|  |
| --- |
|  |

This sample works only with SQL Server 2005 and 2008 with AMO Version 10 installed. It will not work with any version of SQL Server earlier than SQL Server 2005.

The **ascmd** command-line utility enables a database administrator to execute an XMLA script, Multidimensional Expressions (MDX) query, or Data Mining Extensions (DMX) statement against an instance of Microsoft SQL Server Analysis Services. This command-line utility contains functionality for Analysis Services that resembles the **sqlcmd** utility included with SQL Server. For more information, see the topic **sqlcmd Utility** in SQL Server. The execution results of the script, query, or statement can be stored in a file together with relevant SQL Server Profiler trace information.

|  |
| --- |
| **Note:** |
| You cannot use the **ascmd** command-line utility to create a local cube because this functionality is embedded in the MSOLAP OLE DB for OLAP provider, which is not used by the **ascmd** command-line utility. |

# Scenarios

The following scenarios give examples of using the **ascmd** command-line utility.

### Processing a Partition from a Third-Party Tool

A database administrator must process partitions and dimensions as part of a nightly extract, transform, and load (ETL) process. The ETL tool is not a SQL Server tool and the database administrator cannot use SQL Server Agent’s built-in support of XMLA scripts or run a SQL Server Integration Services package. The database administrator wants an automated solution that uses the third-party tool. The solution is a command-line utility to run an XMLA script. The utility is then called from the third-party tool. The database administrator downloads and compiles the **ascmd** command-line utility sample. After compilation, the database administrator can use the **ascmd** command-line utility to execute XMLA scripts that process partitions and dimensions or to attach and detach databases.

### Backing Up an OLAP Database from a Third-Party Tool

Another database administrator at the same company is required to automate the back-up of an Analysis Services database. Again, because the scheduling software that the company is using is not a SQL Server tool, the task has to be run from the command line. The database administrator generates the appropriate XMLA script (using SQL Server Management Studio). Then, the third-party scheduling software uses the **ascmd** command-line utility to run the XMLA script to back up the OLAP database.

### Using XMLA During an Installation

A developer for an independent software vendor is required to integrate the execution of an XMLA script directly into the installation of the firm's product. The developer must run an XMLA script and retrieve status (and trace events) to know that the Analysis Services database was created correctly. The developer can do this by using the **ascmd** command-line utility.

# Languages

* C#, the language that **ascmd** is coded in.
* Batch file commands, which start the **ascmd** command-line utility.

# Prerequisites

To effectively use the **ascmd** command-line utility, you should have installed some or all the software listed in the following table.

Analysis Services

An instance of Analysis Services must be installed and running, because the **ascmd** command-line utilityis used to connect to an instance of Analysis Services and execute MDX queries, XMLA scripts, and DMX statements.

SQL Server Management Studio and Business Intelligence Development Studio

These two work environments provide supporting infrastructure for you to complete any task related to Analysis Services. For any given task, you can approach implementation either through the user interface or programmatically.

Analysis Management Objects (AMO)

AMO, version 10, is required to execute the **ascmd** command-line utility on a computer that does not have Analysis Services 2008 installed. AMO, version 10, is installed with the SQL Server 2008 client tools and can also be installed from the SQL Server Feature Pack, which can be downloaded from the [Microsoft Download Center](http://go.microsoft.com/fwlink/?linkid=11899) at http://www.microsoft.com/downloads.

.NET Framework 2.0 or later

The .NET Framework 2.0 or later is required for the **ascmd** command-line utility to run. It can be downloaded from the [Microsoft Download Center](http://go.microsoft.com/fwlink/?linkid=11899) at http://www.microsoft.com/downloads.

Microsoft Visual Studio 2008

We recommended that you use Visual Studio 2008 when you are building or customizing the **ascmd** sample application.

The Adventure Works DW database included with SQL Server

The Adventure Works DW database and the Adventure Works AS project are useful for experimenting with the **ascmd** command-line utility. You can download these samples from the **Microsoft SQL Server Samples and Community Projects** Web site at [http://www.codepex.com/sqlserversamples](http://go.microsoft.com/fwlink/?LinkId=85384).

# Arguments

The following arguments are supported at the command line for **ascmd**.

**–U** *login\_id*

Is the user login ID, which is case-insensitive.

|  |
| --- |
| **Note:** |
| the use of *login\_id* is different for **sqlcmd** and **ascmd**. In **sqlcmd**, *login\_id* represents a SQL Server login; for **ascmd** it represents a Microsoft Windows login. |

For TCP/IP access, Analysis Services only supports trusted connections. If the **–U** parameter is specified (together with the matching password using the **–P** parameter), the **ascmd** command-line utility logs on to the Windows operating system using the specified account and then impersonates the account when it executes the XMLA script, MDX query, or DMX statement. The login ID must be in the form *<domain>*\*<username>,* and the domain must be specified. If **–U** is not specified, authentication is based on the Windows account of the user who is running the **ascmd** command-line utility.

If an http (or https) connection is specified by the **–S** parameter, the **ascmd** command-line utility does not log on to the Windows operating system. Instead, the **–U** and **–P** parameters (if present) are included as part of the connect string to the Internet Information Services (IIS) server. Depending on how IIS is configured, the **–U** and **–P** parameters can be used for basic authentication. For more information about the “UID” connect string parameter, see **AdomdConnection** Class in SQL Server Books Online.

**–P** *password*

Is a user-specified matching password to the **–U** parameter. If the **–U** parameter is specified and the **–P** parameter is not specified, the password is assumed to be blank (an empty, zero-length string). If the **–P** parameter is specified and the **–U** parameter is not, the **–P** parameter is ignored.

|  |
| --- |
| **Security Note:** |
| Do not use a blank password. Use a strong password. For more information, see **Strong Passwords** in SQL Server Books Online.  The **–P** parameter password is stored as clear text in the script, query, or statement file. It will be visible to anyone who can see the computer monitor or read the file. If you use this feature, put Access Control Lists (ACL) on the files or use other security techniques to make sure that only trusted users can read the files. |

**–S** *server\instance* or **–S** *http[s]://server[:port]/virtualdirectory/msmdpump.dll*

Specifies the Analysis Services instance to which the **ascmd** command-line utility will connect and execute. If the **–P** parameter is not specified, the **ascmd** command-line utility connects to the default instance of Analysis Services on the local computer that is running TCP (connecting to localhost) and executes the XMLA script, MDX query, or DMX statement.

**–d** *database*

Specifies the database against which an MDX query or DMX statement will execute. The **–d** parameter is ignored when the **ascmd** command-line utility executes an XMLA script, because the database name is embedded in the XMLA script.

**–t** *query-timeout*

Specifies the number of seconds before the execution of an XMLA script, MDX query, or DMX statement times out. The **ascmd** command-line utility adds the **TIMEOUT =** *<query-timeout>* clause to the connect string.

**–tc** *connect-timeout*

Specifies the number of seconds before the **ascmd** connection to the Analysis Services instance times out. The **ascmd** command-line utility adds the **CONNECT TIMEOUT =** *<connect-timeout>* clause to the connect string.

**–i** *input-file*

Identifies the file that contains the XMLA script, MDX query, or DMX statement. You must specify a value for either the **–i** or the **–Q** parameter when you use the **ascmd** command-line utility. If you specify no **–i** or**–Q** parameter, or specify both of these parameters, an error is generated.

|  |
| --- |
| **Note:** |
| Similar to the **sqlcmd** command-line utility, the **ascmd** command-line utility can handle multiple input file at a time. Each time you specify –i <file>, the **ascmd** command-line utility simply concatenates the contents together with a GO command between them for multiple batches. |

The input file specified with either the **–i** or the **–Q** parameter must be a valid XML structure and special characters must be HTML-encoded. For example, when you use an ampersand (&) in your text, it must be encoded as &amp;. So [Product].&1922] will be encoded as [Product].&amp;[1922]. Likewise, a less-than sign (<) must be encoded as &lt;, a greater-than sign (>) as &gt;, and double quotation marks (") as &quot;. This is important for MDX queries and DMX statements because the syntax of member keys uses the ampersand character (&). See Scenario 14 for an example of an input file.

|  |
| --- |
| **Note:** |
| If the input text does not start with a valid XMLA command such as <Statement> or <Create> (see the full list later in this document), then the **ascmd** command-line utility assumes that the text is a <Statement> and HTML encodes the text for you and wraps it in a <Statement> … </Statement> XML element tag. This is performed as a convenience so that executing MDX queries and DMX statements are easier. However, you can also use <Statement> elements and write the HTML text yourself. Any valid XMLA command is accepted by the **ascmd** command-line utility. |

An input file can contain multiple batches, separated by GO commands. Each batch in an input file can contain an XMLA script, an MDX query, or a DMX statement. Each GO command must appear on a single line. When a GO command is found, the system sends the input in front of the GO command to the server. An implied GO command is at the end of the input stream. The generated output file is formatted by wrapping the returned XML streams with a <multiple-batches> element. See Scenario 13 for an example of an input file that contains multiple batches.

