Guile-pg Command Reference

Guile-pg

1 Introduction

`Guile-pg` is a Guile module providing a set of procedures to allow access to the PostgreSQL RDBMS from Scheme programs.

Guile is an interpreter for the Scheme programming language: a dialect of Lisp. PostgreSQL is a free SQL database management system. `Guile-pg` is intended as but one of the low-level drivers for the forthcoming Guile database module which will provide a portable interface to many different SQL database servers. In the mean-time it will still be a useful module for those who want just a low-level interface to PostgreSQL.

The `Guile-pg` procedures have an almost one-one correspondence with the PostgreSQL `libpq` C library interface so those familiar with the `libpq` library and other similar interfaces like Edmund Mergl's `Pg` Perl module will find it familiar.

`Guile-pg` is based on Russ McManus's "Sybase glue" code.

2 Examples

To access a PostgreSQL database from Guile, first load the `postgres` database module. This is done using the `use-modules` syntax:

```
(use-modules (database postgres))
```

The next step is to make a connection to the database using the `pg-connectdb` procedure. Then, having created a valid connection to a database server, we can use the `pg-exec` procedure to execute queries and the other `pg-` procedures to retrieve the results of the queries.

The following Guile session opens a connection and executes an SQL statement on the `test` database on the default host (either `localhost` or the host specified in the `PGHOST` environment variable.)

```
guile> (use-modules (database postgres))
guile> (define test (pg-connectdb "dbname=test"))
guile> test
#<PG-CONN:1:test::5432:>
guile> (define result (pg-exec test "SELECT * FROM PEOPLE"))
guile> result
#<PG-RESULT:1:PGRES_TUPLES_OK:3:5>
```

The `test` object is a `PG-CONN` type representing the database connection. Displaying the `PG-CONN` object shows that the serial number of the connection is 1, that the database name is `test` on the default host (the hostname field is empty) on port number 5432 and that the default options were passed to the backend. This object is passed to `pg-exec` which returns a `PG-RESULT` object representing the result of the SQL statement executed on the server. In this case, displaying the `PG-RESULT` object shows us that the serial number of the result is 2, that the return code from the database was `PGRES_TUPLES_OK` and that we have received 3 tuples with 5 fields.

Having obtained a result object from the database we can find out how many rows and how many columns are in the result. We can also find out the names of the columns:

```
guile> (pg-ntuples result)
3
guile> (pg-nfields result)
5
guile> (pg-fname result 0)
"surname"
guile> (pg-fname result 1)
"firstname"
```

To retrieve the data from the result we use the `pg-getvalue` procedure. This takes a result object along with the row and column numbers and returns a string containing that value. `pg-getvalue` always returns a string: it is up to the caller to convert this to whatever type they require.

```
guile> (pg-getvalue result 0 0)
"Bitdiddle"
guile> (pg-getvalue result 0 1)
"Ben"
guile> (pg-getvalue result 1 0)
"Ator"
guile> (pg-getvalue result 1 1)
"Eva Lu"
```

`Guile-pg` connections are closed when Guile's garbage collector collects the discarded object. This only happens when the connection itself and all of the results of `pg-exec` are unreachable. So to close the connection we just rebind the results from that connection and the connection symbol itself:

```
guile> (define result '())
guile> (define test '())
```

Typically though the result and connection variables will be `let` bindings so the connection will usually be closed soon after the body of the `let` returns.

3 Procedures

This section specifies the procedures that `guile-pg` provides to access databases. To access the module, Guile programs must either be run under the `guile-pg` executable, which is a Guile interpreter statically linked with the `guile-pg` interface and the `libpq` library, or they must be run under the ordinary Guile interpreter `guile` and dynamically load the shared library `libpostgres.so` which implements the module. To do this, use the `(use-modules ...)` procedure as follows:

```
(use-modules (database postgres))
```

After which the `pg-` procedures will be accessable.

3.1 Functions for managing connections

pg-connectdb *connect-string* Function

Open a connection to a database. *connect-string* should be a string consisting of zero or more space-separated `name=value` pairs. If the value contains spaces it must be enclosed in single quotes and any single quotes appearing in the value must be escaped using backslashes. Backslashes appearing in the value must similarly be escaped. Note that if the `connect-string` is a Guile string literal then all the backslashes will themselves require to be escaped a second time. The `name` strings can be any of:

`host`
: The host-name or dotted-decimal IP address of the host on which the postmaster is running. If no `host=` sub-string is given then the host is assumed to be the value of the environment variable `PGHOST` or the local host if `PGHOST` is not defined.

