

# Windows PowerShell Research from the Point in Terms of Operational Data Analysis Subsystem Constructing

Hlib Horban<sup>1</sup>[0000-0002-6512-3576], Ihor Kandyba<sup>2</sup>[0000-0002-8589-4028],

Anzhela Boiko<sup>3</sup>[0000-0002-3449-0453], Kateryna Kirey<sup>4</sup>[0000-0002-9338-2380], Viktoriia Chorna<sup>5</sup>

<sup>1-5</sup>Petro Mohyla Black Sea National University, Mykolaiv, Ukraine  
hlib.horban@chmnu.edu.ua<sup>1</sup>, jeo2145@gmail.com<sup>2</sup>,  
anzhela.boiko@chmnu.edu.ua<sup>3</sup>, kirey.kea@gmail.com<sup>4</sup>,  
chornav2008@gmail.com<sup>5</sup>

**Abstract.** Relatively new for the Windows operating system is the PowerShell command shell, that was designed to replace the elder task automation tools, which are the cmd.exe command shell and the Windows Script Host. At the appearance of PowerShell many problems were solved, in particular, the problem of integrating the command line with objects that are supported in the operating system, and working with random data sources. The main feature of the PowerShell command shell is that all results of its commands are represented as sets of objects of certain .NET classes, because this command shell is built on the .NET platform. At the same time, the capabilities of PowerShell are not thoroughly studied. The article presents a study of this command shell in the context of an object database and simple On-Line Analytical Processing (OLAP) system building based on the described classes directly in PowerShell scripts. Main results of building an OLAP system by means of standard PowerShell commands, .NET classes, and creating classes for storing multidimensional data are also presented.

**Keywords:** Command shell, PowerShell, .NET, commandlet, class, object data model, property, connection, links, XML, dimension, measure, fact table, aggregation.

## 1 Review of the contemporary researches

With the dynamic development of the graphical interface of the Windows operating system (Windows OS), there are almost no alternative control methods for this OS based on the use of the command line and different scenarios. First of all, due to the fact that in comparison with UNIX-like operating systems, the command line of the Windows OS has always been its weak point. The main reason for this was that the main efforts of its developers were not aimed at creating a work environment for professionals, but to improve the graphical shell for more comfortable work of ordinary users. At the same time, this control model is not scalable. For example, during admin-

istration of a certain number of servers with the use of standard graphic tools, it is necessary to repeat the same sequence of actions in a certain number of times, and therefore the question arises of automation of the routine operations execution.

If in UNIX systems the automation tool is a standard shell or its modifications (bash, ksh, csh, and others), then in Windows there is no specific tool. On the other hand, there is a certain group of similar tools, which are quite different from each other. In recent versions of Windows OS, these tools are the cmd.exe command-line shell, the Windows Script Host scripting environment, and Microsoft PowerShell. Each tool has its advantages and disadvantages [1, 2].

Chronologically the first tool was the cmd.exe command line shell, which in turn expanded the capabilities of the command.com shell for MS-DOS and the very first versions of Windows, actually replacing it in versions starting with Windows NT. The advantage of cmd.exe is that it is the most universal and easy to learn tool, as well as available in all versions of Windows OS. In turn, its rather serious disadvantage is the fact that this shell does not in any way provide access to objects supported by Windows (COM, .NET, etc.).

The next step in the development of tools for task automation in Windows OS was the Windows Script Host (WSH). This tool allows you to execute scenarios (scripts) directly in the operating system and write them in complete programming languages, which by default are VBScript (script version of the VisualBasic language) and Jscript (some analog of the JavaScript language from Microsoft) [2]. In comparison with the cmd.exe shell, the benefits of WSH are that this technology has its own object model, elements of which have properties and methods that allow solving some daily tasks of the operating system administrator (for example, working with the system registry, network resources, etc.) as well as it allows you to access services of any automation servers applications, which register their objects in the OS.

However, in spite of all benefits, the WSH technology also has strong disadvantages. First of all, Windows operating system does not have complete reference information on WSH objects by default, and in the second place, WSH scenarios present a rather serious potential threat through a security perspective, because there are a great number of viruses, which use WSH for performing destructive actions. Last disadvantage can be avoided by using the encryption described in [3, 4]. Some risks associated with different types of virus attacks can be minimized by the MAS method [5].