Each batch executes and succeeds or fails in its own right. The return status of each batch is recorded in the output file, which you must parse to determine the success or failure of each batch.

**–o** *output-file | NUL | NUL:filename*

Identifies the file that receives (in XML) the results of the XMLA script or the cellset return by the MDX query or DMX statement. If the specified file already exists, the existing file is automatically overwritten. File names that contain spaces must be enclosed in quotation marks (""). If the file name is not valid, an error message is generated, and the **ascmd** command-line utility exits.

The **ascmd** command-line utility does not support concurrent writing of multiple **ascmd** processes to the same file; if this is tried, the file output will be corrupted or incorrect.

If the specified output file is *NUL* or *NUL:filename*, the execution results are discarded unless the **–T** parameter is used to specify a trace file, in which case the execution results are stored in the trace file. Specifying a *NUL* output file and with the **–T** parameter is most useful when specifying a *Duration* trace level with the **–Tl** parameter.

For example, you could create a series of MDX queries and execute them with the **ascmd** command-line utility, ignore the output (which might be very large), record the query durations into a trace file, and then load the query duration values in the trace file into a database. This lets you evaluate performance variations over time. Alternatively, you could use the *Duration-result* trace level with the **–Tl** parameter to include both the duration and the execution results in the trace file.

|  |
| --- |
| **Note:** |
| The **ascmd** command-line utility supports international encoding. Input and output files use UTF-8 encoding with byte-order markers enabled. If your text editor does not support UTF-8 and you have international characters in your MDX query, XMLA script, or DMX statement, you can use Notepad to convert the input file into UTF-8 format. To convert the input file to UTF-8, open the file in Notepad, on the **File** menu select **Save As**, and in the **Encoding** box, select **UTF-8**. You can now use the file with the **–i** parameter. Output and trace files (**–o** and **–T**) are always written with UTF-8 encoding and byte-order markers to ensure that Unicode characters are preserved. |

**–T** *trace-file*

Identifies a file that receives Analysis Services trace events from the **ascmd** command-line utility executing the XMLA script, MDX query, or DMX statement. If the file already exists, it is automatically overwritten (except for the trace files that are created by using the **–Tl** *Duration* and **–Tl** *Duration-result* parameter settings). File names that contain spaces must be enclosed in quotation marks (""). If the file name is not valid, an error message is generated, and the **ascmd** command-line utility exits.

The **ascmd** command-line utility does not support concurrent writing of multiple **ascmd** processes to the same file; if this is tried, the file output will be corrupted or incorrect. If the **–T** parameter is not specified, the trace output is not captured and the **–Tf**, **–Tl**, **Td** and **–Tt** parameters are ignored.

|  |
| --- |
| **Note:** |
| the **–T** parameter is unavailable when you are using http or https access. You must use an ordinary client/server connection, by specifying the **–S** parameter. |

**–xc** *extended-connect-string*

Specifies an extended connect string that is inserted directly into the connect string, without any value checking. The string should not contain any leading or trailing semi-colons (;). In the following example, the extended connect string changes the network packet size used between the server and the **ascmd** process from 4096 to 16384. It also requests that the client locale be set to en-US (US English):

-xc "Packet Size=16384;LocaleIdentifier=1033"

The default for the **ascmd** command-line utility is not to add any extended connect string information. Although many of the options of the **ascmd** command-line utility can be implemented as extended connect string setting (for example by setting **Database=***<database name>* directly), we recommend that you use the standard **ascmd** options when you can and only use extended connect string settings when you have no other mechanism available.

**–Tf** *text | csv*

Specifies the file format for the **–T** parameter (if this parameter is specified). The default value is *csv*. The available options are as follows:

* For text, the file is written in a text format. Examples of the format are as follows:   
    
  <current time> <event-class>.<event-subclass>, [name=value]
* For *csv*, the file is written in comma-separated format. The default column delimiter is **|** (pipe, or vertical bar); use the **–Td** parameter to change the default delimiter for a csv file. The first line in the file specifies column headings for the values.

**–Td** *delim-char*

Specifies a single character as the trace file delimiter when you specify csv as the format for the trace file that use the **–Tf** parameter. Default is **|** (pipe, or vertical bar).

**–Tt** *trace-timeout*

Specifies the number of seconds the Analysis Services engine waits before ending the trace (if you specify the **–T** parameter). The trace is considered finished if no trace messages have been recorded during the specified time period. The default trace time-out value is 5 seconds.

**–Tl** *trace-level*

Specifies what data is collected and recorded in the trace file. This parameter has the following five possible values:

* *High* – records all trace events - this is the default setting.
* *Medium* – records all trace events except the ProgressReportCurrent and Notification events.
* *Low* – records only those trace events that contain "End" or "Error" in the event.
* *Duration* – records no trace events, but instead determines the duration of the execution of the script, query, or statement by the **ascmd** process. Writes a single entry into the trace file that includes the current time, duration, execution text, database, and server name.
* *Duration-result* – records the same information as the Duration setting and also records the result of the execution in the last column of the trace file.

|  |
| --- |
| **Note:** |
| The trace files generated with the *Duration* and *Duration-result* settings are not overwritten with each execution (as is the case with trace files generated with the *High*, *Medium,* and *Low* settings). Instead, with the *Duration* and *Duration-result* settings, if an existing trace file exists, it is opened and new values are appended to the end of the file. If the trace file does not already exist, it is created. |

**–Q** *"cmdline query or script"*

Specifies the actual script, query, or statement directly on the command line instead of in a file.

|  |
| --- |
| **Note:** |
| The **sqlcmd** command-line utility supports an additional ways to specify the input query (using the **–q** parameter). Unfortunately, because that option reads from sysinput, you cannot write it unless you add more language constructs. For example, **sqlcmd** uses “go” and “exit” to control sysinput commands. This additional way to specify query input is not supported by the **ascmd** command-line utility. |

**–v** var=value...

Specifies additional scripting variables. Each variable is a var = value pair. If the value contains embedded spaces or control characters, it must be enclosed in double-quotation marks (").For the following example,

-v maxparallel=4 option= "degree of freedom"

You can specify zero, one, or more than one var = value pairs.

**–ThinkTimeMin** *sec*

Specifies the minimum amount of think time between batches. Think time is a random number of milliseconds waited between batches. See Scenario 14 for an example of this argument.

**–ThinkTimeMax** *sec*

Specifies the maximum amount of think time between batches. Think time is a random number of milliseconds waited between batches. See Scenario 14 for an example of this argument.

**–ConnectWaitMin** *sec*

Specifies the minimum amount of time waited before making the initial connection. Connect time is a random number of milliseconds between before making the initial connection. See Scenario 14 for an example of this argument.

**–ConnectWaitMax** *sec*

Specifies the maximum amount of time waited before making the initial connection. Connect time is a random number of milliseconds between before making the initial connection. See Scenario 14 for an example of this argument.

**–oResultStat** *statistics-output-file*

Specifies the name for the statistics output file. This output file will hold the statistics for each input file specified. For each file, this statistics output file will contain the following information: Number of Errors, Number of Rows, Number of Columns, Number of Cells Total, Number of NonEmpty Cells, Number of Empty Cells, and Number of Cells with Errors. See Scenario 14 for an example of this argument.

**–NoResultStatHeader**

Disables the column header line at the start of the statistics output file. Specify this option (which takes no parameter) if you are combining the statistics output files from multiple ascmd runs and do not want to manually eliminate the header line.

**–RunInfo** *run-info-string*

Specifies run information to be included in the statistics output file. By default, this string is blank, but if you are building an infrastructure around the use of ascmd, then you might specify the run information to label different run numbers or include the data/time that the run was done. It is up to you. This information can also be set using the ASCMDRUNINFO environment variable. If you use that approach then all executions of ascmd will have the same run information.

**–RandomSeed** *seed-int*

Specifies the value for the random seed. The new format for the input batch allows for random member substitutions into MDX queries. This allows you to write a script and won’t be hitting warm caches because you are executing identical queries across multiple ascmd runs in parallel. Each of the ascmd instances inserts random members into the query, so you get the effect of cold caches. See Scenario 13 for an example of this argument.

