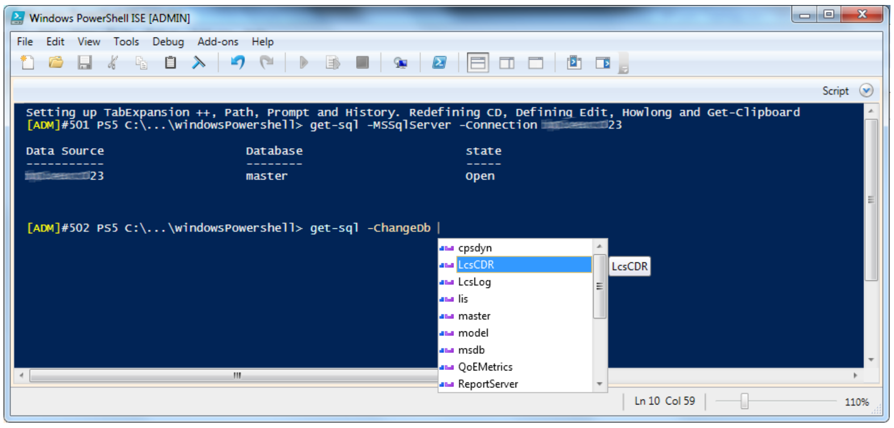
USING GET-SQL

Get-SQL makes it easier to use PowerShell to work with databases, using either ODBC or the SQL-Client for SQL Server. It connects to databases and submits SQL Statements, either taking a prepared SQL statement or building one from multiple intellisense-enabled parameters.

GETTING CONNECTED.

To start using Get-SQL you need to connect to a database as shown in the Figure 1 and Figure 3.

Get-SQL uses `-Connection` to specify a connection-string and `-Session` to name an instance of a link (if you don't specify a session name, Get-SQL will name the session "default"). If a subsequent command provides connection information for a session which already exists, a new connection will <u>only</u> be made if the `–ForceNew` parameter is specified.



```
Setting up TabExpansion ++, Path, Prompt and History. Redefining CD, Defining Edit, Howlong and Get-Clipboard
[ADM]#501 PS5 C:\...\windowsPowershell> get-sql -MSSqlServer -Connection [redacted]23

Data Source                   Database                      State
-----------                   --------                      -----
[redacted]23                  master                        Open


[ADM]#502 PS5 C:\...\windowsPowershell> get-sql -ChangeDb |
```

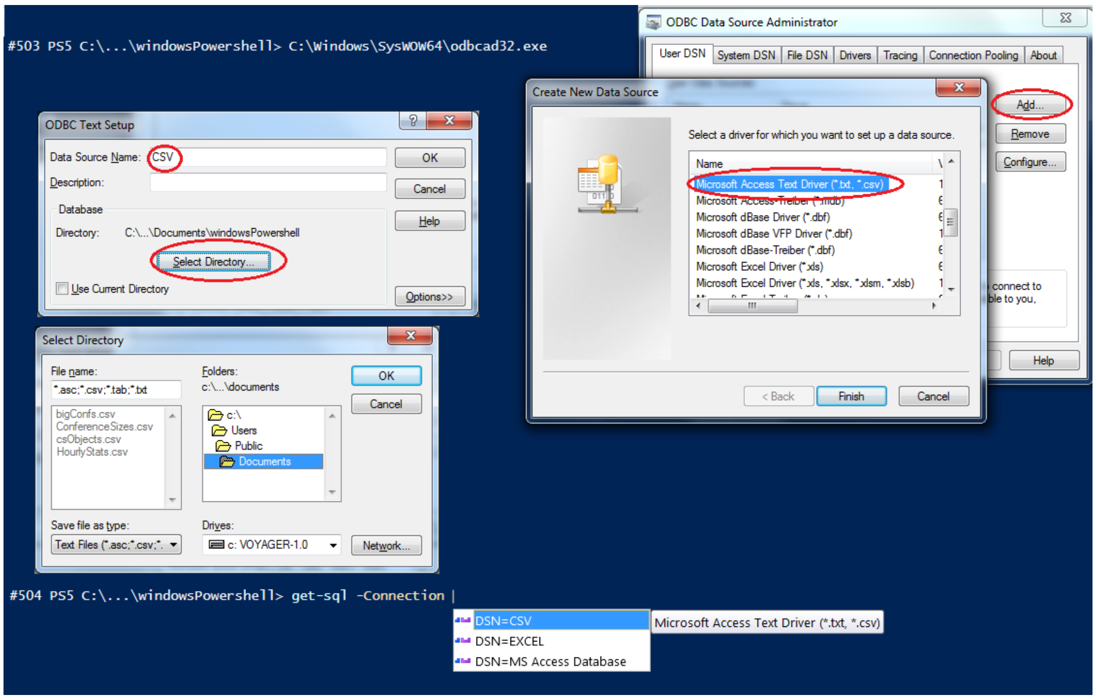
Figure 1 connecting the default session to SQL server