In the early 2000s, the situation with automation tools in Windows was not good enough [1]. On the one hand, the functionality of the cmd.exe shell proved to be insufficient and from the other hand WSH scenarios proved to be overcomplicated for mid-level users and entry level administrators.

The development of a new shell for access to WMI (Windows Management Instrumentation) objects from the command line (WMI Command-line, WMIC) started in 2000. However, this tool was not sufficiently successful, because more emphasis was made not on the user's operational comfort, but on operational characteristics of WMI [1, 2].

Upon completion of the WMIC, Microsoft's experts came to the conclusion that it was possible to implement such a shell, which would not be limited to working only with WMI objects, but could have the ability to work with objects of any classes of the

.NET platform, effectively enabling the ability to use all its capabilities directly from the command line. As can be seen from the above it was the start-up of the development of a completely new task automation tool in Windows, which was named Windows PowerShell.

Windows PowerShell is relatively new. The initial version was implemented in Windows Vista operating system, and starting with Windows 7, PowerShell became its integral part. The current version of this command shell is PowerShell 5.0 [1, 2].

PowerShell itself has solved many problems emerging when automating standard Windows tasks by using cmd.exe and WSH. In particular, some of the solved problems are the work with arbitrary data sources in the command line according to the file system principle and the problem of integration of the command line with different objects of the operating system.

The main innovation of the new command shell is that the result of its work is not the text as it was in the Cmd.exe command shell or UNIX-like systems, but an object [6, 7]. Besides, it is possible to output the required values of the properties of these objects and perform their methods. Since the PowerShell itself built on the .NET platform, any object whose data is output by a particular command line is a .NET object. Again, the internal command in PowerShell is called a commandlet and is also some .NET class, which is a descendant of the Cmdlet class. This class, in its turn, is the base class for all internal commands. A large number of commandlets returns some information that is displayed on the console and represents a set of objects in a certain .NET class.

In Windows PowerShell, the names of all the commandlets consist of a verb and a noun. The verb points out the action to do and the noun in its turn points out the object of the action. For example:

- Get-Process – information output about system processes;
- Set-Location – change of the current directory;
- Get-ChildItem – output of the directory content (its child elements);
- Stop-Service – shutdown of the service.

Commandlets can have parameters and their values (arguments). The structure of parameters in all commandlets is identical, and the same parameters for different commandlets have the same names. All this allows for quite easy memorization and learning of commandlets.

In addition, PowerShell implements a well-developed alias mechanism for more convenient entering of command names. Three types of aliases are supported: names in the style of cmd.exe shell commands, names in the style of shell commands in UNIX-like operating systems, and names representing acronyms – abbreviations formed from the initial letters of words in the name of the commandlet. For example, the Get-ChildItem commandlet has three equivalent aliases: dir (the name of the equivalent command in cmd.exe), ls (the same in shells of UNIX-like operating systems) and gci (acronym of the Get-ChildItem name).

For example, let's perform this commandlet for the C directory C:\Windows:

```
PS C:\Windows> Get-ChildItem
```

The result will be:  
Directory: C:\Windows

Mode	LastWriteTime	Length	Name
d-----	7/16/2016 4:23 PM		ADFS
d-----	7/16/2016 4:23 PM		appcompat
d-----	9/12/2016 2:22 PM		AppPatch
-----			
-a----	7/16/2016 4:19 PM	10240	winhlp32.exe
-a----	7/16/2016 4:18 PM	316640	WMSysPr9.prx
-a----	7/16/2016 4:18 PM	11264	write.exe

As can be seen, this commandlet really displays the content of the directory (if you do not specify a directory, the content of the current directory will be output by default), which includes files and subdirectories. However, each line in the result represents an object of some .NET class, which has corresponding properties, values of which are displayed on the screen. In fact, these are not all properties of this class, but only those that are displayed by default. How to learn about all properties of this class and its name will be described below.

Predominantly the shells of the command line have a conveyor mechanism, the essence of which is the sequential execution of commands in such a way that the result of the previous command line output redirects to the input of the next one. Sufficiently useful property of pipelines is not to depend on the number of transmitted elements, because the pipeline operates separately for each element.

