

# **PostgreSQL Tcl Interface Documentation**

**The PostgreSQL Global Development Group**

**The Tcl Interface Group**

## **PostgreSQL Tcl Interface Documentation**

by The PostgreSQL Global Development Group, The Tcl Interface Group

# Table of Contents

<b>1. pgctl - Tcl Binding Library .....</b>	<b>1</b>
1.1. Overview .....	1
1.2. Loading pgctl into an Application .....	3
1.3. pgctl Command Reference .....	4
pg_connect.....	4
pg_dbinfo.....	8
pg_disconnect.....	10
pg_conndefaults.....	11
pg_exec.....	12
pg_exec_prepared.....	13
pg_result.....	15
pg_select.....	19
pg_execute.....	22
pg_listen.....	24
pg_on_connection_loss.....	25
pg_sendquery.....	26
pg_sendquery_prepared.....	27
pg_getresult.....	28
pg_isbusy.....	29
pg_blocking.....	30
pg_cancelrequest.....	31
pg_null_value_string.....	31
pg_quote.....	32
pg_escape_string.....	34
pg_escape_bytea.....	35
pg_unescape_bytea.....	36
pg_lo_creat.....	36
pg_lo_open.....	37
pg_lo_close.....	38
pg_lo_read.....	39
pg_lo_write.....	40
pg_lo_lseek.....	41
pg_lo_tell.....	42
pg_lo_truncate.....	43
pg_lo_unlink.....	43
pg_lo_import.....	44
pg_lo_export.....	45
pg_sqlite.....	46
pg_copy_complete.....	53
PgGetConnectionId.....	54
1.4. Tcl Namespace Support .....	55
1.5. Connection/result handles as commands .....	55
1.6. Example Program.....	56

# List of Tables

1-1. pgctl Commands.....	1
1-2. pgctl C API.....	3

# Chapter 1. pgctl - Tcl Binding Library

pgctl 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 pgctl. These commands are described further on subsequent pages.

**Table 1-1. pgctl Commands**

Command	Namespace Command	Description
pg_connect	pg::connect	open a connection to the server
pg_dbinfo	pg::dbinfo	returns data about the connection
pg_disconnect	pg::disconnect	close a connection to the server
pg_conndefaults	pg::conndefaults	get connection options and their defaults
pg_exec	pg::sqlexec	send a command to the server
pg_exec_prepared	pg::exec_prepared	send a request to execute a prepared statement, with parameters
pg_result	pg::result	get information about a command result
pg_select	pg::select	loop over the result of a query
pg_execute	pg::execute	send a query and optionally loop over the results
pg_null_value_string	pg::null_value_string	set string to be returned for null values in query results
pg_quote	pg::quote	escape a string for inclusion into SQL statements
pg_escape_string	pg::escape_string	escape a binary string for inclusion into SQL statements
pg_escape_bytea	pg::escape_bytea	escape a binary string for inclusion into SQL statements
pg_unescape_bytea	pg::unescape_bytea	unescape a binary string from the backend
pg_listen	pg::listen	set or change a callback for asynchronous notification messages

Command	Namespace Command	Description
pg_on_connection_loss	pg::on_connection_loss	set or change a callback for unexpected connection loss
pg_sendquery	pg::sendquery	issue pg_exec-style command asynchronously
pg_sendquery_prepared	pg::sendquery_prepared	send an asynchronous request to execute a prepared statement, with parameters
pg_getresult	pg::getresult	check on results from asynchronously issued commands
pg_isbusy	pg::isbusy	check to see if the connection is busy processing a query
pg_blocking	pg::blocking	set a database connection to be either blocking or nonblocking
pg_cancelrequest	pg::cancelrequest	request PostgreSQL abandon processing of the current command
pg_lo_creat	pg::lo_creat	create a large object
pg_lo_open	pg::lo_open	open a large object
pg_lo_close	pg::lo_close	close a large object
pg_lo_read	pg::lo_read	read from a large object
pg_lo_write	pg::lo_write	write to a large object
pg_lo_lseek	pg::lo_lseek	seek to a position in a large object
pg_lo_tell	pg::lo_tell	return the current seek position of a large object
pg_lo_truncate	pg::lo_truncate	Truncate (or pad) a large object to a specified length
pg_lo_unlink	pg::lo_unlink	delete a large object
pg_lo_import	pg::lo_import	import a large object from a file
pg_lo_export	pg::lo_export	export a large object to a file
pg_sqlite	pg::sqlite	bridge between pgctl and the Tcl sqlite package (when compiled with sqlite bridge)
pg_copy_complete	pg::copy_complete	Complete <b>COPY FROM stdin</b> operation after finished writing

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.

The `pg_sqlite` command is only included if Sqlite 3 is installed. It can be disabled at compile time with `./configure --without-sqlite3`.