Figure 1 shows connecting to SQL server without naming the session, so it becomes the default session. `-MSSQLServer` tells Get-SQL to use the *SQLClient* driver. It interprets a `-Connection` parameter which does not contain an = sign, as a server name and builds a simple connection string. This string specifies a trusted connection and a timeout but it doesn't include a database name, so the SQL server driver connects to the **master** database. The second line in Figure 1 shows switching databases - intellisense is available to complete the database name.

Figure 1 shows 64-bit PowerShell (the default), but 32-bit ODBC drivers provided with 32-bit Office, *require* 32-bit PowerShell – and the 64-bit drivers from 64-bit Office only work with 64-bit PowerShell. Office provides the drivers for the following local file types:

- Access (*.mdb, *.accdb) – even if Access itself is not installed
- dBASE / Foxpro (*.dbf, *.ndx, *.mdx)
- Excel (*.xls, *.xlsx, *.xlsm, *.xlsb)
- Paradox (*.db)
- Text (*.txt, *.csv) – the text driver treats a folder as a virtual "database" and files within it as tables.

The text driver appears to work only with Data Source Names (DSNs) created in the "Data sources (ODBC)" tool (the 32-bit version if using 32-bit Office) as in Figure 2, below.


```
#503 PS5 C:\...\windowsPowershell> C:\Windows\SysWOW64\odbcad32.exe
```
ODBC Text Setup — Data Source Name: CSV; Description: ; Database Directory: C:\...\Documents\windowsPowershell; Use Current Directory

Select Directory — File name: *.asc;*.csv;*.tab;*.txt; Folders: c:\...\documents

```
bigConfs.csv
ConferenceSizes.csv
csObjects.csv
HourlyStats.csv
```
Save file as type: Text Files (*.asc;*.csv;*.; Drives: c: VOYAGER-1.0

ODBC Data Source Administrator — User DSN | System DSN | File DSN | Drivers | Tracing | Connection Pooling | About

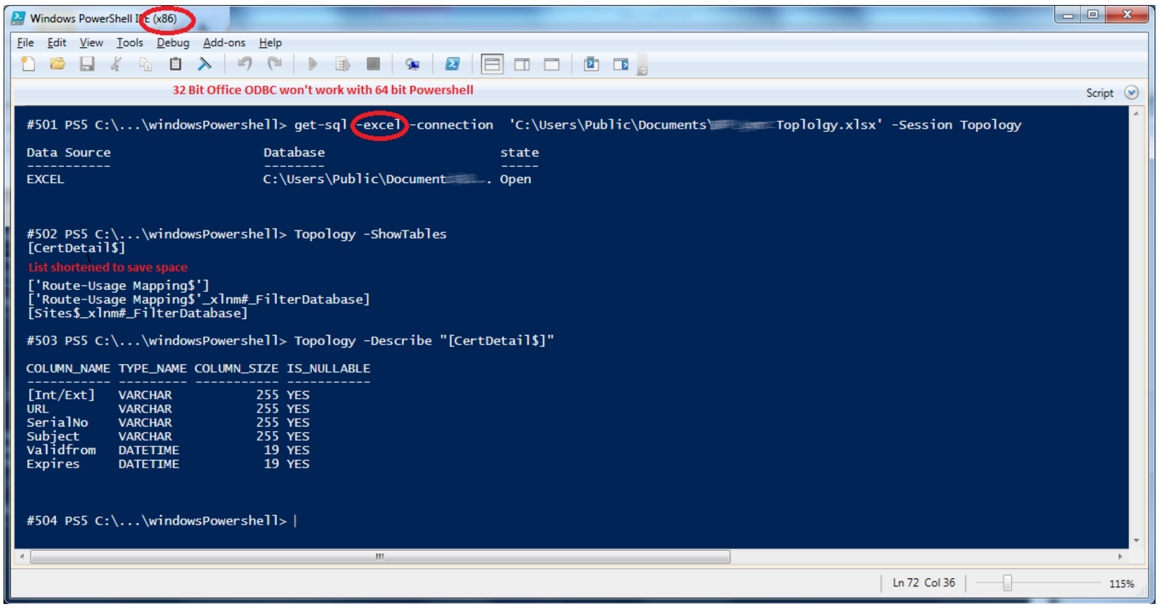
Create New Data Source — Select a driver for which you want to set up a data source.

```
Microsoft Access Text Driver (*.txt, *.csv)
Microsoft Access-Treiber (*.mdb)
Microsoft dBase Driver (*.dbf)
Microsoft dBase VFP Driver (*.dbf)
Microsoft dBase-Treiber (*.dbf)
Microsoft Excel Driver (*.xls)
Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)
```
```
#504 PS5 C:\...\windowsPowershell> get-sql -Connection |
DSN=CSV               Microsoft Access Text Driver (*.txt, *.csv)
DSN=EXCEL
DSN=MS Access Database
```

