execsql Documentation

Release 1.29.3

Dreas Nielsen

Sep 22, 2018

Contents

`execsql.py` is a Python program that applies a SQL script stored in a text file to a PostgreSQL, MS-Access, SQLite, MS-SQL-Server, MySQL, MariaDB, or Firebird database, or an ODBC DSN. execsql.py also supports a set of special commands (*metacommands*) that can import and export data, copy data between databases, and conditionally execute SQL statements and metacommands. These metacommands make up a control language that works the same across all supported database management systems (DBMSs). The metacommands are embedded in SQL comments, so they will be ignored by other script processors (e.g., psql for Postgres and sqlcmd for SQL Server). The metacommands make up a toolbox that can be used to create both automated and interactive data processing applications; some of these uses are illustrated in the *examples*.

CHAPTER 1

Capabilities

You can use execsql to:

- Import data from text files or spreadsheets into a database.
- Export tables and views to text files, OpenDocument spreadsheets, HTML, JSON, LaTeX, or to nine other tabular formats (see *Example 8*).
- Copy data between different databases, even databases using different DBMSs.
- Display tables or views on the terminal or in a GUI dialog window.
- Export data using template processors to produce non-tabular output with customized format and contents.
- Conditionally execute different SQL commands and metacommands based on the DBMS in use, the database in use, data values, user input, and other conditions. Conditional execution can be used with the INCLUDE and EXECUTE SCRIPT metacommands to implement loops (see *Example 6*).
- Prompt the user to select files or directories, answer questions, or enter data values.
- Allow the user to visually compare two tables or views.
- Write messages to the console or to a file during the processing of a SQL script. These messages can be used to display the progress of the script or create a custom log of the operations that have been carried out or results obtained. Status messages and data exported in text format can be combined in a single text file. Data tables can be exported in a text format that is compatible with Markdown pipe tables, so that script output can be converted into a variety of document formats (see *Example 8* and *Example 11*).

Different DBMSs and DBMS-specific client programs provide different and incompatible extensions to SQL, ordinarily to allow interactions with the file system and to allow conditional tests and looping. Some DBMSs do not have any native extensions of this sort. execsql provides these features, as well as features for user interaction, in an identical fashion for all supported DBMSs. This allows standardization of the SQL scripting language used for different types of database management systems.

execsql's features for conditional tests, looping, and sub-scripts allow the script author to write modular, maintainable, and re-usable code.

execsql is inherently a command-line program that can operate in a completely non-interactive mode (except, in some cases, for password prompts). Therefore, it is suitable for incorporation into a toolchain controlled by a shell script

(on Linux), batch file (on Windows), or other system-level scripting application. However, several *metacommands* can be used to generate interactive prompts and data displays, so execsql scripts can be written to provide some user interactivity.

In addition, execsql automatically maintains a log that documents key information about each run of the program, including the databases that are used, the scripts that are run, and the user's choices in response to interactive prompts. Together, the script and the log provide documentation of all actions carried out that may have altered data.

CHAPTER 2

Documentation Guide

The sections of the execsql documentation fall into several categories, described in the following sections. The documentation tracks the latest version on Bibucket, which may be more recent than the version available from the Python Package Index (PyPi).

2.1 Getting Started

These documentation sections contain information that most users will need to read in order to start using execsql.

Syntax and Options: Command-line arguments and flags.
Requirements: Other Python packages that may be needed.

2.2 Tips and Guidance

These documentation sections include information that is pertinent to specific DBMSs and may improve your understanding and usage of execsql's features. If you are encountering unexpected behavior, information in these sections may be of assistance.

Usage Notes: Important but not necessarily essential information about execsql's operation.
SQL Syntax Notes: Details about handling of SQL statements.
Logging: A description of the automatically maintained log file.
Character Encoding: Information on handling of different character encodings.
Using Script Files: Recommendations (really, advocacy) for the use of script files.
Documentation: Information to support the creation of comprehensive documentation.
Debugging: Metacommands to assist with SQL script debugging.

2.3 Reference

These documentation sections contain detailed descriptions of specific execsql features. These sections may be consulted repeatedly when writing SQL scripts. Some tips and guidance are also included within the reference information.

Configuration Files: Fire-and-forget control over execsql's environment and operation.

Substitution Variables: Text substitutions to customize any part of a script.

Metacommands: Import and export data, interact with the user, and dynamically control script flow.

2.4 Examples

This section contains examples of execsql's usage, focusing primarily on *metacommands*. The code snippets in these examples can generally be easily modified for use in other applications.

Examples: Code snippets to illustrate execsql usage.

2.4.1 Syntax and Options

execsql.py should be run at the operating-system command line–i.e., at a shell prompt in Linux or in a command window in Windows. Python may or may not need to be explicitly invoked, and the .py extension may or may not need to be included, depending on your operating system, operating system seetings, and how execsql is *installed*.

execsql.py runs under both Python 2.7 and Python 3.x.

For Linux users: The execsql.py file contains a shebang line pointing to /usr/bin/python, so there should be no need to invoke the Python interpreter. Depending on how execsql.py was obtained and installed, it may need to be made executable with the *chmod* command.

For Windows users: If you are unfamiliar with running Python programs at the command prompt, see https://docs.python.org/2/faq/windows.html.

In the following syntax descriptions, angle brackets identify required replaceable elements, and square brackets identify optional replaceable elements.

```
Commands:
   execsql.py -tp [other options] <sql_script_file> <Postgres_host> <Postgres_db>
   execsql.py -tl [other options] <sql_script_file> <SQLite_db>
   execsql.py -tf [other options] <sql_script_file> <Firebird_host> <Firebird_db>
   execsql.py -ta [other options] <sql_script_file> <Access_db>
   execsql.py -tm [other options] <sql_script_file> <MySQL_host> <MySQL_db>
   execsql.py -ts [other options] <sql_script_file> <SQL_Server_host> <SQL_Server_db>
   execsql.py -td [other options] <sql_script_file> <DSN_name>
Arguments:
   <sql_script_file>
          The name of a text file of SQL commands to be executed.  Required argument.
   <Postgres_host>
          The name of the Postgres host (server) against which to run the SQL.
   <Postgres_db>
          The name of the Postgres database against which to run the SQL.
   <SQLite_db>
          The name of the SQLite database against which to run the SQL.
   <Firebird_host>
          The name of the Firebird host (server) against which to run the SQL.
```

(continues on next page)

(continued from previous page)

```
   <Firebird_db>
          The name of the Firebird database against which to run the SQL.
   <MySQL_host>
          The name of the MySQL or MariaDB host (server) against which to run the SQL.
   <MySQL_db>
          The name of the MySQL or MariaDB database against which to run the SQL.
   <SQL_Server_host>
          The name of the SQL Server host (server) against which to run the SQL.
   <SQL_Server_db>
          The name of the SQL Server database against which to run the SQL.
   <Access_db>
          The name of the Access database against which to run the SQL.
   <DSN_name>
          The name of a DSN data source against which to run the SQL.
Options:
   -a <value>  Define the replacement for a substitution variable $ARG_x.
   -b <value>  Control whether input data columns containing only 0 and 1 are treated
                           as Boolean or integer: 'y'-Yes (default); 'n'-No.
   -d <value>  Make directories used by the EXPORT metacommand: 'n'-No (default); 'y'-
↪Yes.
   -e <value>  Character encoding of the database.  Only used for some database types.
   -f <value>  Character encoding of the script file.
   -g <value>  Character encoding to use for output of the WRITE and EXPORT␣
↪metacommands.
   -i <value>  Character encoding to use for data files imported with the IMPORT␣
↪metacommand.
   -m          Display the allowable metacommands, and exit.
   -n          Create a new SQLite or Postgres database if the specified database␣
↪does not exist.
   -o          Open the online help (http://execsql.readthedocs.io) in the default␣
↪browser.
   -p <value>  The port number to use for client-server databases.
   -s <value>  The number of lines of an IMPORTed file to scan to diagnose the quote␣
↪and
                                delimiter characters.
   -t <value>  Type of database: 'p'-Postgres, 'l'-SQLite, 'f'-Firebird, 'm'-MySQL,
                                's'-SQL Server, 'a'-Access, 'd'-DSN.
   -u <value>  The database user name.
   -v <value>  Use a GUI for interactive prompts.
   -w          Do not prompt for the password when the user is specified.
   -y          List all valid character encodings and exit.
   -z <value>  Buffer size, in kb, to use with the IMPORT metacommand (the default is␣
↪32).
```

If the database type and connection information is specified in a configuration file, then the database type option and the server and database name can be omitted from the command line. The absolute minimum information that must be specified on the command line is the name of the script file to run.

If a server-based database is used (i.e., Postgres, Firebird, MySQL/MariaDB, or SQL Server), then if only one command-line argument is provided in addition to the script file name, that argument will be interpreted as the database name if the server name has been set in a configuration file and the database name has not; otherwise that single argument will be interpreted as the server name.

Following are additional details on some of the command-line options:

-a This option should be followed by text that is to be assigned to a *substitution variable*. Substitution variables can be defined on the command line to provide data or control parameters to a script. The "-a" option can be used repeatedly to de-

fine multiple substitution variables. The value provided with each instance of the "-a" option should be a replacement string. execsql will automatically assign the substitution variable names. The substitution variable names will be "$ARG_1", "$ARG_2", etc., for as many variables are defined on the command line. Use of the "-a" option is illustrated in *Example 9*. Command-line substitution variable assignments are *logged*.

-e, -f, -g, -i These options should each be followed by the name of a *character encoding*. Valid names for character encodings can be displayed using the "-y" option.

-p A port number should be provided if the DBMS is using a port different from the default. The default port numbers are:

- Postgres: 5432
- SQL Server: 1433
- MySQL: 3306
- Firebird: 3050

-u The name of the database user should be provided with this option for password-protected databases; execsql will prompt for a password if a user name is provided, unless the "-w" option is also specified.

-v This option should be followed by an integer indicating the level of GUI interaction that execsql should use. The values allowed are:

- 0: Use the terminal for all prompts (the default).
- 1: Use a GUI dialog for password prompts and the *PAUSE* metacommand.
- 2: Additionally, use a GUI dialog for any message to be displayed with the *HALT* metacommand, and use a GUI dialog to prompt for the initial database to use if no other specifications are provided.
- 3: Additionally, open a GUI *console* when execsql starts.

The prompt for a database password, and the prompt produced by the *PAUSE metacommand*, are both displayed on the terminal by default. When the "-v1" option is used, or the GUI *console* is open, both of these prompts will appear in GUI dialogs instead. If the "-v2" option is specified, then the *HALT* metacommand, if used with a message, will also be displayed in a GUI dialog. In addition, if the "-v2" or "-v3" option is used, and no server name or database name are specified either in a configuration file or on the command line, then execsql will use a GUI dialog to prompt for this information when it starts up.

-w Ordinarily if a user name is specified (with the "-u" option), execsql will prompt for a password for that user. When this option is used, execsql will not prompt for entry of a password.

2.4.2 Requirements

The execsql program uses third-party Python libraries to communicate with different database and spreadsheet software. These libraries must be installed to use those programs with execsql. Only those libraries that are needed, based on the command line arguments and *metacommands*, must be installed. The libraries required for each database or spreadsheet application are:

- PostgreSQL: psycopg2.
- Firebird: fdb.

- MySQL or MariaDB: pymysql.
- SQL Server: pyodbc.
- MS-Access: pyodbc and pywin32.
- DSN data source: pyodbc.
- OpenDocument spreadsheets: odfpy.
- Excel spreadsheets (read only): xlrd.

Connections to SQLite databases are made using Python's standard library, so no additional software is needed.

To use the Jinja or Airspeed template processors with the *EXPORT* metacommand, those software libraries must be installed also.

2.4.3 Configuration Files

In addition to, or as an alternative to, command-line options and arguments, configuration files can be used to specify most of the same information, plus some additional information. Most of the command-line options and arguments can be specified in a configuration file, with the exception of the script name. The script name must always be specified on the command line.

execsql will read information from up to four configuration files in different locations, if they are present. The four locations are:

- The system-wide application data directory. This is `/etc` on Linux, and `%APPDATA%` on Windows.
- The user-specific configuration directory. This is a directory named `.config` under the user's home directory on both Linux and Windows.
- The directory where the script file is located.
- The directory from which execsql was started.

The name of the configuration file, in all locations, is `execsql.conf`.

Configuration data is read from these files in the order listed above. Information in later files may augment or replace information in earlier files. Options and arguments specified on the command line will further augment or override information specified in the configuration files.

Configuration files use the INI file format. Section names are case sensitive and must be all in lowercase. Property names are not case sensitive. Property values are read as-is and may or may not be case sensitive, depending on their use. Comments can be included in configuration files; each comment line must start with the "#" character.

The section and property names that may be used in a configuration file are listed below.

Section `connect`

`db_type` The type of database. This is equivalent to the "-t" command-line option, and the same list of single-character codes are the only valid property values.

`server` The database server name. This is equivalent to the second command-line argument for client-server databases.

`db` The database name. This is equivalent to the third command-line argument for client-server databases

`db_file` The name of the database file. This is equivalent to the second command-line argument for file-based databases.

`port` The port number for the client-server database. This is equivalent to the "-p" command-line option.

username The name of the database user, for client-server databases. This is equivalent to the "-u" command-line option.

access_username The name of the database user, for MS-Access databases only. When using MS-Access, a password will be prompted for only if this configuration option is set or the "-u" command-line option is used, regardless of the setting of the username configuration parameter.

password_prompt Indicates whether or not execsql should prompt for the user's password. The property value should be either "Yes" or "No". This is equivalent to the "-w" command-line option.

new_db Indicates whether or not execsql should create a new PostgreSQL or SQLite database to connect to.

Section `encoding`

database The database encoding to use. This is equivalent to the "-e" command-line option.

script The script encoding to use. This is equivalent to the "-f" command-line option.

import Character encoding for data imported with the IMPORT metacommand. This is equivalent to the "-i" command-line option.

output Character encoding for data exported with the EXPORT metacommand. This is equivalent to the "-h" command-line option.

error_response How to handle conditions where input or output files have incompatible encodings. If not specified, incompatible encodings will cause an error to occur, and execsql will halt. The property values you can use for this setting are:

- "ignore": The inconvertible character will be omitted.
- "replace": The inconvertible character will be replaced with a question mark.
- "xmlcharrefreplace": The inconvertible character will be replaced with the equivalent HTML entity.
- "backslashreplace": The inconvertible character will be replaced with an escape sequence consisting of decimal digits, preceded by a backslash.

Section `input`

access_use_numeric Whether or not to translate decimal (numeric) data types to double precision when the *IMPORT* or *COPY* metacommands construct a CREATE TABLE statement for MS-Access. This property value should be either "Yes" or "No." The default value is "No".

boolean_int Whether or not to consider integer values of 0 and 1 as Booleans when scanning data during import or copying. The property value should be either "Yes" or "No". The default value is "Yes". By default, if a data column contains only values of 0 and 1, it will be considered to have a Boolean data type. By setting this value to "No", such a column will be considered to have an integer data type. This is equivalent to the "-b" command-line option.

boolean_words Whether or not to recognize only full words as Booleans. If this value is "No" (the default), then values of "Y", "N", "T", and "F" will be recognized as Booleans. If this value is "Yes", then only "Yes", "No", "True", and "False" will be recognized as Booleans. This setting is independent of the `boolean_int` setting.

max_int Establishes the maximum value that will be assigned an integer data type when the IMPORT or COPY metacommands create a new data table. Any column with integer values less than or equal to this value (`max_int`) and greater than or equal to $-1 \times$ `max_int` $-$ 1 will be considered to have an 'integer' type. Any column with values outside this range will be considered to have a 'bigint' type. The default value for `max_int` is 2147483647. The `max_int` value can also be altered within a script using the *MAX_INT* metacommand.

empty_strings Determines whether empty strings in the input are preserved or, alternatively, will be replaced by NULL. The property value should be either "Yes" or "No". The default, "Yes", indicates that empty strings are allowed. A value of "No" will cause all empty strings to be replaced by NULL. When this is set to "No", a string value consisting of a sequence of zero or more space characters will be considered to be an empty string. There is no command-line option corresponding to this configuration parameter, but the metacommand *EMPTY_STRINGS* can also be used to change this configuration item.

import_only_common_columns Determines whether the IMPORT metacommand will import data from a CSV file when the file has more data columns than the target table. The property value should be either "Yes" or "No". The default, "No", indicates that the target table must have all of the columns present in the CSV file; if the target table has fewer columns, an error will result. A property value of "Yes" will result in import of only the columns in common between the CSV file and the target table.

scan_lines The number of lines of a data file to scan to determine the quoting character and delimiter character used. This is equivalent to the "-s" command-line option.

import_buffer The size of the import buffer, in kilobytes, to use with the IMPORT metacommand. This is equivalent to the "-z" command-line option.

Section `output`

log_write_messages Specifies whether output of the *WRITE* metacommand will also be written to execsql's log file. The property value should be either "Yes" or "No". This configuration property can also be controlled within a script with the *LOG_WRITE_MESSAGES* metacommand.

make_export_dirs The output directories used in the *EXPORT* and *WRITE* metacommands will be automatically created if they do not exist (and the user has permission). The property value should be either "Yes" or "No". This is equivalent to the "-d" command-line option.

css_file The URI of a CSS file to be included in the header of an HTML file created with the *EXPORT* metacommand. If this is specified, it will replace the CSS styles that execsql would otherwise use.

css_style A set of CSS style specifications to be included in the header of an HTML file created with the *EXPORT* metacommand. If this is specified, it will replace the CSS styles that execsql would otherwise use. Both css_file and css_style may be specified; if they are, they will be included in the header of the HTML file in that order.

template_processor The name of the template processor that will be used with the EXPORT and EXPORT QUERY metacommands. The only valid values for this property are "jinja" and "airspeed". If this property is not specified, the default template processor will be used.

Section `interface`

console_wait_when_done Controls the persistence of any *console window* at the completion of the script when the script either completes normally or exits prematurely as a result of the user's response to a prompt. If the property value is set to "Yes" (the default value is "No"), the console window will remain open until explicitly closed by the user. The message "Script complete; close the console window to exit execsql." will be displayed in the status bar. This setting has the same effect as a *CONSOLE WAIT_WHEN_DONE* metacommand.

console_wait_when_error_halt Controls the persistence of any *console window* at the completion of the script if an error occurs. If the property value is set to "Yes" (the default value is "No"), the console window will remain open until explicitly closed by the user after an error occurs. The message "Script error; close the console window to exit execsql." will be displayed in the status bar. This setting has the same effect as a *CONSOLE WAIT_WHEN_ERROR* metacommand.

gui_level The level of interaction with the user that should be carried out using GUI dialogs. The property value must be 0, 1, 2, or 3. The meanings of these values are:

- 0: Do not use any optional GUI dialogs.

- 1: Use GUI dialogs for password prompts and for the *PAUSE* metacommand.
- 2: Also use a GUI dialog if a message is included with the *HALT* metacommand, and prompt for the initial database to use if no database connection parameters are specified in a configuration file or on the command line.
- 3: Additionally, open a GUI console when execsql starts.

Section `email`

host The SMTP host name to be used to transmit email messages sent using the *EMAIL* metacommand. A host name must be specified to use the *EMAIL* metacommand.

port The port number of the SMTP host to use. If this is omitted, port 25 will be used unless either the `use_ssl` or `use_tls` configuration properties is also specified, in which case ports 465 or 587 may be used.

username The name of the user if the SMTP server requires login authentication.

password An unencrypted password to be used if the SMTP server requires login authentication.

enc_password An encrypted password to be used if the SMTP server required login authentication. The encrypted version of a password should be as is produced by the *SUB_ENCRYPT* metacommand. A suitably encrypted version of a password can be produced by running the script:

```
-- !x! prompt enter_sub pw password message "Enter a password to encrypt"
-- !x! sub_encrypt enc_pw !!pw!!
-- !x! write "The encrypted password is: !!enc_pw!!"
```

If both the `password` and `enc_password` configuration properties are used, the `enc_password` property will take precedence and will be used for SMTP authentication. Note that this is not a cryptographically secure encryption, merely an obfuscation of the password.

use_ssl SSL/TLS encryption will be used from the initiation of the connection.

use_tls SSL/TLS encryption will be used after the initial connection is made using unencrypted text.

email_format Specifies whether the message will be sent as plain text or as HTML email. The only valid values for this property are "plain" and "html". If not specified, emails will be sent in plain text.

message_css A set of CSS rules to be applied to HTML email.

Section `config`

config_file The full name or path to an additional configuration file to be read. If only a path is specified, the name of the configuration file should be `execsql.conf`. The configuration file specified will be read immediately following the configuration file in which it is named. No configuration file will be read more than once.

Section `variables`

There are no fixed properties for this section. All property names and their values that are specified in this section will be used to define substitution variables, just as if a series of SUB metacommands had been used at the beginning of the script.

Section `include_required`

This section lists additional script files that should be automatically included before the main script is run, without the use of any explicit *INCLUDE* metacommand in the main script.

Each property in this section should be an integer, and the property value should be a filename. The integers specify the order in which the files should be included. If any integer is listed more than once, only the last filename associated with that integer in this configuration section will be included. If any of the specified files does not exist, an error will occur and execsql will stop. Each file may be included only once.

Files specified in this section will be included before any files specified in the `include_optional` section. This priority ordering applies to lists of required and optional files specified in all configuration files that are read.

The order in which these files are imported is also affected by the order in which multiple configuration files (if they exist) are read.

Section `include_optional`

This section lists additional script files that will, if they exist, be automatically included before the main script is run, without the use of any explicit *INCLUDE* metacommand in the main script.

Each property in this section should be an integer, and the property value should be a filename. The integers specify the order in which the files should be included. If any integer is listed more than once, only the last filename associated with that integer in this configuration section will be included. If any of the specified files does not exist, it will be ignored. Each file may be included only once.

Files specified in this section will be included after any files specified in the `include_required` section. This priority ordering applies to lists of required and optional files specified in all configuration files that are read.

The order in which these files are imported is also affected by the order in which multiple configuration files (if they exist) are read.

2.4.4 Usage Notes

This section contains miscellaneous notes on execsql usage.

Required Arguments

If the program is run without any arguments it will print a help message on the terminal, similar to the *syntax description*.

At least one argument, the name of the script file to run, is required. This single argument can be used when the database connection information is specified in one or more *configuration files*.

SQL Statement Recognition and SQL Syntax

execsql recognizes a SQL statement as consisting of a sequence of non-comment lines that ends with a line ending with a semicolon. A backslash ("\") at the end of a line is treated as a line continuation character. Backslashes do not need to be used for simple SQL statements, but must be used for procedure and function definitions, where there are semicolons within the body of the definition, and a semicolon appears at the end of lines for readability purposes. Backslashes may not be used as continuation characters for *metacommands*.

With the exception of the "CREATE TEMPORARY QUERY…" statement when used with MS-Access, the execsql program does not parse or interpret SQL syntax in any way.

SQL syntax used in the script must conform to that recognized by the DBMS engine in use. Because execsql can connect to several different DBMSs simultaneously, a single script can contain a mixture of different SQL syntaxes. To minimize this variation (and possible mistakes that could result), execsql metacommands provide some common features of DBMS-specific scripting languages (e.g., pgScript and T-SQL), and execsql turns on ANSI-compatible mode for SQL Server and MySQL when it connects to those databases.

Comments in SQL Scripts

Script files can contain single-line comments, which are identified by two dashes ("--") at the start of a line. Script files can also contain multi-line comments, which begin on a line where the first characters are "/*" and end on a line where the last characters are "*/".

execsql strips single-line and multi-line comments from the script file when compiling SQL statements to send to the DBMS. execsql does not strip comments that follow part of a SQL statement on the same line, such as:

```
select
    scramble(eggs)      -- Use custom aggregate function
from
    refrigerator natural join stove;
```

The DBMS in use must be able to recognize and ignore any such comments. If any such comment occurs on the last line of the SQL statement, following the semicolon, then execsql will not recognize the end of the SQL statement, and an error will result.

Metacommands

Metacommands are directives to execsql that can control script processing, import and export data, report status information, and perform other functions. Metacommands are embedded in single-line SQL comments. These metacommands are identified by the token "!x!" immediately following the SQL comment characters at the beginning of a line, i.e.:

```
-- !x! <metacommand>
```

The special commands that are available are described in the *Metacommands* section.

Autocommit

SQL statements are ordinarily automatically committed by execsql. Consequently, database transactions will not work as expected under default conditions. The *AUTOCOMMIT* and *BATCH* metacommands provide two different ways to alter execsql's default autocommit behavior. Transactions will work as expected either within a batch or after autocommit has been turned off. One difference between these two approaches is that within transactions inside a batch, changes to data tables are not visible to metacommands such as *PROMPT DISPLAY*, whereas these data are visible within transactions that follow an *AUTOCOMMIT OFF* metacommand. This difference in data visibility affects what tests can be done to decide whether to commit or roll back a transaction.

