategory=computer)(name=london))"
```

6. Save and run your script. It should produce a script output that lists all computer accounts named London.
7. This concludes the querying multiple attributes procedure.

What Is Global Catalog?

As you become more proficient in writing your scripts, and as you begin to work your magic on the enterprise on a global scale, you will begin to wonder why some queries seem to take forever and others run rather fast. After configuring some of the parameters you looked at earlier, you might begin to wonder whether you're hitting a Global Catalog (GC) server. A *Global Catalog server* is a server that contains all the objects and their associated attributes from your local domain. If all you have is a single domain, it doesn't matter whether you're connecting to a domain controller or a GC server because the information will be the same. If, however, you are in a multiple domain forest, you might very well be interested in which GC server you are hitting. Depending on your network topology, you could be executing a query that is going across a slow WAN link. You can control replication of attributes by selecting the Global Catalog check box. You can find this option by opening the Active Directory Schema MMC, highlighting the Attributes container. The Active Directory Schema MMC is not available by default in the Administrative Tools program group. For information on how to install it, visit the following URL: *http://technet2.microsoft.com/WindowsServer/en/library/2218144f-bb92-454e-9334-186ee7c740c61033.mspx?mfr=true.*

In addition to controlling the replication of attributes, the erstwhile administrator might also investigate attribute indexing (Fig. 8-1.) Active Directory already has indexes built on certain objects. However, if an attribute is heavily searched on, you might consider an additional index. You should do this, however, with caution because an improperly placed index is worse than no index at all. The reason for this is the time spent building and maintaining an index. Both of these operations use processor time and disk I/O.



assistant

Description: Assistant

Common Name: Assistant

X.500 OID: 1.2.840.113556.1.4.652

Syntax and Range

Syntax:

Minimum:

Maximum:

This attribute is single-valued.

☐ Allow this attribute to be shown in advanced view
☑ Attribute is active
☑ Index this attribute in the Active Directory
☐ Ambiguous Name Resolution (ANR)
☐ Replicate this attribute to the Global Catalog
☑ Attribute is copied when duplicating a user
☐ Index this attribute for containerized searches in the Active Directory

Figure 8-1 Heavily queried attributes often benefit from indexing

Querying a global catalog server

1. Open the BasicQuery.ps1 script in Notepad or another Windows PowerShell editor and save the file as *yourname*QueryGC.ps1

2. On the first noncommented line of your script, declare a variable called *$strBase*. This variable will be used to control the connection into the global catalog server in Active Directory. To do this, instead of using the LDAP moniker, we will use the GC moniker. The rest of the path will be the same because it uses the Distinguished Name of target. For this procedure, let's connect to the OU called MyTestOU in the NwTraders.msft domain. The line of code to do this is shown here:

   ```
   $strBase = <GC://ou=MyTestOU,dc=nwtraders,dc=msft>
   ```

3. On the next line, create a variable called *$strFilter*. This variable will be used to hold the filter portion of the query. The filter will be used to return only objects from Active Directory that have the objectCategory attribute set to User. The line of code that does this is shown here:

   ```
   $strFilter = "(objectCategory=user)"
   ```

4. Create a variable called *$strAttributes* that will be used to hold the attributes to retrieve from Active Directory. For this script, we are only interested in the Name attribute. The line of code that does this is shown here:

   ```
   $strAttributes = "name"
   ```

5. Create a variable called *$strScope*. This variable will hold the string oneLevel that is used to tell Active Directory that we want the script to obtain a list of the users in the MyTestOU only. We do not need to perform a recursive type of query. This line of code is shown here:

   ```
   $strScope = "oneLevel"
   ```

6. Modify the *$strQuery* line so that it uses the four variables we defined for each of the four parts of the LDAP dialect query. The four variables are *$strBase, $strFilter, $strAttributes,* and *$strScope* in this order. Make sure you use a semicolon to separate the four parts from one another. Move the completed line of code to the line immediately beneath the $strScope-“oneLevel” line. The completed line of code is shown here:

   ```
   $strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
   ```

7. Save and run your script. It should run without errors. If it does not produce the expected results, compare your script with the QueryGC.ps1 script.
8. This concludes the querying a global catalog server procedure.

Querying a specific server

1. Open the QueryGC.ps1 script in Notepad or your favorite Windows PowerShell script editor, and save the script as *yourname*QuerySpecificServer.ps1.
2. Edit the string assigned to the *$strBase* variable so that you use the LDAP moniker instead of the GC moniker. After the ://, type the name of the server. Do not use CN=, as would normally be used for the Distinguished Name attribute. Instead, just type the name of the server followed by a forward slash (\). The completed line of code is shown here:

   ```
   $strBase = "<LDAP://London/ou=MyTestOU,dc=nwtraders,dc=msft>"
   ```

3. Save and run your script. If your script does not work properly, compare it with the QuerySpecificServer.ps1 script.
4. This concludes the querying a specific server procedure.

Querying a specific server by IP address

1. Open the QuerySpecificServer.ps1 script in Notepad or your favorite Windows PowerShell script editor and save it as *yourname*QuerySpecificServerByIP.ps1.
2. Edit the string that is assigned to the *$strBase* variable so that you are supplying the LDAP moniker with an IP address instead of a Host name. The revised line of code is shown here:

   ```
   $strBase = "<LDAP://192.168.1.1/ou=MyTestOU,dc=nwtraders,dc=msft>"
   ```

3. Save and run your script. If your script does not run properly, compare it with the QuerySpecificServerByIP.ps1 script.
4. This concludes the querying a specific server by IP address procedure.

Using the base search scope

1. Open the QuerySpecificServer.ps1 script in Notepad, or some other Windows PowerShell script editor. Save the script as *yourname*SearchBase.ps1.
2. Change the $strScope line so that it will point to base instead of oneLevel. This revised line of code is shown here:

   ```
   $strScope = "base"
   ```

3. Because a base query connects to a specific object, there is no point in having a filter. Delete the $strFilter line, and remove the *$strFilter* variable from the second position of the *$strQuery* string that is used for the LDAP dialect query. The revised $strQuery line of code is shown here:

   ```
   $strQuery = "$strBase;;$strAttributes;$strScope"
   ```

4. Because the base query will only return a single object, it does not make sense to perform a *do … until* loop. Delete the line that has the opening *Do*, and delete the line with the *Until ($objRecordSet.eof)*.
5. Delete the opening and the closing curly brackets. Delete the *$objRecordSet.MoveNext()* command because there are no more records to move to.
6. Go to the *$strAttributes* variable and modify it so that we retrieve both the objectCategory and the Name attributes. The revised line of code is shown here:

   ```
   $strAttributes = "objectCategory,name"
   ```

7. Copy the $objRecordSet.Fields.item("name") |Select-Object value line of code, and paste it just below the first one. Edit the first $objRecordSet.Fields.item line of code so that it will retrieve the objectCategory attribute from the recordset. The two lines of code are shown here:

   ```
   $objRecordSet.Fields.item("objectCategory") |Select-Object value
   $objRecordSet.Fields.item("name") |Select-Object value
   ```

8. Save and run your script. If it does not perform correctly, compare it with the Search Base.ps1 script.
9. This concludes the using the base search scope procedure.

Using the SQL Dialect to Query Active Directory

For many network professionals, the rather cryptic way of expressing the query into Active Directory is at once confusing and irritating. Because of this confusion, we also have an SQL dialect we can use to query Active Directory. The parts that make up an SQL dialect query are listed in Table 8-5. Of the four parts that can make up an SQL dialect query, only two parts are required: the *Select* statement and the *from* keyword that indicates the base for the search. An example of this use is shown here. A complete script that uses this type of query is the SelectNameSQL.ps1 script.