**–-Xf** *exit-file*

Specifies the name for the exit file. The idea of the “exit file” is that each time the **ascmd** command-line utility executes a batch, it looks to see if the exit file exists. If so, it does not execute the remaining batches – it just exits. This is our mechanism for stopping a large multi-user test early. In the “driver” batch file that is controlling lots of ascmd processes running in parallel, the exit file might be the results file that the driver collects. As each run completes, the driver batch file starts saving the statistics to the ‘exit file’ and once that file exists, the ascmd batches start stopping themselves. See Scenarios 12 and 13 for examples of this argument.

**–?** or **/?**

Displays the syntax summary of the **ascmd** command-line utility options.

# Key Encryption and Building the Sample

If you have not already created a strong name key file, generate the key file using the following instructions.

#### To generate a strong name key file

1. Open a Microsoft Visual Studio 2008 command prompt. Click **Start**, point to **All Programs**, point to **Microsoft SQL Server 2008**, point to **Visual Studio Tools** and then click **Visual Studio Command Prompt**.
2. Use the change directory command (CD) to change the current directory of the command prompt window to the folder where the samples are installed.

|  |
| --- |
| **Note:** |
| If the default installation location was used, the samples are located in <system\_drive>:\Program Files\Microsoft SQL Server\100\Samples. |

1. At the command prompt, run the following command to generate the key file:

sn -k SampleKey.snk

|  |
| --- |
| **Important:** |
| For more information about the strong-name key pair, see "Security Briefs: Strong Names and Security in the .NET Framework" in the .NET Development Center on MSDN. |

#### To build the sample

After generating the strong name key file, build the sample using the following instructions.

1. From the **File | Open** menu, click **Project** and open the solution file ascmd.sln.
2. From the **Build** menu, click **Build ASCMD**.

|  |
| --- |
| **Note:** |
| Microsoft Visual Studio is fully supported on x86 and x64-based computers, but is not supported on Itanium-based computers. As soon as the **ascmd** command-line utility is compiled, the **ascmd** command-line utility can be executed on any x86, x64, or Itanium-based computer. |

It is a best practice to compile the appropriate version of the **ascmd** command-line utility because performance might be decreased when you execute 32-bit code on a 64-bit computer.

|  |
| --- |
| **Note:** |
| If you are compiling the **ascmd** command-line utility on a computer whose architecture differs from the target computer (for example, compiling the **ascmd** command-line utility on a 32-bit computer that uses either the *x64* or *Itanium* parameter value), you will receive three warning messages that indicate that three separate system DLLs are not available ("…targets a different processor"). This is typical and expected. After you compile the **ascmd** command-line utility, copy the compiled executable to your target server and execute it from the target server (where the appropriate DLLs are available). |

# Using Scripting and Environment Variables

The **ascmd** command-line utility supports system-reserved and user-defined scripting variables that you can use in XMLA scripts, MDX queries, and DMS statements. Values for these variables can be populated by specifying values for environment variables or by specifying values for command-line parameters.

The following rules apply to user-defined scripting variables and environment variables:

* A variable can contain any number of lowercase characters, uppercase characters, digits, dashes (-), or underscores (\_).
* A variable cannot contain embedded characters or control characters, for example CR, LF, TAB.

### System-Reserved Scripting Variables

System-reserved scripting variables are scripting variables that are defined by the **ascmd** command-line utility to hold the values associated with each command-line parameter. In some cases, environment variables can also be used to hold the values for these system-reserved scripting variables. For system-reserved scripting variables that can be populated or derived from both environment variables and command-line parameters, the value specified for the command-line parameter (if it is specified) overwrites any specified environment variable value.

The following table describes the system-reserved scripting variables, the associated command-line parameters, and where applicable, the associated environment variables.

|  |  |  |
| --- | --- | --- |
| **Note:** | | |
| There are three system-reserved scripting variables that can only be set by using a command-line parameter (the **–i**, **–o**, and **–T** parameters). There is no corresponding ASCMD environment variable that you can use to populate the system-reserved scripting variable that corresponds to those three command-line parameters. | | |
| **System-Reserved Scripting Variable** | | **Parameter** | **Environment Variable (if any)** | |
| **ASCMDUSER** | | **–U** | **ASCMDUSER** | |
| **ASCMDDOMAIN** | | **–U** | **ASCMDUSER** | |
| **ASCMDPASSWORD** | | **–P** | **ASCMDPASSWORD** | |
| **ASCMDSERVER** | | **–S** | **ASCMDSERVER** | |
| **ASCMDINSTANCE** | | **–S** | **ASCMDSERVER** | |
| **ASCMDHTTPCONNECTION** | | **–S** | **ASCMDSERVER** | |
| **ASCMDDBNAME** | | –**d** | **ASCMDDBNAME** | |
| **ASCMDINPUTFILE** | | **–i** |  | |
| **ASCMDOUTPUTFILE** | | **–o** |  | |
| **ASCMDQUERYTIMEOUT** | | **–t** | **ASCMDQUERYTIMEOUT** | |
| **ASCMDCONNECTTIMEOUT** | | **–tc** | **ASCMDCONNECTTIMEOUT** | |
| **ASCMDTRACEFILE** | | **–T** |  | |
| **ASCMDTRACEFORMAT** | | **–Tf** | **ASCMDTRACEFORMAT** | |
| **ASCMETRACEDELIM** | | **–Td** | **ASCMDTRACEDELIM** | |
| **ASCMDTRACELEVEL** | | **–Tl** | **ASCMDTRACELEVEL** | |
| **ASCMDTRACETIMEOUT** | | **–Tt** | **ASCMDTRACETIMEOUT** | |
| **ASCMDEXTENDEDCONNECTION** | | **–xc** | **ASCMDEXTENDEDCONNECTSTRING** | |
| **ASCMDOUTPUTRESULTSTATFILE** | | **–oResultStat** |  | |
| **ASCMDRANDOMSEED** | | **–RandomSeed** | **ASCMDRANDOMSEED** | |
| **ASCMDTHINKTIMEMIN** | | **–ThinkTimeMin** | **ASCMDTHINKTIMEMIN** | |
| **ASCMDTHINKTIMEMAX** | | **–ThinkTimeMax** | **ASCMDTHINKTIMEMAX** | |
| **ASCMDCONNECTWAITMIN** | | **–ConnectWaitMin** | **ASCMDCONNECTWAITMIN** | |
| **ASCMDCONNECTWAITMAX** | | **–ConnectWaitMax** | **ASCMDCONNECTWAITMAX** | |
| **ASCMDEXITFILE** | | **–Xf** | **ASCMDEXITFILE** | |
| **ASCMDRUNINFO** | | **-RunInfo** | **ASCMDRUNINFO** | |

Notice that in some cases in the previous table, multiple system-reserved scripting variables are derived from a single parameter or environment variable. In the following example, three system-reserved scripting variables are derived from the ASCMDSERVER environment variable setting.

* C:\>SET ASCMDSERVER=http://myserver/my\_virtual\_dir/msmdpump.dll

The previous SET statement specifying a value for the ASCMDSERVER environment variable sets the following values for the following three system-reserved scripting variables:

* ASCMDSERVER="http://myserver/my\_virtual\_dir/msmdpump.dll"
* ASCMDINSTANCE=""
* ASCMDHTTPCONNECTION="true"

In a following example, the same three system-reserved scripting variables are populated with different values by using a different SET statement:

* C:\>SET ASCMDSERVER=myserver\myinstance

The previous SET statement specifying a value for the ASCMDSERVER environment variable sets values for the following three system-reserved scripting variables:

* ASCMDSERVER="myserver"
* ASCMDINSTANCE="myinstance"
* ASCMDHTTPCONNECTION="false"

#### Using System-Reserved Scripting Variables at the Command Prompt

If an environment variable exists that matches a system-reserved scripting variable (matching is case-insensitive), the environment variable’s value is used as the default value for the system-reserved scripting variable and for the associated command-line parameter. For example, you can use the following SET statement was executed to set the ASCMDDBNAME environment variable:

* C:\>SET ASCMDDBNAME="Adventure Works DW"

In this case, “Adventure Works DW” will be used as the default database (**–d** parameter) when you execute the **ascmd** command-line utility (unless you specify a different value at the command-line).

#### Using System-Reserved Scripting Variables in Scripts, Queries, or Statements

System-defined scripting variables can also be used in an XMLA script, an MDX query, or a DMX statement. The following examples illustrate sample command-line invocations of the **ascmd** command-line utility that use scripting variables. More examples appear later in this document to illustrate usage scenarios.