Figure 2 Setting up a text ODBC connection for CSV files

When specifying the `-connection` parameter it can be `DSN= «Name»` or a detailed connection string, like this:
`'Driver={Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)};DriverId=790;ReadOnly=0;Dbq «FileName»;'`

To save having to set-up DSNs or write connection strings in full, there are `-Excel` and `-Access` switches to tell Get-SQL to treat the `-Connection` parameter as file name (as seen in Figure 3)
If you always use the same database, you can set `$DefaultDBConnection` in your profile to the value for the `-Connection` parameter (though if it is not an ODBC connection string, you may need to specify `-SQLServer -Excel or -Access` explicitly.)


Windows PowerShell ISE (x86)

32 Bit Office ODBC won't work with 64 bit Powershell

```
#501 PS5 C:\...\windowsPowershell> get-sql -excel -connection  'C:\Users\Public\Documents\        Toplolgy.xlsx' -Session Topology

Data Source                   Database                      state
-----------                   --------                      -----
EXCEL                         C:\Users\Public\Document      . Open


#502 PS5 C:\...\windowsPowershell> Topology -ShowTables
[CertDetail$]
List shortened to save space
['Route-Usage Mapping$']
['Route-Usage Mapping$'_xlnm#_FilterDatabase]
[Sites$_xlnm#_FilterDatabase]

#503 PS5 C:\...\windowsPowershell> Topology -Describe "[CertDetail$]"

COLUMN_NAME TYPE_NAME COLUMN_SIZE IS_NULLABLE
----------- --------- ----------- -----------
[Int/Ext]   VARCHAR           255 YES
URL         VARCHAR           255 YES
SerialNo    VARCHAR           255 YES
Subject     VARCHAR           255 YES
Validfrom   DATETIME           19 YES
Expires     DATETIME           19 YES


#504 PS5 C:\...\windowsPowershell> |
```

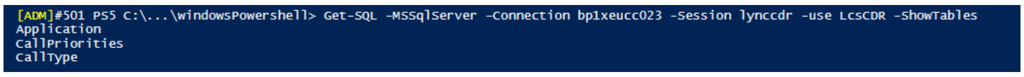
Figure 3 32-bit PowerShell connecting a named session to an Excel database

In Figure 3 the Get-SQL session is named "Topology"; you access this database using
`Get-SQL -Session Topology «command»`

When Get-SQL connects to a database it defines a new PowerShell alias for itself, using the same name as the session. Calling it with the alias makes the same name the default session name so
`Topology -ShowTables`
is equivalent to `Get-SQL -Session topology -ShowTables`. When there is no command "SQL", PowerShell will try adding "Get-" to the start of the command so `SQL` is an automatic alias for Get-SQL

`-ShowTables` returns a list of tables and is one of two parameters for discovering things about the database; the other is `-Describe «tablename»`

The `-Close` switch tells Get-SQL to disconnect from the database after running any command; some ODBC drivers lock files, so forgetting to close the file can prevent other programs from opening it. The opening or closing of a session can be combined with running a command - Figure 1 showed connecting to a server and changing database as two separate PowerShell commands, but it is possible to run commands together; Figure 4 combines the initial connection with a change DB command (using its alias of "`-USE`" to be closer to SQL Server) and the `-ShowTables` switch



```
[ADM]#501 PS5 C:\...\windowsPowershell> Get-SQL -MSSqlServer -Connection bp1xeucc023 -Session lynccdr -use LcsCDR -ShowTables
Application
CallPriorities
CallType
```