```
Select name from 'LDAP://ou=MyTestOu,dc=nwtraders,dc=msft'
```

Table 8-5 SQL Dialect

Select	From	Where	Order by
Comma separated list of attributes	AdsPath for the base of the search enclosed in single quotation marks	Optional Used for the filter	Optional. Used for server side sort control. A comma separated list of attributes

The *Where* statement is used to specify the filter for the Active Directory query. This is similar to the filter used in the LDAP dialect queries. The basic syntax of the filter is attributeName = value. But as in any SQL query, we are free to use various operators, as well as *and*, or *or*, and even wild cards. An example of a query using *Where* is shown here (keep in mind this is a single line of code that was wrapped for readability). A complete script that uses this type of query is the QueryComputersSQL.ps1 script.

```
Select name from 'LDAP://ou=MyTestOu,dc=nwtraders,dc=msft'where
objectCategory='computer'
```

The order by clause is the fourth part of the SQL dialect query. Just like the *Where* clause, it is also optional. In addition to selecting the property to order by, you can also specify two keywords: *ASC* for ascending and *DESC* for descending. An example of using the order by clause is shown here. A complete script using this query is the QueryUsersSQL.ps1 script.

```
Select adsPath, cn from 'LDAP://dc=nwtraders,dc=msft' where
objectCategory='user'order by sn DESC
```

The SQLDialectQuery.ps1 script is different from the BasicQuery.ps1 script only in the dialect used for the query language. The script still creates both a Connection object and a Command object, and works with a RecordSet object in the output portion of the script. In the SQLDialectQuery.ps1 script, we hold the SQL dialect query in three different variables. The *$strAttributes* variable holds the select portion of the script. *$strBase* is used to hold the AdsPath attribute, which contains the complete path to the target of operation. The last variable used in holding the query is the *$strFilter* variable, which holds the filter portion of the query. Using these different variables makes the script easier to modify and easier to read. The *$strQuery* variable is used to hold the entire SQL dialect query. If you are curious to see the query put together in its entirety, you can simply print out the value of the variable by adding the $strQuery line under the line where the query is put back together.

SQLDialectQuery.ps1

```
$strAttributes = "Select name from "
$strBase = "'LDAP://ou=MyTestOu,dc=nwtraders,dc=msft'"
$strFilter = " where objectCategory='computer'"
$strQuery = "$strAttributes$strBase$strFilter"

$objConnection = New-Object -comObject "ADODB.Connection"
$objCommand = New-Object -comObject "ADODB.Command"
$objConnection.Open("Provider=ADsDSOObject;")
$objCommand.ActiveConnection = $objConnection
$objCommand.CommandText = $strQuery
$objRecordSet = $objCommand.Execute()

Do
{
    $objRecordSet.Fields.item("name") |Select-Object Name,Value
    $objRecordSet.MoveNext()
}
Until ($objRecordSet.eof)

$objConnection.Close()
```

Creating an ADO Query into Active Directory: Step-by-Step Exercises

In this exercise, we will explore the use of various queries against Active Directory. We will use both simple and compound query filters as we return data, beginning with the generic and moving on to the more specific.

1. Launch the CreateMultipleUsers.ps1 script from the scripts folder for this chapter. This script will create 60 users with city locations from three different cities, and four different departments. We will use the different users and departments and cities in our Active Directory queries. By default, the script will create the users in the MyTestOU in the NwTraders.msft domain. If your Active Directory configuration is different, then edit the Active Directory Service Interfaces (ADSI) connection string shown here. If you are unsure of how to do this, refer back to Chapter 7, “Working with Active Directory.”

   ```
   $objADSI = [ADSI]"LDAP://ou=myTestOU,dc=nwtraders,dc=msft"
   ```

2. Open Notepad or another Windows PowerShell script editor.
3. On the first line, declare a variable called *$strBase*. This variable will be used to hold the base for our LDAP syntax query into Active Directory. The string will use angle brackets at the beginning and the end of the string. We will be connecting to the MyTestOU in the NwTraders.msft domain. The line of code that does this is shown here:

   ```
   $strBase = "<LDAP://ou=mytestOU,dc=nwtraders,dc=msft>"
   ```

4. On the next line, declare a variable called *$strFilter*. This variable will hold the string that will be used for the query filter. It will filter out every object that is not a User object. The line of code that does this is shown here:

   ```
   $strFilter = "(objectCategory=User)"
   ```