**Table 1-2. pgctl C API**

Function	Description
<code>PGconn *PgGetConnectionId(interp, id, connid)</code>	Given a Tcl handle to an open Pgctl connection, return the underlying libpq connection

## 1.2. Loading pgctl into an Application

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

```
package require Pgctl 1.5
```

`package require` loads the `libpgctl` shared library, and loads any additional Tcl code that is part of the `Pgctl` package. Note that you can manually generate the `pkgIndex.tcl` file, or use **make pkgIndex.tcl** or **make pkgIndex.tcl-hand** to have make generate it.

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

```
load libpgctl[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 `libpgctl` 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.

`libpgctl` 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 `libpgctl` into the executable and thereby avoid the `load` command and the potential problems of dynamic linking. See the source code for `pgctlsh` for an example.

If you want to use the `pg_sqlite` bridge, you must still explicitly `"package require sqlite3"` before executing any `sqlite3` commands.

## 1.3. pgctl Command Reference

### pg\_connect

#### Name

`pg_connect` — open a connection to the server

#### Synopsis

```
pg_connect -conninfo connectOptions [-connhandle connectionHandleName] [-async bool]  
pg_connect dbName [-host hostName] [-port portNumber] [-tty tty] [-options serverOptions] [-  
pg_connect -connlist connectNameValueList [-connhandle connectionHandleName] [-async bool]
```

#### Description

`pg_connect` opens a connection to the PostgreSQL server.

Three 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. The third form takes the parameters as a name value Tcl list. `pg_conndefaults` can be used to retrieve information about the available options in the newer syntax.

#### Arguments

##### New style

*connectOptions*

`pg_connect` opens a new database connection using the parameters taken from the `connectOptions` string. Unlike the old-style usage of `pg_connect`, with the new-style usage the parameter set can be extended without requiring changes to either `libpgctl` or the underlying `libpq` library, so use of the new style (or its nonexistent nonblocking analogues `pg_connect_start` and `pg_connect_poll`) is preferred for new application programming.

The passed string can be empty to use all default parameters, or it can contain one or more parameter settings separated by whitespace. Each parameter setting is in the form `keyword =`



value. (To write an empty value or a value containing spaces, surround it with single quotes, e.g., `keyword = 'a value'`. Single quotes and backslashes within the value must be escaped with a backslash, i.e., `\'` and `\\`.) Spaces around the equal sign are optional.

The currently recognized parameter key words are:

`host`

Name of host to connect to. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored. The default is to connect to a Unix-domain socket in `/tmp`.

`hostaddr`

Numeric IP address of host to connect to. This should be in the standard IPv4 address format, e.g., `172.28.40.9`. If your machine supports IPv6, you can also use IPv6 address format, e.g., `fe80::203:93ff:fedb:49bc`. TCP/IP communication is always used when a nonempty string is specified for this parameter.

Using `hostaddr` instead of `host` allows the application to avoid a host name lookup, which may be important in applications with time constraints. However, Kerberos authentication requires the host name. The following therefore applies: If `host` is specified without `hostaddr`, a host name lookup occurs. If `hostaddr` is specified without `host`, the value for `hostaddr` gives the remote address. When Kerberos is used, a reverse name query occurs to obtain the host name for Kerberos. If both `host` and `hostaddr` are specified, the value for `hostaddr` gives the remote address; the value for `host` is ignored, unless Kerberos is used, in which case that value is used for Kerberos authentication. (Note that authentication is likely to fail if `libpq` is passed a host name that is not the name of the machine at `hostaddr`.) Also, `host` rather than `hostaddr` is used to identify the connection in `$HOME/.pgpass`.

Without either a host name or host address, Pgctl will connect using a local Unix domain socket.

`port`

Port number to connect to at the server host, or socket file name extension for Unix-domain connections.

`dbname`

The database name. Defaults to be the same as the user name.

`user`

PostgreSQL user name to connect as.

`password`

Password to be used if the server demands password authentication.

`connect_timeout`

Maximum wait for connection, in seconds (write as a decimal integer string). Zero or not specified means wait indefinitely. It is not recommended to use a timeout of less than 2 seconds.

`options`

Command-line options to be sent to the server.

`tty`

Ignored (formerly, this specified where to send server debug output).

`sslmode`

This option determines whether or with what priority an SSL connection will be negotiated with the server. There are four modes: `disable` will attempt only an unencrypted SSL connection; `allow` will negotiate, trying first a non-SSL connection, then if that fails, trying an SSL connection; `prefer` (the default) will negotiate, trying first an SSL connection, then if that fails, trying a regular non-SSL connection; `require` will try only an SSL connection.

If PostgreSQL is compiled without SSL support, using option `require` will cause an error, and options `allow` and `prefer` will be tolerated but libpq will be unable to negotiate an SSL connection.

`requiressl`

This option is deprecated in favor of the `sslmode` setting.

If set to 1, an SSL connection to the server is required (this is equivalent to `sslmode require`). libpq will then refuse to connect if the server does not accept an SSL connection. If set to 0 (default), libpq will negotiate the connection type with the server (equivalent to `sslmode prefer`). This option is only available if PostgreSQL is compiled with SSL support.

`service`

Service name to use for additional parameters. It specifies a service name in `pg_service.conf` that holds additional connection parameters. This allows applications to specify only a service name so connection parameters can be centrally maintained. See `PREFIX/share/pg_service.conf.sample` for information on how to set up the file.

`-connhandle connectionHandleName`

Name to use for the connection handle, instead of pgctl generating the name automatically. Without the option, the name is auto-generated, prefixed with `pgsql`, and with a numeric id at the end. This gives the programmer control over the name of the connection handle.

`-async bool`

Connect asynchronously if [bool] is true.

If any parameter is unspecified, then the corresponding environment variable (see `libpq` documentation in the PostgreSQL manual) is checked. If the environment variable is not set either, then built-in defaults are used.

## 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.

`-connhandle connectionHandleName`

Name to use for the connection handle, instead of `pgctl` generating the name automatically. Without the option, the name is auto-generated, prefixed with `pgsql`, and with a numeric id at the end. This gives the programmer control over the name of the connection handle.

`-async bool`

Connect asynchronously if `[bool]` is true.

## Third style (most recent one added)

`-connlist connectNameValuelist`

`pg_connect` opens a new database connection using the parameters taken from the `connectNameValuelist` list. The parameters are exactly the same for the New Style, but they are stored as a Tcl list, instead of a string. The list is a name value pair, for example: **[list host localhost port 5400 dbname template1]**.