Rollback on Exit

When execsql exits, or closes a database connection because the same alias will be used again in a *CONNECT* metacommand, a rollback command will be sent to the database immediately before each connection is closed. Therefore if, for example, a *PROMPT DISPLAY* metacommand is used within a transaction, and the user cancels the display, and thus the script, a rollback command will be sent to the database, thereby terminating the transaction. This prevents transactions from being left open and incomplete, which may cause problems in some circumstances.

Exit Status

If execsql finishes normally, without errors and without being halted either by script conditions or the user, the system exit status will be set to 0 (zero). If an error occurs that causes the script to halt, the exit status will be set to 1. If the user cancels script processing in response to any prompt, the exit status will be set to 2. If the script is halted with either the *HALT* or *HALT DISPLAY* metacommands, the system exit status will be set to 3 unless an alternate value is specified as part of the metacommand.

DSN Connections

When a DSN is used as a data source, execsql has no information about the features or SQL syntax used by the underlying DBMS. In the expectation that a DSN connection will most commonly be used for Access databases, a DSN connection will use Access' syntax when issuing a CREATE TABLE statement in response to a COPY or IMPORT metacommand. However, a DSN connection does not (and cannot) use DAO to manage queries in a target Access database, so all data manipulations must be carried out using SQL statements. The EXECUTE metacommand uses the same approach for DSN connections as is used for SQL Server.

MS-Access-Specific Considerations

Temporary Queries

The syntax of the "CREATE TEMPORARY QUERY" DDL supported by execsql when used with an MS-Access database is:

```
CREATE [TEMP[ORARY]] QUERY|VIEW <query_name> AS <sql_command>
```

The "TEMPORARY" specification is optional: if it is included, the query will be deleted after the entire script has been executed, and if it is not, the query will remain defined in the database after the script completes. If a query of the same name is already defined in the Access database when the script runs, the existing query will be deleted before the new one is created—no check is performed to determine whether the new and old queries have the same definition, and no warning is issued by execsql that a query definition has been replaced. The keyword "VIEW" can be used in place of the keyword "QUERY". This alternative provides compatibility with the "CREATE TEMPORARY VIEW" command in PostgreSQL, and minimizes the need to edit any scripts that are intended to be run against both Access and PostgreSQL databases.

Scripts for Microsoft Access that use temporary queries will result in those queries being created in the Access database, and then removed, every time the scripts are run. This will lead to a gradual increase in the size of the Access database file. If the script halts unexpectedly because of an error, the temporary queries will remain in the Access database. This may assist in debugging the error, but if the temporary queries are not created conditional on their non-existence, you may have to remove them manually before re-running the script.

Password-Protected Databases

The user name for password-protected Access databases is "Admin" by default (i.e., if no other user name was explicitly specified when the password was applied). To ensure that execsql prompts for a password for password-protected Access databases, a user name must be specified either on the command line with the "-u" option or in a configuration file with the `access_username` *configuration item*. When the user name in Access is "Admin", any user name can be provided to execsql.

ODBC and DAO Connections

With Access databases, an ODBC connection is used for SELECT queries, to allow errors to be caught, and a DAO connection to the Jet engine is used when saved action queries (UPDATE, INSERT, DELETE) are created or modified. Because the Jet engine only flushes its buffers every five seconds, execsql will ensure that at least five seconds have passed between the last use of DAO and the execution of a SELECT statement via ODBC.

Boolean Columns

Boolean (Yes/No) columns in Access databases cannot contain NULL values. If you *IMPORT* boolean data into a column having Access' boolean data type, any NULL values in the input data will be converted to *False* boolean values. This is a potentially serious data integrity issue. To help avoid this, when the NEW or REPLACEMENT keywords are used with the *IMPORT* or *COPY* metacommands, and execsql determines that the input file contains boolean data, execsql will create that column in Access with an integer data type rather than a boolean data type, and when adding data will convert non-integer *True* values to 1, and *False* values to 0.

2.4.5 SQL Syntax Notes

ANSI Compatibility

When execsql connects to a SQL Server or MySQL/MariaDB database, it automatically configures the DBMS to expect ANSI-compatible SQL, to allow the use of more standards-compliant, and thus consistent, SQL. In particular, for MySQL/MariaDB, note that the double-quote character, rather than the backtick character, must be used to quote table, schema, and column names, and only the apostrophe can be used to quote character data.

Implicit Commits

By default, execsql immediately commits all SQL statements. The *AUTOCOMMIT* metacommand can be used to turn off automatic commits, and the *BATCH* metacommand can be used to delay commits until the end of a batch. *IMPORT* and *COPY* are the only metacommands that change data, and they also automatically commit their changes when complete (unless *AUTOCOMMIT* has been turned off). If a new table is created with either of these metacommands (through the use of the NEW or REPLACEMENT keywords), the CREATE TABLE statement will not be committed separately from the data addition, except when using Firebird. Thus, if an error occurs during addition of the data, the new target table will not exist—except when using Firebird.

When adding a very large amount of data with the *IMPORT* or *COPY* metacommands, internal transaction limits may be exceeded for some DBMSs. For example, MS-Access may produce a 'file sharing lock count exceeded' error when large data sets are loaded.

Implicit DROP TABLE Statements

The "REPLACEMENT" keyword for the *IMPORT* and *COPY* metacommands allows a previously existing table to be replaced. To accomplish this, execsql issues a "DROP TABLE" statement to the database in use. PostgreSQL, SQLite, MySQL, and MariaDB support a form of the "DROP TABLE" statement that automatically removes all foreign keys to the named table. execsql uses these forms of the "DROP TABLE" statement for these DBMSs, and therefore use of the "REPLACEMENT" keyword always succeeds at removing the named table before trying to create a new table with the same name. SQL Server, MS-Access, and Firebird do not have a form of the "DROP TABLE" statement that automatically removes foreign keys. Therefore, if the "REPLACEMENT" keyword is used with any of these three DBMSs, for a table that has foreign keys into it, that table will not be dropped, and an error will subsequently occur when execsql issues a "CREATE TABLE" statement to create a new table of the same name. To avoid this, when

using any of these three DBMSs, you should include in the script the appropriate SQL commands to remove foreign keys (and possibly even to remove the table) before using the IMPORT or COPY metacommands.

Boolean Data Types

Not all DBMSs have explicit support for a boolean data type. When execsql creates a new table as a result of the NEW or REPLACEMENT keyword in *IMPORT* and *COPY* metacommands, it uses the following data type for boolean values in each DBMS:

- Postgres: boolean.
- SQLite: integer. *True* values are converted to 1, and *False* values are converted to 0.
- Access: integer. Although Access supports a "bit" data type, bit values are non-nullable, and so to preserve null boolean values, execsql uses the integer type instead. *True* values are converted to 1, and *False* values are converted to 0.
- SQL Server: bit.
- MySQL and MariaDB: boolean
- Firebird: integer. *True* values are converted to 1, and *False* values are converted to 0.

If boolean values are imported to some other data type in an existing table, the conversion to that data type may or may not be successful.

When scanning input data to determine data types, execsql will consider a column to contain boolean values if it contains only values of 0, 1, '0', '1', 'true', 'false', 't', 'f', 'yes', 'no', 'y', or 'n'. Character matching is case-insensitive. This behavior can be altered with the `boolean_int` and `boolean_words` *configuration settings* or with the *BOOLEAN_INT* and *BOOLEAN_WORDS* metacommands.

Schemas, the IMPORT and COPY Metacommands, and Schema-less DBMSs

If a schema name is used with the table specifications for the *IMPORT* or *COPY* metacommands, when the command is run against either MS-Access or SQLite, the schema name will be ignored. No error or warning message will be issued. Such irrelevant schema specifications are ignored to reduce the need to customize metacommands for use with different DBMSs.

MS-Access Quirks

The version of SQL that is used by the Jet engine when accessed via DAO or ODBC, and thus that must be used in the script files executed with execsql, is generally equivalent to that used within Access itself, but is not identical, and is also not the same in all respects as standard SQL. There are also differences in the SQL syntax accepted by the DAO and ODBC interfaces. To help avoid inconsistencies and errors, here are a few points to keep in mind when creating SQL scripts for use with Access:

- The Jet engine can fail to correctly parse multi-table JOIN expressions. In these cases you will need to give it some help by parenthesizing parts of the JOIN expression. This means that you have some responsibility for constructing optimized (or at least acceptably good) SQL.
- Not all functions that you can use in Access are available via DAO or ODBC. Sometimes these can be worked around with slightly lengthier code. For example, the `Nz()` function is not available in an ODBC connection, but it can be replaced with an expression such as `Iif([Column] is null, 0, [Column])`. The list of ODBC functions that can be used is listed here: https://msdn.microsoft.com/en-us/library/office/ff835353.aspx. When creating (temporary) queries—i.e., when using DAO—the functions available are equivalent to those available in Access' GUI interface. A partial list of the differences between Access and ANSI SQL is here: https://msdn.microsoft.com/en-us/library/bb208890%28v=office.12%29.aspx

- Literal string values in SQL statements should be enclosed in single quotes, not double quotes. Although Access allows double quotes to be used, the ANSI SQL standard and the connector libraries used for execsql require that single quotes be used.
- Square brackets must be used around column names that contain embedded spaces when a temporary query is being used (i.e., DAO is used). At all other times, double quotes will work.
- Expressions that should produce a floating-point result ('Double') sometimes do not, with the output being truncated or rounded to an integer. A workaround is to multiply and then divide the expression by the same floating-point number; for example: '1.00000001 * <expression> / 1.00000001'.
- The wildcard character to use with the LIKE expression, in a `CREATE [TEMPORARY] QUERY` statement, differs under different circumstances:
  - "*" must be used in action queries (UPDATE, INSERT, DELETE).
  - "*" must be used in simple SELECT queries.
  - "%" must be used in subqueries of SELECT queries.

  These differences are due to the different syntaxes supported by the ODBC and DAO connections, and circumstances (such a subqueries) in which the text of a saved query is recompiled by the ODBC driver. When you are not creating a (temporary) query, "%" should always be used as the wildcard. In particular, avoid creating a query that will be used both directly and as a subquery in another query–this situation is very likely to result in errors. *Because of the potentially adverse consequences of improper interpretation of wildcards with Access databases, you should always test such statements very carefully.*

2.4.6 Substitution Variables

Substitution variables are words that have been defined to be equivalent to some other text, so that when they are used, those words will be replaced (substituted) by the other text in a SQL statement or metacommand before that statement or metacommand is executed. Substitution variables can be defined using the SUB metacommand, as follows:

```
SUB <match_string> <replacement_string>
```

The <match_string> is the word (substitution variable) that will be matched, and the <replacement_string> is the text that will be substituted for the matching word. Substitution variables are only recognized in SQL statements and metacommands when the match string is preceded and followed by two exclamation points ("!!"). For example:

```
-- !x! SUB author Date
create or replace temporary view docs as
select * from documents
where author = '!!author!!';
```

Substitution variable names may contain only letters, digits, and the underscore character. Substitutions are processed in the order in which they are defined. Substitution variable definitions can themselves include substitution variables. SQL statements and metacommands may contain nested references to substitution variables, as illustrated in *Example 7*. Complex expressions using substitution variables can be evaluated using SQL, as illustrated in *Example 16*.

In addition to user-defined substitution variables, there are three additional kinds of substitution variables that are defined automatically by execsql or by specific metacommands. These are *system variables*, *data variables*, and *environment variables*. System, data, and environment variable names are prefixed with "$", "@", and "&" respectively. Because these prefixes cannot be used when defining substitution variables with the SUB metacommand, system variable, data variable, and environment variable names will not conflict with user-created variable names.

System Variables

Several special substitutions (pairs of matching strings and replacement strings) are automatically defined and maintained by execsql. The names and definitions of these substitution variables are:

$ARG_x The value of a substitution variable that has been assigned on the command line using the "-a" command-line option. The value of <x> must be an integer greater than or equal to 1. See *Example 9* for an illustration of the use of "$ARG_x" variables.

$AUTOCOMMIT_STATE A value indicating whether or not execsql will automatically commit each SQL statement as it is executed. This will be either "ON" or "OFF". The autocommit state is database specific, and the value applies only to the database currently in use.

$CANCEL_HALT_STATE The value of the status flag that is set by the *CANCEL_HALT* metacommand. The value of this variable is always either "ON" or "OFF". A modularlized sub-script can use this variable to access and save (in another substitution variable) the CANCEL_HALT state before changing it, so that the previous state can be restored.

$CONSOLE_WAIT_WHEN_DONE_STATE The value of the status flag that is set by the `console_wait_when_done` configuration setting or by the *CONSOLE WAIT_WHEN_DONE* metacommand. The value of this variable is always either "ON" or "OFF".

$CONSOLE_WAIT_WHEN_ERROR_STATE The value of the status flag that is set by the `console_wait_when_error_halt` configuration setting or by the *CONSOLE WAIT_WHEN_ERROR* metacommand. The value of this variable is always either "ON" or "OFF".

$COUNTER_x An integer value that is automatically incremented for every command that references the counter variable. As many counter variables as desired can be used. The value of x must be an integer that identifies the counter variable. Counter variable names do not have to be used sequentially. The first time that a counter variable is referenced, it returns the value 1. If a counter variable is referenced multiple times in one command, each reference will have the same value. The *RESET COUNTER* and *RESET COUNTERS* metacommands can be used to reset counter variables. See examples *6*, *7*, *11*, and *19* for illustrations of the use of counter variables.

$CURRENT_ALIAS The alias of the database currently in use, as defined by the *CONNECT* metacommand, or "initial" if no CONNECT metacommand has been used. This value will change if a different database is *USEd*.

$CURRENT_DATABASE The DBMS type and the name of the current database. This value will change if a different database is *USEd*.

$CURRENT_DBMS The DBMS type of the database in use. This value may change if a different database is *USEd*.

$CURRENT_DIR The full path to the current directory. The value will not have a directory separator character (i.e., "/" or "\") at the end.

$CURRENT_SCRIPT The file name of the script from which the current command originated. This value will change if a different script is *INCLUDEd*. This file name may or may not include a path, depending on how the script file was identified on the command line or in an INCLUDE metacommand.

$CURRENT_SCRIPT_NAME The base file name, without a path, of the script from which the current command originated. This value will change if a different script is *INCLUDEd*.

$CURRENT_SCRIPT_PATH The complete path of the script from which the current command originated, including a terminating path separator character. This value will change if a different script is *INCLUDEd*.

$CURRENT_TIME The date and time at which the current script line is run. See *Example 3* for an illustration of its use.

$DATE_TAG The date on which execsql started processing the current script, in the format YYYYMMDD. This is intended to be a convenient short form of the date that can be used to apply sequential version indicators to directory names or file names (e.g., of exported data). See *Example 2* for an illustration of its use.

$DATETIME_TAG The date and time at which execsql started processing the current script, in the format YYYYMMDD_hhmm. This is intended to be a convenient short form of the date and time that can be used to apply sequential versions to directory names or file names. See *Example 8* for an illustration of its use.

$DB_NAME The name of the database currently in use, as specified on the command line or in a *CONNECT* metacommand. This will be the database name for server-based databases, and the file name for file-based databases.

$DB_NEED_PWD A string equal to "TRUE" or "FALSE" indicating whether or not a password was required for the database currently in use.

$DB_SERVER The name of the database server for the database currently in use, as specified on the command line or in a CONNECT metacommand. If the database in use is not server-based, the result will be an empty string.

$DB_USER The name of the database user for the database currently in use, as specified on the command line or in a *CONNECT* metacommand. If the database connection does not require a user name, the result will be an empty string.

$ERROR_HALT_STATE The value of the status flag that is set by the *ERROR_HALT* metacommand. The value of this variable is always either "ON" or "OFF". A modularlized sub-script can use this variable to access and save (in another substitution variable) the ERROR_HALT state before changing it, so that the previous state can be restored.

$ERROR_MESSAGE The message generated by any error, as it would be printed on the terminal by default. This is initially an empty string, and is set by any SQL error or metacommand error. If an error occurs, the error message is only accessible if the *ERROR_HALT OFF* or *METACOMMAND_ERROR_HALT OFF* metacommands have been used, or in an *ON ERROR_HALT EMAIL* or *ON ERROR_HALT WRITE* metacommand.

$LAST_ERROR The text of the last SQL statement that encountered an error. This value will only be available if the *ERROR_HALT OFF* metacommand has been used.

$LAST_ROWCOUNT The number of rows that were affected by the last INSERT, UPDATE, or SELECT statement. Note that support for $LAST_ROWCOUNT varies among DBMSs. For example, for SELECT statements, Postgres provides an accurate count, SQLite always returns -1, and Firebird always returns 0.

$LAST_SQL The text of the last SQL statement that ran without error.

$METACOMMAND_ERROR_HALT_STATE The value of the status flag that is set by the *METACOMMAND_ERROR_HALT* metacommand. The value of this variable is always either "ON" or "OFF".

$OS The name of the operating system. This will be "linux", "windows", "cygwin", "darwin", "os2", "os2emx", "riscos", or "atheos".

$PYTHON_EXECUTABLE The path and name of the Python interpreter that is running execsql. This can be used with the *SYSTEM_CMD* metacommand to run a Python program in an operating-system-independent manner.

$RANDOM A random real number in the semi-open interval [0.0, 1.0).

$RUN_ID The run identifier that is used in execsql's log file.

$SCRIPT_LINE The line number of the current script for the current command.

$SCRIPT_START_TIME The date and time at which execsql started processing the current script. This value never changes within a single run of execsql.

$STARTING_SCRIPT The file name of the script specified on the command line when execsql is run. This value never changes within a single run of execsql. This file name may or may not include a path, depending on how it was specified on the command line.

$STARTING_SCRIPT_NAME The base file name of the script specified on the command line when execsql is run, without any path specification. This value never changes within a single run of execsql. This may or may not be the same as $STARTING_SCRIPT; the latter may include a path.

$SYSTEM_CMD_EXIT_STATUS The exit status of the command executed by the *SYSTEM_CMD* metacommand. The value is "0" (zero) prior to the first use of the SYSTEM_CMD metacommand.

$TIMER The elapsed time of the script timer. If the *TIMER ON* command has never been used, this value will be zero. If the timer has been started but not stopped, this value will be the elapsed time since the timer was started. If the timer has been started and stopped, this value will be the elapsed time when the timer was stopped.

$USER The name of the person logged in when the script is started. This is not necessarily the same as the user name used with any database.

$UUID A random 128-bit Universally Unique Identifier in the canonical form of 32 hexadecimal digits.

The system variables can be used for conditional execution of different SQL commands or metacommands, and for custom logging of a script's actions using the *WRITE* metacommand.

Data Variables

Two metacommands, *SELECT_SUB* and *PROMPT SELECT_SUB*, will each create a set of substitution variables that correspond to the data values in a single row of a data table. The column names of the data table, prefixed with "@", will be automatically assigned as the names of these data variables. The prefix of "@" cannot be assigned using *SUB* or similar metacommands, and so will prevent data variables from overwriting any user-defined substitution variables that may have the same name as a data table column. See *Example 8* for an illustration of the use of a data variable. All assignments to data variables are automatically logged.

Evironment Variables

The operating system environment variables that are defined when execsql starts will be available as substitution variables prefixed with "&". New environment variables cannot be added by any metacommand.

Metacommands to Assign Substitution Variables

In addition to the *SUB* metacommand, several other metacommands can be used to define substitution variables based on values in a data table, user input, or a combination of the two. All of the metacommands that can be used to define substitution variables are:

PROMPT DIRECTORY Opens a dialog box and prompts the user to identify an existing directory on the file system. The name of the substitution variable is specified in the metacommand, and the full path to the selected directory will be used as the replacement string.

PROMPT ENTER_SUB Opens a dialog box and prompts the user to interactively enter the text that will be used as a replacement string. The name of the substitution variable is specified in the metacommand.

PROMPT ENTRY_FORM Displays a custom data entry form and assigns each of the values entered to a specified substitution variable.

PROMPT OPENFILE Opens a dialog box and prompts the user to select an existing file. The name of the substitution variable is specified in the metacommand, and the full path to the selected file will be used as a replacement string.

PROMPT SAVEFILE Opens a dialog box and prompts the user to enter the name of a new or existing file; the full path to this file will be used as a replacement string.

PROMPT SELECT_SUB Opens a dialog box, displays a data table or view, and prompts the user to select a row. The data values on the selected row will be assigned to a set of data variables.

SELECT_SUB The data values on the first row of a specified table or view will be assigned to a set of data variables. No prompt is displayed.

SUB Directly assigns a replacement string to a substitution variable.

SUB_APPEND Appends text to a substitution variable. The appended text is separated from the existing text with a newline.

SUB_TEMPFILE Assigns a temporary file name to the specified substitution variable.

SUBDATA The data value in the first column of the first row of a specified table or view will be assigned to a user-specified substitution variable.

Substitution variables can also be defined in *configuration files*.

2.4.7 Metacommands

The execsql program supports several special commands–metacommands–that import and export data, conditionally execute parts of the script, report status information, and perform other actions. Some of the things that can be done with metacommands are:

- Include the contents of another SQL script file.
- Import data from a text file or spreadsheet to a new or existing table.
- Export data to the terminal or a file in a variety of formats.
- Connect to multiple databases and copy data between them.
- Write text out to the console or to a file.
- Stop or pause script processing.
- Display a data table for the user to review.
- Display a pair of data tables for the user to compare.
- Prompt the user to respond to a question or enter a value.
- Prompt for the names of files or directories to be used.
- Create sub-scripts that can be executed repeatedly.
- Conditionally execute SQL or metacommands based on data values or user input.
- Execute an operating system command.

Whereas SQL is often embedded in programs written in other languages, execsql inverts this paradigm through the use of metacommands (and *substitution variables*). These allow database operations to be interleaved with user interactions and file system access in a way that may be easier to develop, easier to *re-use*, and more accessible to multiple users than embedded SQL in a high-level programming language.

Metacommands recognized by execsql are embedded in SQL comments, and are identified by the token "`!x!`" immediately following the comment characters at the beginning of the line. Each metacommand must be completely on a single line. An example metacommand is:

```
-- !x! import to staging.weather from billings2012.csv
```

Other illustrations of metacommand usage are in the *examples*.

Because metacommands are embedded in comments, they are hidden from other SQL script processors such as *psql* for Postgres, *mysql* for MySQL/MariaDB, and *sqlcmd* for SQL Server. Thus, a script containing execsql metacommands can potentially also be run using a DBMS's own native script processor. Scripts that make extensive use of execsql's features, however, may not run satisfactorily with other script processors. Scripts that use metacommands such as *IF*, *IMPORT*, *INCLUDE*, *EXECUTE SCRIPT*, or *USE*, or that use *substitution variables* in SQL statements, are not likely to run as expected with other script processors.

Metacommands can appear anywhere in a SQL script except embedded inside a SQL statement. This restriction prohibits constructions such as:

```
select * from d_labresult
where
    lab = '!!selected_lab!!'
-- !x! if(sub_defined(selected_sdg))
    and sdg = '!!selected_sdg!!'
-- !x! endif
    ;
```

This will not work because metacommands are not executed at the time that SQL statements are read from the script file, but are run after the script has been parsed into separate SQL statements and metacommands. Instead, SQL statements can be dynamically constructed using substitution variables to modify them at runtime, like this:

```
-- !x! sub whereclause lab = '!!selected_lab!!'
-- !x! if(sub_defined(selected_sdg))
-- !x!     sub whereclause !!whereclause!! and sdg = '!!selected_sdg!!'
-- !x! endif
select * from d_labresult
where !!whereclause!!;
```

The metacommands are described in the following sections. Metacommand names are shown here in all uppercase, but execsql is not case-sensitive when evaluating the metacommands. The syntax descriptions for the metacommands use angle brackets to identify required replaceable elements, and square brackets to identify optional replaceable elements.

ASK

```
ASK "<question>" SUB <match_string>
```

Prompts for a yes or no response to the specified question, presenting the prompt on the console, and assigns the result, as either "Yes" or "No", to the substitution variable specified. The "Y" and "N" keys will select the corresponding response. The <Esc> key will cancel the script. The selection is also logged. If the prompt is canceled, script processing is halted, and the system exit value is set to 2.

See the *PROMPT ASK* metacommand for a version of this command that uses a GUI window and that can display a data table.

AUTOCOMMIT

```
AUTOCOMMIT ON|OFF
```

By default, execsql automatically commits each SQL statement individually. Setting AUTOCOMMIT off will change this behavior. The user is then responsible for explicitly issuing a "COMMIT;" statement to the database to ensure that all preceding SQL statements are executed.

Unlike *BATCH* metacommands, the SQL statements issued while AUTOCOMMIT is off will not be queued up and automatically run when AUTOCOMMIT is turned back on again. However, any SQL statements that are run after AUTOCOMMIT is turned back on will be automatically committed, and that commit operation will also commit any SQL statements that were issued while AUTOCOMMIT was off, unless a rollback statement was used as the last SQL statement while AUTOCOMMIT was off.

