

# cbl2pg manual

**Info:** See <<http://www.pegueroles.com/cbl2pg>>  
**Author:** Ferran Pegueroles <[ferran@pegueroles.com](mailto:ferran@pegueroles.com)>  
**Revision:** manual.txt 12 2009-07-14 16:07:59Z ferran  
**Description:** A cbl2pg manual, explains motivations and usage

## Introduction

cbl2pg is a library to connect COBOL programs with a PostgreSQL database.

It has been tested with AcuCOBOL on Windows and with TinyCOBOL on windows and Linux but it can work with any other COBOL program that can call a C library.

This library is now very mature, and has been used in production environments for many years.

## Motivation

I have developed cbl2pg because at that moment I needed to access a PostgreSQL database from a COBOL program and the only 2 options I had were to go with a commercial and expensive solution accessing the database via ODBC or the ecpg precompiler.

The problem with the commercial version was the price and that you cannot execute arbitrary SQL commands. The problem with the ecpg was that it generated a COBOL code that was not compatible with the compilers that I was using and that I cannot define dynamic SQL.

All these reasons make me develop this solution.

## How it works

cbl2pg works as a gateway between the C PostgreSQL library and your COBOL program. It adapts the data from COBOL to C and from C to COBOL.

The most difficult work is to adapt the data row from the C structure to the COBOL record, converting each SQL field to a field in a COBOL record.

Table 1 shows these conversions :

SQL Type	COBOL Picture
small integer	S9(4) SIGN TRAILING SEPARATE
integer	S9(9) SIGN TRAILING SEPARATE
big integer	S9(18) SIGN TRAILING SEPARATE
char(N)	X(N)
varchar(N)	X(N)
numeric(N,M)	S9(P)V(M) SIGN TRAILING SEPARATE (P is N - M)
date	9(8) format DDMMYYYY
time	9(6) format HHMMSS
timestamp	x(20)

By now only this types are suported but if needed more can be added.

The COBOL program can generate a SQL statement in a PICTURE X field and then pass this field to a route tahat executes the query and returns a pointer that is used to access the data.

The data is returned as a COBOL record with the fields formates as shown in [Table 1](#). Usually you know the format anb define a COBOL record to receive the data. If you don't know the format you can access the returned data field by field with the format.

## Using cbl2pg

### Initializing

Befor using any function of cbl2pg you need to load the library. In AcuCOBOL you have to do:

```
CALL 'cbl2pg.dll'.
```

or:

```
CALL '\path\to\cbl2pg.dll'.
```

In TinyCOBOL:

```
CALL 'libcbl2pg.so'
```

Nou you can access all the cbl2pg routines.

### Connect to the database

To connect to the database you have to call the sql\_connect routine. This routine has 2 parameters, the connection handler and the connection strng. The connection handler is a pointer to the database connection, in COBOL it can be a USAGE POINTER field or a PIC X field with a size bigger than the size of a pointer. (A PIC X(10) should always work).

The connection string contains the variables to connect to the database. These variables are the same used to connect to the database from any other program.

This variables are defines like this:

```
01 CONNECT-SQL          PIC X(200).
01 DBHANDLER-SQL       USAGE POINTER.
```

You call the sql\_connect like this

```
MOVE "user=john password=secret host=somehost dbname=foo "
    TO CONNECT-SQL.
CALL 'sql_connect' USING DBHANDLER-SQL CONNECT-SQL
```

If the returned dhhandler-sql is zero, the connection is not possible. Is is not zero the connection has been established and you can work with the database.

It is possible to connect from a program to more than one database or connect various times to the same database (for working inside and outside a transaction). To do this, you only need to work with diferent dbhandler-sql variables.

### Execute a query

If you have a handler to a database you can make queries. To make a query to the database you need to put the query on a PIC X field (terminated by a null values x'00') and then call sql\_query.

The parameters of sql\_query are the dbhandler, the query handler, that is another USAGE POINTER variable thet is used as a pointer to the query resultset, a SQLCA structure to hold the response meta-data and the query string terminated by a null value. The query string can be arbitrary but needs to be terminarted by a null value (the C string terminator). This can be done by using the STRING command to string a X'00' at the end of the string or you can define a structure like this:

```

01 QUERY-SQL.
02 SQL-SQL          PIC X(1000).
02 FILLER           PIC X VALUE X'00'

```

The [SQLCA](#) is a structure that holds the number of records read, and the possible error status and error messages. The [SQLCA](#) structure is explained below.