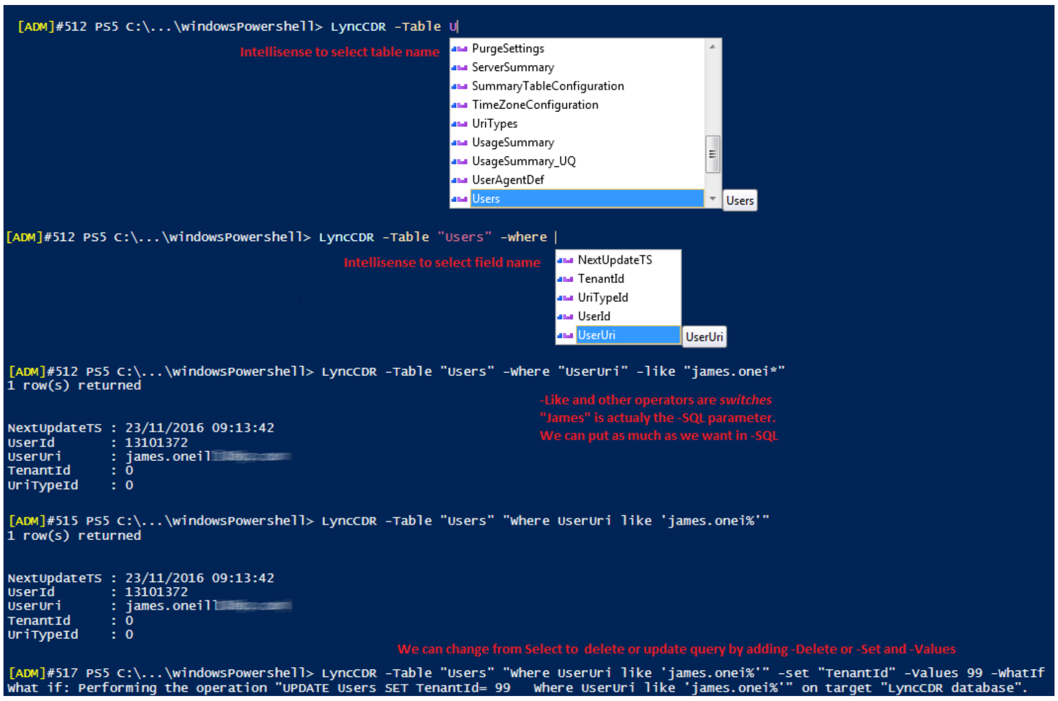
Figure 4 Combing connecting, changing database and a query in one line

RUNNING SQL COMMANDS

Any SQL can be run sent by passing the it in the `-SQL` parameter, `-SQL` will accept the first **unnamed** parameter so the connection can be used like this so each of the following will run the same query:

- `Get-SQL -Session LyncCDR -SQL "SELECT * From Users"`
- `LyncCDR -SQL "SELECT * From Users"`
- `LyncCDR "SELECT * From Users"`
- `"SELECT * From Users" | lynccdr | Select-Object -ExpandProperty userURI`

Notice that SQL statements can be piped into `Get-SQL`; it also allows the `-SQL` parameter to contain multiple statements. Get-SQL also allows a complete query to come from the clipboard using `-Paste`
Figure 1 and Figure 2 showed that `Get-SQL` supports intellisense to provide ODBC DSNs and SQL server databases. It also uses intellisense to help to construct single-table *Select*, *Update*, *Delete* and *Insert* queries from the PowerShell command line. Figure 5 and Figure 6 show this in action.



```
[ADM]#512 PS5 C:\...\windowsPowershell> LyncCDR -Table U
```

Intellisense to select table name

```
[ADM]#512 PS5 C:\...\windowsPowershell> LyncCDR -Table "Users" -where |
```

Intellisense to select field name

```
[ADM]#512 PS5 C:\...\windowsPowershell> LyncCDR -Table "Users" -Where "UserUri" -like "james.onei*"
1 row(s) returned


NextUpdateTS : 23/11/2016 09:13:42
UserId       : 13101372
UserUri      : james.oneil[redacted]
TenantId     : 0
UriTypeId    : 0
```