* C:\>ascmd -S *<server name>* -i process.xmla -v cube=*<CubeID>*

##### process.xmla (simplified)

|  |
| --- |
|  |
| <Batch>  <Parallel>  <Process>  <Object>  <DatabaseID>$(ASCMDDBNAME)</DatabaseID>  <CubeID>($CUBE)</CubeID>  . . .  </Process>  </Parallel>  </Batch> |

### User-Defined Scripting Variables

A user-defined scripting variable is a scripting variable that is defined by using the **–v** parameter at the command line, or is defined as an environment variable. When the **ascmd** command-line utility encounters a variable in an XMLA script, an MDX query, or a DMX statement, and the variable has not been populated by using the **–v** parameter, the utility checks for an environment variable of the same name and uses that variable's value. If the **ascmd** command-line utility does not find a matching environment variable, the scripting variable is eliminated by replacing it with a blank string ("").

The following rules apply to user-defined scripting variables defined by using the **–v** parameter at the command line:

* Leading and trailing spaces are removed from the “value” section of a variable.
* The variable cannot start with the string “ascmd”.

# Using MDX, XMLA, and DMX in Input Files

The **ascmd** command-line utility supports the execution of MDX queries, XMLA scripts, and DMX statements within input files. The input script that you pass to the **ascmd** command-line utility is actually an XMLA Command element.

Command elements are as follows:

* Alter
* Attach
* Backup
* Batch
* BeginTransaction
* Cancel
* ClearCache
* CommitTransaction
* Create
* Delete
* Detach
* DesignAggregations
* Drop
* Insert
* Lock
* MergePartitions
* NotifyTableChange
* Process
* Restore
* RollbackTransaction
* Statement (used to execute MDX queries and DMX statements)
* Subscribe
* Synchronize
* Unlock
* Update
* UpdateCells

To perform commands on more than one object at a time, use the <Batch> command. To execute for MDX queries and DMX statements, use the <Statement> command. For more information, see **Command Element (XMLA)** in SQL Server Books Online. The following examples show how to structure MDX queries, DMX statements, and XMLA scripts.

|  |
| --- |
| **Important:** |
| Like all XML structures, commands are case sensitive. Therefore, for example, you must enclose all MDX queries in <Statement> …. </Statement> tags and the command must be “Statement”, it cannot be “statement” or “STATEMENT”. |

In addition to XMLA Commands, the **ascmd** command-line utility can also be used to execute custom XMLA requests to execute almost any request that can be expressed in XMLA. For example, the **ascmd** command-line utility can be used to issue either of the following XMLA requests:

* Discover XMLA requests to query Analysis Services metadata. This metadata includes information about the following:
  + Objects stored in an Analysis Services database, such as the cubes defined on the server; and
  + Resources being used, such as the connections that are open on the server.
* Execute requests that perform Commands but modify them by specifying a Property List and a Parameters List. An example of this kind of request is provided later in this document **–** see the Execute Example.

If the input text is not formatted as an XMLA Command, a Discover request, or an Execute request, the **ascmd** command-line utility assumes that the input text is a MDX query or DMX statement. In this case, the **ascmd** command-line utility HTML encodes the text and wraps a <Statement> … </Statement> element around it and processes it as an XMLA Command. This enables you to easily enter a MDX query or DMX statement. See Scenario 1 "Querying an Analysis Services Cube" later in this document for an example of how to use this capability.

### MDX Example:

|  |
| --- |
|  |
| <Statement>  SELECT NON EMPTY  [Employees].Members ON ROWS,  [Measures].[Internet Gross Profit] ON COLUMNS  FROM [Adventure Works]  </Statement> |

This example uses an MDX query in an XMLA Statement to return the Internet Gross Profit measure for each member of the Employees attribute hierarchy that is not empty from the Adventure Works cube.

### DMX Example:

|  |
| --- |
|  |
| <Statement>  ALTER MINING STRUCTURE [Bike Buyer]  ADD MINING MODEL [Decision Tree]  (  [Customer Key],  [Age],  [Bike Buyer] PREDICT,  [Commute Distance],  [Education],  [Gender],  [House Owner Flag],  [Marital Status],  [Number Cars Owned],  [Number Children At Home],  [Occupation],  [Region],  [Total Children],  [Yearly Income]  ) USING Microsoft\_Decision\_Trees  WITH DRILLTHROUGH  </Statement> |

This example uses a DMX query in an XMLA Statement change the [Bike Buyer] mining structure by adding a new mining model.

### XMLA Example:

|  |
| --- |
|  |
| <Batch xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">  <Parallel>  <Process xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  <Object>  <DatabaseID>Adventure Works DW</DatabaseID>  <CubeID>Adventure Works DW</CubeID>  <MeasureGroupID>Fact Internet Sales 1</MeasureGroupID>  <PartitionID>Internet\_Sales\_2001</PartitionID>  </Object>  <Type>ProcessFull</Type>  <WriteBackTableCreation>UseExisting</WriteBackTableCreation>  </Process>  </Parallel>  </Batch> |

This example uses an XMLA Statement to fully process the Internet\_Sales\_2001 partition.

### Discover Example:

|  |
| --- |
|  |
| <Discover xmlns="urn:schemas-microsoft-com:xml-analysis">  <RequestType>MDSCHEMA\_CUBES</RequestType>  <Restrictions>  <RestrictionList>  <CATALOG\_NAME>Adventure Works DW</CATALOG\_NAME>  </RestrictionList>  </Restrictions>  <Properties>  <PropertyList>  <Catalog>Adventure Works DW</Catalog>  <Format>Tabular</Format>  </PropertyList>  </Properties>  </Discover> |

This example uses an XMLA Discover request to return what cubes are available in the Adventure Works DW database. Because Perspectives are returned to applications as if they were cubes, the returned data actually includes both cubes and perspectives.

### Execute Example:

|  |
| --- |
|  |
| <Execute xmlns="urn:schemas-microsoft-com:xml-analysis">  <Command>  <Statement>  SELECT [Measures].MEMBERS ON COLUMNS FROM [Adventure Works]  </Statement>  </Command>  <Properties>  <PropertyList>  <Catalog>Adventure Works DW</Catalog>  <Format>Tabular</Format>  <AxisFormat>ClusterFormat</AxisFormat>  </PropertyList>  </Properties>  </Execute> |

This example uses an MDX query in an XMLA Statement. However, notice that the Property List part of the XMLA request specifies that the return format is Tabular instead of Multidimensional. The multidimensional format is the default for an XMLA Statement command. Because the return format is in tabular (rowset) format, the output file could be used by an application that understands xsd flattened rowsets instead of a cellset, and the flattened rowset could be more easily loaded into a SQL relational database because it is now formatted as a table.

# ASCMD Scenario Examples

The following scenarios demonstrate uses of the **ascmd** command-line utility.

### Scenario 1: Querying an Analysis Services Cube

In this scenario, you create an input file that contains an MDX query (the query.mdx file) that contains user-defined scripting variable (cube) in the MDX query. You then call this input file from the **ascmd** command-line utility and specify a value for this variable at the command-line by using the **–v** parameter.

##### query.mdx file:

Format 1:

|  |
| --- |
|  |
| <Statement>  /\* THIS IS AN MDX COMMENT \*/  SELECT [Measures].[Internet Sales Amount] ON COLUMNS  FROM $(cube)  WHERE [Customer].[Country].&amp;[United States]  </Statement> |

Format 2:

|  |
| --- |
|  |
| /\* THIS IS AN MDX COMMENT \*/  SELECT [Measures].[Internet Sales Amount] ON COLUMNS  FROM $(cube)  WHERE [Customer].[Country].&[United States] |

##### Command-line example:

**C:\>ascmd -S myserver -d "Adventure Works DW" -i query.mdx -o result.xml -v cube="[Adventure Works]"**

Notice that using Format 1, the key for the United States is handled by replacing the MDX "&" (which indicates that it is the member key and not the name) with &amp; (as required for HTML encoding) and that the <Statement> element is specified. Notice that using Format 2, neither the HTML encoding nor the <Statement> element is needed. This is because the input text does not start with a valid XMLA command.The **ascmd** command-line utility therefore assumes that the input text is a Statement, automatically HTML encodes the input, and wraps it in a <Statement> element before execution.

### Scenario 2: Backing Up a Database in an Untrusted Domain

In this scenario, you back up a database on a server in an untrusted domain by using the **ascmd** command-line utility. Because the database is in an untrusted domain, this scenario requires http access. In this scenario, the remote server (called "myserver") has both Internet Information Services (IIS) and Analysis Services running, and has an IIS virtual directory named "olapadmin", which is configured to use BASIC authentication. Additionally, the remote server has a local account called "olapadmin" with appropriate backup permissions. You specify the database name, access method, username, password, and backup file at the command line by using **ascmd** command-line parameters, and specify an XMLA input file (backup.xmla) that contains the scripting variables for the database and backup file.