The AUTOCOMMIT metacommand is database-specific, and affects only the database in use when the metacommand is used. This contrasts with the *BATCH* metacommand, which affects all databases.

The *IMPORT* and *COPY* metacommands do not commit data changes while AUTOCOMMIT is off. The SQL statements generated by the *IMPORT* and *COPY* metacommands are sent to the database, however. Therefore the AUTOCOMMIT metacommand is recommended when explicit transaction control is to be applied to the *IMPORT* and *COPY* metacommands.

BATCH

```
BEGIN BATCH
```

```
END BATCH
```

```
ROLLBACK [BATCH]
```

The BATCH commands provide a sort of transaction control at the script level, as an alternative to using the *AUTOCOMMIT OFF* metacommand and the DBMS's own transaction commands. execsql ordinarily executes and commits SQL statements immediately (i.e., as if the database connection is set to autocommit, although execsql actually manages commit and rollback statements directly). The BATCH commands allow you to alter this behavior so that SQL statements are not committed until a batch is completed. This allows execsql to emulate tools that operate in batch mode by default (specifically, sqlcmd).

BEGIN BATCH marks the beginning of a set of SQL statements to be executed in a single operation. END BATCH marks the end of that set of statements. ROLLBACK BATCH sends a "rollback" command to the database to revert the action of any SQL statements that have already been executed in the batch, but does not terminate the batch.

Metacommands may be included inside a batch, but note that the *IMPORT* and *COPY* metacommands always commit the changes they make, so if these metacommands are used inside a batch, any preceding SQL statements in the batch will also be committed.

When the END BATCH metacommand is processed by execsql, a "commit" command will be sent to all databases that have been used inside the batch. Multiple databases may be used inside a batch if the *USE* metacommand is used inside the batch. The BATCH metacommands therefore provide a limited sort of cross-database transaction control.

The BEGIN/END BATCH metacommands can be nested. However, the inner END BATCH metacommand will commit all changes to the databases that have been used, which may include databases used in the outer batch as well. Therefore completion of a nested batch may result in premature commitment of some or all SQL statements in the outer batch. Similarly, a ROLLBACK BATCH metacommand within the inner batch will also roll back any SQL commands sent to the same databases in the outer batche. Thus, although the BATCH commands can be nested, database transactions cannot be. Nesting of BATCH metacommands allows a script file or a *SCRIPT* containing a batch to be *INCLUDEd* or *EXECUTEd*, respectively, within another batch.

Alternatives to using batches to control the execution time of SQL statements are:

- The *AUTOCOMMIT* metacommand, which provides a different method of integrating *IMPORT* and *COPY* metacommands with a sequence of SQL statements
- The *IF* metacommand, which provides a way of conditionally executing SQL statements and metacommands such as *IMPORT* and *COPY*
- The *BEGIN/END SCRIPT* and *EXECUTE SCRIPT* metacommands, which allow both SQL statements and metacommands to be grouped together and executed as a group, with AUTOCOMMIT either on or off.

The END BATCH metacommand is equivalent to the "GO" command of SQL Server utilities such as sqlcmd. There is no explicit equivalent to BEGIN BATCH in sqlcmd or other SQL Server utilities. In sqlcmd a new batch is automatically begun at the beginning of the script or immediately after a GO statement. execsqsl only starts a new batch when a BEGIN BATCH statement is encountered.

If the end of the script file is encountered while a batch of statements is being compiled, but there is no END BATCH metacommand, the SQL statements in that incomplete batch will not be committed.

BEGIN SCRIPT and END SCRIPT

```
BEGIN SCRIPT <script_name>
```

```
END SCRIPT
```

The BEGIN SCRIPT and END SCRIPT metacommands define a block of statements (SQL statements and metacommands) that can be subsequently executed (repeatedly, if desired) using the *EXECUTE SCRIPT* metacommand.

The statements within the BEGIN/END SCRIPT block are not executed within the normal flow of the script in which they appear, and, unlike the BEGIN/END BATCH commands, neither are they executed when the END SCRIPT metacommand is encountered. These statements are executed only when the corresponding script is named in an *EXECUTE SCRIPT* metacommand.

A BEGIN/END SCRIPT block can be used in ways similar to a separate script file that is included with the *INCLUDE* metacommand. Both allow the same code to be executed repeatedy, either at different locations in the main script or recursively to perform looping.

The BEGIN SCRIPT and END SCRIPT metacommands are executed when a script file is read, not while the the script is being executed. As a consequence:

- Substitution variables should ordinarily not be used as script names because they will not have been defined yet, unless they were defined in the variables section of a *configuration file*; and
- The BEGIN/END SCRIPT commands are not ordinarily subject to conditional execution.

However, the BEGIN SCRIPT and END SCRIPT metacommands can be used in a separate script file that is *INCLUDEd* in the main script. In this case, both of the previous restrictions are eliminated. In addition the *EXECUTE SCRIPT* metacommand can be included in a conditional statement.

"CREATE SCRIPT" can be used as an alias for "BEGIN SCRIPT".

CANCEL_HALT

```
CANCEL_HALT ON|OFF
```

When CANCEL_HALT is set to ON, which is the default, if the user presses the "Cancel" button on a dialog (such as is presented by the *PROMPT DISPLAY* metacommand), execsql will halt script processing. If CANCEL_HALT is set to OFF, then execsql will not halt script processing, and it is the script author's responsibility to ensure that adverse consequences do not result from the lack of a response to the dialog. *Example 10* illustrates a condition in which setting CANCEL_HALT to OFF is appropriate.

CONFIG

Several of the configuration settings that can be specified either with *command-line options* or in *configuration files* can also be dynamically altered using metacommands.

```
CONFIG BOOLEAN_INT YES|NO
```

Controls whether integer values of 0 and 1 are considered to be Booleans when the *IMPORT* and *COPY* metacommands scan data to determine data types to use when creating a new table (i.e, when either the NEW or REPLACEMENT keyword is used with the *IMPORT* and *COPY* metacommands.) The argument should be either "Yes" or "No". execsql's default behavior is to consider a column with only integer values of 0 and 1 to have a Boolean data type. By setting this value to "No", such a column will be considered to have an integer data type. This is equivalent to the "-b" command-line option and the `boolean_int` *configuration parameter*.

```
CONFIG BOOLEAN_WORDS YES|NO
```

Controls whether execsql will recognize only full words as Booleans when the *IMPORT* and *COPY* metacommands scan data to determine data types to use when creating a new table (i.e, when either the NEW or REPLACEMENT keyword is used with the *IMPORT* and *COPY* metacommands.). The argument should be either "Yes" or "No". execsql's default behavior is to recognize values of "Y", "N", "T", and "F" as Booleans. By setting BOOLEAN_WORDS to "Yes", then only "Yes", "No", "True", and "False" will be recognized as Booleans.

```
CONFIG CONSOLE WAIT_WHEN_DONE ON|OFF
```

Controls the persistence of any *console window* at the completion of the script when the script either completes normally or exits prematurely as a result of the user's response to a prompt. If the value is set to "ON" (the default value is "OFF"), the console window will remain open until explicitly closed by the user. The message "Script complete; close the console window to exit execsql." will be displayed in the status bar. This metacommand has the same action as the `console_wait_when_done` configuration setting. The value of this setting can be evaluated with the "$console_wait_when_done_state" *system variable*.

```
CONFIG CONSOLE WAIT_WHEN_ERROR ON|OFF
```

Controls the persistence of any *console window* at the completion of the script if an error occurs. If the value is set to "ON" (the default value is "OFF"), the console window will remain open until explicitly closed by the user after an error occurs. This metacommand has the same action as the `console_wait_when_error_halt` configuration setting. The value of this setting can be evaluated with the "$console_wait_when_error_state" *system variable*.

```
CONFIG EMPTY_STRINGS YES|NO
```

Controls whether empty strings are allowed in data that is saved using either the *IMPORT* or *COPY* metacommands. The default is to allow empty strings. A metacommand of EMPTY_STRINGS NO will cause all empty strings to be replaced by NULL. A string containing only space characters is considered to be an empty string.

```
CONFIG IMPORT_COMMON_COLUMNS_ONLY YES|NO
```

Controls whether the *IMPORT* metacommand will import CSV files with more columns than the target table. This has the same action as the `` `import_common_columns_only` *configuration setting* ``. The argument should be either "Yes" or "No". The default value is "No", in which case the *IMPORT* metacommand will halt with an error message if the target table does not have all of the columns that are in the file to be imported.

```
CONFIG LOG_WRITE_MESSAGES ON|OFF
```

Controls whether output of the *WRITE* metacommand will also be written to execsql's log file. When this is set to ON (the default value is OFF), all output of the *WRITE* metacommand will also be written to execsql's *log file*. This behavior can also be controlled with the `log_write_messages` configuration option.

```
CONFIG MAKE_EXPORT_DIRS YES|NO
```

Controls whether the *EXPORT* metacommand will automatically create any directories that are named in an output filename and that do not already exist. The user must have appropriate permissions to create those directories.

```
CONFIG MAX_INT <integer_value>
```

Specifies the threshold between integer and bigint data types that is used by the *IMPORT* and *COPY* metacommands when creating a new table. Any column with integer values less than or equal to this value (max_int) and greater than or equal to -1 × max_int - 1 will be considered to have an 'integer' type. Any column with values outside this range will be considered to have a 'bigint' type. The default value for max_int is 2147483647. The max_int value can also be altered using a configuration option.

CONNECT

For PostgreSQL:

```
CONNECT TO POSTGRESQL(SERVER=<server_name>, DB=<database_name>
      [, USER=<user>, NEED_PWD=TRUE|FALSE] [, PORT=<port_number>]
      [, PASSWORD=<password>] [, ENCODING=<encoding>] [, NEW]) AS <alias_name>
```

For SQLite:

```
CONNECT TO SQLITE(FILE=<database_file> [, NEW]) AS <alias_name>
```

For MS-Access:

```
CONNECT TO ACCESS(FILE=<database_file> [, NEED_PWD=TRUE|FALSE]
      [, PASSWORD=<password>] [, ENCODING=<encoding>]) AS <alias_name>
```

For SQL Server:

```
CONNECT TO SQLSERVER(SERVER=<server_name>, DB=<database_name>
      [, USER=<user>, NEED_PWD=TRUE|FALSE]  [, PORT=<port_number>]
      [, ENCODING=<encoding>]) AS <alias_name>
```

For MySQL:

```
CONNECT TO MYSQL(SERVER=<server_name>, DB=<database_name>
      [, USER=<user>, NEED_PWD=TRUE|FALSE]  [, PORT=<port_number>]
      [, PASSWORD=<password>] [, ENCODING=<encoding>]) AS <alias_name>
```

For MariaDB:

```
CONNECT TO MARIADB(SERVER=<server_name>, DB=<database_name>
      [, USER=<user>, NEED_PWD=TRUE|FALSE]  [, PORT=<port_number>]
      [, PASSWORD=<password>] [, ENCODING=<encoding>]) AS <alias_name>
```

For Firebird:

```
CONNECT TO FIREBIRD(SERVER=<server_name>, DB=<database_name>
      [, USER=<user>, NEED_PWD=TRUE|FALSE]  [, PORT=<port_number>]
      [, ENCODING=<encoding>]) AS <alias_name>
```

For a DSN:

```
CONNECT TO DSN(DSN=<DSN_name>,
      [, USER=<user>, NEED_PWD=TRUE|FALSE] [,
      PASSWORD=<password>] [, ENCODING=<encoding>]) AS <alias_name>
```

Establishes a connection to another database. The keyword values are equivalent to arguments and options that can be specified on the command line when execsql is run. The "NEW" keyword, used with PostgreSQL and SQLite, will cause a new database of the given name to be created. There must be no existing database of that name, and (for Postgres) you must have permissions assigned that allow you to create databases.

The CONNECT metacommands for Postgres, MySQL/MariaDB, Access, and DSN connections allow a password to be specified. If a password is needed for any database but is not provided, execsql will display a prompt for the password. Embedding a password in a SQL script is a security weakness, but may be needed when a script is to be run regularly as a system job. This risk can be minimized by either:

- Using the *PROMPT ENTER_SUB* metacommand to prompt for the password when the script starts and using the *PAUSE... CONTINUE* metacommand to control the timing of successive runs of a subscript; or

- Storing an encrypted copy of the password in a substitution variable and decrypting it before passing it to the *CONNECT* metacommand.

The alias name that is specified in this command can be used to refer to this database in the *USE* and *COPY* metacommands. Alias names can consist only of letters, digits, and underscores, and must start with a letter. The alias name "initial" is reserved for the database that is used when execsql starts script processing, and cannot be used with the *CONNECT* metacommand. If you re-use an alias name, the connection to the database to which that name was previously assigned will be closed, and the database will no longer be available. Using the same alias for two different databases allows for mistakes wherein script statements are run on the wrong database, and so is not recommended.

CONSOLE

```
CONSOLE ON|OFF
```

Creates (ON) or destroys (OFF) a GUI console to which subsequent *WRITE* metacommands will send their output. Data tables exported as text will also be written to this console. The console window includes a status line and progress bar indicator that can each be directly controlled by metacommands listed below.

Only one console window can be open at a time. If a "CONSOLE ON" metacommand is used while a console is already visible, the same console will remain open, and no error will be reported.

A GUI console can be automatically opened when execsql is started by using the "-v3" option.

When the GUI console is turned OFF, subsequent output will again be directed to standard output (the terminal window, if there is one open).

If an error occurs while the console is open, the error message will be written on standard error (typically the terminal) rather than in the console, and the console will be closed as execsql terminates.

```
CONSOLE HIDE|SHOW
```

Hides or shows the console window. Text will still be written to the console window while it is hidden, and will be visible if the console is shown again.

```
CONSOLE STATUS "<message>"
```

The specified message is written to the status bar at the bottom of the console window. Use an empty message ("") to clear the status message.

```
CONSOLE PROGRESS <number> [/ <total>]
```

The progress bar at the bottom of the console window will be updated to show the specified value. Values should be numeric, between zero and 100. If the number is followed by a slash and then another number, the two numbers will be taken as a fraction and converted to a percentage for display. Use a value of zero to clear the progress bar.

```
CONSOLE SAVE [APPEND] TO <filename>
```

Saves the text in the console window to the specified file. If the "APPEND TO" keyword is used, the console text will be appended to any existing file of the same name; otherwise, any existing file will be overwritten.

```
CONSOLE WAIT ["<message>"]
```

Script processing will be halted until the user responds to the console window with either the <Enter> key or the <Esc> key, or clicks on the window close button. If an (optional) message is included as part of the command, the message will be written into the status bar. If the user responds with the <Enter> key, the console window will remain open and script processing will resume. The user can close the console window either with the <Esc> key or by clicking on the window close button.

The console window has a single menu item, 'Save as...', that allows the entire console output to be saved as a text file.

COPY

```
COPY <table1_or_view> FROM <alias_name_1>
  TO [NEW|REPLACEMENT] <table2> IN <alias_name_2>
```

Copies the data from a data table or view in one database to a data table in a second database. The two databases between which data are copied are identified by the alias names that are established with the *CONNECT* metacommand. The alias "initial" can be used to refer to the database that is used when execsql starts script processing. Neither the source nor the destination database need be the initial database, or the database currently in use.

The second (destination) table must have column names that are identical to the names of the columns in the first (source) table. The second table may have additional columns; if it does, they will not be affected and their names don't matter. The data types in the columns to be copied must be compatible, though not necessarily identical. The order of the columns in the two tables does not have to be identical.

If the NEW keyword is used, the destination table will be automatically created with column names and data types that are compatible with the first (source) table. The data types used for the columns in the newly created table will be determined by a scan of all of the data in the first table, but may not exactly match those in the first table. If the destination table already exists when the NEW keyword is used, an error will occur.

If the REPLACEMENT keyword is used, the destination table will also be created to be compatible with the source table, but any existing destination table of the same name will be dropped first. execsql uses a "drop table" statement to drop an existing destination table, and this statement may not succeed if there are dependencies on that table (see the discussion of *implicit drop table statements*). If the destination table is not dropped, then data from the source table will be added to the existing table, or an error will occur if the table formats are not compatible.

If there are constraints on the second table that are not met by the data being added, an error will occur. If an error occurs at any point during the data copying process, no new data will be added to the second table.

The data addition to the target table is always committed. Therefore, the COPY metacommand generally should not be used within transactions or *BATCHes*.

COPY QUERY

```
COPY QUERY <<query>> FROM <alias_name_1>
  TO [NEW|REPLACEMENT] <table> IN <alias_name_2>
```

Copies data from one database to another in the same manner as the *COPY* metacommand, except instead of specifying the source table (or view), a SQL query statement is used instead. The SQL statement must be terminated with a semicolon and enclosed in double angle brackets.

Like all metacommands, this metacommand must appear on a single line, although the SQL statement may be quite long. To facilitate readability, the SQL statement may be saved in a *substitution variable* and that substitution variable referenced in the COPY QUERY metacommand.

The data addition to the target table is always committed. Therefore, the COPY QUERY metacommand generally should not be used within transactions or *BATCHes*.

EMAIL

```
EMAIL FROM <from_address> TO <to_addresses>
      SUBJECT "<subject>" MESSAGE "<message_text>"
      [MESSAGE_FILE "<filename>"]
      [ATTACH_FILE "<attachment_filename>"]
```

Sends an email. The *from_address* should be a valid email address (though not necessarily a real one). The *to_addresses* should also be a valid email address, or a comma- or semicolon-delimited list of email addresses. If none of the destination email addresses are valid, an exception will occur and execsql will halt. If at least one of the email addresses is valid, the command will succeed.

The subject and the message_text should both be enclosed in double quotes and should not contain a double quote. Multiline messages can be used if the message text is contained in a *substitution variable*.

If the MESSAGE_FILE keyword is used, the contents of that file will be inserted into the body of the email message in addition to whatever message_text is specified. The filename may be unquoted, but must be quoted if it contains any space characters.

If the ATTACH_FILE keyword is used, the specified file will be attached to the email message. The attachment_filename may be unquoted, but must be quoted if it contains any space characters.

The SMTP host and any other connection information that is necessary must be specified in the *"email" section of a configuration file*.

ERROR_HALT

```
ERROR_HALT ON|OFF
```

When ERROR_HALT is set to ON, which is the default, any errors that occur as a result of executing a SQL statement will cause an error message to be displayed immediately, and execsql will exit. When ERROR_HALT is set to OFF, then SQL errors will be ignored, but can be evaluated with the *IF SQL_ERROR* conditional.

When ERROR_HALT is set to OFF inside a transaction, any SQL error will ordinarily cause the entire transaction to fail.

EXECUTE

```
EXECUTE <procedure_name>
```

Executes the specified stored procedure (or function, or query, depending on the DBMS). Conceptually, the EXECUTE metacommand is intended to be used to execute stored procedures that do not require arguments and do not return any values. The actual operation of this command differs depending on the DBMS that is in use.

Postgres has stored functions. Functions with no return value are equivalent to stored procedures. When using Postgres, execsql treats the argument as the name of a stored function. It appends an empty pair of parentheses to the function name before calling it, so you should not include the parentheses yourself; the reason for this is to maintain as much compatibility as possible in the metacommand syntax across DBMSs.

Access has only stored queries, which may be equivalent to either a view or a stored procedure in other DBMSs. When using Access, the query referenced in this command should be an INSERT, UPDATE, or DELETE statement—executing a SELECT statement in this context would have no purpose.

SQL Server has stored procedures. When using SQL Server, execsqsl treats the argument as the name of a stored procedure.

SQLite does not support stored procedures or functions, and (unlike Access queries), views can only represent SELECT statements. When using SQLite, execsql cannot treat the argument as a stored procedure or function, so it treats

it as a view and carries out a SELECT * FROM <procedure_name>; statement. This is unlikely to be very useful in practice, but it is the only reasonable action to take with SQLite.

MySQL and MariaDB support stored procedures and user-defined functions. User-defined functions can be invoked within SQL statements, so execsql considers the argument to the EXECUTE metacommand to be the name of a stored procedure, and calls it after appending a pair of parentheses to represent an empty argument list.

Firebird supports stored procedures, and execsql executes the procedure with the given name, providing neither input parameters nor output parameters.

EXECUTE SCRIPT

```
EXECUTE SCRIPT <script_name>
```

This metacommand will execute the set of SQL statements and metacommands that was previously defined and named using the *BEGIN/END SCRIPT* metacommands.

EXPORT

```
EXPORT <table_or_view> [TEE] [APPEND] TO <filename>|stdout
    AS <format> [DESCRIPTION "<description>"]
```

```
EXPORT <table_or_view> [TEE] [APPEND] TO <filename>|stdout
    WITH TEMPLATE <template_file>
```

Exports data to a file. The data set named in this command must be an existing table or view. The output filename specified will be overwritten if it exists unless the APPEND keyword is included. If the output name is given as "stdout", the data will be sent to the console instead of to a file. If specified by the "-d" command-line option or the *make_export_dirs* configuration option, execsql will automatically create the output directories if needed.

If the TEE keyword is used, the data will be exported to the terminal in the TXT format (as described below) in addition to whatever other type of output is produced.

The EXPORT metacommand has two forms, as shown above. The first of these will export the data in a variety of established formats, and the second of which will use one of several different template processors with a template specification file. The first form is more convenient if any of the supported formats is suitable, and the latter form allows more flexible customization of the output.

Exporting Data to Specific Supported Formats

The format specification in the first form of the EXPORT metacommand controls how the data table is written. The allowable format specifications and their meanings are:

B64 Data decoded from a base64-encoded format with no headers, quotes, or delimiters between either columns or rows. This is similar to the RAW export option except that base64-decoding is performed. This format is intended to be used for export of base64-encoded binary data such as images, and ordinarily should be used to export a single value. No description text will be included in the output even if it is provided.

CSV Comma-delimited with double quotes around text that contains a comma or a double quote. Column headers will not be written if the APPEND keyword is used. No description text will be included in the output even if it is provided.

HTML Hypertext markup language. If the APPEND keyword is not used, a complete web page will be written, with meta tags in the header to identify the source of the data, author, and creation date; simple CSS will be defined in the header to format the table. If the APPEND keyword is used, only the table will be written to the output

file. If the APPEND keyword is used and the output file contains a </body> tag, the table will be written before that tag rather than at the physical end of the file. The HTML tags used to create the table have no IDs, classes, styles, or other attributes applied. Custom CSS can be specified in *configuration files*. If the DESCRIPTION keyword is used, the given description will be used as the table's caption.

JSON Javascript Object Notation. The data table is represented as an array of JSON objects, where each object represents a row of the table. Each row is represented as a set of key:value pairs, with column names used as the keys. No description text will be included in the output even if it is provided.

LATEX Input for the LaTeX typesetting system. If the APPEND keyword is not used, a complete document (of class article) will be written. If the APPEND keyword is used, only the table definition will be written to the output file. If the APPEND keyword is used and an existing output file contains an \end{document} directive, the table will be written before that directive rather than at the physical end of the file. Wide or long tables may exceed LaTeX's default page size. If the DESCRIPTION keyword is used, the given description will be used as the table's caption.

ODS OpenDocument spreadsheet. When the APPEND keyword is used, each data set that is exported will be on a separate worksheet. The name of the view or table exported will be used as the worksheet name. If this conflicts with a sheet already in the workbook, a number will be appended to make the sheet name unique. (If a workbook with sheet names longer than 31 characters is opened in Excel, the sheet names will be truncated.) A sheet named "Datasheets" will also be created, or updated if it already exists, with information to identify the author, creation date, description, and data source for each data sheet in the workbook.

PLAIN Text with no header row, no quoting, and columns delimited by a single space. This format is appropriate when you want to export text—see *Example 11* for an illustration of its use. No description text will be included in the output even if it is provided.

RAW Data exactly as stored with no headers, quotes, or delimiters between either columns or rows. This format is most suitable for export of binary data, and ordinarily should be used to export a single value. No description text will be included in the output even if it is provided.

TAB or TSV Tab-delimited with no quoting. Column headers will not be written if the APPEND keyword is used. No description text will be included in the output even if it is provided.

TABQ or TSVQ Tab-delimited with double quotes around any text that contains a tab or a double quote. Column headers will not be written if the APPEND keyword is used. No description text will be included in the output even if it is provided.

TXT Text with data delimited and padded with spaces so that values are aligned in columns. Column headers are underlined with a row of dashes. Columns are separated with the pipe character (|). Column headers are always written, even when the APPEND keyword is used. This output is compatible with Markdown pipe tables—see *Example 8*. If the DESCRIPTION keyword is used, the given description will be written as plain text on the line before the table. If any columns of the table contain binary data, a message identifying the size, in bytes, of the data will be displayed instead of the data itsef.

TXT-ND This is the same as the TXT format, except that table cells where data are missing are filled with "ND" instead of being blank. Some tables with blank cells are not parsed correctly by pandoc, and this format ensures that no cells are blank. If the DESCRIPTION keyword is used, the given description will be written as plain text on the line before the table.

