PostgreSQL Tcl Interface Documentation

The PostgreSQL Global Development Group and the Tcl Interface Group

PostgreSQL Tcl Interface Documentation

by The PostgreSQL Global Development Group and the Tcl Interface Group

Table of Contents

List of Tables

List of Examples

Chapter 1. pgtcl - Tcl Binding Library

pgtcl is a Tcl package for client programs to interface with PostgreSQL servers. It makes most of the functionality of libpq available to Tcl scripts.

1.1. Overview

Table 1-1 gives an overview over the commands available in pgtcl. These commands are described further on subsequent pages.

Table 1-1. pgtcl Commands

Command	Description
pg_connect	open a connection to the server
pg_disconnect	close a connection to the server
pg_conndefaults	get connection options and their defaults
pg_exec	send a command to the server
pg_result	get information about a command result
pg_select	loop over the result of a query
pg_execute	send a query and optionally loop over the results
pg_quote	escape a string for inclusion into SQL statements
pg_listen	set or change a callback for asynchronous notification messages
pg_on_connection_loss	set or change a callback for unexpected connection loss
pg_sendquery	issue pg_exec-style command asynchronously
pg_getresult	check on results from asynchronously issued commands
pg_isbusy	check to see if the connection is busy processing a query
pg_blocking	set a database connection to be either blocking or nonblocking
pg_cancelrequest	request PostgreSQL abandon processing of the current command
pg_lo_creat	create a large object
pg_lo_open	open a large object
pg_lo_close	close a large object
pg_lo_read	read from a large object
pg_lo_write	write to a large object
pg_lo_lseek	seek to a position in a large object
pg_lo_tell	return the current seek position of a large object

Command	Description
pg_lo_unlink	delete a large object
pg_lo_import	import a large object from a file
pg_lo_export	export a large object to a file

The pg_lo_* commands are interfaces to the large object features of PostgreSQL. The functions are designed to mimic the analogous file system functions in the standard Unix file system interface. The pg_lo_* commands should be used within a BEGIN/COMMIT transaction block because the descriptor returned by pg_lo_open is only valid for the current transaction. pg_lo_import and pg_lo_export *must* be used in a BEGIN/COMMIT transaction block.

1.2. Loading pgtcl into an Application

Before using pgtcl commands, you must load the libpgtcl library into your Tcl application. This is normally done with the package require command. Here is an example:

```
package require Pgtcl 1.4
```

package require loads the libpgtcl shared library, and loads any dditional Tcl code that is part of the Pgtcl package.

The old way to load the shared library is by using the Tcl load command. Here is an example:

```
load libpgtcl[info sharedlibextension]
```

Although this way of loading the shared library is deprecated, we continue to document it for the time being, because it may help in debugging if, for some reason, package require is failing. The use of info sharedlibextension is recommended in preference to hard-wiring .so or .sl or .dll into the program.

The load command will fail unless the system's dynamic loader knows where to look for the libpgtcl shared library file. You may need to work with ldconfig, or set the environment variable LD_LIBRARY_PATH, or use some equivalent facility for your platform to make it work. Refer to the PostgreSQL installation instructions for more information.

libpgtcl in turn depends on the interface library libpq, so the dynamic loader must also be able to find the libpq shared library. In practice this is seldom an issue, since both of these shared libraries are normally stored in the same directory, but it can be a stumbling block in some configurations.

If you use a custom executable for your application, you might choose to statically bind libpgtcl into the executable and thereby avoid the load command and the potential problems of dynamic linking. See the source code for pgtclsh for an example.

1.3. pgtcl Command Reference

pg_connect

Name

`pg_connect` — open a connection to the server

Synopsis

```
pg_connect -conninfo connectOptions
pg_connect dbName ?-host hostName? ?-port portNumber? ?-tty tty? ?-options serverOption
```

Description

`pg_connect` opens a connection to the PostgreSQL server.

Two syntaxes are available. In the older one, each possible option has a separate option switch in the `pg_connect` command. In the newer form, a single option string is supplied that can contain multiple option values. `pg_conndefaults` can be used to retrieve information about the available options in the newer syntax.

Arguments

New style

`connectOptions`

A string of connection options, each written in the form `keyword = value`. A list of valid options can be found in the description of the libpq function `PQconnectdb`.

Old style

`dbName`

The name of the database to connect to.

`-host` *`hostName`*

The host name of the database server to connect to.

`-port` *`portNumber`*

The TCP port number of the database server to connect to.

`-tty` *`tty`*

A file or TTY for optional debug output from the server.