-Like and other operators are *switches*
"James" is actualy the -SQL parameter.
We can put as much as we want in -SQL

```
[ADM]#515 PS5 C:\...\windowsPowershell> LyncCDR -Table "Users" "where UserUri like 'james.onei%'"
1 row(s) returned


NextUpdateTS : 23/11/2016 09:13:42
UserId       : 13101372
UserUri      : james.oneil[redacted]
TenantId     : 0
UriTypeId    : 0
```

We can change from Select to delete or update query by adding -Delete or -Set and -Values

```
[ADM]#517 PS5 C:\...\windowsPowershell> LyncCDR -Table "Users" "where UserUri like 'james.onei%'" -set "TenantId" -Values 99 -WhatIf
What if: Performing the operation "UPDATE Users SET TenantId= 99   where UserUri like 'james.onei%'" on target "LyncCDR database".
```

Figure 5 Building *Select* and *Update* queries from the command line

The first command in Figure 5 shows another way to return all rows in the *Users* table:
`LyncCDR -Table Users`

Used without other parameters, `-Table Users` produces the query “`Select * from Users`”, but often we want to add more to end of the query, and the `-SQL` parameter can do that; written out in full it looks like this:

```
Get-SQL -Session LyncCDR -Table "Users" -SQL "where userUri like 'james%' "
```

and can be shortened to

```
LyncCDR -Table "Users"   "where userUri like 'james%' "
```

Get-SQL also has a `-where` parameter which can be populated by Intellisense. (PowerShell allows parameters to be entered in any order, but Intellisense requires the session to be identified to choose the table, and the table to be identified to choose fields).
So, `-where` could specify the `userUri` field, and the last part of query could be something like
`"Between 'James.o' and 'james.p'"` so the previous query could be built like this:

```
LyncCDR -Table "Users" -where "userUri" "like 'james%' "
```

But Get-SQL goes a little further and provides the same options as PowerShell’s `Where-Object` cmdlet, you can specify -
`-EQ, -NE, -GE, -LE, GT, -LT, -LIKE` or `-NotLike` so the command above could be:

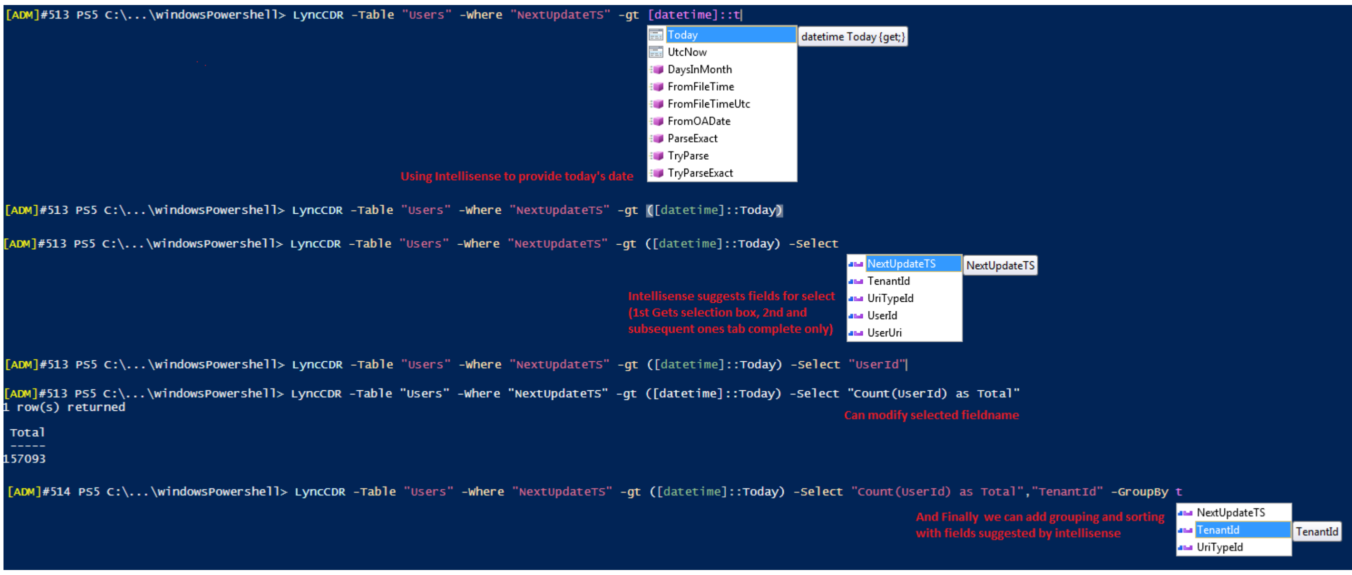
```
LyncCDR -Table Users -where userUri -like james*
```