US Text with the unit separator (Unicode 001F) as the column delimiter, and no quoting. Column headers will not be written if the APPEND keyword is used. No description text will be included in the output even if it is provided.

VALUES Data are written into the output file in the format of a SQL INSERT... VALUES statement. The name of the target table is specified in the form of a substitution variable named target_table; the format of the complete statement is:

```
insert into !!target_table!!
    (<list of column headers>)
```

(continues on next page)

(continued from previous page)

```
values
    (<Row 1 data>),
    (<Row 2 data>),
    ...
    (<Row N data>)
    ;
```

If the DESCRIPTION keyword is used, the description text will be included as a SQL comment before the INSERT statement. The *INCLUDE* metacommand can be used to include a file written in this format, and the target table name filled in with an appropriately-named substitution variable. This output format can also be used to copy data between databases when it is not possible to use execsql's *CONNECT* and *COPY* metacommands.

Exporting Data Using a Template

Template-based exports provide a simple form of report generation or mail-merge capability. The template used for this type of export is a freely-formatted text file containing placeholders for data values, plus whatever additional text is appropriate for the purpose of the report. The exported data will therefore not necessarily be in the form of a table, but may be presented as lists, embedded in paragraphs of text, or in other forms.

execsql supports three different template processors, each with its own syntax. The template processor that will be used is controlled by the `template_processor` *configuration* property. These processors and the syntax they use to refer to exported data values are:

The default (no template processor specified) Data values are referenced in the template by the column name prefixed with a dollar sign or enclosed in curly braces prefixed with a dollar sign. For example if an exported data table contains a column named "vessel", that column could be referred to in either of these ways:

```
Survey operations were conducted from $vessel.
The ${vessel}'s crew ate biscuits for a week.
```

The default template processor does not include any features that allow for conditional tests or iteration within the template. The entire template is processed for each row in the exported data table, and all of the output is combined into the output file.

Jinja Data values are referenced in the template within pairs of curly braces. The Jinja template processor allows conditional tests and iteration, as well as other features, within the template. The entire exported data set is passed to the template processor as an iterable object named "datatable". The names of the column headers are passed as a separate iterable object named "headers". For example, if an exported data table contains a column named "hire_date", that column could be referred to, while iterating over the entire data set, as follows:

```
{% for row in datatable %}
Hire date: {{ row.hire_date }}
. . .
{% endfor %}
```

The template syntax used by Jinja is very similar to that used by Django. Jinja's Template Designer Documentation provides more details about the template syntax.

Airspeed Data values are referenced in the template by the name (or object) name prefixed with a dollar sign, or enclosed in curly braces and prefixed with a dollar sign, just as for the default template processor. The Airspeed template processor also allows conditional tests and iteration, and as with Jinja, the entire exported data set is passed to the template processor as an iterable object named "datatable". The names of the column headers are passed as a separate iterable object named "headers". For example, if an exported data set contains bibliographic information, those columns could be referenced, while iterating over the entire data set, to produce a BibTex bibliography, as follows:

```
#foreach ($doc in $datatable)
@$doc.doc_type {$doc.doc_id,
    author = {$doc.author},
    title  = {$doc.title},
    . . .
    }
#end
```

The template syntax used by Airspeed duplicates that used by Apache Velocity, and the Velocity User's Guide and Reference Guide provide details about the template syntax.

The Jinja and Airspeed template processors are both more powerful than the default, but as a result are also more complex. The different alternatives may be suitable for different purposes, or for different users, based on prior experience. One potentially important difference between Jinja and Airspeed is that Airspeed requires that the entire data set be processed at once, whereas Jinja does not; for very large data sets, therefore, Airspeed could encounter memory limitations.

EXPORT QUERY

```
EXPORT QUERY <<query>> [TEE] [APPEND] TO <filename>|stdout
    AS <format>
    [DESCRIPTION "<description>"]
```

```
EXPORT QUERY <<query>> [TEE] [APPEND] TO <filename>|stdout
    WITH TEMPLATE <template_file>
```

Exports data in the same manner as the *EXPORT* metacommand, except that the data source is a SQL query statement that is contained in the metacommand rather than a database table or view. The SQL query statement must be terminated with a semicolon and enclosed in double angle brackets (i.e., literally "<<" and ">>").

Like all metacommands, this metacommand must appear on a single line, although the SQL statement may be quite long. To facilitate readability, the SQL statement may be saved in a *substitution variable* and that substitution variable referenced in the EXPORT QUERY metacommand.

HALT

```
HALT MESSAGE "<error_message>" [EXIT_STATUS <n>]
```

Script processing is halted, and the execsql.py program terminates. If an error message is provided, it is written to the console, unless the "-v2" or "-v3" option is used, in which case the message is displayed in a dialog. If an EXIT_STATUS value is specified, the system exit status is set to that value, otherwise, the system exit status is set to 3.

Warning: A backward-incompatible change to HALT MESSAGE was made in version 1.26.1.0 (2018-06-13): the default exit status was changed from 2 to 3.

HALT DISPLAY

```
HALT MESSAGE "<error_message>" [DISPLAY <table_or_view>]
  [EXIT_STATUS <n>]
```

Script processing is halted, and the error message is displayed in a GUI window. If a table or view name is provided, the data from that table or view is also displayed. If an EXIT_STATUS value is specified, the system exit status is set to that value, otherwise, the system exit status status is set to 3.

Warning: A backward-incompatible change to HALT DISPLAY was made in version 1.26.1.0 (2018-06-13): the default exit status was changed from 2 to 3.

IF

The IF metacommand allows you to test for certain conditions and control which script statements are subsequently executed. There are two forms of the IF metacommand:

- A single-line IF statement that will conditionally run a single metacommand.
- A multi-line IF statement that must be terminated with an ENDIF metacommand. The multi-line form supports ELSE, ELSEIF, ANDIF, and ORIF clauses.

The syntax for the single-line IF metacommand is:

```
IF([NOT] <conditional test>) {<metacommand>}
```

The conditional tests that can be used are listed below. For the single-line form of the IF metacommand, the metacommand to be executed must be enclosed in curly braces following the conditional test.

The syntax for the multi-line IF metacommand can take several forms, depending on whether the additional ELSE, ELSEIF, ANDIF, and ORIF clauses are used. The simplest form of the multi-line IF metacommand is:

```
IF([NOT] <conditional test>)
    <SQL statements and metacommands>
ENDIF
```

Multi-line IF metacommands can be nested within one another, and single-line IF metacommands can appear within a multi-line IF metacommand.

The ELSE clause allows you to conditionally execute either of two sets of script commands. The form of this set of statements is:

```
IF([NOT] <conditional test>)
    <SQL statements and metacommands>
ELSE
    <SQL statements and metacommands>
ENDIF
```

The ELSEIF clause combines the actions of the ELSE clause with another IF metacommand—effectively, nesting another IF metacommand within the ELSE clause, but not requiring a second ENDIF statement to terminate the nested conditional test. The form of this set of statements is:

```
IF([NOT] <conditional test>)
    <SQL statements and metacommands>
ELSEIF([NOT] <conditional test>)
    <SQL statements and metacommands>
ENDIF
```

Multiple ELSEIF clauses can be used within a single multi-line IF metacommand. An ELSE clause can be used in combination with ELSEIF clauses, but this is not recommended because the results are not likely to be what you expect—the ELSE keyword only inverts the current truth state, it does not provide an alternative to all preceding

ELSEIF clauses. To achieve the effect of a case or switch statement, use only ELSEIF clauses without a final ELSE clause.

The ANDIF clause allows you to test for the conjunction of two conditionals without having to nest IF metacommands and use two ENDIF statements. The simplest form of usage of the ANDIF clause is:

```
IF([NOT] <conditional test>)
ANDIF([NOT] <conditional test>)
    <SQL statements and metacommands>
ENDIF
```

The ANDIF clause does not have to immediately follow the IF metacommand. It could instead follow an ELSE statement, or appear anywhere at all within a multi-line IF metacommand. Usage patterns other than that illustrated above may be difficult to interpret, however, and nested IF metacommands may be preferable to complex uses of the ANDIF clause.

The ORIF clause is similar to the ANDIF clause, but allows you to test the disjunction of two conditionals. The simplest form of usage of the ORIF clause is:

```
IF([NOT] <conditional test>)
ORIF([NOT] <conditional test>)
    <SQL statements and metacommands>
ENDIF
```

The IF metacommands can be used not only to control a single stream of script commands, but also to loop over sets of SQL statements and metacommands, as shown in *Example 6*.

The conditional tests that can be used with IF and WAIT_UNTIL metacommands are listed in the following subsections.

ALIAS_DEFINED test

```
ALIAS_DEFINED(<alias>)
```

Evaluates whether a database connection has been made using the specified alias. Database aliases are defined using the *CONNECT* and *PROMPT CONNECT* metacommands.

COLUMN_EXISTS test

```
COLUMN_EXISTS(<column_name> IN <table_name>)
```

Evaluates whether there is a column of the given name in the specified database table. The table name may include a schema. execsql queries the information schema tables for those DBMSs that have information schema tables. You must have permission to use these system tables. If you do not, an alternative approach is to try to select data from the specified column table and determine if an error occurs.

DATABASE_NAME test

```
DATABASE_NAME(<database_name>)
```

Evaluates whether the current database name matches the one specified. Database names used in this conditional test should exactly match those contained in the "$CURRENT_DATABASE" *substitution variable*.

DBMS test

```
DBMS(<dbms_name>)
```

Evaluates whether the current DBMS matches the one specified. DBMS names used in this conditional test should exactly match those contained in the "$CURRENT_DBMS" *substitution variable*.

DIRECTORY_EXISTS test

```
DIRECTORY_EXISTS(<directory_name>)
```

Evaluates whether there is an existing directory with the given name.

EQUAL test

```
EQUAL("<string_1>", "<string_2>")
```

Evaluates whether the two values are equal. The two string representations of the values first are converted to a normalized Unicode form (Normal Form C) and then are compared as integers, floating-point values, date/time values with a time zone, date/time values, dates, Boolean values, and strings. String comparisons are case insensitive. The first of these data types to which both values can be successfully converted is the basis for determining whether the values are equal. This test is as forgiving as possible, and returns True whenever the two values are plausibly the same. See also *IDENTICAL*.

FILE_EXISTS test

```
FILE_EXISTS(<filename>)
```

Evaluates whether there is a disk file of the given name.

HASROWS test

```
HASROWS(<table_or_view)
```

Evaluates whether the specified table or view has a non-zero number of rows.

IDENTICAL test

```
IDENTICAL("<string_1>", "<string_2>")
```

Evaluates whether the two quoted strings are exactly identical. No Unicode normalization is done, and the comparison is case-sensitive. This test is as unforgiving as possible, and returns False whenever the two values are not exactly the same. See also *EQUAL*.

IS_GT test

```
IS_GT(<value1>, <value2>)
```

Evaluates whether or not the first of the specified values is greater than the second value. If the values are not numeric, an error will occur, and script processing will halt.

IS_GTE test

```
IS_GTE(<value1>, <value2>)
```

Evaluates whether or not the first of the specified values is greater than or equal to the second value. If the values are not numeric, an error will occur, and script processing will halt.

IS_NULL test

```
IS_NULL("<value>")
```

Evaluates whether or not the specified value is null—that is, whether it is a zero-length string.

IS_TRUE test

```
IS_TRUE(<value>)
```

Evaluates whether or not the specified value represents a Boolean value of True. Values of "Yes", "Y", "True", "T", and "1" are considered to represent True values; anything else is considered to represent a False value. The values are not case-sensitive and should not be quoted.

IS_ZERO test

```
IS_ZERO(<value>)
```

Evaluates whether or not the specified value is equal to zero. If the value is not numeric, an error will occur, and script processing will halt.

METACOMMAND_ERROR test

```
METACOMMAND_ERROR()
```

Evaluates whether the previous metacommand generated an error. This test for SQL errors will only be effective if the *METACOMMAND_ERROR_HALT OFF* metacommand has previously been issued. This conditional must be used in the first metacommand after any metacommand that might have encountered an error.

NEWER_DATE test

```
NEWER_DATE(<filename>, <date>)
```

Evaluates whether the specified file was last modified after the given date. This can be used, for example, to compare the date of an output file to the latest revision date of all the data rows that should be included in the output; if the data have been revised after the output file was created, the output file should be regenerated.

NEWER_FILE test

```
NEWER_FILE(<filename1>, <filename2>)
```

Evaluates whether the first of the specified files was last modified after the second of the files. This can be used, for example, to compare the date of an output file to the date of the script file that produces that output; if the script is newer, it may be *INCLUDEd* to run it again.

SCHEMA_EXISTS test

```
SCHEMA_EXISTS(<schema_name>)
```

Evaluates whether or not the specified schema already exists in the database. For DBMSs that do not support schemas (SQLite, MySQL, MariaDB, Firebird, and Access), this will always return a value of False. execsql queries the information schema tables, or analogous tables, for this information. You must have permission to use these system tables.

SQL_ERROR test

```
SQL_ERROR()
```

Evaluates whether the previous SQL statement generated an error. Errors will result from badly-formed SQL, reference to non-existent database objects, lack of permissions, or database locks. A query (e.g., an update query) that does not do exactly what you expect it to will not necessarily cause an error to occur that can be identified with this statement. This test for SQL errors will only be effective if the *ERROR_HALT OFF* metacommand has previously been issued.

Errors in metacommands and some other errors encountered by execsql will cause the program to halt immediately, regardless of the setting of *ERROR_HALT* or the use of the IF(SQL_ERROR()) test.

SUB_DEFINED test

```
SUB_DEFINED(<match_string>)
```

Evaluates whether a replacement string has been defined for the specified substitution variable (matching string).

TABLE_EXISTS test

```
TABLE_EXISTS(<tablename>)
```

Evaluates whether there is a database table of the given name. execsql queries the information schema tables, or analogous tables, for this information. You must have permission to use these system tables. If you do not, an alternative approach is to try to select data from the table and determine if an error occurs; for example:

```
-- !x! error_halt off
select count(*) from maybe_not_a_real_table;
-- !x! error_halt on
-- !x! if(sql_error())
```

VIEW_EXISTS test

```
VIEW_EXISTS(<viewname>)
```

Evaluates whether there is a database view of the given name. For Access, this tests for the existence of a query of the given name. execsql queries the information schema tables, or analogous tables, for this information. You must have permission to use these system tables. If you do not, the alternative approach described for the *TABLE_EXISTS* conditional can be used.

IMPORT

Imports tabular data from a file into a new or existing database table. Data can be imported from either a text file or a spreadsheet. The syntax of the IMPORT metacommand for importing data from a text file is:

```
IMPORT TO [NEW|REPLACEMENT] <table_name> FROM <file_name>
    [WITH [QUOTE <quote_char> DELIMITER <delim_char>]
    [ENCODING <encoding>]]
    [SKIP <lines>]
```

The syntax for importing data from an OpenDocument spreadsheet is:

```
IMPORT TO [NEW|REPLACEMENT] <table_name> FROM <file_name>
    SHEET <sheet_name> [SKIP <rows>]
```

The syntax for importing data from an Excel spreadsheet is:

```
IMPORT TO [NEW|REPLACEMENT] <table_name> FROM EXCEL <file_name>
    SHEET <sheet_name>
```

Column names in the input must be valid for the DBMS in use.

If the "WITH QUOTE <quote_char> DELIMITER <delim_char>" clause is not used with text files, execsql will scan the text file to determine the quote and delimiter characters that are used in the file. By default, the first 100 lines of the file will be scanned. You can control the number of lines scanned with the "-s" option on execsql's command line. If the "WITH…" clause is used, the file will not be scanned to identify the quote and delimiter characters regardless of the setting of the "-s" option.

execsql will read CSV files containing newlines embeded in delimited text values. Scanning of a CSV file to determine the quote and delimiter characters may produce incorrect results if most of the physical lines scanned consist of text that makes up only part of a logical data column.

The quoting characters that will be recognized in a text file, and that can be specified in the "WITH…" clause are the double quote (`"`) and the single quote (`'`). If no quote character is used in the file, this can be specified in the metacommand as "WITH QUOTE NONE".

The delimiter characters that will be recognized in a text file, and that can be specified in the "WITH…" clause are the comma (`,`), semicolon (`;`), vertical rule (`|`), tab, and the unit separator (Unicode 001F). To specify that the tab character is used as a delimiter, use "WITH…DELIMITER TAB", and to specify that the unit separator is used as a delimiter, use "WITH…DELIMITER US".

The SKIP key phrase specifies the number of lines (or rows) at the beginning of the file (or worksheet) to discard before evaluating the remainder of the input as a data table.

If the NEW keyword is used, the input will be scanned to determine the data type of each column, and a CREATE TABLE statement run to create a new table for the data. Scanning of the file to determine data formats is separate from the scanning that is done to determine the quote and delimiter characters. If the table already exists when the NEW keyword is used, a fatal exception will result. If the REPLACEMENT keyword is used, the result is the same as if the NEW keyword were used, except that an existing table of the given name will be deleted first. If the table does not exist, an informational message will be written to the log.

If a table is scanned to determine data types, any column that is completely empty (all null) will be created with the text data type. This provides the greatest flexibility for subsequent addition of data to the table. However, if that column ought to have a different data type, and a WHERE clause is applied to that column assuming a different data type, the DBMS may report an error because of incomparable data types.

The handling of Boolean data types when data are imported depends on the capabilities of the DBMS in use. See the relevant section of the *SQL syntax notes*.

If a column of imported data contains only numeric values, but any non-zero value has a leading digit of "0", that column will be imported as a text data type (character, character varying, or text).

When execsql generates a CREATE TABLE statement, it will quote column names that contain any characters other than letters, digits, or the underscore ("_"). A mixture of uppercase and lowercase letters in a column name is not taken as an indication that a quoted identifier should be used for the column name, and execsql does not fold column names to either uppercase or lowercase. Case sensitivity and case-folding behavior varies between DBMSs, and execsql leaves it to the user to manage these differences.

The case-folding behavior of the DBMS should also be considered when specifying the table name in the IMPORT metacommand. When execsql checks to see if a table exists, it queries the information schema using the table name exactly as given (i.e., execsql does not do any case folding); if the actual table name differs because of case folding by the DBMS, the check will fail and an error will occur.

If neither the NEW or REPLACEMENT keywords are used, the table must already exist, and have column names identical to those in the file, and in the same order. The data types in the table must also be compatible with those in the file.

If the NEW keyword is used, the target table will be created without a primary key or other constraints. If data are imported to an existing table, they must meet any constraints already in place on that table. If data are imported to an existing table, the imported data will be added to any already-existing data. If existing data are to be replaced, they should be deleted before the IMPORT metacommand is run.

The NEW keyword cannot be used within a *batch* with Firebird. Firebird requires that the CREATE TABLE statement be committed—the table actually created—before data can be added. There is only one commit statement for a batch, at the end of the batch, and therefore the CREATE TABLE statement is not committed before data are added.

If the ENCODING keyword is not used, the character encoding of text files imported with the IMPORT metacommand is as specified with the "-i" command-line option or the corresponding *configuration file option*. If not specified in either of these ways, the encoding is assumed to be UTF-8. If a UTF byte order mark is found at the start of a data file, the encoding indicated by that marker will be taken as definitive regardless of the ENCODING keyword or the "-i" option.

By default, the target table must have all of the columns that are present in the text file to be imported. If it does not, an error will result. The `import_common_columns_only` *configuration parameter* can be used to allow import of data from text files with more columns than the target table. In either case, the target table may have more columns than the text file being imported.

Under some circumstances, import of data from text files to Postgres and MySQL/MariaDB uses the fast file reading features provided by both of those databases: Postgres' COPY command and MySQL's LOAD DATA LOCAL INFILE command. The text file and the target table must have exactly the same columns, in the same order, for the fast file reading routines to be used. In addition, the "empty_strings" configuration setting must be set to "Yes" (the default). If

these conditions are not satisfied, or, for Postgres, if the file encoding is of a type that is not recognized by Postgres (see https://www.postgresql.org/docs/current/static/multibyte.html), a slower loading routine will be used, with encoding conversion handled by execsql. Explicitly setting either the encoding or the quote and delimiter characters in the metacommand will cause execsql to use its own import routine instead of the fast file reading features of Postgres or MySQL.

The sheet name used when importing data from a spreadsheet can be either the sheet name, as it appears on the tab at the bottom of the sheet, or the sheet number. Comparison of the actual sheet names to the value given is case-insensitive. Sheet numbers start at 1.

When MS-Excel saves an OpenDocument spreadsheet, it may create an additional empty column to the right of all data columns. This spurious column is not eliminated by opening and re-saving the spreadsheet using LibreOffice Calc (as of version 5.0.2 at least). The IMPORT metacommand will report an error with such a file because of the absence of a column header on the extra column. To avoid this problem, as well as other issues related to incorrect implementation of the OpenDocument standard in Excel, and the data corruption that can occur when Excel imports and exports CSV files, and the ambiguous representation of dates in Excel, Excel should not be used for data that may be transferred to or from databases or other formats. Import of data from Excel may also take 10-100 times longer—or more—than import from a text file.

Because the data addition to the target table is always committed, the IMPORT metacommand generally should not be used within transactions or *BATCHes*.

Some performance considerations when using IMPORT are:

- Creating the table using a separate CREATE TABLE statement before the IMPORT metacommand will be faster than using the NEW or REPLACEMENT keywords. The time required for execsql to scan an entire file to determine data types can be much greater than the time required to import the file.
- When importing to Postgres from a text file that has an encoding that is recognized by Postgres, data are read and processed in chunks that are 32 kb in size. A larger or smaller value may give better performance, depending on system-specific conditions. The "-z" command-line option can be used to alter the buffer size.

In general, if an error occurs while importing data, none of the new data should be in the target table (the operation is not committed). However, MySQL/MariaDB may issue messages about data type incompatibility to the standard error device (ordinarily the terminal), yet load some or all of the data. If the NEW or REPLACEMENT keywords are used, depending on the DBMS and where the error occurred, the target table may be created even if the data are not loaded.

The name, size, and date of the IMPORTed file are written to the execsql.log file.

IMPORT_FILE

Imports an entire file into a single column, on a new row, of an existing database table. The syntax of the IMPORT_FILE metacommand is:

```
IMPORT_FILE TO TABLE <table_name> COLUMN <column_name> FROM <file_name>
```

The data type of the column must allow insertion of binary data. If the table contains any other columns that must be non-null, those columns must have default values.

INCLUDE

```
INCLUDE <filename>
```

The specified file should be a script that contains SQL statements and/or metacommands. Those SQL statements and metacommands will be inserted into the script at the point where the INCLUDE metacommand occurs.

LOG

```
LOG "<message>"
```

Writes the specified message to execsql's *log file*.

METACOMMAND_ERROR_HALT

```
METACOMMAND_ERROR_HALT ON|OFF
```

When METACOMMAND_ERROR_HALT is set to ON, which is the default, any errors that occur during execution of a metacommand will cause an error message to be displayed immediately, and execsql to exit. When METACOMMAND_ERROR_HALT is set to OFF, then metacommand errors will be ignored, but can be evaluated with the IF METACOMMAND_ERROR conditional.

The METACOMMAND_ERROR_HALT metacommand does not itself set or reset the internal flag that indicates whether a metacommand has encountered an error. This is so that a "METACOMMAND_ERROR_HALT Off" command can be used immediately after a potentially-failing metacommand and not alter the error flag that is set by the previous command. Specifically, constructions like this:

```
-- !x! metacommand_error_halt off
-- !x! connect to postgresql(server=none, db=imaginary, user=nobody, need_pw=False)␣
↪as pg
-- !x! metacommand_error_halt on
-- !x! if(metacommand_error())
```

can be used. If the CONNECT metacommand fails, the following IF metacommand will identify that error despite the intervening "METACOMMAND_ERROR_HALT On" command.

ON CANCEL_HALT EMAIL

```
ON CANCEL_HALT EMAIL FROM <from_address> TO <to_addresses>
      SUBJECT "<subject>" MESSAGE "<message_text>"
      [MESSAGE_FILE "<filename>"]
      [ATTACH_FILE "<attachment_filename>"]
```

```
ON CANCEL_HALT EMAIL CLEAR
```

Sends the specified email only if the user cancels the script at a prompt. This command operates similarly to the *EMAIL* metacommand, except for its deferred operation.

The form of the metacommand with the "CLEAR" keyword will eliminate any email specification that was previously established.

The email specification is scanned for *substitution variables* at two different times: first, when the ON CANCEL_HALT EMAIL metacommand is invoked, and second, when the specified email is going to be sent. Substitution variables to be replaced when the email is sent must not be defined at the time that the metacommand is invoked. See the *ON ERROR_HALT WRITE* metacommand for an example of the use of deferred substitution of metacommands.