##### backup.xmla file:

|  |
| --- |
|  |
| <Backup xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">  <Object>  <DatabaseID>$(ascmddbname)</DatabaseID>  </Object>  <File>$(backupfile).abf</File>  </Backup> |

##### Command-line example:

**C:\>ascmd -S https://myserver/msolap90/msmdpump.dll -U myserver\olapadmin -P #1PWD -d "Adventure Works DW" -i backup.xmla -v backupfile="AdvWorks"**

Notice that in the command-line example, https is used so that the password is encrypted when it is sent over the network to the remote server.

### Scenario 3: Processing Multiple Partitions

In this scenario, you process multiple partitions by using the **ascmd** command-line utility. You use scripting variables in the XMLA processing script (process.xmla) to specify the degree of parallelism, the database and cube names, and the process type. This XMLA script also demonstrates the use of comments in an XMLA script. When you call the process.xmla processing script from the **ascmd** command-line utility, you specify the server and database name, an output file for XMLA results, a trace file for trace events, the trace level, and the degree of parallelism in a batch bat (process.bat). The trace file will contain the same events and information as SQL Server Profiler would return if an administrator was monitoring the system during the processing.

##### process.xmla file:

|  |
| --- |
|  |
| <Batch xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">  <Parallel maxparallel="$(MAXPARALLEL)">  <!-- SEE ABOVE FOR HOW MANY PARITIONS PROCESSED IN PARALLEL -->  <Process xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  <Object>  <DatabaseID>$(ASCMDDBNAME)</DatabaseID>  <CubeID>$(ASCMDDBNAME)</CubeID>  <!-- Just so happens CubeID=DatabaseID=Database name :-) -->  <MeasureGroupID>Fact Internet Sales 1</MeasureGroupID>  <PartitionID>Internet\_Sales\_2001</PartitionID>  </Object>  <Type>$(PROCESSTYPE)</Type>  <WriteBackTableCreation>UseExisting</WriteBackTableCreation>  </Process>  <Process xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  <Object>  <DatabaseID>$(ASCMDDBNAME)</DatabaseID>  <CubeID>$(ASCMDDBNAME)</CubeID>  <MeasureGroupID>Fact Internet Sales 1</MeasureGroupID>  <PartitionID>Internet\_Sales\_2002</PartitionID>  </Object>  <Type>$(PROCESSTYPE)</Type>  <WriteBackTableCreation>UseExisting</WriteBackTableCreation>  </Process>  <Process xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  <Object>  <DatabaseID>$(ASCMDDBNAME)</DatabaseID>  <CubeID>$(ASCMDDBNAME)</CubeID>  <MeasureGroupID>Fact Internet Sales 1</MeasureGroupID>  <PartitionID>Internet\_Sales\_2004</PartitionID>  </Object>  <Type>$(PROCESSTYPE)</Type>  <WriteBackTableCreation>UseExisting</WriteBackTableCreation>  </Process>  <Process xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  <Object>  <DatabaseID>$(ASCMDDBNAME)</DatabaseID>  <CubeID>$(ASCMDDBNAME)</CubeID>  <MeasureGroupID>Fact Internet Sales 1</MeasureGroupID>  <PartitionID>Internet\_Sales\_2003</PartitionID>  </Object>  <Type>$(PROCESSTYPE)</Type>  <WriteBackTableCreation>UseExisting</WriteBackTableCreation>  </Process>  </Parallel>  </Batch> |

##### process.bat file:

|  |
| --- |
|  |
| @echo off  call :generate-timestamp  ascmd -S myserver -d "Adventure Works DW" -i process.xmla  -o process.xml -T process-%timestamp%.csv -Tl medium  -v maxparallel=4 processtype=ProcessFull  if ERRORLEVEL 1 goto errseen  goto :EOF  :errseen  echo \*\* Error seen in processing  goto :EOF  :generate-timestamp  set now\_date=%date%  set now\_time=%time%  set now\_Year=%now\_date:~10,4%  set now\_Month=%now\_date:~4,2%  set now\_Day=%now\_date:~7,2%  set now\_Hour=%now\_time:~0,2%  set now\_Min=%now\_time:~3,2%  if "%now\_Hour:~0,1%"==" " set now\_Hour=0%now\_Hour:~1,1%  set timestamp=%now\_year%%now\_month%%now\_day%\_%now\_hour%%now\_min%  goto :EOF |

Notice that the batch file uses a timestamp in the output file so that multiple runs can be recorded at the same time.

### Scenario 4: Creating a New Database on a Server

In this scenario, you use the **ascmd** command-line utility to call an XMLA script file (create.xmla) to create a new database on a server. The database name is defined in the XMLA script using a user-defined scripting variable, and value for this variable is defined at the command line using the **–v** parameter.

##### create.xmla file:

The file was created from SQL Server Management Studio. To create your own file, right-click the database and from the **Script** menu, click **Create**.

|  |
| --- |
|  |
| <Create xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">  <ObjectDefinition>  <Database xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  <ID>$(dbname)</ID>  <Name>$(dbname)</Name>  <Description>A Unified Dimensional Model that encompasses the Adventure Works data warehouse.</Description>  <Language>1033</Language>  <Collation>Latin1\_General\_CI\_AS</Collation>  <DataSourceImpersonationInfo>  <ImpersonationMode>Default</ImpersonationMode>  </DataSourceImpersonationInfo>  <Dimensions>  <Dimension>  <ID>Dim Promotion</ID>  <Name>Promotion</Name>  <Annotations>  . . . |

##### Command-line example:

**C:\>ascmd -S myserver -i create.xmla -v dbname="My Adventure Works DW"**

In the previous XMLA script, you could also use scripting variables to configure objects such as the connect string to a data source, the server and database name that are used in the data source, or field names in the data source view.

### Scenario 5: Creating a Cache Warmer Application

In this scenario, you use a batch file (cache\_warmer.bat) to call the **ascmd** command-line utility to call several MDX queries that warm the Analysis Services data cache. For example, you might call this batch file by using SQL Server Agent daily at 2:00 A.M. or after your night batch load. In the batch file, you set environment variables for the server, database, and cube names. Because the server and database names specified as environment variables exactly match the names of system-reserved scripting variables, they become the default values for the **–S** and **–d** command-line parameters. The user-defined scripting variable for the cube name is used in all the MDX queries.

##### query1.mdx file:

Files: query1.mdx to query6.mdx in the format of query1.txt

|  |
| --- |
|  |
| <Statement>  SELECT [Measures].[Internet Sales Amount] ON COLUMNS  FROM $(cube)  WHERE [Customer].[Country].&amp;[United States]  </Statement> |

Create additional query files by replacing [United States] with the other countries in Adventure Works: [Australia], [Canada], [France], [Germany], or [United Kingdom].

##### cache\_warmer.bat file:

|  |
| --- |
|  |
| set ascmdserver=myserver  set ascmddbname=Adventure Works DW  set cube=[Adventure Works]  set QUERYDIR=..\queries  set OUTPUTDIR=..\queries  echo -------------------------  set f=  for %%f in (%QUERYDIR%\\*.mdx) do (  call :query %%f  if ERRORLEVEL 1 goto :EOF  )  echo -------------------------  echo Done.  goto :EOF  :query  echo Query: %1  echo ---------  ascmd -T %OUTPUTDIR%\querylog.txt -Tl duration  -Tf text -o %OUTPUTDIR%\%~n1.xml -i %1  echo Errorlevel: %ERRORLEVEL%  echo -------------------------  if ERRORLEVEL 1 goto :errseen  goto :EOF  :errseen  echo -------------------------  echo \*\*\*\*\*\*  echo \*\*\*\*\*\* ERROR SEEN \*\*\*\*\*\*  echo \*\*\*\*\*\* Exiting \*\*\*\*\*\*  goto :EOF |

### Scenario 6: Creating a Validation Procedure

In this scenario, you use the **ascmd** command-line utility to call several MDX query files (similar to the previous scenario) at the end of a nightly ETL run. You use the **–Tl** duration parameter to record the duration of each MDX query into a trace file together with directing the MDX script output to a nul file (**–o** *NUL*). You could also use the **–Tl** duration parameter together with recording the execution results into a trace log. Use of the **ascmd** command-line utility in this manner lets you track the length of time required for each MDX query, and to compare these results daily to ensure that values in the same range are being returned. If duration results for a given day are significantly out of range, this might indicate that the results of the ETL run have to be backed out.

##### Command-line example:

**C:\>ascmd -i %queryfile% -o NUL -T querylog.csv -Tl duration**

### Scenario 7: Automating the Building and Training of a Data Mining Model

In this scenario, you use the **ascmd** command-line utility to call a series of DMX statement as follows:

* A DMX statement that creates a mining structure (Bike Buyer Structure.DMX) and uses the environment variables to set the server and database names.
* A DMX statement (Clustering\_Model.dmx) that adds a clustering mining model to the structure.
* A DMX statement (DT\_Model.dmx) that adds a decision tree mining model to the structure.
* A DMX statement (Process Bike Buyer Structure.dmx) to process the mining structure and mining models.