```
array set conninfo {
    host      192.168.123.180
    port      5801
    dbname    template1
    user      postgres
}
set conn [pg::connect -connlist [array get ::conninfo]]
```

`-async bool`

Connect asynchronously if [bool] is true.

`-connhandle connectionHandleName`

Name to use for the connection handle, instead of pgtcl generating the name automatically. Without the option, the name is auto-generated, prefixed with `pgsql`, and with a numeric id at the end. This gives the programmer control over the name of the connection handle.

## Return Value

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

# pg\_dbinfo

## Name

`pg_dbinfo` — returns data about the connection

## Synopsis

`pg_dbinfo command ?conn? ?paramname?`

## Description

`pg_dbinfo` returns data about the connection. The first argument is a command, and the second and third argument depend on the command chosen.

## Arguments

*command*

*connections*

Return a list of connection handles.

*results connHandle*

Return a list of result handles for the named connection.

*version connHandle*

Return server version for the connection.

*protocol connHandle*

Return protocol version for the connection.

*param connHandle name*

Return connection's value for the named parameter.

*backendpid connHandle*

Return server process ID for the connection.

*socket connHandle*

Return socket file handle for the connection.

*sql\_count connHandle*

Return number of SQL queries that have been made for the connection.

*dbname connHandle*

Return name of the connected database.

*user connHandle*

Return logged in user name.

*password connHandle*

Return logged in user's password.

*host connHandle*

Return address of the connected host.

*port connHandle*

Return host port for the connection.

*options connHandle*

Return command line options passed in the connection request.

*status connHandle*

Return connection status.

*transaction\_status connHandle*

Return transaction status.

*error\_message connHandle*

Return the most recent error message on the connection.

*needs\_password connHandle*

Return true if the connection required a password but none was available.

This function can be applied after a failed connection attempt to decide whether to prompt the user for a password.

*used\_password connHandle*

Return true if the connection used a password.

This function can be applied after either a failed or successful connection attempt to detect whether the server demanded a password.

*used\_ssl connHandle*

Return true if the connection uses SSL.

*conn*

The handle of the connection, when required.

*param*

The connection parameter name, when the command "param" is provided.

## Return Value

A Tcl list of connection handle names

## 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 [-paramarray arrayVar] [-variables] conn commandString [args]
```

### 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`. Check for `{[pg_result $result -status] == PGRES_COMMAND_OK}`.

If the `[-paramarray]` flag is provided, then a substitution is performed on the query, securely replacing each back-quote delimited name with the corresponding entry from the named array. If the array does not contain the named element, then NULL is substituted (similarly to the way an array created by `-withoutnulls` is generated). Each such name must occur in a location where a value or field name could appear. See `pg_select` for more info.



If the [-variables] flag is provided, then embedded Tcl variables are passed as parameters. Variables can be flagged with ':' and array references and namespaces are supported. Complex variable names can usually be embedded with :{nasty-name-here} though no attempt at duplicating Tcl brace-escaping is made. If the variable does not exist, then NULL is substituted.

## Arguments

[-paramarray arrayname]

Perform parameter substitution via 'quoted' elements of the array using PQexecParams or PQSendQueryParams.

[-variables]

Substitute Tcl variables found in the SQL string using PQexecParams or PQSendQueryParams.

*conn*

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

*commandString*

The SQL command to execute.

*args*

For PostgreSQL versions greater than 7.4, *args* consists of zero or more optional values that can be inserted, unquoted, into the SQL statement using \$-style substitution. Nulls are represented by the string "NULL".

## Return Value

A result handle. A Tcl error will be returned if pgctl 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.