This metacommand sends email after any action triggered by an *ON CANCEL_HALT WRITE* metacommand has completed. This allows any output file created by the *ON CANCEL_HALT WRITE* metacommand to be included in the email message or as an attachment.

If an error occurs during the sending of email (for example, if no SMTP port is defined in a *configuration file*), then the email will not be sent and no error message describing this failure will be issued. An error message describing the error

that triggered the sending of email will be issued, as it would be if the ON CANCEL_HALT EMAIL metacommand had not been used. The execsql *log file* will contain a message describing the failure of the ON CANCEL_HALT EMAIL metacommand.

ON CANCEL_HALT WRITE

```
ON CANCEL_HALT WRITE "<text>" [[TEE] TO <output>]
```

```
ON CANCEL_HALT WRITE CLEAR
```

Writes the specified text only if the user cancels the script in response to a prompt. The given text is written immediately before the standard error message is displayed. This command operates similarly to the *WRITE* metacommand, except for its deferred operation.

The text to be written may be enclosed in double quotes (as shown above), in single quotes, or in matching square brackets.

The form of the metacommand with the "CLEAR" keyword will eliminate any message that was previously established.

The text to be written is scanned for *substitution variables* at two different times: first, when the ON CANCEL_HALT WRITE metacommand is invoked, and second, when the specified text is actually written. Substitution variables to be replaced when the text is written must not be defined at the time that the metacommand is invoked.

If an error occurs when the text is written (for example if an attempt is made to write to a read-only file), then the text will not be written and no error message describing this failure will be issued. An error message describing the error that triggered the CANCEL_HALT WRITE action will be issued, as it would be if the ON CANCEL_HALT WRITE metacommand had not been used. The execsql *log file* will contain a message describing the failure of the ON CANCEL_HALT WRITE metacommand. The execsql log file will contain a message describing the failure of the ON CANCEL_HALT WRITE metacommand.

ON ERROR_HALT EMAIL

```
ON ERROR_HALT EMAIL FROM <from_address> TO <to_addresses>
      SUBJECT "<subject>" MESSAGE "<message_text>"
      [MESSAGE_FILE "<filename>"]
      [ATTACH_FILE "<attachment_filename>"]
```

```
ON ERROR_HALT EMAIL CLEAR
```

Sends the specified email only if an error occurs in the definition or processing of a SQL statement or metacommand. This command operates similarly to the *EMAIL* metacommand, except for its deferred operation.

The form of the metacommand with the "CLEAR" keyword will eliminate any email specification that was previously established.

The email specification is scanned for *substitution variables* at two different times: first, when the ON ERROR_HALT EMAIL metacommand is invoked, and second, when the specified email is going to be sent. Substitution variables to be replaced when the email is sent must not be defined at the time that the metacommand is invoked. See the *ON ERROR_HALT WRITE* metacommand for an example of the use of deferred substitution of metacommands.

This metacommand sends email after any action triggered by an *ON ERROR_HALT WRITE* metacommand has completed. This allows any output file created by the *ON ERROR_HALT WRITE* metacommand to be included in the email message or as an attachment.

If an error occurs during the sending of email (for example, if no SMTP port is defined in a *configuration file*), then the email will not be sent and no error message describing this failure will be issued. An error message describing the error that triggered the sending of email will be issued, as it would be if the ON ERROR_HALT EMAIL metacommand had not been used. The execsql *log file* will contain a message describing the failure of the ON ERROR_HALT EMAIL metacommand.

ON ERROR_HALT WRITE

```
ON ERROR_HALT WRITE "<text>" [[TEE] TO <output>]
```

```
ON ERROR_HALT WRITE CLEAR
```

Writes the specified text only if an error occurs in the definition or processing of a SQL statement or metacommand. The given text is written immediately before the standard error message is displayed. This command operates similarly to the *WRITE* metacommand, except for its deferred operation.

The text to be written may be enclosed in double quotes (as shown above), in single quotes, or in matching square brackets.

The form of the metacommand with the "CLEAR" keyword will eliminate any message that was previously established.

The text to be written is scanned for *substitution variables* at two different times: first, when the ON ERROR_HALT WRITE metacommand is invoked, and second, when the specified text is actually written. Substitution variables to be replaced when the text is written must not be defined at the time that the metacommand is invoked. For example, this script:

```
create temporary view ok as
select long_text from some_data where row_number < 4;

-- !x! sub errmsg **** An error occurred:
-- !x! sub_append errmsg Last SQL (by simple substitution): !!$last_sql!!
-- !x! sub_append errmsg Last SQL (deferred): !!!!deferred_msg!!!!
-- !x! on error_halt write "!!errmsg!!"
-- !x! sub deferred_msg $last_sql

create temporary view not_ok as
select long_text from some_data where no_such_function(row_id);

-- !x! export not_ok to stdout as txt
```

will produce the following output when the *EXPORT* metacommand is run:

```
**** An error occurred:
Last SQL (by simple substitution): create temporary view ok as
select long_text from some_data where  row_number < 4;
Last SQL (deferred): create temporary view not_ok as
select long_text from some_data where no_such_function(row_id);
**** Error in metacommand.
     Line 69 of script on_error_halt_write.sql
     OperationalError: no such function: no_such_function in ../../execsql/execsql.py␣
↪on line 2434 of execsql.
     export not_ok to stdout as txt
     Metacommand: export not_ok to stdout as txt
     Error occurred at 2017-10-07 17:56:24 UTC.
```

The "deferred_msg" substitution variable displays the SQL command that actually caused the error because that variable was not defined at the time that the ON ERROR_HALT WRITE metacommand was run.

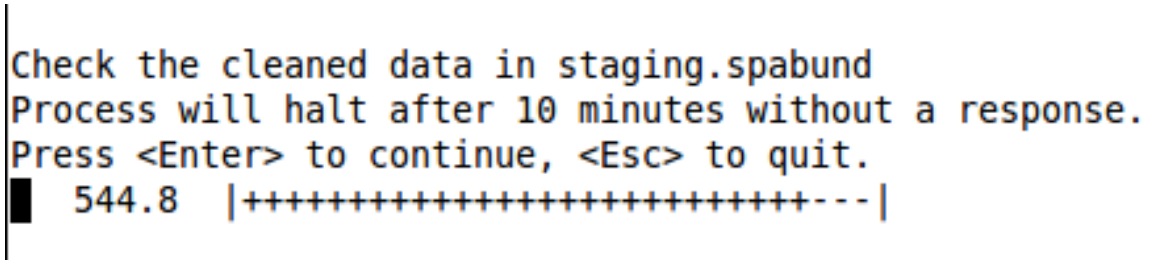
If an error occurs when the text is written (for example if an attempt is made to write to a read-only file), then the text will not be written and no error message describing this failure will be issued. An error message describing the error that triggered the ERROR_HALT WRITE action will be issued, as it would be if the ON ERROR_HALT WRITE metacommand had not been used. The execsql *log file* will contain a message describing the failure of the ON ERROR_HALT WRITE metacommand. The execsql log file will contain a message describing the failure of the ON ERROR_HALT WRITE metacommand.

PAUSE

```
PAUSE "<text>" [HALT|CONTINUE AFTER <n> MINUTES|SECONDS]
```

Displays the specified text and pauses script processing. You can continue script processing with the <Enter> key, or halt script processing with the <Esc> key. The message will be displayed on the console by default; if the "-v" command-line option is used, the message will be displayed in a GUI dialog.

If the "HALT|CONTINUE. . . " clause is used, the PAUSE prompt will disappear after the specified time, regardless of whether the <Enter> or <Esc> keys were struck. If the PAUSE prompt times out in this way, script processing will be either halted or continued, as specified. The prompt with a timeout limit will look like this on the console:



The countdown of time remaining is always displayed in seconds.

If the "-v1", "-v2", or "-v3" command-line option is used, the prompt will appear in a GUI dialog instead of on the console.

If the "HALT" action is taken, either as a result of user input or as a result of a timeout, the effect on the script depends on the CANCEL_HALT setting. If script processing is halted, the system exit value will be set to 2.

PG_VACUUM

```
PG_VACUUM <vacuum arguments>
```

Runs the 'vacuum' command on the current database if the current DBMS is Postgres. The 'vacuum' command will not execute successfully as a SQL command because it requires a change in the configuration of the (psycopg2) connection. This metacommand makes that change, runs the 'vacuum' metacommand, and restores the connection configuration to its default setting.

This metacommand has no effect if the current DBMS is not PostgreSQL.

Note: The PG_VACUUM metacommand commits all pending transactions when it runs.

PROMPT ASK

```
PROMPT ASK "<question>" SUB <match_string> [DISPLAY <table_or_view>]
```

Prompts for a yes or no response to the specified question, using a dialog box, and assigns the result, as either "Yes" or "No", to the substitution variable specified. A data table or view can optionally be displayed with the question (as shown for the *PROMPT DISPLAY* metacommand). The "Y" and "N" keys will select the corresponding response, and the <Enter> key will also select the "Yes" response. The <Esc> key will cancel the script. The selection is also logged. If the prompt is canceled, script processing is halted, and the system exit value is set to 2.

See the *ASK* metacommand for a version of this command that presents the prompt on the console.

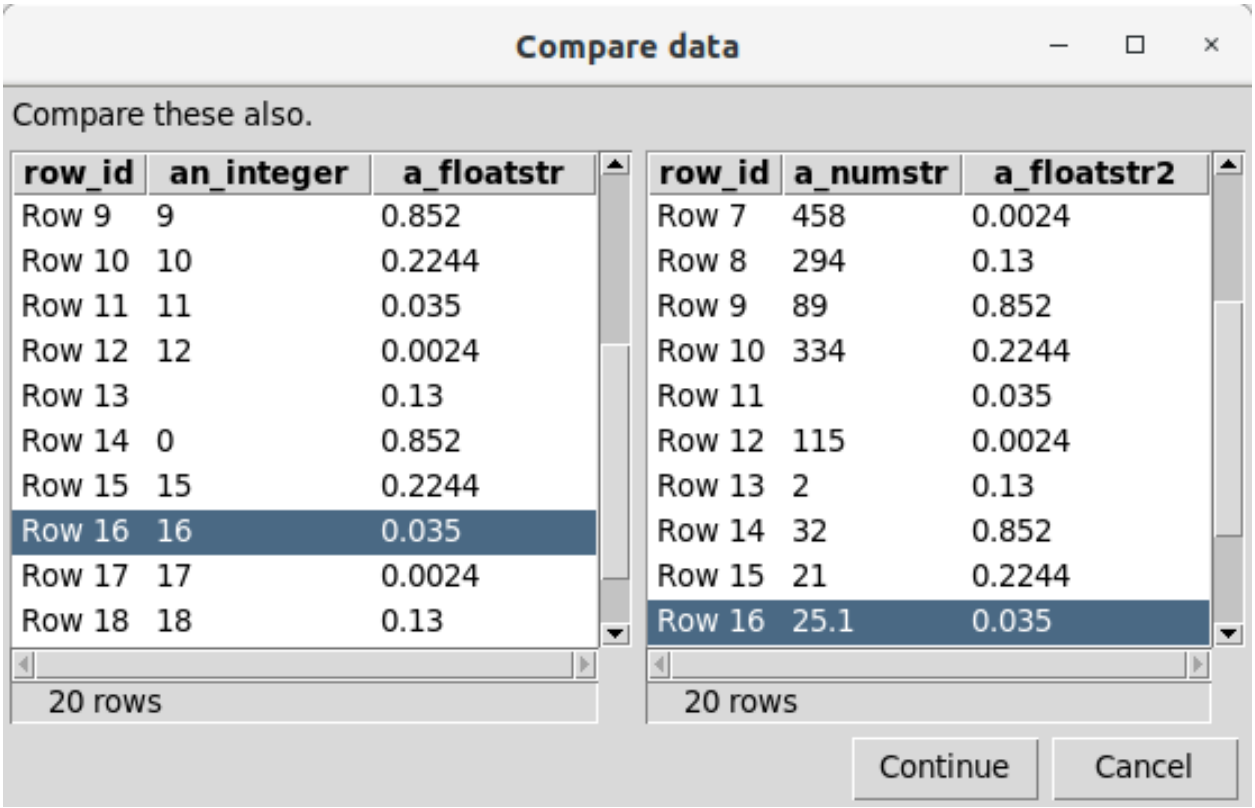
PROMPT COMPARE

```
PROMPT COMPARE <table1> AND|BESIDE <table2> PK(<col1>[, col2[, col3...]]) [MESSAGE "
↪<text>"]
```

Displays the two specified tables in a graphical interface. The two tables must have at least one column name in common. Clicking on a row in one of the tables will highlight that row and the *first* matching row in the other table. The names of all columns that are to be used to match rows must be specified within the parentheses of the `PK()` phrase. When more than one column name is listed, each additional colum name must be preceded by a comma.

When the `AND` keyword is used, the second table is displayed below the first table. When the `BESIDE` keyword is used, the second table is displayed to the right of the first table.

The display looks like this with the `BESIDE` orientation:



If the 'Continue' button is selected, the script will continue to run. If the 'Cancel' button is selected, the script will immediately halt. The Enter key also carries out the action of the 'Continue' button, and the Escape key carries out

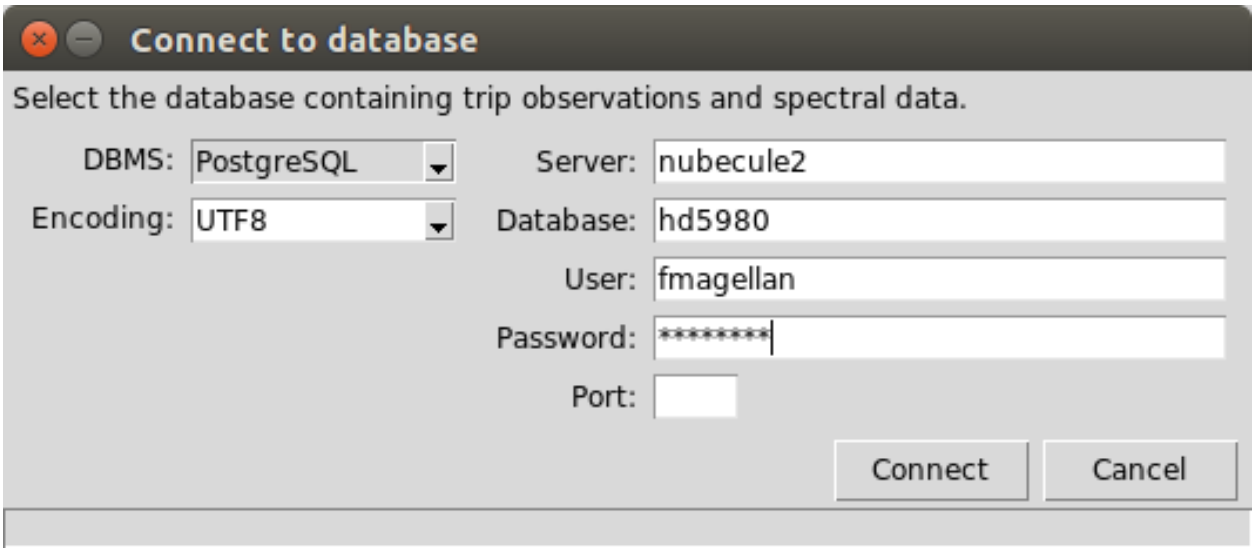
the action of the 'Cancel' button.

PROMPT CONNECT

```
PROMPT [MESSAGE "<text>"] CONNECT AS <alias>
```

Prompts for database connection parameters in a dialog box, and assigns that connection to the specified database alias. Any database connection previously associated with this alias will be closed, even if the prompt is canceled.

The connection dialog looks like this:



The prompt provides several common options for the database encoding. If the database uses a different encoding, you can type in the name of that encoding.

If the port is not specified, the default port for the selected DBMS will be used.

If a password is not provided, a connection will be attempted without using any password; there will be no additional prompt for a password.

If a file-based DBMS (MS-Access or SQLite) is selected, the prompt for the server and other information will be replaced by a prompt for a file name.

PROMPT DIRECTORY

```
PROMPT DIRECTORY SUB <match_string>
```

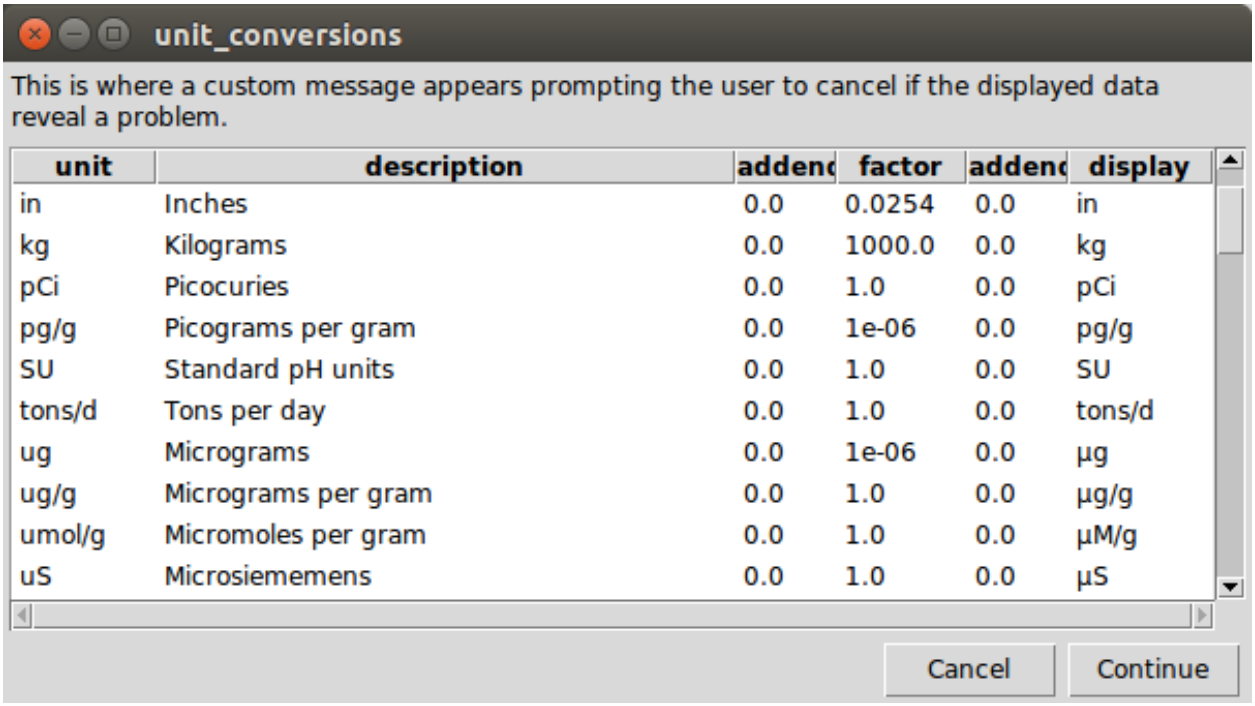
Prompts for the name of an existing directory, using a dialog box, and assigns the selected directory name (including the full path) to the substitution variable specified. The selection is also logged. If the prompt is canceled, unless *CANCEL_HALT* is set to OFF, script processing is halted, and the system exit value is set to 2. If *CANCEL_HALT* is set to ON, the specified substitution variable will be undefined.

PROMPT DISPLAY

```
PROMPT MESSAGE "<text>" DISPLAY <table_or_view_name>
```

Displays the selected view or table in a window with the specified message and both 'Continue' and 'Cancel' buttons. If the 'Continue' button is selected, the script will continue to run. If the 'Cancel' button is selected, the script will immediately halt. The Enter key also carries out the action of the 'Continue' button, and the Escape key carries out the action of the 'Cancel' button.

The prompt display looks like this:



If any columns of the table contain binary data, a message identifying the size, in bytes, of the data will be displayed instead of the data itsef.

PROMPT ENTER_SUB

```
PROMPT ENTER_SUB <match_string> [PASSWORD] MESSAGE "<text>"
    [DISPLAY <table_or_view>]
    [TYPE INT|FLOAT|BOOL|IDENT]
    [LCASE|UCASE]
```

Prompts for a replacement string to be assigned to the specified substitution variable (matching string). Entry of a value is not required; the 'OK' button (or 'Enter' key) can be used to continue even when nothing has been entered, and if this is done, an empty string will be returned.

If the "PASSWORD" keyword is included, the characters that are typed in response to the prompt will be displayed as asterisks.

The "TYPE" keyword can be used to limit the type of entry provided. This keyword takes any of the following arguments, and constrains the entry as described:

- INT: Only digits may be entered, optionally preceded by a minus sign.
- FLOAT: Only digits and a single decimal point may be entered, optionally preceded by a minus sign.
- BOOL: Only the values "True" or "False", or any prefix of those words, may be entered. Characters may be entered in upper- or lower-case. If only a prefix of the words is entered, the return value will be forced to an entire word, either "True" or "False" as appropriate.

- IDENT: Only letters, digits, and the underscore character may be entered, and the first character must be a letter.

The keywords "LCASE" and "UCASE" force the returned value to be all lowercase or all uppercase, respectively.

PROMPT ENTRY_FORM

```
PROMPT ENTRY_FORM <specification_table> MESSAGE "<text>"
    [DISPLAY <table_or_view>]
```

Dynamically creates a data entry form following the specifications in the specification_table and assigns the entered values to the substitution variables named in the specification table.

The data entry form will have one data entry prompt for every row in the specification table. The following columns in the specification table will be used to construct the data entry form:

sub_var The name of the substitution variable to which the entered value will be assigned. This column is required, and must contain non-null text.

prompt The text to display on the form as a prompt to the user to indicate what information should be entered. This column is required, and must contain non-null text.

required An indicator of whether a non-null value must be provided. This column is optional. If present, it should have a Boolean data type. If the column is missing or the contents are null, the value will not be required.

initial_value The initial, or current, value. It will be displayed on the form and may be replaced. This column is optional, and if present, its contents may be null.

width An integer specifying the width of the entry area for this value, in characters. This column is optional, and if present, its contents may be null.

entry_type Text specifying the type of entry control to use on the form. This may take the values "checkbox" or "textarea". The former will present a checkbox on the form, and the latter will present a multi-line text entry area. If this column has any other value, or is null, or is missing, either a text entry control will be used, or a dropdown control will be used if a lookup table is specified. If "checkbox" is specified, the values returned in the substitution variable will always be either "0", indicating that the checkbox was cleared, or "1", indicating that the checkbox was checked.

lookup_table The name of a table or view containing, in its first column, a set of valid values for this entry. This column is optional, and if present, its contents may be null. If present, the entry will be constrained to only members of the given list.

validation_regex A regular expression pattern to be used to validate the entry. This validation check will be applied when the entry is about to lose focus; if the entered value does not match the regular expression, the entry will retain focus until it is corrected. This column is optional, and if present, its contents may be null.

validation_key_regex A regular expression pattern to be used to validate each keystroke for the entry. This validation check will be applied for each keystroke while the entry has the focus. The entire value, with the additional keystroke applied, must match the regular expression. If it does not match, the keystroke will not change the entry. This column is optional, and if present, its contents may be null.

sequence A value used to specify the order in which values should appear on the form. This column is optional; if absent, the order of values on the form is indeterminate.

The order of the columns in the specification table does not matter. The specification table may contain additional columns other than those listed above; if it does, those columns will be ignored.

After data entry is complete and the data entry form is closed with the "Continue" button that appears on the form, the designated substitution variables will be defined to have the corresponding values that were entered. Substitution variables will not be defined for values that were not entered (were left empty on the form) even if they had been

defined previously—except for checkboxes, for which the substitution variable is always defined and assigned a value of "0" or "1".

Although the PROMPT ENTRY_FORM metacommand supports validation of individual entries through the use of either a list of valid values or a regular expression, it does not support cross-column validation or foreign key checks (except for single valid values). The primary purpose of execsql is to facilitate scripting, and therefore documentation, of data modifications, and interactive data entry runs counter to that purpose. There are nevertheless circumstances in which a data entry form is an appropriate tool to collect user input. Use of a simple custom data entry form is illustrated in *Example 18* and *Example 23*.

PROMPT OPENFILE

```
PROMPT OPENFILE SUB <match_string>
```

Prompts for the name of an existing file (implicitly, to be opened), using a dialog box, and assigns the selected filename (including the full path) to the substitution variable specified. The selection is also logged. If the prompt is canceled, unless *CANCEL_HALT* is set to OFF, script processing is halted, and the system exit value is set to 2. If CANCEL_HALT is set to ON, the specified substitution variable will be undefined.

PROMPT SAVEFILE

```
PROMPT SAVEFILE SUB <match_string>
```

Prompts for the name of a new or existing file, using a dialog box, and assigns the selected filename (including the full path) to the substitution variable specified. The selection is also logged. If the prompt is canceled, unless *CANCEL_HALT* is set to OFF, script processing is halted, and the system exit value is set to 2. If CANCEL_HALT is set to ON, the specified substitution variable will be undefined.