In Windows PowerShell objects pass down the pipeline, and in the cmd.exe shell and UNIX-like systems, a text stream passes down the pipeline. Such organization of pipeline operation gives the advantage, because the command, which receives the result of the previous command, analyzes it and allocates the necessary information. In the case of presentation of a result in the form of a text, it is sometimes difficult to perform its analysis, because usually output results of the commands are mainly oriented not to the convenience of the further text review, but to the convenient visual perception of the users. In the case of presenting of a result in the form of objects this problem does not arise, because the necessary information for the next command can be obtained with the help of simple access to the corresponding properties of objects.

Pipeline mechanisms often use commands that somehow process input information. Such commands are usually called filters, examples of which in PowerShell are the Where-Object, Select-Object and Group-Object commandlets.

The Where-Object commandlet sets a defined condition for object retrieval. As an example of using this commandlet, let's present information output only with respect to executable files in the C:\Windows folder (they contain .exe extensions).

```
Get-ChildItem C:\Windows | Where-Object {$_.Extension -eq  
".exe"}
```

Directory: C:\Windows

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a----	7/16/2016 4:18 PM	61440	bfsvc.exe
-a----	7/16/2016 4:18 PM	4673304	explorer.exe
-----	-----	-----	-----
-a----	7/16/2016 4:19 PM	10240	winhlp32.exe
-a----	7/16/2016 4:18 PM	11264	write.exe

Another filter in PowerShell is the `Select-Object` commandlet. It can be used to output only the properties of objects specified as arguments. In the following example, let's add this commandlet to the outputs of the previous commandlets, which receives input information down the previous pipeline from the `Where-Object` commandlet. In its turn, let's order the `Select-Object` commandlet to output only the names and sizes of the files with `.exe` extension.

```
Get-ChildItem C:\Windows | Where-Object {$_.Extension -eq ".exe"} | Select-Object Name,Length
```

Name	Length
----	-----
bfsvc.exe	61440
explorer.exe	4673304
-----	-----
winhlp32.exe	10240
write.exe	11264

With the help of PowerShell it is possible to group objects. This functionality acts using the `Group-Object` commandlet. You should specify to it one of the properties as arguments, by which the grouping will be carried out. In the following example, grouping of the content of the `C:\Windows` directory by extension is performed.

```
Get-ChildItem C:\Windows | Group-Object Extension
```

Count	Name	Group
-----	----	-----
70		{ADFS, appcompat, AppPatch, AppReadiness...}
1	.NET	{Microsoft.NET}
11	.exe	{bfsvc.exe, explorer.exe, HelpPane.exe, hh.exe...}
1	.dat	{bootstat.dat}
7	.log	{DtcInstall.log, iis.log, lsasetup.log, PFR0.log...}
1	.bin	{mib.bin}
4	.INI	{ODBC.INI, ODBCINST.INI, system.ini, win.ini}
2	.dll	{pyshellext.amd64.dll, twain_32.dll}
1	.xml	{ServerDataCenter.xml}

```
1 .prx {WMSysPr9.prx}
```

The output of the above-mentioned sequence of commandlets represents a table with three fields, which in fact is the object of the definite .NET class with three properties. The value of the Count property in each line represents the number of elements that are included into the corresponding group. In its turn, the value of the Name property represents the value of the corresponding property of the source objects that was specified as an argument for the Group-Object commandlet.

## 2 Main part

The results of these commandlets resemble the action of SQL queries for relational databases. In objects that return as a result of the certain commandlets execution, it is possible to draw an analogy with the relational table. However, in this case, this is not about relational tables after all, but about class objects, because any commandlet in PowerShell rep-represents its result in the form of certain sets of objects of a certain .NET class, data on which is simply represented in the table form.

At this, between the object and relational data models it is possible to draw corresponding analogies of concepts used in them [8, 9]. For example, the class in the object data model is represented by a table in the relational data model, but in its turn the object or instance of this class is nothing else than one row of the table, and the class property is the column of the table, respectively. However, the object data model is completely built on classes, and one of its main differences from the relational data model is that the links between objects are carried out not on fields (properties) as between the relational tables, but by references [8, 9]. In its turn, references are the same properties but not simple data types are used as their types, but entity classes. Another difference between the object data model and relational data model is that the links between objects are bidirectional.