## Example

```
pg_exec $conn {select * from table1 where id = $1 and user = $2} $id $user
```

## pg\_exec\_prepared

### Name

`pg_exec_prepared` — send a request to execute a prepared SQL statement to the server

### Synopsis

```
pg_exec_prepared conn statementName [args]
```

### Description

`pg_exec_prepared` submits a command to the PostgreSQL server and returns a result.

`pg_exec_prepared` functions identically to `pg_exec`, except that it operates using statements prepared by the **PREPARE** SQL command.

Note that prepared statements are only support under PostgreSQL 7.4 and later.

### Arguments

*conn*

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

*statementName*

The name of the prepared statement to execute.

*args*

*args* consists of zero or more optional values that can be inserted, unquoted, into the SQL statement using `$`-style substitution.

### Return Value

A result handle. See `pg_exec` for details.

## Example

```
pg_exec $conn {prepare insert_people
    (varchar, varchar, varchar, varchar, varchar, varchar)
    as insert into people values ($1, $2, $3, $4, $5, $6);}

pg_exec_prepared $conn insert_people $email $name $address $city $state $zip
```

## 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 `pgctl` 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` [*diagCode*]

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

*diagCode*, if specified, requests data for a specific diagnostic code:

*severity*

The severity; the field contents are `ERROR`, `FATAL`, or `PANIC`, in an error message, or `WARNING`, `NOTICE`, `DEBUG`, `INFO`, or `LOG`, in a notice message, or a localized translation of one of these.

*sqlstate*

The `SQLSTATE` code for the error. (See PostgreSQL manual Appendix A).

*primary*

The primary human-readable error message (typically one line).

*detail*

An optional secondary error message carrying more detail about the problem, which may run to multiple lines.

*hint*

An optional suggestion about what to do about the problem. This is intended to differ from detail in that it offers advice (potentially inappropriate) rather than hard facts.

The result may run to multiple lines.

*position*

A string containing a decimal integer indicating an error cursor position as an index into the original statement string.

The first character has index 1, and positions are measured in characters not bytes.

*internal\_position*

This is the same as "position", but it is used when the cursor position refers to an internally generated command rather than the one submitted by the client.

The first character has index 1, and positions are measured in characters not bytes.

`internal_query`

This is the text of a failed internally generated command. This could be, for example, a SQL query issued by a PL/pgSQL function.

`context`

An indication of the context in which the error occurred. Presently this includes a call stack traceback of active PL functions. The trace is one entry per line, most recent first.

`file`

The filename of the source code location where the error was reported.

`line`

The line number of the source code location where the error was reported.

`function`

The name of the source code function reporting the error.

`-foreach arrayName tclCode`

Iterates through each row of the result, filling `arrayName` with the columns and their values and executing `tclCode` for each row in turn. Null columns will be not be present in the array.

`-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. (This is similar to `-numTuples` but relevant to **INSERT** and **UPDATE** commands.)

`-numAttrs`

The number of columns (attributes) in each row.

`-assign arrayName`

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

`-foreach arrayName code`

For each resulting row assigns the results to the named array, using subscripts matching the column names, then executes the code body.

`-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 `(firstColumnNameValue, 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. If a field's value is null, sets an empty string or the default string, if a default string has been defined.

`-tupleArrayWithoutNulls rowNumber arrayName`

Stores the columns of the row in array `arrayName`, indexed by column names. Row numbers start at zero. If a field's value is null, unsets the column from the array.

`-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.

`-list`

Returns one list containing all the data returned by the query.

`-lList`

Returns a list of lists, where each embedded list represents a tuple in the result.

`-dict`

Returns a dict object with the results. This needs to have dictionary support built into Tcl (Tcl 8.5), and is experimental right now, since Tcl 8.5 has not been release yet, and the API could change. In order to enable this, you need to add `-DHAVE_TCL_NEWDICTOBJ` to the Makefile in the `DEFS` variable.

`-null_value_string [string]`

Defines or retrieves the string that will be returned for null values in query results. Defaults to whatever was set by `pg_null_value_string` but can be set here and, in this case, affects only this query result.

`-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 [-rowbyrow] [-nodotfields] [-withoutnulls] [-paramarray var] [-variables] [-params pa`

### 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 (unless the `[-nodotfields]` flag is provided):

`.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.

If the [-param] flag is provided, then it contains a list of parameters that will replace "\$1", "\$2" and so on in the query string, as if it were a prepared statement. Be sure to properly escape or quote the "\$" in the query. :)

If the [-paramarray] flag is provided, then a substitution is performed on the query, securely replacing each back-quote delimited name with the corresponding entry from the named array. If the array does not contain the named element, then NULL is substituted (similarly to the way an array created by -withoutnulls is generated). Each such name must occur in a location where a value or field name could appear.

If the [-variables] flag is provided, then embedded Tcl variables are passed as parameters. Variables can be flagged with ':' and array references and namespaces are supported. Complex variable names can usually be embedded with :{nasty-name-here} though no attempt at duplicating Tcl brace-escaping is made. If the variable does not exist, then NULL is substituted.