`port`
: The TCP or Unix socket on which the backend is listening. If this is not specified then the value of the `PGHOST` environment variable is used. If that too is not defined then the default port 5432 is assumed.

`options`
: A string containing the options to the backend server. The options given here are in addition to the options given by the environment variable `PGOPTIONS`. The options string should be a set of command line switches as would be passed to the backend. See the `postgres (1)` man page for more details.

`tty`
: A string defining the file or device on which error messages from the backend are to be displayed. If this is empty (`""`) then the environment variable `PGTTY` is checked. If the specified `tty` is a file then the file will be readable only by the user the postmaster runs as (usually `postgres`). Similarly, if the specified `tty` is a device then it must have permissions allowing the postmaster user to write to it.

`dbname`
: The name of the database. If no `dbname=` sub-string is given then the database name is assumed to be that given by the value of

the PGDATABASE environment variable, or the USER environment variable if the PGDATABASE environment variable is not defined. If the USER environment variable is not specified either then the value of the user option is taken as the database name.

`user` The login name of the user to authenticate. If none is given then the PGUSER environment variable is checked. If that is not given then the login of the user owning the process is used.

`password` The password. Whether or not this is used depends upon the contents of the pg_hba.conf file. See the pg_hba.conf (5) man page for details.

`authtype` This must be set to password if password authentication is in use, otherwise it must not be specified.

Example

```
(define (user-count host)
  (let* ((conn (pg-connectdb
                 (string-append
                    "host=" host
                    " port=5432 dbname=template1")))
         (result (pg-exec conn
                    "SELECT COUNT(*) FROM pg_user")))
        (if (and result
                 (eq? (pg-result-status result)
                      PGRES_TUPLES_OK))
            (string->number (pg-getvalue result 0 0))
            #f)))
```

pg-reset *conn* Function

Resets the connection with the backend. Equivalent to closing the connection and re-opening it again with the same connect options as given to pg-connectdb. *conn* must be a valid PG_CONN object returned by pg-connectdb.

pg-get-db *conn* Function

Returns a string containing the name of the database to which *conn* represents a connection.

pg-get-user *conn* Function

Returns a string containing the user name used to authenticate the connection *conn*.

pg-get-pass *conn* Function

Returns a string containing the password used to authenticate the connection *conn*.

pg-get-host *conn* Function

Returns a string containing the name of the host to which *conn* represents a connection.

pg-get-port *conn* Function

Returns a string containing the port number to which *conn* represents a connection.

pg-get-tty *conn* Function

Returns a string containing the the name of the diagnostic tty for *conn*.

pg-get-options *conn* Function

Returns a string containing the the options string for *conn*.

pg-backend-pid *conn* Function

Returns an integer which is the the PID of the backend process for *conn*.

pg-get-client-data *conn* Function

Returns the the client data associated with *conn*.

pg-set-client-data! *conn data* Function

Associates *data* with *conn*.

Example

The following is a set of three procedures which implement a database connection object that logs all the SQL statements it executes.

```
(define (pg-connectdb-logged options)
  (let ((conn (pg-connectdb options)))
       (if conn (begin (pg-set-client-data! conn '())
                       conn)
                #f)))

(define (pg-exec-logged conn sql)
  (pg-set-client-data! conn
                       (append
                         (pg-get-client-data conn)
                         (list sql)))
  (pg-exec conn sql))

(define (pg-conn-log conn)
  (pg-get-client-data conn))
```

3.2 Miscellaneous Functions

pg-guile-pg-loaded Function

Returns `#t` indicating that the binary part of `guile-pg` is loaded. Thus to test if `guile-pg` is loaded, use

```
(defined? 'pg-guile-pg-loaded)
```

Note

This function is guaranteed to be present in all future versions of `guile-pg`.

pg-guile-pg-version Function

Returns a string giving the version of `guile-pg`. The string is always of the form "M.m" giving major and minor versions.

pg-guile-pg-module-version Function

Returns a string giving the version of the scheme module `postgres.scm`. The string is always of the form "M.m" giving major and minor versions. Unless something is wrong with the installation, this should be the same as `pg-guile-pg-version`.

pg-guile-pg-module-config-stamp Function

Returns a string containing the date and time at which `guile-pg` was configured (this is probably the build time.)

pg-trace *conn port* Function

Starts outputting low-level trace information on the connection *conn* to *port*, which must have been opened for writing. This trace is more useful for debugging Postgres than it is for debugging applications. The return value is unspecified.