It is worthy of note that it was impossible to create own classes in PowerShell language for a long time, though it was possible to use predefined .NET classes with the purpose to create their objects. Already in the PowerShell 5.0 version for the first time there appeared a possibility of own classes declaration that removed restrictions of PowerShell functionality as a software programming language of CLI scripts. Here is an example of the declaration of Student (a student) and Group (a group) classes, and we will create a link between them.

```
class Student
{
    [int]$id; [string]$surname
    [string]$name; [string]$midname
    [Group]$group;
}
class Group
{
    [int]$number; [string]$specialty
```

```
[Student[]]$students  
}
```

In such a way the object of the "Student" class contains a reference to the object of the "Group" class, because one student can enter only one group. However, on the other hand, a certain group consists of a plurality of students, so the "Group" class describes the property, which is an array of references to objects of the "Student" class.

There are two different methods to create a new instance of the self-described class in PowerShell: the New-Object commandlet and the static New() method that is a member of any class described in the PowerShell:

```
$s=New-Object Student  
$g=[Group]::New()
```

As the result, new instances of the "Student" and "Group" classes were created. Now we need to create relationship between them. This can be done as follows:

```
$s.Group=$g
```

In such a way with the help of the PowerShell command line scripts it is possible to describe entity classes that correspond to certain entities; link them to each other by declaration of corresponding links and to create their instances. In such a way with the help of the above structures it is possible to design a complete object database. In PowerShell there are services for XML format in which it is convenient enough to store the data which present objects. For this purpose, the Get-Content commandlet is used to read the contents of the file and present it in a corresponding format. As well the .NET classes designed to work with XML data are used.

In such a way, the possibilities of the object database building by means of the PowerShell command shell by way of description of their own entity classes have been considered herein before. Upon that analogs of queries are some PowerShell commandlets that are commonly used in pipelines and in a certain way perform filtering of objects or their properties. Such commandlets are Where-Object, Select-Object and Group-Object.

The Measure-Object commandlet accepts a corresponding set of digits as the input data, processes it, and allows you to perform aggregation functions on it to calculate the sum, the average value, and the maximum and minimum values. To perform a corresponding operation for this commandlet, you should specify the -Sum, -Average, -Minimum and -Maximum keys, respectively. As a rule this commandlet is quite convenient to use for calculation of the total size of files in a certain directory, as well as the average size of files in the directory, if necessary. As an example, we will calculate the statistical data for the C:\Windows directory with the use of the commandlet described above as the -Property key for which we will indicate the Length property, which is the file size, and also specify the keys for the calculation of all aggregated values:

```
Get-ChildItem C:\Windows | Measure-Object -Property Length -Sum  
-Average -Minimum -Maximum
```

```
Count      : 24
Average    : 299407,541666667
Sum        : 7185781
Maximum    : 4675384
Minimum    : 0
Property   : Length
```

The result of the Measure-Object commandlet implementation is presented in a quite convenient form, because it provides information about the number of elements, the corresponding aggregated value, and the property name of the objects of a certain class, upon which the calculation of aggregated values was performed. The application of this commandlet is not confined to calculating the total and average size of the file. It is quite convenient to use it in PowerShell scripting to compute the aggregated values of numeric arrays. As an example, let's declare an array of five numbers and calculate its sum, as well as average, maximum and minimum values.

```
$a=3,7,5,2,8
$a | Measure-Object -Sum -Average -Minimum -Maximum
```

```
Count      : 5
Average    : 5
Sum        : 25
Maximum    : 8
Minimum    : 2
Property   :
```

The result does not require additional comments. We can suppose that when using the above commandlets, it is quite effective to carry out calculations of aggregate functions in multidimensional arrays that are the basis of OLAP systems [10]. OLAP is a technology of operational analytical processing of data, which uses methods and means for collecting, storing and analyzing multidimensional data with the purpose to support decision-making processes.

At present OLAP systems are most often used in conjunction with relational Database Management Systems (DBMS) because the links between them are already well studied. However, after consideration of the Measure-Object commandlet and other commandlets designed for data processing in PowerShell, the question now arises of whether the OLAP-system, at least primitive, can be implemented with their help. Besides, such a system would not be bound to any DBMS.

To answer this question, let us consider the structure of OLAP systems in more detail. Their basis is multidimensional databases, which in their turn are built based on the fact and dimension tables [10]. Fact tables contain external keys that are primary keys in dimension tables and quantitative values that are called measures and usually represent the value of profit, cost, etc. At the same time, the dimension in the OLAP-systems seems to be a definite sequence of values of some parameter, which should be analyzed. It is from the fact and dimension tables a multidimensional data structure is



formed that takes the form of a hypercube, in which defined actual values of variables that represent measures, are located at the intersection of dimensions. Besides the actual values, hypercubes also store the summary data that can be calculated using the aggregation function (sum, mean value, etc.). In this way, you can calculate the summary data for one or more dimensions, and it also will contain one summary value for all dimensions at once.