Notes: This substitution is performed by generating a positional parameter list and calling PQExecParams with a modified query containing \$1, \$2, ... where the original 'names' appeared. This is a straight substitution, so if this mechanism is used the back-quote character (`) can not appear elsewhere in the query, even in a quoted string. There are a maximum of 99,999 names.

## Arguments

[-params list]

Perform parameter substitution using PQExecParams or PQSendQueryParams.

[-paramarray arrayname]

Perform parameter substitution via 'quoted' elements of the array using PQExecParams or PQSendQueryParams.

[-variables]

Substitute Tcl variables found in the SQL string using PQExecParams or PQSendQueryParams.

[-rowbyrow]

Perform the select in row-by-row mode. This means that the code block is called immediately results become available, rather than waiting for the query to complete.

[-nodotfields]

Suppress generation of the pseudo-fields .headers, .numcols, and .tupno.



`[-withoutnulls]`

If specified null columns will be unset from the array rather than being defined and containing the null string, typically an empty string.

`[-count countVar]`

Set the variable "countVar" to the number of tuples returned for use in the block.

*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

Number of rows actually processed.

## 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)]  
}
```

This example demonstrates how to use named parameters to securely perform queries on an SQL database:

```
# An array imported from some hive of scum and villainy like a web form.  
set form(first) {Andrew'}; DROP TABLE students;--}  
set form(last) {Randall}  
  
# Secure extraction of data  
pg_select -paramarray form $pgconn "SELECT * from students WHERE firstname = 'first' AND  
    lappend candidates $row(student_id) $row(firstname) $row(lastname) $row(age)  
}
```

## 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`. `pgctl` 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 [-paramarray [-variables] arrayVar] conn commandString [args]
```

## 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.

If the `[-paramarray]` flag is provided, then a substitution is performed on the query, securely replacing each back-quote delimited name with the corresponding entry from the named array. If the array does not contain the named element, then NULL is substituted (similarly to the way an array created by `-withoutnulls` is generated). Each such name must occur in a location where a value or field name could appear. See `pg_select` for more info.

If the `[-variables]` flag is provided, then embedded Tcl variables are passed as parameters. Variables can be flagged with `'.'` and array references and namespaces are supported. Complex variable names can usually be embedded with `:{nasty-name-here}` though no attempt at duplicating Tcl brace-escaping is made. If the variable does not exist, then NULL is substituted. Each such name must occur in a location where a value or field name could appear. See `pg_select` for more info.

## Arguments

`[-paramarray arrayname]`

Perform parameter substitution via ‘quoted’ elements of the array using `PQexecParams` or `PQSendQueryParams`.

`[-variables]`

Substitute Tcl variables found in the SQL string using `PQexecParams` or `PQSendQueryParams`.

*conn*

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

*commandString*

The SQL command to execute.

*args*

For PostgreSQL versions greater than 7.4, *args* consists of zero or more optional values that can be inserted, unquoted, into the SQL statement using `$`-style substitution. Nulls are represented by the string "NULL".

## Return Value

A Tcl error will be returned if `pgctl` 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\_sendquery\_prepared

### Name

`pg_sendquery_prepared` — send a request to execute a prepared statement to the backend connection, without waiting for a result

## Synopsis

```
pg_sendquery_prepared conn statementName [args]
```

## Description

`pg_sendquery_prepared` 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.

*statementName*

The name of the prepared SQL statement to execute asynchronously.

*args*

*args* consists of zero or more optional values that can be inserted, unquoted, into the SQL statement using `$`-style substitution.

## Return Value

A Tcl error will be returned if `pgctl` 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\_getresult

### Name

`pg_getresult` — process asynchronous results



## Synopsis

`pg_getresult conn`

## Description

`pg_getresult` checks to see if any commands issued by `pg_sendquery` have completed.

This will return the same sort of result handle that `pg_exec` returns.

If there is no query currently being processed or all of the results have been obtained, `pg_getresult` returns nothing.

## Arguments

*conn*

The handle of a connection to the database to which asynchronous requests are being issued.

## Return Value

If a query result is available, a command result object is returned. This handle can be passed to `pg_result` to obtain the results of the command.

If there is no query currently being processed or all of the results have been obtained, `pg_getresult` returns nothing.

# 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\_null\_value\_string

### Name

`pg_null_value_string` — define a value to be returned for NULL fields distinct from the default value of an empty string.

### Synopsis

`pg_null_value_string` [*string*]

### Description

`pg_null_value_string` sets or retrieves a string to be returned in query results for fields whose value is NULL, making it possible to distinguish between NULL values and values that are not null but are comprised of an empty string. Without setting an alternative null value with this or with `pg_result`'s `-null_value_string`, it is impossible to tell the difference between a NULL field value and one that is not null but empty.

### Arguments

*string*

The string to be set that will be returned for null fields.

### Return Value

Returns the string that is currently being returned for null fields. It will be the passed string value if one was passed, or, otherwise, the value currently being used.