Example

This example defines a pair of procedures `pg-trace-on` and `pg-trace-off` which implement a higher-level trace function which opens the required file and starts/stops the trace.

```
(define (pg-trace-on conn filename)
  (let ((port (open-file filename "w")))
       (pg-set-client-data! conn port)
       (pg-trace conn port)))

(define (pg-trace-off conn)
  (let ((port (pg-get-client-data conn)))
    (if port
        (begin
           (close-port port)
           (pg-untrace conn))
        #f)))
```

pg-untrace *conn* Function

Stops tracing on a given connection. The return value is unspecified.

3.3 Functions for Retrieving Data

pg-exec *conn statement* Function

Executes the SQL string *statement* on a given connection (*conn*,) returning either a `PG_RESULT` object containing a `pg-result-status` or `#f` if an error occurred, in which case the error message can be obtained using `pg-error-message`, passing it the `PG_CONN` object on which the statement was attempted. Note that the error message is available only until the next call to `pg-exec` on this connection.

Example

This example defines a function `pg-execute` which wraps `pg-exec` so that a `misc-error` is thrown instead of returning false. There are numerous other examples of `pg-exec` calls throughout this chapter.

```
(define (pg-execute conn sql)
  (let ((result (pg-exec conn sql)))
       (if result result
         (error (pg-error-message conn)))))
```

Notes

The entire result set is returned at once from a call to to `pg-exec`. If a `SELECT` results in a very large number of tuples then this can be a problem because it requires a large amount of memory. In these cases it is better to `DECLARE` a cursor over the `SELECT` and retrive small numbers of rows at a time using `FETCH`. These commands can only be issued between `BEGIN TRANSACTION/END TRANSACTION` pairs. See the PostgreSQL `declare (l)` and `fetch (l)` man pages for more details.

pg-error-message *conn* Function

Returns the most-recent error message that occurred on this connection, or an empty string if the previous `pg-exec` succeeded.

pg-result-status *result* Function

Returns the integer status of a `PG_RESULT` object returned by `pg-exec`. One of:

`PGRES_TUPLES_OK`

The statement returned zero or more tuples. The number of fields and tuples returned can be determined from calls to `pg-nfields` and `pg-ntuples` respectively. The value of a given attribute (field) of a given tuple can be determined by calling `pg-getvalue`.

`PGRES_COMMAND_OK`

The statement was a command (`INSERT`, `UPDATE`, `DELETE CREATE TABLE`, `DROP TABLE` etc.), which was executed without error. The number of tuples affected by the command can be determined by a call to `pg-cmdtuples`.

`PGRES_EMPTY_QUERY`
: It is not known under which circumstances this result-status is returned.

`PGRES_COPY_OUT`
: The statement was a `COPY <table> TO stdout`. The data can be read using `pg-getline`.

`PGRES_COPY_IN`
: The statement was a `COPY <table> FROM stdin`. The rows should be written using `pg-putline`.

`PGRES_BAD_RESPONSE`
: This occurs when the `libpq` interface receives an unexpected response from the backend. It indicates a problem with Postgres.

`PGRES_NONFATAL_ERROR`
: It is not known under which circumstances this result-status is returned.

`PGRES_FATAL_ERROR`
: The command was not executable for some reason. This is the returned status when a syntax error is detected in the command, for example.

Example

This contrived example defines a procedure to insert records into a given table, returning the number of tuples inserted (always one) or `#f` if an error occurred.

```
(define (pg-insert conn table record)
  (let ((result (pg-exec conn
                 (string-append "INSERT INTO "
                                table
                                " VALUES "
                                record))))
       (if (and result
                (eq? (pg-result-status result)
                     PGRES_COMMAND_OK))
           (string->number (pg-cmdtuples result))
           #f)))
```

The procedure could be called as follows

```
(pg-insert conn "people" "('Warbucks', 'Oliver')")
```

pg-get-connection *result* Function

Returns the `PG_CONN` object representing the connection from which a *result* was returned.

pg-binary-tuples? *result* Function

Returns `#t` if *result* contains binary tuple data, `#f` otherwise.

pg-fmod *result field-number* Function

Returns the integer type-specific modification data for the given field (*field-number*) of *result*.

pg-ntuples *result* Function

Returns the number of tuples in *result*.

pg-nfields *result* Function

Returns the number of fields in *result*.

pg-cmdtuples *result* Function

Returns the number of tuples affected by a command. This is a string which is empty in the case of commands like `CREATE TABLE`, `GRANT`, `REVOKE` etc. which don't affect tuples.

pg-oid-status *result* Function

Returns a string which contains the integer OID (greater than or equal to 0) of the tuple inserted, or is empty if the command to which *result* pertains was not `INSERT`.