When you have the mining structure in place, you can use the **ascmd** command-line utility to call several DMX statements that query the mining structure using different mining models.

#### Create the mining structure

##### Bike Buyer Structure.dmx file:

|  |
| --- |
|  |
| <Statement>  CREATE MINING STRUCTURE [Bike Buyer]  (  [Customer Key] LONG KEY,  [Age]LONG DISCRETIZED(Automatic,10),  [Bike Buyer] LONG DISCRETE,  [Commute Distance] TEXT DISCRETE,  [Education] TEXT DISCRETE,  [Gender] TEXT DISCRETE,  [House Owner Flag] TEXT DISCRETE,  [Marital Status] TEXT DISCRETE,  [Number Cars Owned]LONG DISCRETE,  [Number Children At Home]LONG DISCRETE,  [Occupation] TEXT DISCRETE,  [Region] TEXT DISCRETE,  [Total Children]LONG DISCRETE,  [Yearly Income] DOUBLE CONTINUOUS  )  </Statement> |

Command-line example:

**C:\>set ascmdserver=myserver**

**C:\>set ascmddbname=Adventure Works DW**

**C:\>ascmd -i "Bike Buyer Structure.dmx"**

#### Add a clustering mining model to the structure

##### Clustering\_Model.dmx file:

|  |
| --- |
|  |
| <Statement>  ALTER MINING STRUCTURE [Bike Buyer]  ADD MINING MODEL [Clustering]  USING Microsoft\_Clustering  </Statement> |

Command-line example:

**C:\>ascmd -i "Clustering\_Model.dmx"**

#### Add a decision tree mining model to the structure

##### DT\_Model.dmx file

|  |
| --- |
|  |
| <Statement>  ALTER MINING STRUCTURE [Bike Buyer]  ADD MINING MODEL [Decision Tree]  (  [Customer Key],  [Age],  [Bike Buyer] PREDICT,  [Commute Distance],  [Education],  [Gender],  [House Owner Flag],  [Marital Status],  [Number Cars Owned],  [Number Children At Home],  [Occupation],  [Region],  [Total Children],  [Yearly Income]  ) USING Microsoft\_Decision\_Trees  WITH DRILLTHROUGH  </Statement> |

##### Command-line example:

**C:\>ascmd -i "DT\_Model.dmx"**

#### Process the mining structure and mining models

##### Process Bike Buyer Structure.dmx file:

|  |
| --- |
|  |
| <Statement>  INSERT INTO MINING STRUCTURE [Bike Buyer]  (  [Customer Key],  [Age],  [Bike Buyer],  [Commute Distance],  [Education],  [Gender],  [House Owner Flag],  [Marital Status],  [Number Cars Owned],  [Number Children At Home],  [Occupation],  [Region],  [Total Children],  [Yearly Income]  )  OPENQUERY([$(ASCMDDBNAME)],  'SELECT CustomerKey, Age, BikeBuyer,  CommuteDistance,EnglishEducation,  Gender,HouseOwnerFlag,MaritalStatus,  NumberCarsOwned,NumberChildrenAtHome,  EnglishOccupation,Region,TotalChildren,  YearlyIncome  FROM dbo.vTargetMail')  </Statement> |

##### Command-line example:

**C:\>ascmd -i "DT\_Model.dmx"**

#### Query the structure using the decision tree mining model

##### SELECT\_DRILLTHROUGH.dmx file:

|  |
| --- |
|  |
| <Statement>  SELECT \*  FROM [Decision Tree].CASES  </Statement> |

##### BATCH\_PREDICTION.dmx file:

|  |
| --- |
|  |
| <Statement>  SELECT  TOP 10  t.[LastName],  t.[FirstName],  [Decision Tree].[Bike Buyer],  PredictProbability([Bike Buyer])  From  [Decision Tree]  PREDICTION JOIN  OPENQUERY([$(ASCMDDBNAME)],  'SELECT  [LastName],  [FirstName],  [MaritalStatus],  [Gender],  [YearlyIncome],  [TotalChildren],  [NumberChildrenAtHome],  [Education],  [Occupation],  [HouseOwnerFlag],  [NumberCarsOwned]  FROM  [dbo].[ProspectiveBuyer]  ') AS t  ON  [Decision Tree].[Marital Status] = t.[MaritalStatus] AND  [Decision Tree].[Gender] = t.[Gender] AND  [Decision Tree].[Yearly Income] = t.[YearlyIncome] AND  [Decision Tree].[Total Children] = t.[TotalChildren] AND  [Decision Tree].[Number Children At Home] = t.[NumberChildrenAtHome] AND  [Decision Tree].[Education] = t.[Education] AND  [Decision Tree].[Occupation] = t.[Occupation] AND  [Decision Tree].[House Owner Flag] = t.[HouseOwnerFlag] AND  [Decision Tree].[Number Cars Owned] = t.[NumberCarsOwned]  WHERE [Decision Tree].[Bike Buyer] =1  ORDER BY PredictProbability([Bike Buyer]) DESC  </Statement> |

##### SELECT\_DISCRETE.dmx file:

|  |
| --- |
|  |
| <Statement>  SELECT DISTINCT [Bike Buyer]  FROM [Decision Tree]  </Statement> |

##### Command-line example:

**C:\>ascmd -i SELECT\_DRILLTHROUGH.dmx**

**C:\>ascmd -i BATCH\_PERDICTION.dmx**

**C:\>ascmd -i SELECT\_DISCRETE.dmx**

# Scenario 8: Clearing the Analysis Services Data Cache

In this scenario, you use the **ascmd** command-line utility to call an XMLA script (ClearCache.xmla) that clears the Analysis Services data cache between performance runs when doing performance studies. The ClearCache.xmla file contains scripting variables for the database and cube names. This XMLA script is called by a batch file (ClearCache.bat) that specifies the server and instance name, the database name, the input file name, the output file name, and the cube name.

##### ClearCache.xmla file:

|  |
| --- |
|  |
| <Batch xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">  <ClearCache>  <Object>  <DatabaseID>$(ASCMDDBNAME)</DatabaseID>  <CubeID>$(CUBE)</CubeID>  </Object>  </ClearCache>  </Batch> |

##### ClearCache.bat file:

|  |
| --- |
|  |
| @echo off  ascmd -S myserver\myinstance -d "Adventure Works DW" -i ClearCache.xmla  -o ClearCache.xml -v cube="Adventure Works DW"  if ERRORLEVEL 1 goto :errseen  goto :EOF  :errseen  echo \*\*\*\* Error seen \*\*\*\*  echo \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  goto :EOF |

### Scenario 9: Determining who is currently connected to your server

In this scenario, you use the **ascmd** command-line utility to retrieve the list of active connections on the server. An application might use this information to delay processing until specific users are disconnected, or to send an e-mail to the operators if anyone has a current connection (other than the connection for the nightly batch run).

##### connections.xmla file:

|  |
| --- |
|  |
| <Discover xmlns="urn:schemas-microsoft-com:xml-analysis">  <RequestType>DISCOVER\_CONNECTIONS</RequestType>  <Restrictions />  <Properties>  <PropertyList>  <Content>Data</Content> <!-- Only the data; no schema -->  </PropertyList>  </Properties>  </Discover> |

##### Command-line example:

**C:\>ascmd -S myserver -i connections.xmla -o current\_connections.xml**

### Scenario 10: Is a partition processed and, if so, when was it last processed

In this scenario, you use the **ascmd** command-line utility to determine whether a partition has been processed and when it was processed. This information can easily be retrieved because it is stored as a property of the partition object. Thus a DISCOVER\_XML\_METADATA request can be used to retrieve this information.

##### Processed\_partition.xmla file:

|  |
| --- |
|  |
| <Discover xmlns="urn:schemas-microsoft-com:xml-analysis">  <RequestType>DISCOVER\_XML\_METADATA</RequestType>  <Restrictions>  <RestrictionList>  <DatabaseID>$(DatabaseID)</DatabaseID>  <CubeID>$(CubeID)</CubeID>  <MeasureGroupID>$(MeasureGroupID)</MeasureGroupID>  <PartitionID>$(PartitionID)</PartitionID>  <!-- Ask for just this object referenced -->  <ObjectExpansion>ReferenceOnly</ObjectExpansion>  </RestrictionList>  </Restrictions>  <Properties>  <PropertyList>  <Content>Data</Content> <!-- Only the data; no schema -->  </PropertyList>  </Properties>  </Discover> |

##### Command-line example:

**C:\>ascmd -S myserver -i processed\_partition.xmla -o processed\_partition\_data.xml**

### Scenario 11: Recording Server or Local Cube Metadata

In this scenario, you use the **ascmd** command-line utility to record the entire server or local cube metadata into an XML file. This DISCOVER\_XML\_METADATA request can be used to retrieve this information.