## pg\_quote

### Name

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

## Synopsis

```
pg_quote [-null] [connection] 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 a *connection* is provided, the connection is used to customize the quoting process for the database referenced by the connection.

If the `[-null]` option is provided, then if the text matches the null string (either the empty string, or the null string specified in the *connection*) then the SQL keyword NULL is returned, rather than a quoted string.

If you're doing something like

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

and *name* contains text including an unescaped single quote, such as Bob's House, at best the insert will fail, and at worst your software will be exploited via an SQL injection attack. Passing value strings through `pg_quote` will properly quote them for insertion into SQL commands.

```
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.

## See Also

In most cases, with recent versions of SQL, it is better to use the native parameter insertion capabilities of the SQL server and protocol. If you are using a version of PostgreSQL more recent than 7.4, consider the optional parameter arguments to `pg_exec` and `pg_sendquery`, and the `paramarray` option to `pg_exec`, `pg_sendquery`, and `pg_select`.

## pg\_escape\_string

### Name

`pg_escape_string` — escapes a string for inclusion into SQL statements. This is the same as `pg_quote`. It was added for consistency.

### Synopsis

```
pg_escape_string string
```

### Description

`pg_escape_string` 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 including an unescaped single quote, such as `Bob's House`, at best the insert will fail, and at worst your software will be exploited via an SQL injection attack. Passing value strings through `pg_escape_string` will properly quote them for insertion into SQL commands.

```
pg_exec $conn "insert into foo values ([pg_escape_string $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.

## See Also

In most cases, with recent versions of SQL, it is better to use the native parameter insertion capabilities of the SQL server and protocol. If you are using a version of PostgreSQL more recent than 7.4, consider the optional parameter arguments to `pg_exec` and `pg_sendquery`, and the `paramarray` option to `pg_exec`, `pg_sendquery`, and `pg_select`.

# pg\_escape\_bytea

## Name

`pg_escape_bytea` — escapes a binary string for inclusion into SQL statements.

## Synopsis

`pg_escape_bytea string`

## Description

`pg_escape_bytea` escapes a binary string, making it safe for inclusion into SQL statements.

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

## Arguments

*binary\_string*

The binary string to be escaped.

## Return Value

Returns the binary string, escaped for inclusion into SQL queries.

# pg\_unescape\_bytea

## Name

`pg_unescape_bytea` — unescapes a binary string.

## Synopsis

`pg_unescape_bytea string`

## Description

`pg_unescape_bytea` unescapes a binary string, when retrieving from the backend.

## Arguments

*binary\_string*

The string to be unescaped.

## Return Value

Returns the binary string.



## 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\_truncate

## Name

`pg_lo_truncate` — Truncate a large object to a given length

## Synopsis

```
pg_lo_truncate conn descriptor length
```

## Description

`pg_lo_truncate` truncates the specified large object to the given length. If the length is greater than the current large object length, the large object is extended with null bytes.

## 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`.

*length*

The length to which the large object is to be truncated or padded.

## 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.

# pg\_sqlite

## Name

`pg_sqlite` — implements a bridge between PostgreSQL and Sqlite3 using the Pgtcl and sqlite3 packages.

## Synopsis

`pg_sqlite sqlite_db command [args]`

## Description

`pg_sqlite` can import the results of a Postgres SQL query (previously made via `pg_exec`) directly into an `sqlite3` table, export the results of an `Sqlite` SQL query into a Postgres table (via `write_tabsep`), or import TSV files directly into `sqlite3`.

The commands currently implemented are `import_postgres_result`, `read_tabsep`, `write_tabsep`, and `read_tabsep_keylist`.

## Arguments

*sqlite\_db*

An `Sqlite3` database handle previously created via the `sqlite3` command.

*command*

The command, one of `info`, `import_postgres_result`, `read_tabsep`, `write_tabsep`, or `read_tabsep_keylist`,

*args*

Command-specific arguments.

## Commands

### info

```
pg_sqlite sqlite_db info
[-busy]
[-filename]
[-db database_name]
```

Request information from an `SQLITE` database connection. Returns a key-value list containing the values requested if available. With no arguments, all available info is returned.

`-filename`

Request the filename associated with a database.

`-db database`

For the `-filename` option, specify the database name (default "main").

**-busy**

Request a list of prepared commands that are currently busy.

### **import\_postgres\_result**