5. Create a variable called *$strAttributes*. This variable will hold the attribute we wish to retrieve from Active Directory. For this lab, we only want the name of the object. This line of code is shown here:

   ```
   $strAttributes = "name"
   ```

6. On the next line, we need to declare a variable called *$strScope* that will hold the search scope parameter. For this exercise, we will use the subtree parameter. This line of code is shown here:

   ```
   $strScope = "subtree"
   ```

7. On the next line, we put the four variables together to form our query string for the ADO query into Active Directory. Hold the completed string in a variable called *$strQuery*. Inside quotes, separate each of the four variables by a semicolon, which is used by the LDAP dialect to distinguish the four parts of the LDAP dialect query. The line of code to do this is shown here:

   ```
   $strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
   ```

8. Create a variable called *$objConnection*. The *$objConnection* variable will be used to hold an ADODB.Connection COM object. To create the object, use the *New-Object* cmdlet. This line of code is shown here:

   ```
   $objConnection = New-Object -comObject "ADODB.Connection"
   ```

9. Create a variable called *$objCommand* that will be used to hold a new instance of the COM object “ADODB.Command”. The code to do this is shown here:

   ```
   $objCommand = New-Object -comObject "ADODB.Command"
   ```

10. Open the Connection object by calling the Open method. Supply the name of the provider to use while opening the connection. For this lab, we will use the AdsDSOObject provider. The line of code that does is shown here:

    ```
    $objConnection.Open("Provider=ADsDSOObject")
    ```

11. Now we need to associate the Connection object we just opened with the ActiveConnection property of the Command object. To do this, simply supply the Connection object contained in the *$objConnection* variable to the ActiveConnection property of the Command object. The code that does this is shown here:

    ```
    $objCommand.ActiveConnection = $objConnection
    ```

12. Now we need to supply the text for the Command object. To do this, we will use the query contained in the variable *$strQuery* and assign it to the CommandText property of the Command object held in the *$objCommand* variable. The code that does this is shown here:

    ```
    $objCommand.CommandText = $strQuery
    ```

13. It is time to execute the query. To do this, call the Execute method of the Command object. It will return a RecordSet object, so use the variable *$objRecordSet* to hold the RecordSet object that comes back from the query.

    ```
    $objRecordSet = $objCommand.Execute()
    ```

14. Use a *do … until* statement to walk through the recordset until you reach the end of file. While you are typing this, go ahead and open and close the curly brackets. This will take four lines of code, which are shown here:

    ```
    Do
    {


    }
    Until ($objRecordSet.eof)
    ```

15. Inside the curly brackets, retrieve the Name attribute from the recordset by using the Item method from the Fields property. Pipeline the resulting object into a *Select-Object* cmdlet and retrieve only the value property. This line of code is shown here:

    ```
    $objRecordSet.Fields.item("name") |Select-Object Value
    ```

16. Call the MoveNext method to move to the next record in the RecordSet object contained in the *$objRecordSet* variable. This line of code is shown here:

    ```
    $objRecordSet.MoveNext()
    ```

17. After the until ($objRecordSet.eof) line of code, call the Close method from the RecordSet object to close the connection into Active Directory. This line of code is shown here:

    ```
    $objConnection.Close()
    ```

18. Save your script as *yourname*QueryUsersStepByStep.ps1. Run your script. You should see the name of 60 users come scrolling forth from the Windows PowerShell console. If this is not the case, compare your script with the QueryUsersStepByStep.ps1 script. Note, in the QueryUsersStepByStep.ps1 script, there are five $strFilter lines … only one that is not commented out. This is so you will have documentation on the next steps. When this code is working, it is time to move on to a few more steps.

19. Now we want to modify the filter so that it will only return users who are in the Charlotte location. To do this, copy the $strFilter line and paste it below the current line of code. Now, comment out the original $strFilter. We now want to use a compound query: objects in Active Directory that are of the category user, and a location attribute of Charlotte. From Chapter 7, you may recall the attribute for location is l. To make a compound

query, enter the search parameter inside parentheses, inside the grouping parentheses, after the first search filter. This modified line of code is shown here:

```
$strFilter = "(&(objectCategory=User)(l=charlotte))"
```

20. Save and run your script. Now, we want to add an additional search parameter. Copy your modified $strFilter line, and paste it beneath the line you just finished working on. Comment out the previous $strFilter line. Just after the location filter of Charlotte, add a filter for only users in Charlotte who are in the HR department. This revised line of code is shown here:

```
$strFilter = "(&(objectCategory=User)(l=charlotte)(department=hr))"
```

21. Save and run your script. Now copy your previous $strFilter line of code, and paste it below the line you just modified. This change is easy. You want all users in Charlotte who are not in HR. To make a not query, place the exclamation mark (bang) operator inside the parentheses you wish the operator to affect. This modified line of code is shown here:

```
$strFilter = "(&(objectCategory=User)(l=charlotte)(!department=hr))"
```

22. Save and run your script. Because this is going so well, let's add one more parameter to our search filter. So, once again copy the $strFilter line of code you just modified, and paste it beneath the line you just finished working on. This time, we want users who are in Charlotte or Dallas and who are not in the HR department. To do this, add a l=dallas filter behind the l=charlotte filter. Put parentheses around the two locations, and then add the pipeline character (|) in front of the l=charlotte parameter. This revised line of code is shown here. Keep in mind that it is wrapped for readability, but should be on one logical line in the script.

```
$strFilter = "(&(objectCategory=User)(|(l=charlotte)(l=dallas))(!department=hr))"
```

23. Save and run your script. In case you were getting a little confused by all the copying and pasting, here are all the *$strFilter* commands you have typed in this section of the step-by-step exercise:

```
$strFilter = "(objectCategory=User)"
#$strFilter = "(&(objectCategory=User)(l=charlotte))"
#$strFilter = "(&(objectCategory=User)(l=charlotte)(department=hr))"
#$strFilter = "(&(objectCategory=User)(l=charlotte)(!department=hr))"
#$strFilter = "(&(objectCategory=User)(|(l=charlotte)(l=dallas))(!department=hr))"
```

24. This concludes this step-by-step exercise.

One Step Further: Controlling How a Script Executes Against Active Directory

In this exercise, we will control the way we return data from Active Directory.

1. To make it easier to keep up the number of users returned from our Active Directory queries, run the DeleteMultipleUsers.ps1 script. This will delete the 60 users we created for the previous step-by-step exercise.

2. Run the Create2000Users.ps1 script. This script will create 2000 users for you to use in the MyTestOU OU.

3. Open the QueryUsersStepbyStep.ps1 script and save it as *yourname*OneStepFurther-QueryUsers.ps1.

4. Because we are not interested in running finely crafted queries in this exercise (rather, we are interested in how to handle large amounts of objects that come back), delete all the *$strFilter* commands except for the one that filters out User objects. This line of code is shown here:

   ```
   $strFilter = "(objectCategory=User)"
   ```

5. Save and run your script. You will see 1000 user names scroll by in your Windows PowerShell console window. After about 30 seconds (on my machine anyway), you will finally see MyLabUser997 show up. The reason it is MyLabUser997 instead of MyLabUser1000 is that this OU already had three users when we started (myBoss, myDirect1, and myDirect2). This is OK; it is easy to see that the query returned the system default of 1000 objects.

6. We know, however, there are more than 2000 users in the MyTestOU, and we have only been able to retrieve 1000 of them. To get past the query limit that is set for Active Directory, we need to turn on paging. This is simple. We assign a value for the PageSize property to be less than the 1000 object limit. To do this, we use the Item method of the properties collection on the Command object and assign the value of 500 to the PageSize property. This line of code is shown here. Place this code just above this line, which creates the RecordSet object: $objRecordSet = $objCommand.Execute().

   ```
   $objCommand.Properties.item("Page Size") = 500
   ```

7. After you have made the change, save and run your script. You should see all 2000 user objects show up … however, the results may be a little jumbled. Without using a *Sort-Object* or specifying the Sort property on the server, the values are not guaranteed to be in order. This script takes about a minute or so on my computer.

8. To tell Active Directory we do not want any size limit, specify the SizeLimit property as 0. We can do this by using the Item method of the properties collection on the Command object. This line of code is shown here:

   ```
   $objCommand.Properties.item("Size Limit") = 0
   ```

9. To make the script a bit more efficient, change the script to perform an asynchronous query (synchronous being the default). This will reduce the network bandwidth consumed and will even out the processor load on your server. To do this, declare a variable called *$blnTrue* and set it equal to the Boolean type. Assign the value −1 to it. Place this code just under the line that creates the *$strQuery* variable. This line of code is shown here:

   ```
   $blnTrue = [bool]-1
   ```

10. Under the line of code that sets the size limit, use the Item method of the properties collection to assign the value true to the asynchronous property of the Command object.

Use the Boolean value you created and stored in the *$blnTrue* variable. This line of code is shown here:

```
$objCommand.Properties.item("Asynchronous") = $blnTrue
```

11. Save and run your script. You should see the script run perhaps a little faster because it is doing an asynchronous query. If your script does not run properly, compare your script with the OneStepFurtherQueryUsers.ps1 script.
12. To clean up after this lab, run the Delete2000Users.ps1 script. It will delete the 2000 users we created at the beginning of the exercise.
13. This concludes this one step further exercise.

Chapter 8 Quick Reference

To	Do This
Make an ADO connection into Active Directory	Use the ADsDSOObject provider with ADO to talk to Active Directory
Perform an Active Directory query	Use the Field object to hold attribute data
Tell ADO search to cache results on the client side of the connection	Use the "Cache results" property
Directly query a Global Catalog (GC) server	Use GC:// in your connection moniker, instead of using LDAP://, as shown here: `GC://`
Directly query a specific server in Active Directory	Use LDAP:// in your connection moniker, followed by a trailing backslash (/), as shown here: `LDAP://London/`
Query for multiple attributes in Active Directory using the LDAP dialect	Open a set of parentheses. Inside the set of parentheses, type your attribute name and value for each of the attributes you wish to query. Enclose them in parentheses. At the beginning of the expression between the first two sets of parentheses, use the ampersand (&) operator, as shown here: `(&(objectCategory=computer)(name=london))`
Use server side sorting when using the SQL dialect	Use the order by parameter followed by either the ASC or the DESC keyword, as shown here: `'user'order by sn DESC`
Return more than 1000 objects from an Active Directory ADO query	Turn on paging by specifying the PageSize property on the Command object, and supply a value for SizeLimit property
Connect to Active Directory using alternative credentials	Specify the User ID and Password properties on the Connection object