`-options` *`serverOptions`*

Additional configuration options to pass to the server.

Return Value

If successful, a handle for a database connection is returned. Handles start with the prefix `pgsql`.

pg_disconnect

Name

`pg_disconnect` — close a connection to the server

Synopsis

```
pg_disconnect conn
```

Description

`pg_disconnect` closes a connection to the PostgreSQL server.

Arguments

`conn`

The handle of the connection to be closed.

Return Value

None

pg_conndefaults

Name

`pg_conndefaults` — get connection options and their defaults

Synopsis

```
pg_conndefaults
```

Description

`pg_conndefaults` returns information about the connection options available in `pg_connect -conninfo` and the current default value for each option.

Arguments

None

Return Value

The result is a list describing the possible connection options and their current default values. Each entry in the list is a sublist of the format:

```
{optname label dispchar dispsize value}
```

where the *`optname`* is usable as an option in `pg_connect -conninfo`.

pg_exec

Name

`pg_exec` — send a command to the server

Synopsis

```
pg_exec conn commandString
```

Description

`pg_exec` submits a command to the PostgreSQL server and returns a result. Command result handles start with the connection handle and add a period and a result number.

Note that lack of a Tcl error is not proof that the command succeeded! An error message returned by the server will be processed as a command result with failure status, not by generating a Tcl error in `pg_exec`.

Arguments

`conn`

The handle of the connection on which to execute the command.

`commandString`

The SQL command to execute.

Return Value

A result handle. A Tcl error will be returned if pgtcl was unable to obtain a server response. Otherwise, a command result object is created and a handle for it is returned. This handle can be passed to `pg_result` to obtain the results of the command.

pg_result

Name

`pg_result` — get information about a command result

Synopsis

```
pg_result resultHandle resultOption
```

Description

`pg_result` returns information about a command result created by a prior `pg_exec`.

You can keep a command result around for as long as you need it, but when you are done with it, be sure to free it by executing `pg_result -clear`. Otherwise, you have a memory leak, and pgtcl will eventually start complaining that you have created too many command result objects.

Arguments

`resultHandle`

The handle of the command result.

`resultOption`

One of the following options, specifying which piece of result information to return:

`-status`

The status of the result.

`-error`

The error message, if the status indicates an error, otherwise an empty string.

`-conn`

The connection that produced the result.

`-oid`

If the command was an `INSERT`, the OID of the inserted row, otherwise 0.

`-numTuples`

The number of rows (tuples) returned by the query.

`-cmdTuples`

The number of rows (tuples) affected by the command.

`-numAttrs`

The number of columns (attributes) in each row.

`-assign` *`arrayName`*

Assign the results to an array, using subscripts of the form `(rowNumber, columnName)`.

`-assignbyidx` *`arrayName ?appendstr?`*

Assign the results to an array using the values of the first column and the names of the remaining column as keys. If *`appendstr`* is given then it is appended to each key. In short, all but the first column of each row are stored into the array, using subscripts of the form `(firstColumnValue, columnNameAppendStr)`.

`-getTuple` *`rowNumber`*

Returns the columns of the indicated row in a list. Row numbers start at zero.

`-tupleArray` *`rowNumber arrayName`*

Stores the columns of the row in array *`arrayName`*, indexed by column names. Row numbers start at zero.

`-attributes`

Returns a list of the names of the columns in the result.

`-lAttributes`

Returns a list of sublists, `{name typeOid typeSize}` for each column.

`-clear`

Clear the command result object.

Return Value

The result depends on the selected option, as described above.

pg_select

Name

pg_select — loop over the result of a query

Synopsis

```
pg_select conn commandString arrayVar procedure
```

Description

pg_select submits a query (SELECT statement) to the PostgreSQL server and executes a given chunk of code for each row in the result. The *commandString* must be a SELECT statement; anything else returns an error. The *arrayVar* variable is an array name used in the loop. For each row, *arrayVar* is filled in with the row values, using the column names as the array indices. Then the *procedure* is executed.

In addition to the column values, the following special entries are made in the array:

.headers

A list of the column names returned by the query.

.numcols

The number of columns returned by the query.

.tupno

The current row number, starting at zero and incrementing for each iteration of the loop body.

Arguments

conn

The handle of the connection on which to execute the query.

commandString

The SQL query to execute.

arrayVar

An array variable for returned rows.

procedure

The procedure to run for each returned row.