Mathematically the hypercube is appropriate to represent by following sets:

1.  $D$  – a set of hypercube dimensions for a specific subject area:

$$D = \{D_1, D_2, \dots, D_i, \dots, D_n\}, \quad (1)$$

where  $D_i$  –  $i$ -dimension,  $n$  – the number of dimensions;

2.  $A$  – a set of attributes (values of elements) of hypercube dimensions:

$$A = A_1 \cup A_2 \cup \dots \cup A_i \cup \dots \cup A_n \quad (2)$$

where  $A_i$  – a set of attributes of dimension  $D_i$ , which in turn can be represented as:

$$A_i = \{A_i^1, A_i^2, \dots, A_i^k, \dots, A_i^m\}, \quad (3)$$

where  $A_i^k$  –  $k$ -attribute of  $i$ -dimension,  $m$  – the number of attributes in  $i$ -dimension;

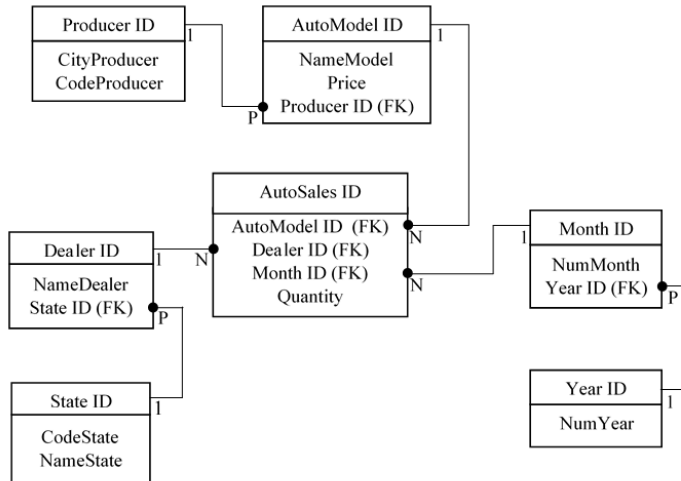
3.  $M$  – a set of values of hypercube measures:

$$M = \{M_{I_1, I_2, \dots, I_i, \dots, I_n}^1, \dots, M_{I_1, I_2, \dots, I_i, \dots, I_n}^l, \dots, M_{I_1, I_2, \dots, I_i, \dots, I_n}^z\}, \quad (4)$$

where  $I_i$  – attribute index of  $i$ - dimension,  $n$  – the number of dimensions,

$M_{I_1, I_2, \dots, I_i, \dots, I_n}^l$  –  $l$ -measure for the cube cell with  $I_1, I_2, \dots, I_i, \dots, I_n$  index,  $z$  – the number of hypercube measures.

As an example of creating an OLAP system, let us consider a database where corresponding car sales information is stored. The hypercube that is built for the above database will have three dimensions and one measure. The hypercube that will build for the above database will have three dimensions and one measure. The second dimension is the dealer information, which will have the "dealer" and "location" hierarchy levels. The third dimension is the date of sale with the "month" and "year" hierarchy levels. In its turn, the measure will be the number of vehicles of a defined model sold at specified date by specific dealer. Fig. 1 shows the corresponding Integrated DEFinition1 (IDEF1) diagram for this database.



**Fig. 1.** IDEF1 diagram of the car sales database.

For the building of such a database with the help of PowerShell, all the entities presented in the diagram above must be described in their classes. Connections between these classes will be presented as links. In this way, it is possible to connect the actual data with the dimension data, which in its turn can be connected to the different hierarchy levels. The fact table also can be presented in the form of a class; in this case it would be better called a fact class. Let us present in PowerShell language descriptions of classes, which are the entities used in the hierarchy levels of corresponding dimensions.

```

class AutoModel
{
    [int]$id
    [string]$nameModel
    [int]$price
    [Producer]$producer
}

class Producer
{
    [int]$id
    [string]$cityProducer
    [string]$codeProducer
    [AutoModel[]]$autoModels
}

```

For example, for the AutoModel class, its connection with the Producer class is described. This connection is bidirectional: the AutoModel class describes a link to an object of the Producer class, while the Producer class describes an array of links to the corresponding objects of the AutoModel class. The entities of all other dimensions and connections between different hierarchy levels in them are described in the same way.