```
pg_sqlite sqlite_db import_postgres_result handle
[-rowbyrow]
[-sql target_sql]
[-create new_table]
[-into table]
[-replace]
[-as name_type_list]
[-types type_list]
[-names name_list]
[-pkey primary_key]
[-sep separator]
[-null null_string]
[-poll_interval rows]
[-recommit rows]
[-check]
[-max column-name variable-name]
```

Import the result of a PostgreSQL request into an sqlite3 table.

*handle*

A database or result handle. Normally, you would call `pg_exec` and pass the result handle to `pg_sqlite` here. If you are using `row_by_row` mode, though, you use `pg_sendquery` and pass the PostgreSQL database handle here.

**-rowbyrow**

Perform the request using row-by-row mode. This injects the data to sqlite directly without making an internal copy, but the result may be incomplete or inconsistent if an error occurs during the request.

**-sep** *separator*

String to use to separate columns. Default is "\t" (tab).

**-null** *null\_string*

String to use to indicate a null value. Default is to treat all strings literally.

**-sql** *target\_sql*

An INSERT statement, suitable to be compiled into a prepared statement to be applied to each row. For example "INSERT INTO newtable (id, name, value) VALUES (?, ?, ?)"

`-create new_table`

A table to be created in sqlite. The names and types of columns, and the primary key, must be provided.

`-into table`

An existing sqlite table to insert the data. The names of the columns must be provided, if the type can not be inferred it will be assumed to be "text".

`-replace`

When performing the insert on an existing or newly create table, use "INSERT OR REPLACE" semantics. Not compatible with "-sql".

`-as name-type-list`

A list of alternating column names and types. See note on types.

`-types type-list`

A list of column types. See note on types.

`-names name-list`

A list of column names.

`-pkey primary_key`

A list containing key names and optional sorting to indicate primary key where needed. For example **-pkey {{clock ASC} {sequence ASC}}**.

`-poll_interval count`

Call DoOneEvent() every *count* rows to keep the event loop alive during long transactions.

`-recommit count`

Chunk the operation in transactions, with one transaction every *count* rows.

`-check`

Check (via a SELECT) whether the exact row is already present, and skip inserting the row if so. This avoids bloating the WAL log during large re-loads of databases, at some performance cost.

`-max column-name variable-name`

Set \$*variable-name* to the maximum value of *column-name* imported.

## **write\_tabsep**

```
pg_sqlite sqlite_db write_tabsep handle sql
[sql]
[-sep separator]
[-null null_string]
[-poll_interval rows]
```

Write the results of the provided *sql* to a file handle

This command may be used to export sqlite3 data to postgres. You issue a **COPY FROM STDIN ... FORMAT text** command, then use **pg\_sqlite write\_tabsep ...** to write the data directly to the Postgresql handle, followed by writing the terminator line to the same handle. See the PostgreSQL documentation on the "COPY" command for more details.

*handle*

File handle.

**-sep** *separator*

String to use to separate columns. Default is "\t" (tab).

**-null** *null\_string*

String to use to indicate a null value. Default is to treat all strings literally.

**-poll\_interval** *count*

Call DoOneEvent() every *count* rows to keep the event loop alive during long transactions.

### read\_tabsep

```
pg_sqlite sqlite_db read_tabsep
[-row single_tab_separated_row]
[-file file_handle]
[-sql target_sql]
[-create new_table]
[-into table]
[-replace]
[-as name_type_list]
[-types type_list]
[-names name_list]
[-pkey primary_key]
[-sep separator]
[-null null_string]
[-poll_interval rows]
[-recommit rows]
[-check]
```

Read a previously opened file into an sqlite3 table.

`-row single_tab_separated_row`

An already read tab-separated line.

`-file file_handle`

An open file consistion of tab-separated rows.

`-sep separator`

String to use to separate columns. Default is "\t" (tab).

`-null null_string`

String to use to indicate a null value. Default is to treat all strings literally.

`-sql target_sql`

An INSERT statement, suitable to be compiled into a prepared statement to be applied to each row. For example "INSERT INTO newtable (id, name, value) VALUES (?, ?, ?)"

`-create new_table`

A table to be created in sqlite. The names and types of columns, and the primary key, must be provided.

`-into table`

An existing sqlite table to insert the data. The names of the columns must be provided, if the type can not be inferred it will be assumed to be "text".

`-replace`

When performing the insert on an existing or newly create table, use "INSERT OR REPLACE" semantics. Not compatible with "-sql".

`-as name-type-list`

A list of alternating column names and types. See note on types.

`-types type-list`

A list of column types. See note on types.

`-names name-list`

A list of column names.

`-pkey primary_key`

A list containing key names and optional sorting to indicate primary key where needed. For example **-pkey {{clock ASC} {sequence ASC}}**.

`-poll_interval count`

Call DoOneEvent() every *count* rows to keep the event loop alive during long transactions.

**-recommit** *count*

Chunk the operation in transactions, with one transaction every *count* rows.

**-check**

Check (via a SELECT) whether the exact row is already present, and skip inserting the row if so. This avoids bloating the WAL log during large re-loads of databases, at some performance cost.

### **read\_tabsep\_keylist**