Return Value

None

Examples

This examples assumes that the table `table1` has columns `control` and `name` (and perhaps others):

```
pg_select $pgconn "SELECT * FROM table1;" array {
    puts [format "%5d %s" $array(control) $array(name)]
}
```

pg_execute

Name

pg_execute — send a query and optionally loop over the results

Synopsis

```
pg_execute ?-array arrayVar? ?-oid oidVar? conn commandString ?procedure?
```

Description

pg_execute submits a command to the PostgreSQL server.

If the command is not a SELECT statement, the number of rows affected by the command is returned. If the command is an INSERT statement and a single row is inserted, the OID of the inserted row is stored in the variable *oidVar* if the optional *-oid* argument is supplied.

If the command is a SELECT statement, then, for each row in the result, the row values are stored in the *arrayVar* variable, if supplied, using the column names as the array indices, else in variables named by the column names, and then the optional *procedure* is executed if supplied. (Omitting the *procedure* probably makes sense only if the query will return a single row.) The number of rows selected is returned.

The *procedure* can use the Tcl commands break, continue, and return with the expected behavior. Note that if the *procedure* executes return, then pg_execute does not return the number of affected rows.

pg_execute is a newer function which provides a superset of the features of pg_select and can replace pg_exec in many cases where access to the result handle is not needed.

For server-handled errors, pg_execute will throw a Tcl error and return a two-element list. The first element is an error code, such as PGRES_FATAL_ERROR, and the second element is the server error text. For more serious errors, such as failure to communicate with the server, pg_execute will throw a Tcl error and return just the error message text.

Arguments

-array *arrayVar*

Specifies the name of an array variable where result rows are stored, indexed by the column names. This is ignored if *commandString* is not a SELECT statement.

-oid *oidVar*

Specifies the name of a variable into which the OID from an INSERT statement will be stored.

conn

The handle of the connection on which to execute the command.

commandString

The SQL command to execute.

procedure

Optional procedure to execute for each result row of a `SELECT` statement.

Return Value

The number of rows affected or returned by the command.

Examples

In the following examples, error checking with `catch` has been omitted for clarity.

Insert a row and save the OID in `result_oid`:

```
pg_execute -oid result_oid $pgconn "INSERT INTO mytable VALUES (1);"
```

Print the columns `item` and `value` from each row:

```
pg_execute -array d $pgconn "SELECT item, value FROM mytable;" {
    puts "Item=$d(item) Value=$d(value)"
}
```

Find the maximum and minimum values and store them in `$s(max)` and `$s(min)`:

```
pg_execute -array s $pgconn "SELECT max(value) AS max, min(value) AS min FROM mytable;"
```

Find the maximum and minimum values and store them in `$max` and `$min`:

```
pg_execute $pgconn "SELECT max(value) AS max, min(value) AS min FROM mytable;"
```

pg_listen

Name

pg_listen — set or change a callback for asynchronous notification messages

Synopsis

```
pg_listen conn notifyName ?callbackCommand?
```

Description

pg_listen creates, changes, or cancels a request to listen for asynchronous notification messages from the PostgreSQL server. With a *callbackCommand* parameter, the request is established, or the command string of an already existing request is replaced. With no *callbackCommand* parameter, a prior request is canceled.

After a pg_listen request is established, the specified command string is executed whenever a notification message bearing the given name arrives from the server. This occurs when any PostgreSQL client application issues a NOTIFY command referencing that name. The command string is executed from the Tcl idle loop. That is the normal idle state of an application written with Tk. In non-Tk Tcl shells, you can execute update or vwait to cause the idle loop to be entered.

You should not invoke the SQL statements LISTEN or UNLISTEN directly when using pg_listen. pgtcl takes care of issuing those statements for you. But if you want to send a notification message yourself, invoke the SQL NOTIFY statement using pg_exec.

Arguments

conn

The handle of the connection on which to listen for notifications.

notifyName

The name of the notification condition to start or stop listening to.

callbackCommand

If present, provides the command string to execute when a matching notification arrives.

Return Value

None

pg_on_connection_loss

Name

`pg_on_connection_loss` — set or change a callback for unexpected connection loss

Synopsis

```
pg_on_connection_loss conn ?callbackCommand?
```

Description

`pg_on_connection_loss` creates, changes, or cancels a request to execute a callback command if an unexpected loss of connection to the database occurs. With a *`callbackCommand`* parameter, the request is established, or the command string of an already existing request is replaced. With no *`callbackCommand`* parameter, a prior request is canceled.