A call to `sql_query` is like this:

```

MOVE SPACES TO SQL-SQL.

STRING 'SELECT * FROM FOO' X'00'
      DELIMITED BY SIZE INTO SQL-SQL.

CALL 'sql_query' USING DBHANDLER-SQL, QHANDLER-SQL,
                    SQLCA, SQL-SQL

```

Now you can use the `QHANDLER` to retrieve the values from the resultset (if there is any).

## Retrieve the result

You can know if the query has returned any values by examining [SQLCODE](#) in [SQLCA](#). If the value of [SQLCODE](#) is zero, then there is a result of the query and this result can be obtained using one of the functions in [Table 2](#)

Function	Description
<a href="#">sql_get_line</a>	Get one row from the result
<a href="#">sql_get_item</a>	Get one field from the result by position
<a href="#">sql_get_item_by_name</a>	Get one field from the result by name
<a href="#">sql_get_info</a>	Get information from the result

### sql\_get\_line

This is the main routine to get values from the result. You call this routine with the resultset handler, the row number to retrieve and the record to get the data in. The routine fills the record with the data from the database following the structure from [Table 1](#).

The row

You call the routine from COBOL as:

```
CALL 'sql_get_line' USING QHANDLER-SQL, LIN-SQL, RECORD-DATA.
```

`QHANDLER-SQL` is the query handler of a select that returned some data. `LIN-SQL` is an integer type (`PIC INTEGER` or `PIC 9 COMP-X`) with the name of the line to retrieve indexed by zero (first line is line 0).

### sql\_get\_item

With this function we can retrieve a value from the result set, not an entire row but a single column of a row. The column is returned with the data from the database following the structure from [Table 1](#).

You call the routine from COBOL as

```
CALL 'sql_get_item' USING QHANDLER-SQL, LIN-SQL, COL-SQL, FIELD-DATA.
```

The parameters are the same as in [sql\\_get\\_line](#), adding the `COL-SQL` that is the index beginning from zero of the column to return.

### sql\_get\_item\_by\_name

This routine works like [sql\\_get\\_item](#) but retrieves data by column name instead of by column position. You call the routine from COBOL as

```
CALL 'sql_get_item_by_name' USING QHANDLER-SQL,LIN-SQL,COLNAME-SQL,FIELD-DATA.
```

The COLNAME-SQL is a NULL terminated string that contains the name of the column to retrieve.

### sql\_get\_info

This function is used to retrieve information about the format of a field in the result set. The parameters are the same as [sql\\_get\\_item](#).

You call the routine from COBOL as

```
CALL 'sql_get_info' USING QHANDLER-SQL,LIN-SQL,COL-SQL,INFO-DATA.
```

## Update and insert shortcuts

To make easier the operations of insert and update to the database from a COBOL record we created to simple functions [sql\\_make\\_update](#) and [sql\\_make\\_insert](#)

This functions get the database table structure from a table in memory, create the SQL INSERT or UPDATE statement with the data from a COBOL record and sent it to the database.

The memory table that defines the format of the database is used to make is possible to the routine to interpret the data in the record and to get the field names.

This table can be created from the metadata in the database with the [sql\\_create\\_field\\_table](#) function or can be written by hand. The format of this table is shown in [Table Format](#)

[Table 1](#) shows the functions involved in this proces.

Function	Description
<a href="#">sql_make_update</a>	Helper to make an update of a record in the database
<a href="#">sql_make_insert</a>	Helper to insert a record in the database
<a href="#">sql_create_field_table</a>	Inspect the database to get the table of fields

### sql\_make\_update

This routine does a database update of a row from a COBOL record. The routine makes it very easy to uupdate a record, without the need to create a full SQL string with all the fields correctly edited.

All text fields are also escaped with [sql\\_escape](#) to prevent SQL injection errors.

You call the routine from COBOL as

```
CALL 'sql_make_update' USING DBHANDLER-SQL,SQLCA,DATA-RECORD,TABLE-FORMAT .
```

The parameters of the call are the database connection, the [SQLCA](#) structure like in [sql\\_query](#), the data record to insert and the table format as described in [Table format](#).