PROMPT SELECT_SUB

```
PROMPT SELECT_SUB <table_or_view> MESSAGE "<prompt_text>" [CONTINUE]
```

Displays the selected data table or view, similar to the PROMPT DISPLAY metacommand, but allows you to select a single row of data, and then assigns the data values from that row to a set of substitution variables corresponding to the column names, but prefixed with the "@" character. This prefix prevents any conflict between these automatically-assigned substitution variables and any others that you may have created with the SUB command or by any other means—except for the SELECT_SUB metacommand, which uses the same prefix for substitution variables.

Data can be selected from the display either by highlighting the row with a single mouse click and then clicking on the "OK" button, or by double-clicking on a row. If data selection is canceled either with the "Cancel" button or by hitting the Escape key, script processing will be halted and the system exit value will be set to 2, unless *CANCEL_HALT* has been set to OFF.

Null values in the selected data row will be represented by substitution variables with zero-length string values.

If the CONTINUE keyword is used, then a "Continue" button will also be displayed in the dialog box. This option allows the user to close the dialog without either selecting an item or canceling the script.

If no data value is selected (i.e., either the "Continue" button has been used, or the "Cancel" button has been used and CANCEL_HALT has been set to OFF), all data values corresponding to column names of the table that was displayed will be undefined, even if they were defined before the table was displayed.

See *Example 8* for an illustration of the use of this metacommand, and *Example 17* for an illustration of the use of the CONTINUE keyword.

RESET COUNTER

```
RESET COUNTER <counter_no>
```

Resets the specified counter variable so that the next reference to it will return a value of 1.

RESET COUNTERS

```
RESET COUNTERS
```

Resets all counter variables so that the next reference to any of them will return a value of 1.

RM_FILE

```
RM_FILE <file_name>
```

Deletes the specified file. Although execsql is not intended to be a file management tool, there are occasions when deletion of a file from within the script may be a useful workflow step—for example, if header information was written to an output file in anticipation of subsequent addition of error messages, but no errors were later encountered. Whereas the *EXPORT* metacommand will automatically overwrite an existing file if it exists, the *WRITE* metacommand always appends text to an existing file. The RM_FILE metacommand is therefore useful to remove an existing output text file that you wish to rewrite. The RM_FILE metacommand is also useful when you want to create a new SQLite database (using the NEW keyword) and want to ensure that the SQLite file does not already exist, to avoid the error that the *CONNECT* metacommand would otherwise raise.

If the file that is to be deleted does not actually exist, no error will occur.

RM_SUB

```
RM_SUB <match_string>
```

Deletes the specified user-created subsitution variable.

SELECT_SUB

```
SELECT_SUB <table_or_view>
```

Assigns data values from the first row of the specified table or view to a set of substitution variables corresponding to the column names, but prefixed with the "@" character. This prefix prevents any conflict between these automatically-assigned substitution variables and any others that you may have created with the *SUB* command or by any other means—except for the *PROMPT SELECT_SUB* metacommand, which uses the same prefix for substitution variables.

Null values in the selected data row will be represented by substitution variables with zero-length string values.

If the selected table or view contains no data, an error will occur, and script processing will be halted.

SET COUNTER

```
SET COUNTER <counter_no> TO <value>
```

Assigns the specified value to the counter. The next time that this counter is referenced, the value returned will be one larger than the value to which it is set by this metacommand.

SUB

```
SUB <match_string> <replacement_string>
```

Defines a *substitution variable* (the <match_string>) which, if matched on any line of the script, will be replaced by the specified replacement string. Replacement will occur on all following lines of the script (and all included scripts) before the lines are evaluated in any other way. Every occurrence of the <match_string>, when immediately preceded and followed by two exclamation points ("!!"), will be replaced by the replacement string. Substitutions are processed in the order in which they are defined.

SUB_ADD

```
SUB_ADD <match_string> <numeric_value>
```

Adds the specified numeric value to the value of the substitution variable, which should also be numeric.

Although SQL can be used to perform computations with numeric substitution variables (see *Example 16*), incrementing a variable is a commonly useful operation, and this metacommand allows it to be carried out without a round trip to the database.

If the substitution variable is not numeric, it will be redefined with a suffix of "+" and the numeric value.

SUB_APPEND

```
SUB_APPEND <match_string> <new_line>
```

Appends the given *new_line* text to the specified *substitution variable* (the *match_string*), separated with a newline character. This metacommand allows the creation of multi-line messages that can be used with the *WRITE*, *PROMPT*, and *EMAIL* metacommands.

This metacommand may change the order in which substitution variables are defined. The substitution variable specified in this metacommand will become the most recently defined substitution variable.

SUB_DECRYPT

```
SUB_DECRYPT <sub_var_name> <encrypted_text>
```

Creates a substitution variable containing an unencrypted version of the given encrypted_text. The encrypted_text must have been produced by the *SUB_ENCRYPT* metacommand. The encryption method used is not of cryptographic quality, and is primarily intended to provide simple obfuscation of email passwords or other sensitive information that may appear in configuration or script files.

SUB_EMPTY

```
SUB_EMPTY <match_string>
```

Defines a *substitution variable* containing an empty string.

SUB_ENCRYPT

```
SUB_ENCRYPT <sub_var_name> <plaintext>
```

Creates a substitution variable containing an encrypted version of the given plaintext. The encryption method used is not of cryptographic quality, and is primarily intended to provide simple obfuscation of email passwords or other sensitive information that may appear in configuration or script files.

SUB_TEMPFILE

```
SUB_TEMPFILE <match_string>
```

Assigns a unique temporary file name to the specified substitution variable (the match_string). The location of (path to) this temporary file is operating-system dependent; the file may not be located in the current working directory. The temporary file will not be created, opened, or used directly by execsql. All temporary files will automatically be deleted when execsql exits (however, a temporary file will not be deleted if it is in use by another process, and then may persist until manually removed). See *Example 12* and *Example 13* for llustrations of the use of temporary files.

SUBDATA

```
SUBDATA <match_string> <table_or_view_name>
```

Defines a *substitution variable* which, if matched on any line of the script, will be replaced by the data value in the first column of the first row of the specified table or view.

If there are no rows in the specified data source, the substitution variable will be undefined. This case can be evaluated with the *SUB_DEFINED* conditional.

Warning: A backward-incompatible change to SUBDATA was made in version 1.24.8.0 (2018-06-03). Previously, if there were no rows in the data source, execsql would halt with an error message.

SYSTEM_CMD

```
SYSTEM_CMD ( <operating system command line> )
```

The specified command line will be passed to the operating system to execute. This command is executed by the system, not by the command shell, so commands that are processed by the shell cannot be used. Internal commands for the Bash shell are listed here: https://www.gnu.org/software/bash/manual/html_node/Bash-Builtins.html (some Bash internal commands also have analogues as external programs), and internal commands for the Windows command interpreter are listed here: https://ss64.com/nt/syntax-internal.html.

Because commands are not processed by the shell, the system path is not searched for executable commands, so full path names must be used for executable files. Execution of the SQL script does not ordinarily continue until the operating system command has completed–however, note that if this command invokes an editor or other software that is already open, the execsql script will continue immediately.

On non-POSIX operating systems (specifically, Windows), any backslashes in the command line will be doubled before the command line is passed to the operating system. Because backslashes are used as directory separators in Windows paths, this automatic alteration of the command line is meant to eliminate the need to double backslashes in path specifications on Windows.

The exit status of the command that is invoked will be stored in the *system variable* $SYSTEM_CMD_EXIT_STATUS.

TIMER

```
TIMER ON|OFF
```

Starts or stops an internal timer. The value of the timer can be obtained with the "$timer" *system variable*. Elapsed time is reported in real-time seconds (not CPU time) to at least the nearest millisecond.

USE

```
USE <alias_name>
```

Causes all subsequent SQL statements and metacommands to be applied to the database identified by the given alias name. The alias name must have been previously established by the *CONNECT* metacommand, or the alias name "initial" can be used to refer to the database that is used when execsql starts script processing.

WAIT_UNTIL

```
WAIT_UNTIL <Boolean expression> HALT|CONTINUE AFTER <n> SECONDS
```

Suspends execution of the SQL script until the specified Boolean expression becomes true. The Boolean expressions that can be used with the WAIT_UNTIL metacommand are the same as those that can be used with the *IF* metacommands.

The condition is tested once per second for up to <n> seconds. If the condition has not become true by that time, then the script either halts or continues, as specified.

The WAIT_UNTIL metacommand can be used to insert a pause in a script without issuing a message, as the *PAUSE* metacommand does:

```
WAIT UNTIL EQUALS("1","0") CONTINUE AFTER 1 SECONDS
```

WRITE

```
WRITE "<text>" [[TEE] TO <output>]
```

Writes the specified text to the console or a file, or both. The text to be written must be enclosed in double quotes. If no output filename is specified, the text will be written to the terminal. If the "TEE" keyword is included, the text will be written to both the console and the specified file. If the "-v3" command-line option is used, or a GUI console is opened explicitly, the text will be written to the GUI console. If the text is written to a file, it will always be appended to any existing file of the given name. The output file directory will be created if it does not exist and the `make_export_dirs` *configuration setting* is set to "Yes".

The text to be written may be enclosed in double quotes (as shown above), in single quotes, or in matching square brackets.

WRITE CREATE_TABLE

For data in a delimited text file:

```
WRITE CREATE_TABLE <table_name> FROM <file_name>
    [WITH QUOTE <quote_char> DELIMITER <delim_char>]
    [SKIP <lines>]
    [COMMENT "<comment_text>"] [TO <output>]
```

For data in an OpenDocument spreadsheet:

```
WRITE CREATE_TABLE <table_name> FROM <file_name>
    SHEET <sheet_name> [SKIP <rows>] [COMMENT "<comment_text>"]
    [TO <output>]
```

For data in an Excel spreadsheet:

```
WRITE CREATE_TABLE <table_name> FROM EXCEL <file_name>
    SHEET <sheet_name> [COMMENT "<comment_text>"] [TO <output>]
```

For data in a table of an aliased database:

```
WRITE CREATE_TABLE <table_name> FROM <table_name>
    IN <alias> [COMMENT "<comment_text>"] [TO <output>]
```

Generates the CREATE TABLE statement that would be executed prior to *importing* data from the specified file or worksheet, or copying data from the specified aliased database, if the NEW or REPLACEMENT keyword were used with the *IMPORT* or *COPY* metacommand. The comment text, if provided, will be written as a SQL comment preceding the CREATE TABLE statement. The comment text must be double-quoted; table, file, and worksheet names can be quoted or unquoted. If no output filename is specified, the text will be written to the console. Text will always be appended to any existing file of the given name. The output file directory will be created if it does not exist and the `make_export_dirs` *configuration setting* is set to "Yes". See *Example 12* for an illustration of the use of this metacommand.

The SKIP key phrase specifies the number of lines at the beginning of the file to discard before evaluating the remainder of the file as a data table.

The WRITE CREATE_TABLE command may report an error when used with ODS files that have been created or edited using Excel—see the description of the *IMPORT* metacommand for additional information about this problem.

WRITE SCRIPT

```
WRITE SCRIPT <script_name> [[APPEND] TO <output_file>]
```

Displays the text of the specified script, which must have been defined with the *BEGIN SCRIPT* metacommand. The lines of the specified script will be written either to the console or to the specified file. The output file directory will be created if it does not exist and the `make_export_dirs` *configuration setting* is set to "Yes".

2.4.8 Logging

execsql.py automatically logs certain actions, conditions, and errors that occur during the processing of a script file. Although a script file provides good documentation of database operations, there are circumstances in which a script file is not a definitive record of what operations were carried out. These circumstances include:

- Errors
- Choices made by the user in response to a *PROMPT* metacommand.
- Cancellation of the script in response to a *PAUSE* metacommand or password prompt from the *CONNECT* metacommand.

Information is logged into a tab-delimited text file named `execsql.log` in the directory from which the script file was run.

Note: Prior to version 1.28.0.5 (2018-09-10), the log file was created in the directory of the starting script.

This file contains several different record types. The first value on each line of the file identifies the record type. The second value on each line is a run identifier. All records that are logged during a single run of execsql.py have the same run identifier. The run identifier is a compact representation of the date and time at which the run started. The record types and the values that each record of that type contains are:

run—Information about the run as a whole:

- Record type
- Run identifier
- Script name
- Script path
- Script file revision date
- Script file size in bytes
- User name
- Command-line options

run_db_file—Information about the file-based database used (Access or SQLite):

- Record type
- Run identifier
- Database file name with full path

run_db_server—Information about the server-based database used (Postgres or SQL Server):

- Record type
- Run identifier
- Server name
- Database name

connect—The type and name of a database to which a connection has been established; this may be either a client-server or file-based database:

- Record type
- Run identifier
- DBMS type and database identifiers

action—Significant actions carried out by the script, primarily those that affect the results.

- Record type
- Run identifier
- Sequence number—The order of actions, status messages, and errors. Automatically generated.
- Action type—One of the following values:
  - export—Execution of an *EXPORT* metacommand.

    - prompt_quit—The user's choice resulting from a *PROMPT DISPLAY* metacommand.
  - Line number—The script line number where the action takes place.
  - Description—Free text describing the action.

status—Status messages; ordinarily these are errors

  - Record type
  - Run identifier
  - Sequence number—The order of actions, status messages, and errors. Automatically generated.
  - Status type—One of the following values:
    - exception
    - error
  - Description—Free text describing the status.

exit—Program status at exit.

  - Record type
  - Run identifier
  - Exit type—One of the following values:
    - end_of_script—A normal exit; the entire script has been processed.
    - prompt_quit—The user chose to cancel the script in response to a PROMPT metacommand.
    - halt—A *HALT* metacommand was executed.
    - error—An error occurred.
    - exception—An exception occurred.
  - Line number—The script line number from which the exit was triggered (may be null).
  - Description—Free text describing the exit condition.

The messages for each run are appended to the end of the log file.

Although logging is performed automatically by execsql, there are three ways to make use of the log file in custom scripts:

- The *LOG* metacommand provides a way to write a custom message into the log file.
- The *LOG_WRITE_MESSAGES* metacommand causes the output of all *WRITE* metacommands to be echoed to the log file.
- The $RUN_ID *system variable* provides a way to link other information (e.g., status or error messages) to the run that is identified in the log file.

2.4.9 Character Encoding

Command-line options allow specification of the encoding used in the database, the encoding used to read the script file and imported data files, and the encoding used to write output text. The encoding of data files to be imported can also be specified with the *IMPORT* metacommand. Database encoding can also be specified with the *CONNECT* metacommand. Specification of appropriate encoding will eliminate errors that would otherwise result from the presence of non-ASCII characters.

For Postgres and SQLite, the database encoding used is determined by interrogating the database itself, and any database encoding specified on the command line is ignored.

If no encodings are specified on the command line, the following default encodings are used:

- Script file: utf8
- Firebird: latin1
- MySQL and MariaDB: latin1
- SQL Server: latin1
- Access: windows_1252
- DSN: None
- Output: utf8
- Import: utf8

If a UTF byte order mark (BOM) is found at the start of the script file or at the start of a data file to be *IMPORTed*, the encoding indicated by the BOM will be taken as definitive regardless of any configuration options that may be used.

There is no default encoding for a DSN connection because the actual data source used is unknown, and because some ODBC drivers may return results in Unicode. If no encoding is specified, the ODBC driver must return result in Unicode or some compatible format (e.g., ASCII).

Some encodings are known by multiple names (aliases). In cases where the performance of the *IMPORT* metacommand is dependent on the compatibility of encoding (specifically, Postgres), execsql will try to match the input file and database encodings using the matching rules of Unicode Technical Starndard #22 and the equivalences documented by WHATWG.

The "-y" command-line option will display all of the encoding names that execsql recognizes. There are some aliases for the displayed encoding names that can also be used, if you know them. The encoding names used by each DBMS may differ from this list.

The log file is always written in UTF-8.

2.4.10 Using Script Files

Using script files to store and execute task-specific SQL statements has a number of advantages over using views, functions, or procedures that are stored within the database itself, particularly for one-off or infrequent tasks, or actions that must be applied to multiple databases. These advantages are:

- When database operations are only part of an overall task, maintenance and management of all components of the task is easier and more reliable when SQL scripts are kept together in the file system with input files, database output, output processing scripts, and final task products. Because all of the SQL needed for a specific data summarization task is kept together, there is little or no risk that one of a set of separate database objects—views or stored procedures—that are needed to complete a specific task will be either deleted or altered. The clutter of queries, functions, and procedures that would otherwise accumulate in a heavily used database can be reduced or eliminated.
- When multiple databases with the same data model are used (e.g., for different projects), only one copy of the scripts tailored for that data model need be maintained, rather than having duplicate procedures or views in every database. This reduces maintenance and ensures consistency.
- Creation of the SQL script for a new task can be simplified by copying and editing a previously existing script. The user's preferred editor can be used to carry out search and replace operations to easily and reliably make changes throughout the entire set of SQL statements and scripts that are needed for a particular task.
- Complete documentation can (and should!) be included in the script files, so that the purpose, assumptions, limitations, and history of changes can be easily reviewed by anybody who might consider using or modifying the query script. This documentation is easily accessible to scanning and searching tools like grep.

- The script can be easily preserved to document the way in which data were selected or summarized. Scripts can be easily archived, backed up, and put under version control independently of the database. Script files can be made read-only so that they cannot easily or accidentally be modified after the script for a particular task has been finalized.
- Data management processes can be more easily automated by integrating a script-processing tool like execsql with other system tools than by using interactive database interfaces. The ability of execsql to export data in CSV, TSV, OpenDocument spreadsheet, readable Markdown-compatible text, HTML, JSON, and LaTeX formats reduces the amount of time that might otherwise be required to interactively open the database, run the appropriate query (not to mention verifying that the query, or any queries that it depends on, have not been altered), export the result, and reformat the result. If the query output will be further processed or used in another scriptable application (e.g., to produce graphics or statistics using R), execsql can be combined with other programs in a system script file to further automate the data summarization and analysis process.
- If a database must be maintained in two different formats (e.g., in PostgreSQL for ordinary use, but downloaded to SQLite for use when a network connection is not available), one script file can potentially be used to carry out exactly the same data selection and summarization operations on both formats of the database.
- The capabilities provided by some of execsql's metacommands surpass the features available in views or stored procedures in most DBMSs, and this additional functionality is only available when script files are used.

The capabilities provided by metacommands may, in some cases, allow a script designed for execsql to take the place of a custom database client program.

2.4.11 Documentation

One of the primary goals of execsql is to facilitate, and even encourage, comprehensive documentation of all actions taken upon a database. Two fundamental aspects of execsql that support this goal are:

- The use of *script files*, which require that SQL statements be saved in a file rather than executed interactively, and which also allow copious comments to be included; and
- Automatic logging of information about the database(s) used, the script file(s) run, and user choices in response to interactive prompts.

Other features of execsql that also support this goal are:

- The *LOG* metacommand, which writes a user-provided message to the standard log file;
- The *WRITE* metacommand, which makes it easy to issue progress and status messages to the terminal or to a file.
- The *LOG_WRITE_MESSAGES* metacommand, which automatically echoes all output of the *WRITE* metacommand to the standard log file;
- The TEE clause of the *WRITE* metacommand, which makes it easy to write progress and status messages to a custom documentation file in addition to the console;
- The $RUN_ID *system variable*, which can be written into a custom documentation file to establish a correspondence between the information in that file and the information in the standard log file;
- Other *system variables* such as $CURRENT_DATABASE, $DB_NAME, $CURRENT_DIR, $CURRENT_SCRIPT, $CURRENT_TIME, $LAST_ROWCOUNT, $LAST_SQL, and $USER, which provide useful contextual and status information that can be written into a custom documentation file;
- The TXT output format of the *EXPORT* metacommand, which displays (or writes to a file) a table or query in the format of a Markdown pipe table, which is an inherently readable format if included in a custom documentation file;
- The *CONSOLE SAVE* metacommand, which allows the entire contents of a GUI console window to be written to a custom documentation file; and

- The $DATE_TAG, $DATETIME_TAG, and $RUN_ID *system variables*, which can be used to construct file names for custom documentation files.

Using these features when writing script files allows easy generation of documentation that can be valuable for establishing exactly what, and how, changes were made to a database.

An example of such a use is the creation of a custom log file to document the actions of a script. A custom log file might be initialized as follows:

```
-- ***************************************************************************
--          Create a custom log file
-- Input substitution variables:
--     CUSTOM_LOG     : The name of the log file to be created.  Required.
--     SCRIPT_PURPOSE : A narrative description of the script's purpose.
--                      Optional.
-- ***************************************************************************
-- !x! rm_file !!CUSTOM_LOG!!
-- !x! write "=====================================================" to !!CUSTOM_LOG!!
-- !x! if(sub_defined(SCRIPT_PURPOSE))
    -- !x! write "!!SCRIPT_PURPOSE!!" to !!CUSTOM_LOG!!
    -- !x! write "-----------------------------------------------------" to !!CUSTOM_
↪LOG!!
    -- !x! write " " to !!CUSTOM_LOG!!
-- !x! endif
-- !x! write "Working dir: !!$CURRENT_DIR!!" to !!CUSTOM_LOG!!
-- !x! write "Script:      !!$CURRENT_SCRIPT!!" to !!CUSTOM_LOG!!
-- !x! write "Database:    !!$CURRENT_DATABASE!!" to !!CUSTOM_LOG!!
-- !x! write "User:        !!$DB_USER!!" to !!CUSTOM_LOG!!
-- !x! write "Run at:      !!$CURRENT_TIME!!" to !!CUSTOM_LOG!!
-- !x! write "Run ID:      !!$RUN_ID!!" to !!CUSTOM_LOG!!
-- !x! write " " to !!CUSTOM_LOG!!
```

Subsequently, throughout the script, WRITE metacommands can be used to append information to the custom log file.

As an alternative to writing documentation to a text file, documentation could be saved to a database that serves as an activity log. *Example 20* illustrates how this can be done for data issues, and a similar technique can be used to record ordinary progress and status information.

2.4.12 Debugging

Execsql includes several metacommands that will display elements of its internal environment, to asssist with script debugging.

```
DEBUG LOG SUBVARS
```

Writes all substitution variables to the log file.

```
DEBUG WRITE SUBVARS [[APPEND] TO <filename>]
```

Writes all substitution variables to the terminal or to the specified text file.

```
DEBUG LOG CONFIG
```

Writes configuration settings to the log file.

```
DEBUG WRITE CONFIG [[APPEND] TO <filename>]
```

Writes configuration settings to the console or to the specified text file.

```
DEBUG WRITE ODBC_DRIVERS [[APPEND] TO <filename>]
```

Writes the names of available ODBC drivers to the console or to the specified text file. ODBC drivers are used with SQL Server and MS-Access.

```
DEBUG WRITE <script_name> [[APPEND] TO <filename>]
```

This is an alias for the *WRITE SCRIPT* metacommand.

2.4.13 Examples

The following examples illustrate some of the features of execsql.

Example 1: Use Temporary Queries to Select and Summarize Data in Access

This example illustrates a script that makes use of several temporary queries to select and summarize data, and a final query that prepares the data for export or further use. The SQL in this example is specific to MS-Access.

```
-- ---------------------------------------------------------------------
-- Get result records that meet specific selection criteria.
-- ---------------------------------------------------------------------
create temporary view v_seldata as
select
  smp.sys_sample_code, rs.test_surrogate_key,
  rs.cas_rn, tst.lab_anl_method_name,
  iif(rs.detect_flag='N', rs.method_detection_limit, rs.result_value) as conc,
  rs.detect_flag='Y' as detected,
  rs.lab_qualifiers like '*J*' as estimated,
  iif(rs.detect_flag='N', rs.detection_limit_unit, rs.result_unit) as unit
from
  ((((dt_result as rs
  inner join dt_test as tst on tst.test_surrogate_key=rs.test_surrogate_key)
  inner join dt_sample as smp on smp.sys_sample_code=tst.sys_sample_code)
  inner join dt_field_sample as fs on fs.sys_sample_code=smp.sys_sample_code)
  inner join dt_location as loc on loc.sys_loc_code=fs.sys_loc_code)
  inner join rt_analyte as anal on anal.cas_rn=rs.cas_rn
where
  (loc.loc_name like 'SG*' or loc.loc_name like 'SC*')
  and smp.sample_type_code='N'
  and smp.sample_matrix_code='SE'
  and anal.analyte_type in ('ABN', 'PEST', 'PCB', 'LPAH', 'HPAH')
  and rs.reportable_result='Yes'
  and not (rs.result_value is null and rs.method_detection_limit is null);

-- ---------------------------------------------------------------------
-- Summarize by sample, taking nondetects at half the detection limit.
-- ---------------------------------------------------------------------
create temporary view v_samp as
select
  sys_sample_code, cas_rn, lab_anl_method_name,
  Avg(iif(detected, conc, conc/2.0)) as concentration,
  Max(iif(detected is null, 0, detected)) as detect,
  Min(iif(estimated is null, 0, estimated)) as estimate,
  unit
from
```

(continues on next page)

(continued from previous page)

```
  v_seldata
group by
  sys_sample_code, cas_rn, lab_anl_method_name, unit;

-- ------------------------------------------------------------------
-- Pull in sample location and date information, decode analyte,
-- and reconstruct qualifiers.
-- ------------------------------------------------------------------
select
  loc.loc_name, fs.sample_date, fs.start_depth, fs.end_depth, fs.depth_unit,
  smp.sample_name, anal.chemical_name, dat.lab_anl_method_name,
  iif(dat.detect, concentration, concentration/2.0) as conc,
  (iif(detect, "", "U") & iif(estimate, "J", "")) as qualifiers,
  unit
from
  (((v_samp as dat
  inner join dt_sample as smp on dat.sys_sample_code=smp.sys_sample_code)
  inner join dt_field_sample as fs on fs.sys_sample_code=smp.sys_sample_code)
  inner join dt_location as loc on loc.sys_loc_code=fs.sys_loc_code)
  inner join rt_analyte as anal on anal.cas_rn=dat.cas_rn;
```

During the execution of this script with Access, the temporary queries will be created in the database. When the script concludes, the temporary queries will be removed. Nothing except the data itself need be kept in the database to use a script like this one.