The description of the entity that represents the actual data is slightly different. This class contains links to entity class objects that store data of a particular dimension in the most detailed way. A property that is a measure is specified separately in the class. The code of the actual data class presented below.

```
class AutoSales
{
    [int]$id
    [AutoModel]$autoModel
    [Dealer]$dealer
    [Month]$month
}
```

The most convenient way to store data in the form of classes is to store it in XML files because the XML format itself is isomorphic towards the object representation of data: the name of the class is a tag, and the names of its corresponding properties are the tag attributes. As for the preservation of object hierarchies in XML, it is easy to implement by enclosing a subordinate object into the main one. Below there is an example of how to save dimension data, representing the dealer and its location, in XML. Here, the main object is the location area, which is the higher level of the hierarchy, and the subordinate objects are dealers.

```
<state id="2" codeState="15" nameState="Mykolaiv region">
<dealer id="3" nameDealer="Velocity" state="2" />
<dealer id="4" nameDealer="Power" state="2" />
<dealer id="5" nameDealer="Supperline" state="2" />
</state>
```

For the building of an OLAP system, it is quite sufficient to create two XML-files, one of which contains dimension data, and the second one contains actual data. The following example shows a corresponding fragment of the XML file which contains the actual data on car sales:

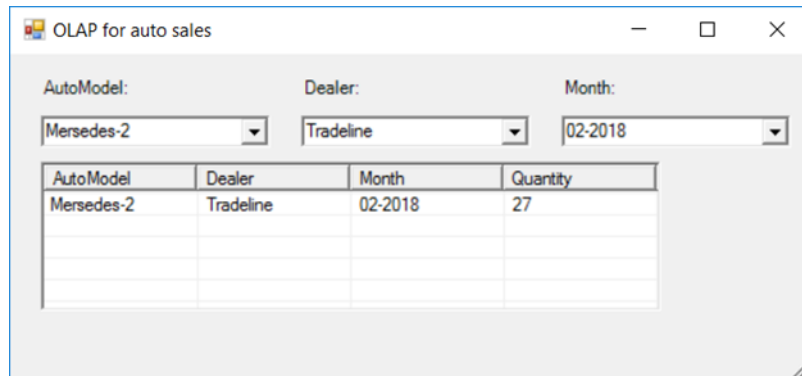
```
<root>
<AutoSel autoModel="1" dealer="1" month="1" quantity="15" />
<AutoSel autoModel="1" dealer="9" month="2" quantity="18" />
<AutoSel autoModel="1" dealer="7" month="4" quantity="11" />
<AutoSel autoModel="1" dealer="5" month="5" quantity="25" />
.....
</root>
```

Now let us consider the type of actual data processing as well as dimension data directly in PowerShell scripts. Since PowerShell allows the use of any .NET class, it is possible to use a form to display the data in a more user-friendly way. However, it is only possible to create a new form in the PowerShell script code after connecting the appropriate build. An example of creating a new form that is preceded by connecting the relevant build is given below.

```
[void] [System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")  
$factForm=New-Object System.Windows.Forms.Form
```

A new object in PowerShell is created with the help of the New-Object commandlet. In such a way, a new form is created in the presented example. Using the above commandlet, it is possible to create new objects of visual components (drop-down list, button, table, etc.) as well, and then place them on the form.

The form of the OLAP application displays dimension data as well as multidimensional data, both actual and aggregated. To do this, it contains three drop-down lists that allow the end user to select specific values for each dimension. If you want to perform aggregation for a defined dimension, the element with the value "ALL" corresponds to this action in the drop-down list. However, the main element on the form is a table which displays the actual or aggregated data.