##### Metadata.xmla file:

|  |
| --- |
|  |
| <Discover xmlns="urn:schemas-microsoft-com:xml-analysis">  <RequestType>DISCOVER\_XML\_METADATA</RequestType>  <Restrictions>  <RestrictionList>  <ObjectExpansion>ExpandFull</ObjectExpansion>  </RestrictionList>  </Restrictions>  <Properties>  <PropertyList>  <Format>Tabular</Format>  </PropertyList>  </Properties>  </Discover> |

##### Command-line example:

**C:\>ascmd -S myserver -i metadata.xmla -o server\_metadata.xml**

**C:\>ascmd -S c:\mylocalcube.cub -i metadata.xmla -o localcube\_metadata.xml**

### Scenario 12: Recording Server Configuration Properties

In this scenario, you use the **ascmd** command-line utility to record server configuration properties. This DISCOVER\_XML\_METADATA request can be used to retrieve this information.

##### Server\_properties.xmla file:

|  |
| --- |
|  |
| <Discover xmlns="urn:schemas-microsoft-com:xml-analysis">  <RequestType>DISCOVER\_XML\_METADATA</RequestType>  <Restrictions>  <RestrictionList>  <ObjectExpansion>ObjectProperties</ObjectExpansion>  </RestrictionList>  </Restrictions>  <Properties>  <PropertyList>  <Format>Tabular</Format>  </PropertyList>  </Properties>  </Discover> |

##### Command-line example:

**C:\>ascmd -S myserver -i serverproperties.xmla -o myserver\_properties.xml**

# Scenario 13: Using the GO Command to Perform a Writeback Operation

In this scenario, you use the **ascmd** command-line utility to break the writeback into two pieces: change the data and then commit it. Writeback requires the GO command because the two MDX statements required for a writeback operation (the Update Cube and the Commit Transaction statements) must be issued one after the other in the same transaction. MDX does not support issuing them in the same batch.

For this scenario, you have to modify the Adventure Works DW database to support writeback. The existing database does not currently have an example of a cube that supports writeback. To create and verify a cube that supports writeback, follow these steps:

1. **To define a new cube called “Writeback”**
2. Open Business Intelligence Development Studio.
3. On the File menu, point to **Open**, and then click **Analysis Services Database**.
4. In the **Connect to Database** dialog box, type your server name in the **Server** text box, select the **Adventure Works DW** database in the **Database** list, and then click **OK**.
5. In the Solution Explorer pane, right-click **Cubes** and then click **New Cube**.
6. In the Cube Wizard, click **Next** on the **Welcome to the Cube Wizard** page, select **Build the cube using a data source**, clear the **Auto build** check box, and then click **Next**.
7. Select **Adventure Works DW** in the **Available data source views** list on the **Select Data Source View** page and then click **Next**.
8. On the **Identify Fact and Dimension Tables** page, select the **Fact** check box for the **FactSalesQuota** table and the **Dimension** check box for the **dbo.DimTime** and **dbo.DimEmployee** table, and then click **Next**.
9. On the **Review Shared Dimensions** page, select **Date** and **Employee** in the **Available Dimensions** list, click **>** to add these dimensions to the **Cube dimensions** list and then click **Next**.
10. On the **Select Measures** page, clear the **Fact Sales Quota** check box, select the **Sales Amount Quota** check box, and then click **Next**.
11. On the **Completing the Wizard** page, change the cube name to **Writeback** and then click **Finish**.
12. **To enable writeback for the Fact Sales Quota measure group**
13. In cube designer, select the **Partitions** tab.
14. Right-click **Fact Sales Quota** partition in the partition list and then click **Writeback Settings**.
15. In the **Enable Writeback - Fact Sales Quota** dialog box, review the default writeback table name and then click **OK** to create this table and enable writeback for this partition.
16. Notice that two partitions now appear: one for the fact table; one for the writeback table.
17. **To process the Writeback cube**
18. Right-click **Writeback** in the **Cubes** node in Solution Explorer and click **Process**.
19. Click **Yes** when you are prompted to save changes.
20. In the **Process Cube - Writeback** dialog box, click **Run**.
21. If you expand the processing commands, you will see the CREATE TABLE SQL statement that is used to create the writeback relational table.
22. When processing is completed, verify that the process succeeded in the **Status** box, and then click **Close**.
23. Click Close again to close the **Process Partition - WriteTable\_Fact Sales Quota** dialog box.
24. Close Business Intelligence Development Studio.
25. **To verify that writeback is working**
26. Open SQL Server Management Studio.
27. Connect to your server, and then in **Object Explorer**, expand **Databases**, right-click **Adventure Works DW**, point to **New Query** and then click **MDX**.
28. In the MDX query window, execute the following MDX query to return the current sales quote for Q1FY2002 and Stephen Y. Jiang:

|  |
| --- |
|  |
| /\* Employee 272 is [Stephen Y. Jiang]\*/  SELECT [Measures].[Sales Amount Quota] ON COLUMNS  FROM [Writeback]  WHERE ([Employee].[Employee].[Stephen Y. Jiang],[Date].[Calendar].[Calendar Quarter].[Q1 CY 2002]) |

1. Modify the cell to return $2,200 by issuing the following MDX statement:

|  |
| --- |
|  |
| UPDATE CUBE [Writeback]  SET ([Employee].[Employee].[Stephen Y. Jiang],  [Date].[Calendar].[Calendar Quarter].[Q1 CY 2002]) = 2200 |

1. Commit the transaction by executing the following MDX statement:

|  |
| --- |
|  |
| COMMIT TRANSACTION |

1. At this point, you can examine the “dbo.WriteTable\_Fact Sales Quota” table in the Adventure Works DW relational database to see what writeback has actually done for the cell. If you do, you will notice that it is the delta (-88800) that is written to this relational table. The original fact table is unchanged.

##### writeback.mdx file:

|  |
| --- |
|  |
| /\* What is the existing value? \*/  SELECT [Measures].[Sales Amount Quota] ON COLUMNS  FROM [Writeback]  WHERE ([Employee].[Employee].&[272],  [Date].[Calendar].[Calendar Quarter].&[2002]&[1])  GO  /\* Update the cube with a new value \*/  UPDATE CUBE [Writeback]  SET ([Employee].[Employee].&[272],  [Date].[Calendar].[Calendar Quarter].&[2002]&[1]) = 33000 /\* some different value \*/  GO  /\* Commit it \*/  Commit Transaction  GO  /\* See what the updated value is \*/  SELECT [Measures].[Sales Amount Quota] ON COLUMNS  FROM [Writeback]  WHERE ([Employee].[Employee].&[272],  [Date].[Calendar].[Calendar Quarter].&[2002]&[1])  GO |

##### Command-line example:

**C:\>ascmd -S myserver -d "Adventure Works DW" -i writeback.mdx -o writeback\_result.xml -v cube="[Writeback]"**

##### Writeback\_result.xml:

|  |
| --- |
|  |
| <multiple-batches>     <return xmlns="urn:schemas-microsoft-com:xml-analysis">        <root xmlns= . . .>           <...metadata about the result set...>  <CellData xmlns="urn:schemas-microsoft-com:xml-analysis:mddataset">    <Cell CellOrdinal="0">       <Value xsi:type="xsd:double" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">0</Value>       <FmtValue>2200</FmtValue>    </Cell>  </CellData>        </root>     </return>     <return xmlns="urn:schemas-microsoft-com:xml-analysis">        <root xmlns="urn:schemas-microsoft-com:xml-analysis:empty" />     </return>     <return xmlns="urn:schemas-microsoft-com:xml-analysis">        <root xmlns="urn:schemas-microsoft-com:xml-analysis:empty" />     </return>     <return xmlns="urn:schemas-microsoft-com:xml-analysis">        <root xmlns= . . .>           <...metadata about the result set...>  <CellData xmlns="urn:schemas-microsoft-com:xml-analysis:mddataset">    <Cell CellOrdinal="0">       <Value xsi:type="xsd:double" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">0</Value>       <FmtValue>33000</FmtValue>    </Cell>  </CellData>        </root>     </return>  </multiple-batches> |

Notice that there are two empty result sets in the middle for the UPDATE CUBE statement and the COMMIT TRANSACTION statement.