Example 2: Execute a Set of QA Queries and Capture the Results

This example illustrates a script that creates several temporary queries to check the codes that are used in a set of staging tables against the appropriate dictionary tables, and, if there are unrecognized codes, writes them out to a text file.

```
create temporary view qa_pty1 as
select distinct stage_party.party_type
from   stage_party
       left join e_partytype
         on stage_party.party_type=e_partytype.party_type
where  e_partytype.party_type is null;

-- !x! if(hasrows(qa_pty1))
-- !x!   write "Unrecognized party types:" to Staging_QA_!!$DATE_TAG!!.txt
-- !x!   export qa_pty1 append to Staging_QA_!!$DATE_TAG!!.txt as tab
-- !x! endif

create temporary view qa_prop1 as
select distinct stage_property.property_type
from   stage_property
       left join e_propertytype
         on stage_property.property_type=e_propertytype.property_type
where  e_propertytype.property_type is null;

-- !x! if(hasrows(qa_prop1))
-- !x!   write "Unrecognized property types:" to Staging_QA_!!$DATE_TAG!!.txt
-- !x!   export qa_prop1 append to Staging_QA_!!$DATE_TAG!!.txt as tab
-- !x! endif
```

(continues on next page)

(continued from previous page)

```
create temporary view qa_partyprop1 as
select distinct stage_partyprop.property_rel
from   stage_partyprop
       left join e_partyproprel
         on stage_partyprop.property_rel=e_partyproprel.property_rel
where  e_partyproprel.property_rel is null;

-- !x! if(hasrows(qa_partyprop1))
-- !x!   write "Unrecognized party-property relationship types:" to Staging_QA_!!
↪$DATE_TAG!!.txt
-- !x!   export qa_partyprop1 append to Staging_QA_!!$DATE_TAG!!.txt as tab
-- !x! endif
```

Example 3: Execute a Set of QA Queries and Display the Results with a Prompt

This example illustrates a script that compiles the results of several QA queries into a single temporary table, then displays the temporary table if it has any rows (i.e., any errors were found), and prompts the user to cancel or continue the script.

```
create temporary table qa_results (
  table_name varchar(64) not null,
  column_name varchar(64) not null,
  severity varchar(20) not null,
  data_value varchar(255) not null,
  description varchar(255) not null,
  time_check_run datetime not null
  );

insert into qa_results
  (table_name, column_name, severity, data_value, description, time_check_run)
select distinct
       'stage_party',
       'party_type',
       'Fatal',
       stage_party.party_type,
       'Unrecognized party type',
       cast('!!$CURRENT_TIME!!' as datetime)
from   stage_party
       left join e_partytype
         on stage_party.party_type=e_partytype.party_type
where  e_partytype.party_type is null;

insert into qa_results
  (table_name, column_name, severity, data_value, description, time_check_run)
select distinct
      'stage_property',
      'property_type',
      'Fatal',
      stage_property.property_type,
      'Unrecognized property type',
      cast('!!$CURRENT_TIME!!' as datetime)
from  stage_property
      left join e_propertytype
        on stage_property.property_type=e_propertytype.property_type
where e_propertytype.property_type is null;
```

(continues on next page)

(continued from previous page)

```
insert into qa_results
  (table_name, column_name, severity, data_value, description, time_check_run)
select distinct
      'stage_partyprop',
      'property_rel',
      'Fatal',
      stage_partyprop.property_rel,
      'Unrecognized property relationship type',
      cast('!!$CURRENT_TIME!!' as datetime)
from  stage_partyprop
      left join e_partyproprel
        on stage_partyprop.property_rel=e_partyproprel.property_rel
where  e_partyproprel.property_rel is null;

-- !x! if(hasrows(qa_results)) { prompt message "Cancel if there are any fatal errors.
↪" display qa_results }
```

Example 4: Include a File if it Exists

This example illustrates how a script file can be modified by inclusion of an additional script only if that script file exists. This might be used when a general-purpose script is used to process data sets, and when some special data-set-specific processing is needed, that processing is coded in a separate script file, which is read into the main script only if it exists.

Each data set to be processed is identified by a unique name, which is defined with a SUB command in a script that is also read into the main script. The definition of the data set name might look like this, in a file named ds_name.sql:

```
-- !x! sub DS_NAME ALS4110-52
```

The main script then would look like this:

```
-- !x! include ds_name.sql
-- !x! if(file_exists(!!DS_NAME!!_fixes.sql)) { include !!DS_NAME!!_fixes.sql }
```

Example 5: Include a File if a Table Exists

Similar to *Example 4*, this example illustrates how a script file can be included if a database table exists. This might be used when carrying out quality assurance checks of data sets that have optional components. In this case, if an optional component has been loaded into a staging table, the script to check that component will be included.

```
-- !x! if(table_exists(staging.bioaccum_samp)) { include qa_bioaccumsamp.sql }
```

Example 6: Looping

Although execsql does not have any metacommands specifically for looping through groups of SQL statements or metacommands, the *IF* metacommand can be used with either the *INCLUDE* or *EXECUTE SCRIPT* metacommands to perform looping. Commands to be executed within a loop must be in a separate script (either in a separate file if the *INCLUDE* metacommand is used, or in a script block defined with the *BEGIN/END SCRIPT* metacommands), and that script should end with another *INCLUDE* or *EXECUTE SCRIPT* metacommand to continue the loop, or should forego re-running itself again to exit the loop.

This approach implements tail recursion. Either a single-line IF metacommand can be used, as shown here, or the script's recursive invocation of itself can be contained within a block IF statement.

A script to control a loop would invoke the inner loop script as follows:

```
-- !x! write "Before looping starts, we can do some stuff."
-- !x! include loop_inner.sql
-- !x! write "After looping is over, we can do more stuff."
```

In this example, the inner part of the loop is contained in a script file named `loop_inner.sql`. The inner loop script should have a structure like:

```
-- !x! write "Loop iteration number !!$counter_1!!"
-- !x! prompt ask "Do you want to loop again?" sub loop_again
-- !x! if(equals("!!loop_again!!", "Yes")) { include loop_inner.sql }
```

Termination of the loop may be controlled by some data condition instead of by an interactive prompt to the user. For example, you could loop for as many times as there are rows in a table by using the *SUBDATA* metacommand to get a count of all of the rows in a table, and then use the *IF(EQUALS())* conditional test to terminate the loop when a counter variable equals the number of rows in the table.

Every loop iteration increases the size of the script in memory, so execsql deallocates the memory used for script commands that have already been executed, to minimize the possibility of an out-of-memory error.

> *To iterate is human, to recurse divine.*
> – L Peter Deutsch

Example 7: Nested Variable Evaluation

This example illustrates nested evaluation of substitution variables, using scripts that print out all of the substitution variables that are assigned with the "-a" command-line option.

Because there may be an indefinite number of command-line variable assignments, a looping technique is used to evaluate them all. The outer level script that initiates the loop is simply:

```
-- !x! include arg_vars_loop.sql
```

The script that is called, `arg_vars_loop.sql`, is:

```
1 -- !x! sub argvar $ARG_!!$counter_1!!
2 -- !x! if(sub_defined(!!argvar!!))
3 -- !x!   write "Argument variable !!argvar!! is: !!!!argvar!!!!"
4 -- !x!   include arg_vars_loop.sql
5 -- !x! endif
```

On line 3 of this script the substitution variable argvar is first evaluated to generate a name for a command-line variable, consuming the inner pair of exclamation points. The resulting variable (which will take on values of "$ARG_1", "$ARG_2", etc.) will then be evaluated, yielding the value of the command-line variable assignment.

Example 8: Prompt the User to Choose an Option

This example illustrates how the *PROMPT SELECT_SUB* metacommand can be used to prompt the user to select among several options. In this example, the options allow the user to choose a format in which to (export and) view a data table or view. For this example, there must be a data table or view in the database named some_data.

```
drop table if exists formats;
create temporary table formats ( format varchar(4) );
insert into formats (format)
values ('csv'), ('tsv'), ('ods'), ('html'), ('txt'), ('pdf'), ('GUI');

-- !x! sub data_table some_data
-- !x! prompt select_sub formats message "Choose the output format you want."
-- !x! if(equals("!!@format!!", "GUI"))
-- !x!   prompt message "Selected data." display !!data_table!!
-- !x! else
-- !x!   sub outfile outfile_!!$DATETIME_TAG!!.!!@format!!
-- !x!   if(equals("!!@format!!", "pdf"))
-- !x!     sub txtfile outfile_!!$DATETIME_TAG!!.txt
-- !x!     write "# Data Table" to !!txtfile!!
-- !x!     export !!data_table!! append to !!txtfile!! as txt
-- !x!     system_cmd(pandoc !!txtfile!! -o !!outfile!!)
-- !x!   else
-- !x!     export !!data_table!! to !!outfile!! as !!@format!!
-- !x!   endif
-- !x!   system_cmd(xdg-open !!outfile!!)
-- !x! endif
```

This example also illustrates that, because the text ("txt") output format of the *EXPORT* metacommand creates a Markdown-compatible table, this type of text output can be combined with output of *WRITE* metacommands and converted to Portable Document Format (PDF). This example also illustrates how the *SYSTEM_CMD* metacommand can be used to immediately open and display a data file that was just exported. (Note that the `xdg-open` command is available in most Linux desktop environments. In Windows, the `start` command is equivalent.)

This example also illustrates how substitution variables can be used to parameterize code to support modularization and code re-use. In this example the substitution variable data_table is assigned a value at the beginning of the script. Alternatively, this variable might be assigned different values at different locations in a main script, and the commands in the remainder of this example placed in a second script that is INCLUDEd where appropriate to allow the export and display of several different data tables or views. *Example 10* illustrates this usage.

Example 9. Using Command-Line Substitution Variables

This example illustrates how substitution variables that are assigned on the command line using the "-a" option can be used in a script.

This example presumes the existence of a SQLite database named todo.db, and a table in that database named todo with columns named todo and date_due. The following script allows a to-do item to be added to the database by specifying the text of the to-do item and its due date on the command line:

```
-- !x! if(sub_defined($arg_1))
-- !x!   if(sub_defined($arg_2))
insert into todo (todo, date_due) values ('!!$arg_1!!', '!!$arg_2!!');
-- !x!   else
insert into todo (todo) values ('!!$arg_1!!');
-- !x! endif
```

This script can be used with a command line like:

```
execsql.py -tl -a "Share your dog food" -a 2015-11-21 add.sql todo.db
```

Example 10. Using CANCEL_HALT to Control Looping with Dialogs

This example illustrates the use of the *CANCEL_HALT* metacommand during user interaction with dialogs. Ordinarily when a user presses the "Cancel" button on a dialog, execsql treats this as an indication that a necessary response was not received, and that further script processing could have adverse consequences—and therefore execsql halts script processing. However, there are certain cases when the "Cancel" button is appropriately used to terminate a user interaction without stopping script processing.

The scripts in this example presents the user with a list of all views in the database, allows the user to select one, and then prompts the user to choose how to see the data. Three scripts are used:

- `view_views.sql`: This is the initial script that starts the process. It turns the *CANCEL_HALT* flag off at the start of the process, and turns it back on again at the end.
- `view_views2.sql`: This script is included by view_views.sql, and acts as an inner loop, repeatedly presenting the user with a list of all the views in the database. The "Cancel" button on this dialog is used to terminate the overall process. If the user selects a view, rather than canceling the process, then the choose_view.sql script is *INCLUDEd* to allow the user to choose how to see the data.
- `choose_view.sql`: This script presents the dialog that allows the user to choose how to see the data from the selected view. This is the same script used in *Example 8*, except that the data_table variable is defined in the `view_views2.sql` script instead of in `choose_view.sql`.

`view_views.sql`

```
create temporary view all_views as
select table_name from information_schema.views;

-- !x! cancel_halt off
-- !x! include view_views2.sql
-- !x! cancel_halt on
```

`view_views2.sql`

```
-- !x! prompt select_sub all_views message "Select the item you would like to view or␣
↪export; Cancel to quit."
-- !x! if(sub_defined(@table_name))
-- !x!    sub data_table !!@table_name!!
-- !x!    include choose_view.sql
-- !x!    include view_views2.sql
-- !x! endif
```

The `choose_view.sql` script can be seen in *Example 8*.

The CONTINUE keyword of the *PROMPT SELECT_SUB* metacommand can also be used to close the dialog without canceling the script.

Example 11. Output Numbering with Counters

This example illustrates how counter variables can be used to automatically number items. This example shows automatic numbering of components of a Markdown document, but the technique can also be used to number database objects such as tables and views.

This example creates a report of the results of a set of QA checks, where the information about the checks is contained in a table with the following columns:

- `check_number`: An integer that uniquely identifies each QA check that is conducted.
- `test_description`: A narrative description of the scope or operation of the check.

- `comment_text`: A narrative description of the results of the check.

The results of each check are also represented by tabular output that is saved in a table named qa_tbl_x where x is the check number.

```
-- !x! sub section !!$counter_1!!
-- !x! sub check_no !!$counter_3!!
-- !x! write "# Section !!section!!. QA Check !!check_no!!" to !!outfile!!
-- !x! reset counter 2
-- !x! write "## Subsection !!section!!.!!$counter_2!!. Test" to !!outfile!!
create or replace temporary view descript as
select test_description from qa_results where check_number = !!check_no!!;
-- !x! export descript append to !!outfile!! as plain
-- !x! write "## Subsection !!section!!.!!$counter_2!!. Results" to !!outfile!!
create or replace temporary view comment as
select comment_text from qa_results where check_number = !!check_no!!;
-- !x! export comment append to !!outfile!! as plain
-- !x! write "Table !!check_no!!." to !!outfile!!
-- !x! write "" to !!outfile!!
-- !x! export qa_tbl_!!check_no!! append to !!outfile!! as txt-nd
-- !x! write "" to !!outfile!!
```

A script like this one could be *INCLUDEd* as many times as there are sets of QA results to report.

This example also illustrates how the value of a counter variable can be preserved for repeated use by assigning it to a user-defined substitution variable.

Example 12. Customize the Table Structure for Data to be Imported

This example illustrates how the structure of a table that would be created by the *IMPORT* metacommand can be customized during the import process. Customization may be necessary because the data types that are automatically selected for the columns of the new table need to be modified. This may occur when:

- A column is entirely null. In this case, execsql will create the column with a text data type, whereas a different data type may be more appropriate.
- A column contains only integers of 1 and 0; execsql will create this column with a Boolean data type, whereas an integer type may be more appropriate.
- A column contains only integers, whereas a floating-point type may be more appropriate.

The technique shown here first writes the CREATE TABLE statement to a temporary file, and then opens that file in an editor so that you can make changes. After the file is edited and closed, the file is *INCLUDEd* to create the table structure, and then the data are loaded into that table.

```
-- !x! sub input_file storm_tracks.csv
-- !x! sub target staging.storm_data

-- !x! sub_tempfile tmpedit
-- !x! write create_table !!target!! from !!input_file!! comment "Modify the␣
↪structure as needed." to !!tmpedit!!
-- !x! system_cmd(vim !!tmpedit!!)
-- !x! include !!tmpedit!!
-- !x! import to !!target!! from !!input_file!!
```

Changes to data types that are incompatible with the data to be loaded will result in an error during the import process. Changes to column names will also prevent the data from being imported.

Although this example shows this process applied to only a single file/table, multiple CREATE TABLE statements can be written into a single file and edited all at once.

This example illustrates the use of a temporary file for the CREATE TABLE statement, although you may wish to save the edited form of this statement in a permanent file to keep a record of all data-handling operations.

Example 13. Import All the CSV Files in a Directory

When a group of related data files are to be loaded together into a database, they can all be loaded automatically with this script if they are first placed in the same directory. This example script operates by:

- Prompting for the directory containing the CSV files to load.
- Creating a text file with the names of all of the CSV files in that directory.
- Importing the text file into a database table.
- Adding columns to that table for the name of the table into which each CSV file is imported and the time of import. The main file name of each CSV file is used as the table name.
- Looping over the list of CSV files, choosing one that does not have an import time, importing that file, and setting the import time.

This process uses two script files. The first one obtains the list of CSV files, and the second one acts as the inner part of the loop, repeatedly loading a single CSV file. The main script looks like this:

```
-- !x! prompt directory sub dir
-- !x! sub_tempfile csvlist
-- !x! write "csvfile" to !!csvlist!!
-- !x! system_cmd(sh -c "ls !!dir!!/*.csv >>!!csvlist!!")
-- !x! import to new staging.csvlist from !!csvlist!!

alter table staging.csvlist
    add tablename character varying(64),
    add import_date timestamp with time zone;

update staging.csvlist
set tablename = replace(substring(csvfile from E'[^/]+\.csv$'), '.csv', '');

create temporary view unloaded as
select * from staging.csvlist
where import_date is null
limit 1;

-- !x! if(hasrows(unloaded))
-- !x!     write "Importing CSV files from !!dir!!"
-- !x!     include import1.sql
-- !x!     write "Done."
-- !x! endif
```

The second script, which must be named `import1.sql`, in accordance with the reference to it in the first script, looks like this:

```
-- !x! select_sub unloaded
-- !x! write "    !!@tablename!!"
-- !x! import to new staging.!!@tablename!! from !!@csvfile!!
update staging.csvlist
set import_date = current_timestamp
```

(continues on next page)

(continued from previous page)

```
where tablename = '!!@tablename!!';
-- !x! if(hasrows(unloaded)) { include import1.sql }
```

This example is designed to run on a Linux system with PostgreSQL, but the technique can be applied in other environments and with other DBMSs.

Example 14. Run a Script from a Library Database

Despite the advantages of storing scripts on the file system, in some cases storing a set of scripts in a library database may be appropriate. Consider a table named scriptlib that is used to store SQL scripts, and that has the following columns:

- `script_name`: A name used as a unique identifier for each script; the primary key of the table.
- `script_group`: A name used to associate related scripts.
- `group_master`: A Boolean used to flag the first script in a group that is to be executed.
- `script_text`: The text of the SQL script.

A single script from such a library database can be run using another script like the following:

```
-- !x! sub_tempfile scriptfile
create temporary view to_run as
select script_text from scriptlib
where script_name = 'check_sdg';
-- !x! export to_run to !!scriptfile!! as plain
-- !x! include !!scriptfile!!
```

This technique could be combined with a prompt for the script to run, using the method illustrated in *Example 8*, to create a tool that allows interactive selection and execution of SQL scripts.

This technique can be extended to export all scripts with the same script_group value, and then to run the master script for that group. To use this approach, the filename used with the *IMPORT* metacommand in each script must be a substitution variable that is to be replaced with the name of a temporary file created with the *SUB_TEMPFILE* metacommand.

Example 15: Prompt for Multiple Values

The *PROMPT SELECT_SUB* metacommand allows the selection of only one row of data at a time. Multiple selections can be obtained, however, by using the *PROMPT SELECT_SUB* metacommand in a loop and accumulating the results in another variable or variables.

This example illustrates that process, using a main script that *INCLUDEs* another script, `choose2.sql`, to present the prompt and accumulate the choices in the desired form.

The main script looks like this:

```
-- !x! sub prompt_msg Choose a sample material, or Cancel to quit.
create or replace temporary view vw_materials as
select sample_material, description from e_sampmaterial;
-- !x! cancel_halt off
-- !x! include choose2.sql
-- !x! cancel_halt on
-- !x! if(sub_defined(in_list))
           create or replace temporary view sel_samps as
```

(continues on next page)

(continued from previous page)

```
            select * from d_sampmain
            where sample_material in (!!in_list!!);
-- !x!      prompt message "Selected samples." display sel_samps
-- !x! endif
```

The script that is included, `choose2.sql`, looks like this:

```
-- !x! prompt select_sub vw_materials message "!!prompt_msg!!"
-- !x! if(sub_defined(@sample_material))
-- !x!      if(not sub_defined(in_list))
-- !x!          sub in_list '!!@sample_material!!'
-- !x!          sub msg_list !!@sample_material!!
-- !x!      else
-- !x!          sub in_list !!in_list!!, '!!@sample_material!!'
-- !x!          sub msg_list !!msg_list!!, !!@sample_material!!
-- !x!      endif
            create or replace temporary view vw_materials as
            select sample_material, description from e_sampmaterial
            where sample_material not in (!!in_list!!);
-- !x!      sub prompt_msg You have chosen !!msg_list!!; choose another, or Cancel to␣
↪quit.
-- !x!      include choose2.sql
-- !x! endif
```

In this example, only one value from each of the multiple selections is accumulated into a single string in the form of a list of SQL character values suitable for use with the SQL in operator. The multiple values could also be accumulated in the form of a values list, if appropriate to the intended use.

Another approach to handling multiple selection is to reassign each selected value to another substitution variable that has a name that is dynamically created using a counter variable, as shown in the following snippet.

```
-- !x!      sub selections !!$counter_1!!
-- !x!      sub sample_material_!!selections!! !!@sample_material!!
-- !x!      sub description_!!selections!! !!@description!!
```

Example 16: Evaluating Complex Expressions with Substitution Variables

Although execsql does not itself process mathematical expressions or other similar operations on substitution variables, all of the functions of SQL and the underlying DBMS can be used to evaluate complex expressions that use substitution variables. For example:

```
-- !x! sub var1 56
-- !x! sub var2 October 23
create temporary view var_sum as
select cast(right('!!var2!!', 2) as integer) + !!var1!! as x;
-- !x! subdata sum var_sum
```

This will assign the result of the expression to the substitution variable "sum". Any mathematical, string, date, or other functions supported by the DBMS in use can be applied to substitution variables in this way.

Example 17: Displaying Summary and Detailed Information

A set of QA checks performed on data may be summarized as a list of all of the checks that failed; however, there may also be detailed information about those results that the user would like to see—such as a list of all the data rows that

failed. Assuming that a view has been created for each QA check, and that the QA check failures have been compiled into a table of this form (see also *Example 3*):

```
create table qa_results (
    description character varying(80),
    detail_view character varying(24)
    );
```

The detail_view column should contain the name of the view with detailed information about the QA failure. Both the summary and detailed information can be presented to the viewer using the following statements in the main script:

```
-- !x! if(hasrows(qa_results))
    -- !x! sub prompt_msg Select an item to view details, Cancel to quit data loading.
    -- !x! include qa_detail.sql
-- !x! endif
```

Where the qa_detail.sql script is as follows:

```
-- !x! prompt select_sub qa_results message "!!prompt_msg!!" continue
-- !x! if(sub_defined(@detail_view))
    -- !x! prompt message "!!@description!!" display !!detail_view!!
    -- !x! include qa_detail.sql
-- !x! endif
```

The user can cancel further script processing using the "Cancel" button on either the summary dialog box or any of the detail displays. If the "Continue" button is chosen on the summary dialog box, script processing will resume.

Example 18: Creating a Simple Entry Form

This example illustrates the creation of a simple data entry form using the *PROMPT ENTRY_FORM* metacommand. In this example, the form is used to get a numeric value and a recognized set of units for that value, and then display that value converted to all compatible units in the database.

This example relies on the presence of a unit dictionary in the database (`e_unit`) that contains the unit code, the dimenension of that unit, and a conversion factor to convert values to a standard unit for the dimension. This example uses two scripts, named unit_conv.sql and unit_conv2.sql, the first of which *INCLUDEs* the second.