AutoModel	Dealer	Month	Quantity
Mercedes-2	Tradeline	02-2018	27

**Fig. 2.** Example of the form image for the OLAP system with the output of defined actual data.

If no element with the value "ALL" has been selected in any of the drop-down lists, it means that the actual data that will be read from the corresponding XML file is displayed. If the end user has changed the current element in any of the three drop-down lists, a corresponding event handler will be called. Its action consists first of reading objects with actual data and then filtering them by conditions that correspond to the value of the elements in the drop-down lists selected by the user. The PowerShell code below is designed for this purpose.

```
$factXml.GetElementsByTagName("AutoSel") | Where-Object {($_.AutoModel -eq $selModel.id) -and ($_.dealer -eq $seldealer.id) -and ($_.month -eq $selmonth.id)}
```

The user can also select the element with the value "ALL" in one or several drop-down lists that means aggregation by one specific dimension or their collection. By way of example, let us give in Fig. 3 for a start the form with the summary of sales by car model.

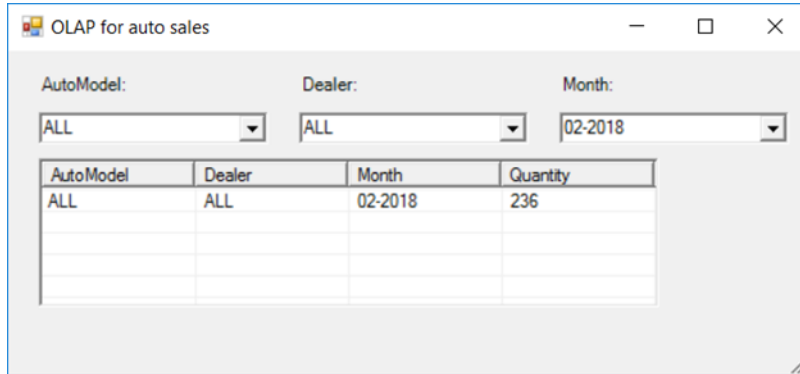
AutoModel	Dealer	Month	Quantity
ALL	Tradeline	02-2018	41

**Fig. 3.** Example of the form image with a summary data output by one dimension.

The above example summarises the car sales of all the models, available in the model database, by the Supperline dealer in December 2017. This is indicated by an element with the value "ALL" in the drop-down list that corresponds to the "AutoModel" dimension. In this case the following procedure is carried out by the program. The actual data at first is read from the XML file and then filtered by the selected dealer and month. After this operation, the obtained set of objects is saved into the \$existFact variable, which is then processed by the Measure-Object commandlet for calculation of the total value of the measure for the corresponding set of objects. Besides the sum, it is possible to find the average, minimum and maximum values of a measure, which the mentioned above commandlet allows you to do. Below there is a code in the PowerShell language, which performs the actions described above.

```
$existFact=$factXml.GetElementsByTagName("AutoSel") | Where-Object {($_.dealer -eq $seldealer.id) -and ($_.month -eq $selmonth.id)}
$sum=($existFact.quantity | Measure-Object -sum).Sum
```

If you select elements with the value "ALL" in any two drop-down lists at once, the aggregation in two dimensions will be carried out, and in this case, only one dimension will have a fixed value. Fig. 4 shows an example of summing up the number of vehicles sold in December 2017, regardless of which model they belong to and by which dealer they were sold.



**Fig. 4.** Example of the form image with the final data output by one dimension.

Calculation of the corresponding aggregated value is carried out with the help of the following program code.

```
$existFact=$factXml.GetElementsByTagName("AutoSel") | Where-Object {$_.month -eq $selmonth.id}
$sum=($existFact.quantity | Measure-Object -sum).Sum
```

In contrast to the previous example, where the aggregated value was calculated by only one dimension, only one condition is checked in the corresponding Where-Object command code (in this case, it is the correspondence of the month to the selected value). If you calculate the final value for the whole cube, the commandlet for checking the condition does not need to be applied at all. In this case, it is necessary to use only the Measure-Object commandlet, which calculates the total value for the set from absolutely all objects with actual values stored in the XML file. The program code that carries out the above action is presented below.

```
$existFact=$factXml.GetElementsByTagName("AutoSel")
$sum=($existFact.quantity | Measure-Object -sum).Sum
```

The corresponding example of the form image with the total value output is shown in Fig. 5.