The callback command string is executed from the Tcl idle loop. That is the normal idle state of an application written with Tk. In non-Tk Tcl shells, you can execute `update` or `vwait` to cause the idle loop to be entered.

Arguments

`conn`

The handle to watch for connection losses.

`callbackCommand`

If present, provides the command string to execute when connection loss is detected.

Return Value

None

pg_sendquery

Name

`pg_sendquery` — send a query string to the backend connection without waiting for a result

Synopsis

```
pg_sendquery conn commandString
```

Description

`pg_sendquery` submits a command to the PostgreSQL server. This function works like `pg_exec`, except that it does not return a result. Rather, the command is issued to the backend asynchronously.

The result is either an error message or nothing. An empty return indicates that the command was dispatched to the backend.

Arguments

`conn`

The handle of the connection on which to execute the command.

`commandString`

The SQL command to execute.

Return Value

A Tcl error will be returned if pgtcl was unable to issue the command. Otherwise, an empty string will be return. It is up to the developer to use `pg_getresult` to obtain results from commands issued with `pg_sendquery`.

pg_isbusy

Name

`pg_isbusy` — see if a query is busy

Synopsis

```
pg_isbusy conn
```

Description

`pg_isbusy` checks to see if the backend is busy handling a query or not.

Arguments

`conn`

The handle of a connection to the database in which the large object exists.

Return Value

Returns `1` if the backend is busy, in which case a call to `pg_getresult` would block, otherwise it returns `0`.

pg_blocking

Name

`pg_blocking` — see or set whether or not a connection is set to blocking or nonblocking

Synopsis

```
pg_blocking conn ?mode?
```

Description

`pg_blocking` can set the connection to either blocking or nonblocking, and it can see which way the connection is currently set.

Arguments

`conn`

The handle of a connection to the database in which the large object exists.

`mode`

If present, sets the mode of the connection to `nonblocking` if `0`. Otherwise it sets the connection to `blocking`.

Return Value

Returns nothing if called with the *`mode`* argument. Otherwise it returns `1` if the connection is set for `blocking`, or `0` if the connection is set for `nonblocking`.

pg_cancelrequest

Name

`pg_cancelrequest` — request that PostgreSQL abandon processing of the current command

Synopsis

```
pg_cancelrequest conn
```

Description

`pg_cancelrequest` requests that the processing of the current command be abandoned.

Arguments

conn

The handle of a connection to the database in which the large object exists.

Return Value

Returns nothing if the command was successfully dispatched or if no query was being processed. Otherwise, returns an error.

pg_quote

Name

`pg_quote` — escapes a string for inclusion into SQL statements

Synopsis

```
pg_quote string
```

Description

`pg_quote` quotes a string and escapes single quotes and backslashes within the string, making it safe for inclusion into SQL statements.

If you're doing something like

```
pg_exec $conn "insert into foo values ('$name');"
```

and `name` contains text includeing an unescaped single quote, such as `Bob's House`, the insert will fail. Passing value strings through `pg_quote` make sure they can be used as values and stuff in Postgres.

```
pg_exec $conn "insert into foo values ([pg_quote $name]);"
```

...will make sure that any special characters that occur in name, such as single quote or backslash, will be properly quoted.

Arguments

`string`

The string to be escaped.

Return Value

Returns the string, escaped for inclusion into SQL queries. Note that it adds a set of single quotes around the outside of the string as well.

pg_lo_creat

Name

`pg_lo_creat` — create a large object

Synopsis

```
pg_lo_creat conn mode
```

Description

`pg_lo_creat` creates a large object.

Arguments

`conn`

The handle of a connection to the database in which to create the large object.

`mode`

The access mode for the large object. It can be any or'ing together of `INV_READ` and `INV_WRITE`. The "or" operator is `|`. For example:

```
[pg_lo_creat $conn "INV_READ|INV_WRITE"]
```

Return Value

The OID of the large object created.

pg_lo_open

Name

`pg_lo_open` — open a large object

Synopsis

```
pg_lo_open conn loid mode
```

Description

`pg_lo_open` opens a large object.

Arguments

`conn`

The handle of a connection to the database in which the large object exists.

`loid`

The OID of the large object.

`mode`

Specifies the access mode for the large object. Mode can be either `r`, `w`, or `rw`.

Return Value

A descriptor for use in later large-object commands.

pg_lo_close

Name

`pg_lo_close` — close a large object

Synopsis

```
pg_lo_close conn descriptor
```

Description

`pg_lo_close` closes a large object.

Arguments

`conn`

The handle of a connection to the database in which the large object exists.