Notice there are no quote marks in the last example: PowerShell can often manage without them but when Get-SQL’s argument completers provide values for intellisense they *are* wrapped in quotes.
Notice also, that if `-Like` and `-NotLike` are used, Get-SQL converts the standard command-line wildcard “*” into the “%” sign that SQL uses. In other places or if you use the format `"like 'James*'"`, the * will remain untouched, so you *can* search for an asterisk.

All the queries so far are expected to return data and Get-SQL writes a message to the console (<u>not</u> to standard output, ensuring that it doesn’t get mixed up with the data) saying how many rows were returned. This can be supressed with the `-Quiet` switch – it’s also often helpful to see the data in PowerShell’s grid view which can apply further sorting and filtering on the returned data to save typing `| Out-GridView` you can specify `-G` (or `-Gridview` in full). `--GridView` can also be used with `-Describe`

Get-SQL isn’t confined to the building *Select* queries.
Adding `-Delete` to the command line will transform the query from a *Select* into a *Delete* query (if the database driver supports it – Excel’s, for example, does not) and adding `-Set «fieldname1 ,fieldname2» -values «value1,value2»` turns it into an *Update*; the last command in Figure 5 does exactly this, and shows Get-SQL’s support for `-Verbose, -WhatIf` and `-Confirm` switches.
`-Verbose` displays the SQL string submitted to the database and if Get-SQL builds a connection string `-Verbose` will display that as well.
If `-Delete` or `-Set` is specified, GET-SQL will require confirmation (unless you specify `-Confirm:$false), -Insert` does not (unless you specify `-Confirm`) and `-whatif` will display the *Delete*, *Insert* or *Update* query which *would* be run.

There are additional options for *Select* queries which can be seen in Figure 6



```
[ADM]#513 PS5 C:\...\windowsPowershell> LyncCDR -Table "Users" -Where "NextUpdateTS" -gt [datetime]::t
```

```
[ADM]#513 PS5 C:\...\windowsPowershell> LyncCDR -Table "Users" -Where "NextUpdateTS" -gt ([datetime]::Today)

[ADM]#513 PS5 C:\...\windowsPowershell> LyncCDR -Table "Users" -Where "NextUpdateTS" -gt ([datetime]::Today) -Select
```

NextUpdateTS
NextUpdateTS
TenantId
UriTypeId
UserId
UserUri
Intellisense suggests fields for select (1st Gets selection box, 2nd and subsequent ones tab complete only)

```
[ADM]#513 PS5 C:\...\windowsPowershell> LyncCDR -Table "Users" -Where "NextUpdateTS" -gt ([datetime]::Today) -Select "UserId"

[ADM]#513 PS5 C:\...\windowsPowershell> LyncCDR -Table "Users" -Where "NextUpdateTS" -gt ([datetime]::Today) -Select "Count(UserId) as Total"
1 row(s) returned

 Total
 -----
157093

[ADM]#514 PS5 C:\...\windowsPowershell> LyncCDR -Table "Users" -Where "NextUpdateTS" -gt ([datetime]::Today) -Select "Count(UserId) as Total","TenantId" -GroupBy t
```

Can modify selected fieldname
And Finally we can add grouping and sorting with fields suggested by intellisense
NextUpdateTS
TenantId
TenantId
UriTypeId

Figure 6 More options in Select queries

SQL date formatting can be problematic and Get-SQL will format `datetime` objects so they can be processed (for special cases the `-dateformat` parameter allows the format to be changed) – PowerShell's Tab Expansion will put brackets around `[datetime]::Today` and you can use, for example, `-LT ([datetime]::Today).AddDays(-7)` for "more than one week ago"

Specifying `-Table` on its own runs a query as "*Select **", to choose specific fields Get-SQL has a `-Select` parameter – the fields can be completed by intellisense and it can be quicker to edit a field name populated that way than to type it; in Figure 6 you can see that `"UserID"` was filled in and then changed to `"Count(UserID) as Total"`, and then a second field was added. Note that the separate field names in `-Select` become changed a single string holding a comma separated list in the final query so anything which legitimate in a *Select* clause can be used for example: `-Select "TOP 5 *"`.