NOTE: This function is deprecated. Use `pg-oid-value` instead.

pg-oid-value *result* Function

If the *result* is that of an SQL `INSERT` command, this function returns the integer OID of the inserted tuple, otherwise it returns `#f`.

pg-fname *result field-number* Function

Returns a string containing the canonical lower-case name of the field *field-number* in *result*. SQL variables and field names are not case-sensitive.

pg-fnumber *result field-name* Function

Returns the integer field-number corresponding to *field-name* if this exists in *result*, or `-1` otherwise.

pg-getvalue *result tuple field-number* Function

Returns a string containing the value of the attribute *field-number*, tuple *tuple* of *result*. It is up to the caller to convert this to the required type.

This example defines a procedure `pg-gettuple` which returns a given tuple as an alist, keyed on the field name. It's not an especially efficient procedure because it constructs the list of field-names afresh each time it's called.

```
(define (pg-fnames result)
  (let ((nfields (pg-nfields result)))
    (define (iter i)
      (if (= i nfields) '()
          (cons (pg-fname result i) (iter (+ i 1)))))
    (iter 0)))

(define (pg-getvalues result tuple)
  (let ((nfields (pg-nfields result)))
    (define (iter i)
```

```
          (if (= i nfields) '()
              (cons (pg-getvalue result tuple i) (iter (+ i 1)))))
        (iter 0)))

  (define (pg-gettuple result tuple)
    (map (lambda (n v) (cons (string->symbol n) v))
         (pg-fnames result)
         (pg-getvalues result tuple)))
```

Field values can be extracted from the tuple using `assq-ref`, eg:

```
(define tup (pg-gettuple result 0))
(assq-ref tup 'firstname)
```

Using this procedure we can define a version of `for-each` which iterates through the tuples returned from a given `SELECT` query:

```
(define (pg-for-each proc result)
  (let ((ntuples (pg-ntuples result)))
    (define (iter i)
      (cond ((= i ntuples) ntuples)
            (else (proc (pg-gettuple result i))
                  (iter (+ i 1)))))
    (iter 0)))
```

This implementation of `pg-for-each` inherits inefficiency from the `pg-gettuple` procedure.

Notes

The `pg-getvalue` procedure throws `out-of-range` errors if either the `tuple` or `field-number` arguments are out of range.

pg-getisnull *result tuple field-number* — Function

Returns `#t` if the attribute is `NULL`, `#f` otherwise.

pg-ftype *result field-number* — Function

Returns the PostgreSQL internal integer representation of the type of the given attribute. The integer is actually an OID (object ID) which can be used as the primary key to reference a tuple from the system table `pg_type`. A `misc-error` is thrown if the `field-number` is not valid for the given `result`.

Example

This defines a procedure `pg-ftype-name` which returns the type name of a given attribute in a string.

A more efficient implementation would be to define a type of connection which, when opened, issued the query to retrieve the `pg_type` system table once. An alist for looking up type names could then be associated with each connection using `pg-set-client-data!`.

```
(define (pg-ftype-name result fnum)
  (let ((result (pg-exec
                (pg-get-connection result)
                (string-append
                  "SELECT typname FROM pg_type WHERE oid = "
                  (number->string
                     (pg-ftype result fnum))))))
       (if (and
             result
             (eq? (pg-result-status result) PGRES_TUPLES_OK)
             (> (pg-ntuples result) 0))
           (pg-getvalue result 0 0)
           "")))
```

pg-fsize *result field-number* Function

Returns the size of a given attribute in bytes, or -1 if the field is variable-length.

pg-getlength *result tuple field-number* Function

The size of the datum in bytes.

3.4 Functions for Copying Data

pg-getline *conn* Function

Reads a line from a connection on which a `COPY <table> TO STDOUT` has been issued. Returns a string from the connection. If the returned string consists of only a backslash followed by a full stop, then the results from the `COPY` command have all been read and `pg-endcopy` should be called to resynchronise the connection before any further calls to `pg-exec` on this connection.

Example

This example defines a procedure `pg-copy-to-port` which can be used to retrieve the results of a `COPY <table> TO STDOUT` command.

```
(define (pg-copy-to-port conn port)
  (define (iter)
    (let ((line (pg-getline conn)))
         (if (string=? line "\\.")
             (pg-endcopy conn)
             (begin (write-line line port)
                    (iter)))))
  (iter))
```

Notes

It is an error to call `pg-getline` on a connection without first executing a `COPY <table> TO STDOUT` command on that connection. It is also an error to call `pg-getline` after a terminating line has been received, without an intervening `COPY` command being issued on that connection.

pg-putline *conn string* Function

Writes a line to the connection on which a COPY <table> FROM STDIN has been issued. The lines written should include the final newline characters.