`descriptor`

A descriptor for the large object from `pg_lo_open`.

Return Value

None

pg_lo_read

Name

`pg_lo_read` — read from a large object

Synopsis

```
pg_lo_read conn descriptor bufVar len
```

Description

`pg_lo_read` reads at most *`len`* bytes from a large object into a variable named *`bufVar`*.

Arguments

`conn`

The handle of a connection to the database in which the large object exists.

`descriptor`

A descriptor for the large object from `pg_lo_open`.

`bufVar`

The name of a buffer variable to contain the large object segment.

`len`

The maximum number of bytes to read.

Return Value

The number of bytes actually read is returned; this could be less than the number requested if the end of the large object is reached first. In event of an error, the return value is negative.

pg_lo_write

Name

`pg_lo_write` — write to a large object

Synopsis

```
pg_lo_write conn descriptor buf len
```

Description

`pg_lo_write` writes at most *`len`* bytes from a variable *`buf`* to a large object.

Arguments

`conn`

The handle of a connection to the database in which the large object exists.

`descriptor`

A descriptor for the large object from `pg_lo_open`.

`buf`

The string to write to the large object (not a variable name, but the value itself).

`len`

The maximum number of bytes to write. The number written will be the smaller of this value and the length of the string.

Return Value

The number of bytes actually written is returned; this will ordinarily be the same as the number requested. In event of an error, the return value is negative.

pg_lo_lseek

Name

`pg_lo_lseek` — seek to a position of a large object

Synopsis

```
pg_lo_lseek conn descriptor offset whence
```

Description

`pg_lo_lseek` moves the current read/write position to *`offset`* bytes from the position specified by *`whence`*.

Arguments

`conn`

The handle of a connection to the database in which the large object exists.

`descriptor`

A descriptor for the large object from `pg_lo_open`.

`offset`

The new seek position in bytes.

`whence`

Specified from where to calculate the new seek position: `SEEK_CUR` (from current position), `SEEK_END` (from end), or `SEEK_SET` (from start).

Return Value

None

pg_lo_tell

Name

`pg_lo_tell` — return the current seek position of a large object

Synopsis

```
pg_lo_tell conn descriptor
```

Description

`pg_lo_tell` returns the current read/write position in bytes from the beginning of the large object.

Arguments

`conn`

The handle of a connection to the database in which the large object exists.

`descriptor`

A descriptor for the large object from `pg_lo_open`.

Return Value

A zero-based offset in bytes suitable for input to `pg_lo_lseek`.

pg_lo_unlink

Name

`pg_lo_unlink` — delete a large object

Synopsis

```
pg_lo_unlink conn loid
```

Description

`pg_lo_unlink` deletes the specified large object.

Arguments

`conn`

The handle of a connection to the database in which the large object exists.

`loid`

The OID of the large object.

Return Value

None

pg_lo_import

Name

`pg_lo_import` — import a large object from a file

Synopsis

```
pg_lo_import conn filename
```

Description

`pg_lo_import` reads the specified file and places the contents into a new large object.

Arguments

`conn`

The handle of a connection to the database in which to create the large object.

`filename`

Specified the file from which to import the data.

Return Value

The OID of the large object created.

Notes

`pg_lo_import` must be called within a `BEGIN`/`COMMIT` transaction block.

pg_lo_export

Name

`pg_lo_export` — export a large object to a file

Synopsis

```
pg_lo_export conn loid filename
```

Description

`pg_lo_export` writes the specified large object into a file.

Arguments

`conn`

The handle of a connection to the database in which the large object exists.

`loid`

The OID of the large object.

`filename`

Specifies the file into which the data is to be exported.

Return Value

None

Notes

`pg_lo_export` must be called within a `BEGIN`/`COMMIT` transaction block.

1.4. Example Program

Example 1-1 shows a small example of how to use the pgtcl commands.

Example 1-1. pgtcl Example Program

```
# getDBs :
#   get the names of all the databases at a given host and port number
#   with the defaults being the localhost and port 5432
#   return them in alphabetical order
proc getDBs { {host "localhost"} {port "5432"} } {
    # datnames is the list to be result
    set conn [pg_connect template1 -host $host -port $port]
    set res [pg_exec $conn "SELECT datname FROM pg_database ORDER BY datname;"]
    set ntups [pg_result $res -numTuples]
    for {set i 0} {$i < $ntups} {incr i} {
 lappend datnames [pg_result $res -getTuple $i]
    }
    pg_result $res -clear
    pg_disconnect $conn
    return $datnames
}
```