`-GroupBy` and `-OrderBy` parameters also modify how *Select* queries are built, and both support intellisense; fields for *Order By* can be modified to contain "*DESC*" so they sort in descending order. The last parameter which affects *Select* queries is `-Distinct` which changes the *Select* statement to a *Select Distinct* one

Get-SQL won't check a *Delete* or *Update* query which is passed in the `-SQL` parameter – it allows any SQL and assumes database engine supports it and you intend to run it - but when asked to build such a query from parameters it requires that there is a *Where* condition (so a whole table isn't deleted or overwritten by mistake). If `-Select, -Distinct -Orderby` or `-GroupBy` have been specified then the parameter definitions for GET-SQL won't allow `-Set ... -Values` or `-Delete`

As well as building *Select*, *Delete* and *Update* queries, Get-SQL has an `-INSERT` parameter to allow it to add new rows. For all other kinds of queries the `-SQL` parameter contains either a complete SQL query or something to append to the end of query built from other parameters, with `-Insert` it must be one or more PSObjects or Hash Tables. With a PSobject the property names match column names in the table and the value of each goes in that column: with a hash table the *Keys* are column names.

So, an insert command might look like this
```
Get-SQL -Insert TestTable @{Name="File.ext";length=1024}
```
or this
```
dir *.ps1 | Select-object -property -Name, Length | Get-SQL -Insert TestTable
```
Passing an object through `Select-Object -property` is a quick way to create the required PSobject

Get-SQL has a lot of choices: Figure 7 shows each different type of operation (top to bottom) and the different groups of parameters which build up to create the command (left to right)

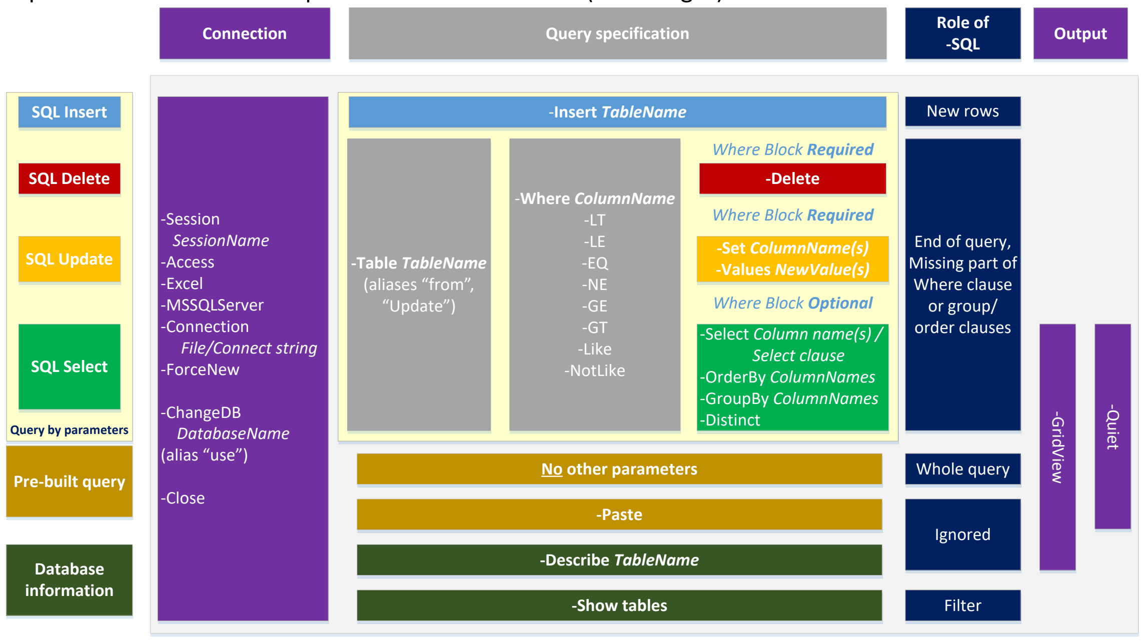


Figure 7 How the parameters fit together

One final parameter was added in version 1.2. When a table object is returned it is expanded into its rows and there are use cases where the table is needed `-OutputVariable MyTable` puts the table into a variable so that after Get-SQL has finished, the table can be accessed as `$MyTable`.