The last line should be a backslash, followed by a full-stop. After this, the pg-endcopy procedure should be called for this connection before any further pg-exec call is made. The return value is undefined.

Example

This example defines a procedure pg-copy-from-port which can be used to supply data to a COPY <table> TO STDIN command.

```
(define (pg-copy-from-port conn port)
  (define (iter)
    (let ((line (read-line port)))
         (if (eof-object? line)
             (begin (pg-putline conn "\\.\n")
                    (pg-endcopy conn))
             (begin (pg-putline conn line)
                    (pg-putline conn "\n")
                    (if (string=? line "\\.")
                        (pg-endcopy conn)
                        (iter))))))
  (iter))
```

pg-endcopy *conn* Function

Resynchronises with the backend process. This procedure must be called after the last line of a table has been transferred using pg-getline or Returns an integer: zero if successful, non-zero otherwise.

3.5 Functions for Managing Large Objects

Note

All of the operations on large objects must be carried out inside a BEGIN TRANSACTION/END TRANSACTION pair. Failure to do this will result in a loss of synchronisation between the backend and the libpq library, resulting in an unusable connection to the database, and possible corruption of data.

pg-lo-creat *conn modes* Function

Creates a new large object and opens a port over it for reading and/or writing. *modes* is a string describing the mode in which the port is to be opened. The mode string must include one of “r” for reading, “w” for writing or “a” for append (but since the object is empty to start with this is the same as “w”.) The return value is a large object port which can be used to read or write data to/from the object, or #f on failure in which case pg-error-message from the connection should give some idea of what happened.

In addition to returning `#f` on failure this procedure throws a `misc-error` if the `modes` string is invalid.

pg-lo-open *conn oid modes* Function

Opens a port over an existing large object. The port can be used to read or write data from/to the object. *oid* should be an integer identifier representing the large object. *modes* must be a string describing the mode in which the port is to be opened. The mode string must include one of `r` for reading, `w` for writing, `a` for append or `+` with any of the above indicating both reading and writing/appending. `A` is equivalent to opening the port for writing and immediately doing a `(pg-lo-seek)` to the end. The return value is either an open large object port or `#f` on failure in which case `pg-error-message` from the connection should give some idea of what happened.

Notes

This procedure throws a `misc-error` exception if the `modes` string is invalid.

pg-lo-unlink *conn oid* Function

Deletes the large object identified by *oid*. Returns `#t` if the object was successfully deleted, `#f` otherwise, in which case `pg-error-message` applied to `conn` should give an idea of what went wrong.

pg-lo-get-connection *port* Function

Returns the connection associated with a given large object port. *Port* must be a large object port returned from `pg-lo-creat` or `pg-lo-open`.

pg-lo-get-oid *port* Function

Returns the integer identifier of the object to which a given port applies. *Port* must be a large object port returned from `pg-lo-creat` or `pg-lo-open`.

pg-lo-tell *port* Function

Returns the position of the file pointer for the given large object port. *Port* must be a large object port returned from `pg-lo-creat` or `pg-lo-open`. The return value is either an integer greater than or equal to zero or `#f` if an error occurred. In the latter case `pg-error-message` applied to `conn` should explain what went wrong.

pg-lo-seek *port where whence* Function

Sets the position of the next read or write to/from the given large object port. *Port* must be a large object port returned from `pg-lo-creat` or `pg-lo-open`. *Where* is the position to set the pointer. *Whence* must be one of

`SEEK_SET` Relative to the beginning of the file.

`SEEK_CUR` Relative to the current position.

`SEEK_END` Relative to the end of the file.

The return value is an integer which is the new position relative to the beginning of the object, or a number less than zero if an error occurred.

Notes

It is possible to seek beyond the end of file opened only for reading. In which case subsequent reads of the port will return an EOF object.

pg-lo-read *size number port* Function

Reads *number* objects each of length *size* from *port*. Returns a string containing the data read from the port or `#f` if an error occurred.

pg-lo-import *conn filename* Function

Creates a new large object and loads it with the contents of the specified file. *Filename* must be a string containing the name of the file to be loaded into the new object. The function returns the integer identifier (OID) of the newly created large object, or `#f` if an error occurred, in which case `pg-error-message` should be consulted to determine the failure.

pg-lo-export *conn oid filename* Function

Write the contents of a given large object to a file. *Oid* is the integer identifying the large object to be exported and *filename* the name of the file to contain the object data. The function returns `#t` on success, `#f` otherwise, in which case `pg-error-message` may offer an explanation of the failure.

Concept Index

Function Index

P

Table of Contents