# Scenario 14: Using ASCMD to Execute Many Queries Sequentially or in Parallel

In this scenario, you use the **ascmd** command-line utility to execute many MDX queries either sequentially or in parallel. You may want to execute a query workload sequentially to individually evaluate performance statistics about each query in preparation for a stress test. For example, you may need to eliminate randomly generated queries that only hit null values and then provide a mix of simple, medium and complex queries. Thereafter, you may want to execute selected queries in parallel for a multi-user stress test. The following batch file can be used as a template for accomplishing either sequential or parallel execution.

##### WorkLoadTesting.cmd:

@echo off

setlocal

rem --+++

rem -- Run ascmd with one or more client streams with one or more queries per stream.

rem -- This launches one or more client versions of ascmd.

rem --

rem -- You specify:

rem --

rem -- NumClients = number of simulated clients. We will launch an instance of ascmd for each simulated client.

rem --

rem -- NumInputFilesPerClient = number of input files, for each client.

rem -- This is important because it allows you to change NumClients and the first 'n' clients still have exactly

rem -- the same input files.

rem --

set FDATETIME=%DATE:~12,2%%DATE:~4,2%%DATE:~7,2%\_%TIME:~0,2%%TIME:~3,2%%TIME:~6,2%

if {%date:~8,1%}=={-} set FDATETIME=%DATE:~6,2%%DATE:~9,2%%DATE:~12,2%\_%TIME:~0,2%%TIME:~3,2%%TIME:~6,2%

rem -- Parameters

set /a NumClients=5

set /a NumInputFilesPerClient=15

set ASCMDDIR=%~dp0

set SRCDIR=%~dp0MDX\_Queries

set DSTDIR=%~dp0Output\_ascmd\_%FDATETIME%

set BASECLIENTOUTFILENAME=Q

set FILENAMESEARCH=\*.xml

set DATABASE=AdventureWorksDW

rem -- Create the destination directory. (Useful if it is based on timestamp.)

if not exist "%DSTDIR%" md "%DSTDIR%"

rem -- Loop through all the files in the source directory.

rem -- We build up the list and when we have enough, we launch ascmd with the list.

set ICLIENT=1

set IFILE=0

set INPUTFILELIST=

for %%G in (%SRCDIR%\%FILENAMESEARCH%) do call :AddSingleFile %%G

rem -- Check if we did not have enough files for the clients.

rem -- Better to check first but that is more code to write.

if %ICLIENT% LEQ %NUMCLIENTS% (

@echo.

@echo ERROR: Not enough files exist for each client.

@echo.

)

goto :eof

:----------

:AddSingleFile

rem -- For files at the end that aren't used, we just ignore them.

rem -- Ideally we would break out of the calling loop.

if %ICLIENT% GTR %NUMCLIENTS% goto :eof

rem -- Append the file to the list of files for the client we are getting ready to launch.

set INPUTFILELIST=%INPUTFILELIST% -i "%1"

set /a IFILE = %IFILE% + 1

if %IFILE% LSS %NumInputFilesPerClient% goto :eof

rem -- Build up a set of other command line options.

set OTHER=

set OTHER=%OTHER% -o "%DSTDIR%\%BASECLIENTOUTFILENAME%\_%ICLIENT%.xml"

set OTHER=%OTHER% -oResultStat "%DSTDIR%\ResultStat\_%ICLIENT%.csv"

set OTHER=%OTHER% -RandomSeed %ICLIENT%

set OTHER=%OTHER% -ConnectWaitMin 5

set OTHER=%OTHER% -ConnectWaitMax 15

set OTHER=%OTHER% -ThinkTimeMin 15

set OTHER=%OTHER% -ThinkTimeMax 45

set OTHER=%OTHER% -RunInfo "Workload\_5\_Run\_1"

set OTHER=%OTHER% -NoResultStatHeader

rem -- Launch ascmd to simulate this client.

rem @echo start "RunASCMD %ICLIENT%" /MIN %ASCMDDIR%ascmd -d "%DATABASE%" %INPUTFILELIST% %OTHER% -Xf quit.txt

@echo on

start "RunASCMD %ICLIENT%" /MIN %ASCMDDIR%ascmd.exe -d "%DATABASE%" %INPUTFILELIST% %OTHER% -Xf quit.txt

@echo off

rem -- Reset the variables for the current client.

set /a ICLIENT = %ICLIENT% + 1

set IFILE=0

set INPUTFILELIST=

goto :eof

## Sequential Query Execution

You can modify and use the previous batch file to generate and execute the following command-line string that executes all MDX queries stored in XML files in a specified folder.

##### Command-line example:

**C:\StressTest>start "RunASCMD 1" /MIN ascmd.exe -d "Adventure Works 2008" -i "Stress\_Queries\QuerySet1.XML" -i "Stress\_Queries\QuerySet2.XML" -i " Stress\_Queries\QuerySet3.XML" -i "Stress\_Queries\QuerySet4.XML" -o "C:\ASCMD\Output\_ascmd\_081010\_141502\Q\_1.xml" -oResultStat "C:\ASCMD\Output\_ ascmd\_081010\_140146\** **ResultStat\_1.csv" -ConnectWaitMin 0 -ConnectWaitMax 0 -ThinkTimeMin 0 -ThinkTimeMax 0 -RunInfo "Workload\_5\_Run\_1" -Xf quit.txt**

Notice that this command-line example specifies multiple input files using multiple –i arguments, specifies the output file for the query, the output file for the query statistics, and the run name. Notice also that, since the input files specified are being executed serially, no wait times or think times are specified such that the input files are executed as quickly as Analysis Services can respond to each query. Finally, notice the exit file is specified as “quit.txt” – if this file is placed in the source folder, the execution of the string of input files will stop as soon as the currently executing query completes.

## Parallel Query Execution

You can modfy and use the previous batch file to generate and execute the following command-line strings for a specified number of query streams based on the MDX query files stored in XML (or MDX) files in a specified folder.

##### Command-line example, Stream 1:

**C:\StressTest>start "RunASCMD 1" /MIN C:\StressTest\ascmd.exe -d "Adventure Works 2008" -i "C:\StressTest\Stress\_Queries\QuerySet1.xml" -i "C:\StressTest\Stress\_Queries\QuerySet2.xml" -o "C:\ASCMD\Output\_ascmd\_081010\_141502\Q\_1.xml" -oResultStat "C:\ASCMD\Output\_ ascmd\_081010\_140146\** **ResultStat\_1.csv" -ConnectWaitMin 5 -ConnectWaitMax 15 -ThinkTimeMin 15 -ThinkTimeMax 45 -RunInfo "Workload\_5\_Run\_1" -NoResultsStatHeader -Xf quit.txt**

##### Command-line example, Stream 2:

**C:\StressTest>start "RunASCMD 2" /MIN C:\StressTest\ascmd.exe -d "Adventure Works 2008" -i "C:\StressTest\Stress\_Queries\QuerySet3.xml" -i "C:\StressTest\Stress\_Queries\QuerySet4.xml" -o "C:\ASCMD\Output\_ascmd\_081010\_141502\Q\_2.xml" -oResultStat "C:\ASCMD\Output\_ ascmd\_081010\_140146\** **ResultStat\_2.csv" -ConnectWaitMin 5 -ConnectWaitMax 15 -ThinkTimeMin 15 -ThinkTimeMax 45 -RunInfo "Workload\_5\_Run\_1" -NoResultsStatHeader -Xf quit.txt**

Notice that each command-line example specifies multiple input files using multiple –i arguments, specifies the output file for the query, and the output file for the query statistics. Note also that a run name was specified and that the output header in the CSV files was suppressed. Since each command-line example is to be executed in parallel, wait times and think times are specified such that the connect times for each connection and the execution time for each query in each stream are staggered. Finally, notice the exit file is specified as “quit.txt” for each stream – if this file is placed in the source folder, the execution of the string of input files for all streams will stop as soon as the currently executing query completes.

## Sample Input File

The following is an example of an XML input file.

##### Input file example:

<Queries>

<Query>/\* Query 1 \*/

/\* Time \*/

SELECT [Measures].[Internet Sales Amount] ON COLUMNS ,

[Date].[Calendar].[Calendar Quarter].&amp;[2003].&amp;[1] ON ROWS

FROM [Adventure Works 2008]

CELL PROPERTIES VALUE, FORMATTED\_VALUE, CELL\_ORDINAL, ACTION\_TYPE

</Query>

<Query>/\* Query 2 \*/

/\* Time \*/

SELECT [Measures].[Internet Sales Amount] ON COLUMNS ,

[Date].[Calendar].[Calendar Quarter].&amp;[2003].&amp;[2] ON ROWS

FROM [Adventure Works 2008]

CELL PROPERTIES VALUE, FORMATTED\_VALUE, CELL\_ORDINAL, ACTION\_TYPE

</Query>

</Queries>

© 2008 Microsoft Corporation. All rights reserved.