`unit_conv.sql`

```
create temporary table inp_spec (
    sub_var text,
    prompt text,
    required boolean,
    initial_value text,
    width integer,
    lookup_table text,
    validation_regex text,
    validation_key_regex text,
    sequence integer
    );

insert into inp_spec
    (sub_var, prompt, required, width, lookup_table, validation_regex, validation_key_
↪regex, sequence)
values
    ('value', 'Numeric value', True, 12, null, '-?[0-9]+\\.?[0-9]*', '-?[0-9]?\\.?[0-
↪9]*', 1),
```

(continues on next page)

(continued from previous page)

```
    ('unit', 'Unit', True, null, 'e_unit', null, null, 2),
    ('comment', 'Comment', False, 40, null, null, null, 3)
    ;

create or replace temporary view entries as
select * from inp_spec order by sequence;

-- !x! prompt entry_form entries message "Enter a value, unit, and comment."

-- !x! include unit_conv2.sql
```

Note that to include a decimal point in the regular expressions for the numeric value, the decimal point must be escaped twice: once for SQL, and once for the regular expression itself. Also note that in this case, the validation_regex and the validation_key_regex are identical except that all subexpressions in the latter are optional. If the first digit character class were not optional, then at least one digit would always be required, and entry of a leading negative sign would not be possible (though a negative sign could be added after at least one digit was entered).

`unit_conv2.sql`

```
create or replace temporary view conversions as
select
    !!value!! * su.factor / du.factor as converted_value,
    du.unit
from
    e_unit as su
    inner join e_unit as du on du.dimension=su.dimension and du.unit <> su.unit
where
    su.unit='!!unit!!'
order by
    du.unit;

update inp_spec
set initial_value='!!value!!'
where sub_var = 'value';
-- !x! sub old_value !!value!!

update inp_spec
set initial_value='!!unit!!'
where sub_var = 'unit';
-- !x! sub old_unit !!unit!!

-- !x! if(sub_defined(comment))
update inp_spec
set initial_value='!!comment!!'
where sub_var = 'comment';
-- !x! endif

-- !x! prompt entry_form entries message "The conversions for !!value!! !!unit!! are:
↪"display conversions

-- !x! if(not equal("!!unit!!", "!!old_unit!!"))
-- !x! orif(not equal("!!value!!", "!!old_value!!"))
-- !x! include unit_conv2.sql
-- !x! endif
```

The unit_conv2.sql script will continue to display conversions for as long as either the value or the unit is changed.

Example 19: Dynamically Altering a Table Structure to Fit Data

Example contributed by E. Shea.

This example illustrates automatic (scripted) revisions to a table structure, wherein a number of additional columns are added to a table; the number of columns added is determined by the data. The case illustrated here is of a Postgres table containing a PostGIS multipoint geometry column. The purpose of this script is to extract the coordinate points from the geometry column and store each point as a pair of columns containing latitude and longitude values. The number of points in the multipoint column varies from row to row, and the maximum number of points across all rows is not known (and need not be known) when this script is run.

This example assumes the existence of a database table named `sample_points` that contains the following two columns:

- `sample`: a text value uniquely identifying each row.
- `locations`: a multi-point PostGIS geometry value.

This operation is carried out using two scripts, named `expand_geom.sql` and `expand_geom2.sql`. The first of these calls the second. Looping and a counter variable are used to create and fill as many additional columns as are needed to contain all the point coordinates.

`expand_geom.sql`

```
create temporary view vw_pointcount as
select max(ST_NumGeometries(locations)) as point_len
from sample_points;

--!x! subdata point_len vw_pointcount

drop table if exists tt_samp_loc_points;
create temporary table tt_samp_loc_points (
    sample text,
    locations geometry,
    location_srid integer,
    point_count integer
    );

insert into tt_samp_loc_points
    (sample, locations, location_srid, point_count)
select
    sample, locations, ST_SRID(locations), ST_NumGeometries(locations)
from sample_points;

--!x! include expand_geom2.sql
```

`expand_geom2.sql`

```
--!x! sub point_number !!$counter_1!!

alter table tt_samp_loc_points
    add column point_!!point_number!!_longitude double precision,
    add column point_!!point_number!!_latitude double precision;

update tt_samp_loc_points
set
    point_!!point_number!!_longitude = ST_X(ST_GeometryN(locations, !!point_number!!
↪)),
    point_!!point_number!!_latitude = ST_Y(ST_GeometryN(locations, !!point_number!!))
```

(continues on next page)

(continued from previous page)

```
where
    point_count>=!!point_number!!;

--!x! if(is_gt(!!point_len!!, !!point_number!!)) {include expand_geom2.sql}
```

Example 20: Logging Data Quality Issues

This example illustrates how data quality issues that are encountered during data loading or cleaning operations can be logged for later evaluation and resolution. Issues are logged in a SQLite database in the working directory. This database is named issue_log.sqlite and is automatically created if necessary. The database contains one table named issue_log in which all issues are recorded. The issue log database may also contain additional tables that provide data to illustrate the issues. Each of these additional tables has a name that starts with "issue_", followed by an automatically-assigned issue number.

The script that logs the issues is named `log_issue.sql`. It should be included in the main script at every point where an issue is to be logged. Three substitution variables are used to pass information to this script:

- `dataset`: The name of the data set to which this issue applies.
- `issue`: A description of the issue.
- `issue_data`: The name of a table or view containing a data summary that illustrates the issue. This substitution variable need not be defined if no illustrative data are necessary or applicable (use the *RM_SUB* metacommand to un-define this variable if it has been previously used).

Each issue is logged only once. The issue_log table is created with additional columns that may be used to record the resolution of each issue, and these are not overwritten if an issue is encountered repeatedly.

The `log\_issue.sql` script uses several substitution variables with names that start with "iss_", and uses counter variable 15503. Other parts of the loading and cleaning scripts should avoid collisions with these values.

`log_issue.sql`

```
-- ------------------------------------------------------------------
-- Save the current alias to allow switching back from the SQLite issue log.
-- ------------------------------------------------------------------
-- !x! sub iss_calleralias !!$CURRENT_ALIAS!!

-- ------------------------------------------------------------------
-- Connect to the issue log.  Create it if it doesn't exist.
-- ------------------------------------------------------------------
-- !x! if(alias_defined(iss_log))
-- !x!      use iss_log
-- !x! else
-- !x!      if(file_exists(issue_log.sqlite))
-- !x!              connect to sqlite(file=issue_log.sqlite) as iss_log
-- !x!              use iss_log
        create temporary view if not exists iss_no_max as
        select max(issue_no) from issue_log;
-- !x!              subdata iss_max iss_no_max
-- !x!              set counter 15503 to !!iss_max!!
-- !x!      else
-- !x!              connect to sqlite(file=issue_log.sqlite, new) as iss_log
-- !x!              use iss_log
        create table issue_log (
            issue_no integer,
            data_set text,
```

(continues on next page)

(continued from previous page)

```
            issue_summary text,
            issue_details text,
            resolution_strategy text,
            resolution_reviewer text,
            reviewer_comments text,
            final_resolution text,
            resolution_implemented date,
            resolution_script text
            );
-- !x!      endif
-- !x! endif

-- ---------------------------------------------------------------------
-- Add the new issue if it does not already exist.
-- ---------------------------------------------------------------------
drop view if exists check_issue;
create temporary view check_issue as
    select issue_no
    from issue_log
    where data_set='!!dataset!!'
        and issue_summary='!!issue!!';

-- !x! if(not hasrows(check_issue))
-- !x!      sub iss_no !!$counter_15503!!
    insert into issue_log
        (issue_no, data_set, issue_summary)
    values (
        !!iss_no!!,
        '!!dataset!!',
        '!!issue!!'
        );
-- !x! else
-- !x!      subdata iss_no check_issue
-- !x! endif

-- ---------------------------------------------------------------------
-- Save the issue data if there is any and it has not already
-- been saved.
-- ---------------------------------------------------------------------
-- !x! if(sub_defined(issue_data))
-- !x!      sub iss_table issue_!!iss_no!!
-- !x!      if(not table_exists(!!iss_table!!))
-- !x!              copy !!issue_data!! from !!iss_calleralias!! to replacement !!iss_
↪table!! in iss_log
-- !x!      endif
-- !x! endif

-- ---------------------------------------------------------------------
-- Restore the caller alias
-- ---------------------------------------------------------------------
-- !x! use !!iss_calleralias!!
```

This script would be used during a data loading or cleaing process as illustrated in the following code snippet.

```
-- ---------------------------------------------------------------------
-- Define the data set both for import and for issue logging.
-- ---------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
-- !x! sub dataset data_set_524.csv
-- !x! write "Importing data."
-- !x! import to replacement data524 from !!dataset!!

-- ---------------------------------------------------------------
--  Check for multiple stations per sample.
-- ---------------------------------------------------------------
-- !x! write "Checking for multiple stations per sample."
create temporary view samploc as
    select distinct loc_name, sample_name
    from data524;

create temporary view dupsamplocs as
    select sample_name
    from samploc
    group by sample_name
    having count(*) > 1;

create or replace temporary view allsampdups as
    select samploc.sample_name, loc_name
    from samploc
        inner join dupsamplocs as d on d.sample_name=samploc.sample_name
    order by samploc.sample_name;

-- !x! if(hasrows(dupsamplocs))
-- !x!      sub issue Multiple locations for a sample.
-- !x!      sub issue_data allsampdups
-- !x!      include log_issue.sql
-- !x! endif
```

Example 21: Updating Multiple Databases with a Cross-Database Transaction

This example illustrates how the same SQL script can be applied to multiple databases, and the changes committed only if they were successful for all databases. This makes use of the looping technique illustrated in *Example 6*, but using sub-scripts defined with *BEGIN/END SCRIPT* metacommands instead of *INCLUDE* metacommands. The same approach, of committing changes only if there were no errors in any database, could also be done without looping, simply by unrolling the loops to apply the updates to each database in turn. This latter approach would be necessary when different changes were to be made to each database—though even in that case, the commit statements could all be executed in a loop.

```
-- =========================================================
--          Setup
-- Put database names in a table so that we can loop through
-- the list and process the databases one by one.
-- ---------------------------------------------------------
create temporary table dblist (
    database text,
    db_alias text,
    updated boolean default false
    );

insert into dblist
    (database)
values
    ('db1'), ('db2'), ('db3');
```

(continues on next page)

(continued from previous page)

```
update dblist
set db_alias = 'conn_' || database;

create temporary view unupdated as
select *
from dblist
where not updated
limit 1;
-- ===========================================================


-- ===========================================================
--        Script: Update a single database
-- Data variables @database and @db_alias should be defined.
-- -----------------------------------------------------------
-- !x! BEGIN SCRIPT update_database
-- !x! connect to postgresql(server=myserver, db=!!@database!!, user=!!$db_user!!,␣
↪need_pwd=True) as !!@db_alias!!
-- !x! use !!@db_alias!!

-- Begin a transaction
begin;

<SQL commands to carry out the update>

-- !x! use initial
-- !x! END SCRIPT
-- ===========================================================


-- ===========================================================
--        Script: Update all databases
-- -----------------------------------------------------------
-- !x! BEGIN SCRIPT update_all_dbs
-- !x! select_sub unupdated
-- !x! if(sub_defined(@database))
    -- !x! write "Updating !!@database!!"
    -- !x! execute script update_db
    update dblist set updated=True where database='!!@database!!';
    -- !x! execute script update_all_dbs
-- !x! endif
-- !x! END SCRIPT
-- ===========================================================


-- ===========================================================
--        Script: Commit changes to all databases
-- -----------------------------------------------------------
-- !x! BEGIN SCRIPT commit_all_dbs
-- !x! select_sub unupdated
-- !x! if(sub_defined(@db_alias))
    -- !x! write "Committing changes to !!@database!!"
    -- !x! use !!@db_alias!!
    commit;
    -- !x! use initial
    update dblist set updated=True where database='!!@database!!';
```

(continues on next page)

(continued from previous page)

```
    -- !x! execute script commit_all_dbs
-- !x! endif
-- !x! END SCRIPT
-- ========================================================


-- ========================================================
--        Update all databases
-- --------------------------------------------------------
-- !x! autocommit off
-- !x! execute script update_all_dbs

-- If there was an error during updating of any database,
-- the error should halt execsql, and the following code
-- will not run.
update dblist set updated=False;
-- !x! execute script commit_all_dbs
-- ========================================================
```

Example 22: Exporting with a Template to Create a Wikipedia Table

This example illustrates how the *EXPORT* metacommand can be used with the Jinja2 template processor to create a simple Wikipedia table. To run this example, Jinja2 must be specified as the template processor to use; the *configuration file* must contain the following lines:

```
[output]
template_processor=jinja
```

The following template file to be used with the *EXPORT* metacommand will result in output that will render the exported data as a Wikipedia table.

```
{| class="wikitable"{% for hdr in headers %}
! scope="col" | {{ hdr }}{% endfor %}
|-
{% for row in datatable %}{% for hdr in headers %}{% for key, value in row.
↪iteritems() %}{% if  key == hdr %}| {{ value }}
{% endif %}{% endfor %}{% endfor %}|-
{% endfor %}
```

The output produced using this template will look like:

```
{| class="wikitable"
! scope="col" | row_id
! scope="col" | row_number
! scope="col" | long_text
! scope="col" | some_date
! scope="col" | some_number
|-
| Row 1
| 1
| Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
| 1951-03-18
| 67.7593972309
|-
| Row 2
```

(continues on next page)

(continued from previous page)

```
| 2
| Aenean commodo ligula eget dolor. Aenean massa.
| 1951-03-19
| 26.7195924129
|-
. . .
|}
```

This template will work with any exported data table.

Example 23: Validation of PROMPT ENTRY_FORM Entries

Example contributed by E. Shea.

This example illustrates how validation of interactively entered data can be performed, and the user prompted repeatedly until the entered data are valid.

Although the *PROMPT ENTRY_FORM* metacommand allows validation of individual entries, this capability is limited, and cannot be used to cross-validate different entries. The following code demonstrates a prompt for a pair of dates, where validation of both the individual entries and a properly-ordered relationship between the two entries is carried out using a *SCRIPT* that re-runs itself if the data are not valid.

```
create temporary table tt_datespecs (
    sub_var text,
    prompt text,
    required boolean default True,
    sequence integer
    );
insert into tt_datespecs (sub_var, prompt, sequence)
values
    ('min_date', 'Start date', 1),
    ('max_date', 'End_date', 2);

-- !x! begin script getdates
   -- !x! if(sub_defined(getdate_error)) { sub getdate_prompt !!getdate_error!! }
   -- !x! sub_append getdate_prompt Enter date range to include:
   -- !x! prompt entry_form tt_datespecs message "!!getdate_prompt!!"

   drop view if exists qa_date1;
   drop view if exists qa_date2;

   -- !x! error_halt off
    create or replace temporary view qa_date1 as
    select
        cast('!!min_date!!' as date) as start_date,
        cast('!!max_date!!' as date) as end_date;
    -- !x! error_halt on
    -- !x! if(sql_error())
        -- !x! sub getdate_error ** ERROR **  One or both of the values you entered␣
↪is not interpretable as a date.
        -- !x! execute script getdates
    -- !x! else
            create or replace temporary view qa_date2 as
            select 1
            where  cast('!!min_date!!' as date) > cast('!!max_date!!' as date);
            -- !x! if(hasrows(qa_date2))
```

(continues on next page)

(continued from previous page)

```
            -- !x!     sub getdate_error ** ERROR **  The end date must be equal to␣
↪or later than the start date.
            -- !x!     execute script getdates
            -- !x! endif
    -- !x! endif

    -- !x! rm_sub getdate_error

-- !x! end script
```

2.4.14 Availability

The execsql program is available on PyPi. It can be installed with:

```
pip install execsql
```

The latest code is available from the Bibucket repository.

2.4.15 Copyright and License

2.4.16 Contributors

Elizabeth Shea Examples and corrections to the *PROMPT ENTRY_FORM* metacommand.

2.4.17 Change Log

Version	Date	Features
1.30.0.0	2018-09-22	Changed to write a description of the binary data length when binary data are used with PROMPT DISPLA
1.29.3.0	2018-09-22	Modified the WRITE metacommand to use the 'make_export_dirs' configuration setting.
1.29.2.0	2018-09-19	Added the SUB_ADD metacommand. Made the WITH keyword optional in the IMPORT metacommand.
1.29.0.0	2018-09-12	Added the IMPORT_FILE metacommand.
1.28.0.0	2018-08-19	Modified to run under Python 3.x as well as 2.7.
1.27.4.0	2018-08-19	Now writes Python version number into execsql.log.
1.27.3.0	2018-07-31	Changed to read configuration files from both the script directory and the starting directory, if different.
1.27.2.0	2018-07-30	Added the SUB_EMPTY metacommand.
1.27.1.0	2018-07-29	Allowed single quotes and square brackets in ON ERROR_HALT WRITE and ON CANCEL_HALT WRI
1.27.0.0	2018-07-29	Internal script processing routines rewritten.
1.26.8.0	2018-07-25	Allowed single quotes and square brackets in WRITE metacommand. Fixed stripping of extra spaces from
1.26.5.0	2018-07-20	Added $PYTHON_EXECUTABLE system variable. Trimmed any trailing space from last column header
1.26.4.3	2018-07-12	Corrected handling of double-quoted filenames by the ON ERROR_HALT WRITE and ON CANCEL_HA

Version	Date	Features
1.26.4.2	2018-07-09	Corrected handling of double-quoted filenames by the WRITE and RM_FILE metacommands.
1.26.4.0	2018-06-27	Added $STARTING_SCRIPT_NAME and $CURRENT_SCRIPT_NAME system variables. Added IS_TR
1.26.2.0	2018-06-25	Added a $CURRENT_SCRIPT_PATH system variable that returns the path only of the current script file.
1.26.1.0	2018-06-13	Corrected hang on uppercase counter references. Changed HALT metacommands to set the exit code to 3.
1.26.0.0	2018-06-13	Added ON CANCEL_HALT WRITE and ON CANCEL_HALT EMAIL metacommands.
1.25.0.0	2018-06-10	Added the PROMPT COMPARE metacommand.
1.24.12.0	2018-06-09	Added a MAKE_EXPORT_DIRS metacommand. Grouped all metacommands corresponding to configura
1.24.9.0	2018-06-03	Modified the IMPORT metacommand to write the file name, file size, and file date to execsql.log.
1.24.8.0	2018-06-03	Corrected 'is_null()', 'equals()', and 'identical()' to strip quotes. Added filename to error message when th
1.24.7.0	2018-04-03	Added the $SYSTEM_CMD_EXIT_STATUS system variable.
1.24.6.0	2018-04-01	Added the "B64" format to the EXPORT and EXPORT_QUERY metacommands.
1.24.5.0	2018-03-15	Added the 'textarea' entry type to the PROMPT ENTRY_FORM metacommand.
1.24.4.0	2017-12-31	Allowed "CREATE SCRIPT" as an alias for "BEGIN SCRIPT". Allowed "DEBUG WRITE SCRIPT" as a
1.24.2.0	2017-12-30	Modified characters allowed in user names for Postgres and ODBC connections. Added the TYPE and LC
1.24.0.0	2017-11-04	Added the 'include_required' and 'include_optional' configuration settings.
1.23.3.0	2017-11-03	Added the CONSOLE_WAIT_WHEN_ERROR_HALT setting and associated metacommand and system v
1.23.2.0	2017-11-02	Added the $ERROR_MESSAGE system variable.
1.23.1.0	2017-10-20	Added the ASK metacommand.
1.23.0.0	2017-10-09	Added the ON ERROR_HALT EMAIL metacommand.
1.22.0.0	2017-10-07	Added the ON ERROR_HALT WRITE metacommand.
1.21.13.0	2017-09-29	Added the SUB_APPEND and WRITE SCRIPT metacommands. Modified all metacommand messages to
1.21.12.0	2017-09-24	Added the PG_VACUUM metacommand.
1.21.11.0	2017-09-23	Modified error message content and format.
1.21.10.0	2017-09-12	Added the "error_response" configuration setting for encoding mismatches.
1.21.9.0	2017-09-06	Modified to handle trailing comments on SQL script lines.
1.21.8.0	2017-08-11	Modified to allow a password to be specified in the CONNECT metacommand for MySQL.
1.21.7.0	2017-08-05	Modified to allow import of CSV files with more columns than the target table. Added DEBUG metacomm
1.21.1.0	2017-07-04	Passed column headers to template processors as a separate object.
1.21.0.0	2017-07-01	Extended the EXPORT metacommand to allow several different template processors to be used.
1.20.0.0	2017-06-30	Added the EMAIL, SUB_ENCRYPT, and SUB_DECRYPT metacommands, and configuration properties
1.18.0.0	2017-06-24	Improved the speed of import of CSV files to Postgres and MySQL/MariaDB. Modified the EXPORT... Al
1.17.0.0	2017-05-28	Modified the specifications for the PROMPT ENTRY_FORM to allow checkboxes to be used.
1.16.9.0	2017-05-27	Added a DESCRIPTION keyword to the EXPORT metacommands.
1.16.8.0	2017-05-20	Added the VALUES export format.
1.16.7.0	2017-05-20	Added BOOLEAN_INT and BOOLEAN_WORDS metacommands. Allowed the PAUSE metacommand t
1.16.3.0	2017-04-23	Added a configuration option allowing the specification of additional configuration files to read. Added a M
1.16.0.0	2017-03-25	Added the BEGIN SCRIPT, END SCRIPT, and EXECUTE SCRIPT metacommands.
1.15.0.0	2017-03-09	Added the TEE keyword to the WRITE, EXPORT, and EXPORT QUERY metacommands.
1.13.0.0	2017-03-05	Added the LOG_WRITE_MESSAGES metacommand and configuration parameter.
1.12.0.0	2017-03-04	Added a 'boolean_words' configuration option. Enabled reading of CSV files with newlines within delimit
1.8.15.0	2017-01-14	Added a $LAST_ROWCOUNT system variable.
1.8.14.0	2016-11-13	Added evaluation of numeric types in input. Added 'empty_strings' configuration parameter and metacom
1.8.13.0	2016-11-07	Added the "-b" command-line option and configuration parameter.
1.8.12.0	2016-10-22	Added the RM_SUB metacommand.
1.8.11.0	2016-10-19	Added the SET COUNTER metacommand.
1.8.10.2	2016-10-17	Added $RUN_ID system variable Modified to recognize as text any imported data that contains only nume
1.8.8.0	2016-09-28	Added $CURRENT_ALIAS, $RANDOM, and $UUID system variables.
1.8.4.0	2016-08-13	Added logging of database close when autocommit is off. Added import from MS-Excel. Corrected parsin
1.7.3.0	2016-08-05	Added $OS system variable.

Version	Date	Features
1.7.2.0	2016-06-11	Added DIRECTORY_EXISTS conditional and option to automatically make directories used by the EXPO
1.7.0.0	2016-05-20	Added NEWER_DATE and NEWER_FILE conditionals.
1.6.0.0	2016-05-15	Added CONSOLE SAVE metacommand. Added DSN connections. Added COPY QUERY and EXPORT
1.4.4.0	2016-05-02	Added CONSOLE HIDE SHOW metacommands and allowed <Enter> in response to the CONSOLE WAI
1.4.2.0	2016-05-02	Added a "Save as…" menu to the GUI console and changed the PAUSE and HALT metacommands to use
1.4.0.0	2016-04-30	Added a GUI console with a status bar and progress bar to which WRITE output and exported text will be
1.3.3.0	2016-04-09	Additions to 'Save as…' options in PROMPT DISPLAY metacommand, and date/time values exported to
1.3.2.0	2016-02-28	Enabled the use of a backslash as a line continuation character for SQL statements.
1.3.1.0	2016-02-20	Added PROMPT ENTRY_FORM and LOG metacommands.
1.2.15.0	2016-02-14	Added $DB_NAME, $DB_NEED_PWD, $DB_SERVER, and $DB_USER system variables. Added RAW
1.2.10.0	2016-01-23	Added ENCODING keyword to IMPORT metacommand. Added TIMER metacommand and $TIMER sys
1.2.8.2	2016-01-21	Fixed extra quoting in drop table method. Fixed str coercion in TXT export.
1.2.8.0	2016-01-11	Suppressed column headers when EXPORTing to CSV and TSV with APPEND. Eliminated %H%M patte
1.2.7.1	2016-01-03	Modified import of integers to Postgres; added the AUTOCOMMIT metacommand and modified the BAT
1.2.4.6	2015-12-19	Modified quoting of column names for the COPY and IMPORT metacommands.
1.2.4.5	2015-12-17	Fixed asterisks in PROMPT ENTER_SUB.
1.2.4.4	2015-12-14	Fixed regexes for quoted filenames.
1.2.4.3	2015-12-13	Fixed -y option display; fixed parsing of WRITE CREATE_TABLE comment option; fixed parsing of back
1.2.4.0	2015-11-21	Added connections to PostgreSQL, SQL Server, MySQL, MariaDB, SQLite, and Firebird. Added numerou
0.4.4.0	2010-06-20	Added INCLUDE, WRITE, EXPORT, SUB, EXECUTE, HALT, and IF (HASROWS, SQL_ERROR) meta
0.3.1.0	2008-12-19	Executes SQL against Access, captures output of the last statement.

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W