```
pg_sqlite sqlite_db read_tabsep_keylist
[-row single_tab_separated_row]
[-file file_handle]
[-create new_table]
[-into table]
[-replace]
[-as name_type_list]
[-names name_list]
[-pkey primary_key]
[-sep separator]
[-null null_string]
[-poll_interval rows]
[-recommit rows]
```

Read a previously opened file containing alternating key-value columns into an sqlite3 table.

**-row** *single\_tab\_separated\_row*

An already read tab-separated key-value list line.

**-file** *file\_handle*

An open file consistion of tab-separated key-value list rows.

**-sep** *separator*

String to use to separate columns. Default is "\t" (tab).

**-null** *null\_string*

String to use to indicate a null value. Default is to treat all strings literally.

**-create** *new\_table*

A table to be created in sqlite. The names and types of columns, and the primary key, must be provided.

**-into** *table*

An existing sqlite table to insert the data. The names of the columns must be provided, if the type can not be inferred it will be assumed to be "text".



`-replace`

When performing the insert on an existing or newly create table, use "INSERT OR REPLACE" semantics. Not compatible with "-sql".

`-as name-type-list`

A list of alternating column names and types. See note on types.

`-names name-list`

A list of column names.

`-pkey primary_key`

A list containing key names and optional sorting to indicate primary key where needed. For example **-pkey {{clock ASC} {sequence ASC}}**.

`-poll_interval count`

Call DoOneEvent() every *count* rows to keep the event loop alive during long transactions.

`-recommit count`

Chunk the operation in transactions, with one transaction every *count* rows.

## Types

*A note on types:* `pg_sqlite` supports four types: `integer` (or `int`), `boolean` (or `bool`), `double` (or `real`), and `text`. `Integer`, `double`, and `text` match both `sqlite` and `postgresql` types. `Boolean` is an integer type in `sqlite`, and `boolean` in `postgresql`, and converts `postgresql` boolean values (such as 'yes', 'no', 'true', or 'false') to integer 1 and 0.

In addition, `pg_sqlite` will accept PostgreSQL boolean values for the integer type. This may be made conditional on a "strict" mode in the future.

## Return Value

Number of rows imported or exported.

## See Also

Sqlite 3

## pg\_copy\_complete

### Name

`pg_compy_complete` — Completes a **COPY FROM stdin** operation

### Synopsis

```
pg_copy_complete conn
```

### Description

`pg_copy_complete` completes a **COPY FROM stdin** operation. After writing the rows to the postgres connection handle, this tells postgres that the copy is completed and it can return to normal operation.

### Arguments

*conn*

An postgresql connection handle.

## PgGetConnectionId

### Name

`PgGetConnectionId` — Provides access to the underlying `libpq` SQL connection handle.

### Synopsis

```
extern PGconn *PgGetConnectionId(Tcl_Interp *interp, const char *handle, Pg_ConnectionId
```

### Description

`PgGetConnectionId` returns the underlying `PGconn` handle. This is intended to allow Tcl extensions to Pgctl to use the existing database connection.

The `Pg_ConnectionId` structure provides access to more internals of the Pgtcl handle, but may be ignored in most cases.

## Parameters

`Tcl_Interp *interp`

Pointer to Tcl interpreter.

`char *handle`

The name of the Pgtcl database handle command.

`Pg_ConnectionId **connid`

Pointer to address to hold Pgtcl connection handle. Must be provided even if not used.

## Returns

`PGconn *conn` is a pointer to the libpq SQL database connection.

## 1.4. Tcl Namespace Support

With version 1.5, there is proper Tcl namespace support built into pgtcl. There are commands now that mirror the `pg_` commands, but use the Tcl namespace convention. For example, there are commands now called: `pg::connect`, `pg::result`, etc. However, due to this, there are some incapacibilities. For example, `pg_exec` has a counterpart called `pg::sqlexec`, since doing a namespace import `::pg::*` would clobber the builtin Tcl command `exec`. The old command names, `pg_*`, are still there for backwards compatibility, but might be phased out eventually.

So, one can use Tcl's namespace mechanisms now with pgtcl. For example, you can import that namespace:

```
namespace import ::pg::*
```

```
set conn [connect template1 -host $host -port $port]
```

## 1.5. Connection/result handles as commands

Starting with version 1.5, you can use the connection/result handle as a Tcl command. What this means is that when a handle for a connection or result is generated, a corresponding Tcl command is also generate. For example, you can do the following:

```
set conn [pg::connect template1 -host $host -port $port]
set res [$conn exec "SELECT datname FROM pg_database ORDER BY datname;"]
set datnames [$res -list]
$res -clear
rename $conn {} ;# or $conn disconnect
```

Note that deleting the command (**rename \$conn {}**), has the same effect as **pg::result \$res -clear** (if it is a result handle), and **pg::disconnect** (if it is a connection handle). Also, if that command gets overloaded with a proc definition, then that has the same effect as deleting the command.

## 1.6. Example Program

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

### Example 1-1. pgctl 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
}

## OR an alternative

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 datnames [pg_result $res -list]
    pg_result $res -clear
    pg_disconnect $conn
}
```

```
        return $datnames
    }

    ## OR an alternative

    proc getDBs { {host "localhost"} {port "5432"} } {
        # datnames is the list to be result
        set conn [pg_connect template1 -host $host -port $port]
        set res [$conn exec "SELECT datname FROM pg_database ORDER BY datname;"]

        set datnames [$res -dict]
        $res -clear
        rename $conn {}
        return [dict get $datnames]
    }
```