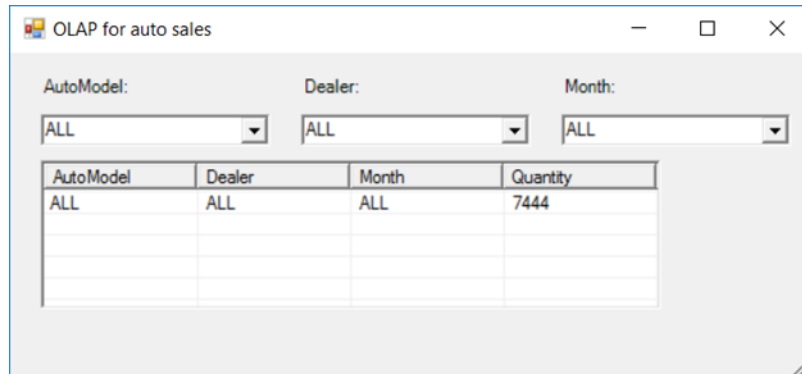


Fig. 5. The image of the form with the total value output for the whole hypercube.

Thus, by using the Where-Object and Measure-Object cmdlets you can output any data from the OLAP cube, both actual and aggregated.

### 3 Conclusions and Perspectives of Further Research

Thus, this article studies the PowerShell, which is increasingly being used in Microsoft Windows OS instead of the traditional cmd.exe command shell. Even though all its capabilities are not yet studied well enough, the conclusion can be drawn that this command shell is more functional than cmd.exe. In particular, PowerShell has a data management tool that is similar to the DBMS query. The Measure-Object commandlet was studied, which has an opportunity to process multidimensional data in the same way as in OLAP-systems, which are now increasingly used in industrial DBMS. Combining PowerShell scripts of own classes descriptions in the code became possible in version 5.0, and the use of standard commandlets opens up the prospects for designing and implementing such structures, which can serve as a substitute for databases and data storages at least in the simplest cases. In particular, the article describes the principles of creating a simple OLAP system in the PowerShell environment. At the same time, this system certainly does not claim to be universal because it is bound to a specific database. For the building of a universal OLAP system, it is necessary to design such an architecture that would allow the end-users to create their entities automatically, rather than to describe them manually with the use of PowerShell classes. All of this the system should do automatically. For implementing such actions, it is necessary to think over the structure of metadata carefully and implement it, which is a part of the further plans of the authors.

### References

1. Shepard, M., Venkatesan, C., Talaat, S., Blawat, B.: PowerShell: Automating Administrative Tasks. Packt Publishing Ltd. (2017)

2. Payette, B., Siddaway, R.: Windows PowerShell in Action. Third Edition. Manning Publications Co (2018)
3. Krainyk, Y., Perov, V., Musiyenko, M., Davydenko, Y.: Hardware-oriented turbo-product codes decoder architecture. In: 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), pp. 151-154. IEEE Press, Bucharest, Romania (2017). doi: 10.1109/IDAACS.2017.8095067
4. Krainyk, Y., Davydenko, Y., Starchenko, V.: Message-level Decoding of Error Patterns for Turbo-Product Codes. In: 39th International Conference on Electronics and Nanotechnology (ELNANO), pp. 660-663. IEEE Press, Kyiv, Ukraine (2019). doi: 10.1109/ELNANO.2019.8783849
5. Burlachenko, I., Zhuravska, I., Davydenko, Y., Savinov, V.: Vulnerabilities Analysis and Defense Based on MAS Method in Fast Dynamic Wireless Networks. In: 4th International Symposium on Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS), pp. 98-102. IEEE Press, Lviv, Ukraine (2018). doi: 10.1109/IDAACS-SWS.2018.8525692
6. Knittel, B.: Windows 7. Scripts, automation and command line. St. Petersburg (2012). (in Russian)
7. Popov, A. V.: Introduction to Windows PowerShell. St. Petersburg (2009). (in Russian)
8. Fisun, M., Horban, H.: Generation of the association rules among multidimensional data in DBMS caché environment. *Advances in Intelligent Systems and Computing*, 63-79 (2016).
9. Fisun, M., Dvoretzkyi, M., Shved, A., Davydenko, Y.: Query parsing in order to optimize distributed DB structure. In: 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), pp. 172-178. IEEE Press, Bucharest, Romania (2017). doi: 10.1109/IDAACS.2017.8095071
10. Fisun, M., Horban, H.: Implementation of the information system of the association rules generation from OLAP-cubes in the post-relational DBMS cache. In: XIth International Scientific and Technical Conference Computer Sciences and Information Technologies (CSIT), pp. 40-44. IEEE Press, Lviv, Ukraine (2016). doi: 10.1109/STC-CSIT.2016.7589864