### sql\_make\_insert

This function is used to make an update of a row in the database from a COBOL record.

All the columns of the row are updated except the rows marked as primary keys in the table fotmat.

You call the routine from COBOL as

```
CALL 'sql_make_insert' USING DBHANDLER-SQL,SQLCA,DATA-RECORD,TABLE-FORMAT .
```

Like `sql_make_update` this function takes the database connection, the `SQLCA` structure , the data record to insert and the table format as described in [Table format](#).

### Table format specification

For `sql_make_update` and `sql_make_insert` to work we need a structure to describe the format of the table we are working on.

This structure is defined on a COBOL structure and can be used in all the calls to `sql_make_update` and `sql_make_insert`.

The table is structured with 27 char long items. The first item contains the table name. This is a limitation of the format because only supports 27 chars long table names.

The next items in the table describe table fields.

Position	Size	Description
1	20	Field name
21	3	Field total size in the cobol record, zero padded If the field is numeric it includes decimals and the sign.
24	2	Decimal positions. Only meaningfull with numeric fields
25	1	Primary key indicator. Indicates if field is a primeray key Possible values : <ul style="list-style-type: none"> <li>• K : Is a primary key</li> <li>• k : Is a primary key, but not used on insert. used for autonumeric primary keys.</li> <li>• blank : Not a primary key</li> </ul>
26	1	Field type indicator . Indicated the type of the database field Possible values : <ul style="list-style-type: none"> <li>• C : Text or char field</li> <li>• N : Numeric field</li> <li>• D : Date field (Format DDMMYYYY)</li> <li>• T : Time field (Format HHMMSS)</li> <li>• S : Timestamp field</li> <li>• blank : Field no used for insert or update. used for automatic like auto-timestamps or un-updatable fields</li> </ul>

This is an sample of this a table and the format to update it

```
CREATE TABLE product {
  id serial,
  created date,
  description char(128),
  price numeric(9,2),
}
```

```

01 llista-expedi.
  02 filler pic x(27) value 'product          ' .
  02 filler pic x(27) value 'id              01000kN'.
  02 filler pic x(27) value 'created         00800 D'.
  02 filler pic x(27) value 'description    12800 C'.
  02 filler pic x(27) value 'price         01002 N'.
  02 filler pic x(27) value x'00'.

```

There is a lot of work to write and maintain by hand the structure of all the fields of this table. This work can be reduced using [sql\\_create\\_field\\_table](#).

### sql\_create\_field\_table

With this function we can get all the information to fill the [Table format](#) from the database. We only need to define a data record with enough space to get all the data, put in the table name and call the function.

You call the routine from COBOL as

```
CALL 'sql_create_field_table' USING DBHANDLER-SQL, TABLE-FORMAT .
```

All the fields and field types are read from the database and the field table is ready to be used.

### Miscellaneous functions

Function	Description
<a href="#">sql_make_create</a>	Create a table from a field table description
<a href="#">sql_escape</a>	Escape a text string to be used in a SQL query
<a href="#">sql_exec_file</a>	Execute SQL from a text file.

#### sql\_make\_create

TODO

#### sql\_escape

TODO

#### sql\_exec\_file

TODO

### The SQLCA

TODO

The sqlca structure

```

*
*  SQLCA
*
01 sqlca.
  05 sqlcaid          pic x(8).
  05 sqlcabc          pic s9(9) comp-5.
  05 sqlcode          pic s9(9) comp-5.

```

```

05 sqlerrm.
    10 sqlerrml      pic s9(4) comp-5.
    10 sqlerrmc      pic x(70).
05 sqlerrp          pic x(8).
*
* 1. empty
* 2. empty
* 3. Num. rows processed (update, delete, inset)
* 4. Number of rows returned (select , fetch)
*
*
*
05 sqlerrd          occurs 6
                    pic s9(9) comp-5.

05 sqlwarn.
    10 sqlwarn0      pic x.
    10 sqlwarn1      pic x.
    10 sqlwarn2      pic x.
    10 sqlwarn3      pic x.
    10 sqlwarn4      pic x.
    10 sqlwarn5      pic x.
    10 sqlwarn6      pic x.
    10 sqlwarn7      pic x.
05 sqlext.
    10 sqlwarn8      pic x.
    10 sqlwarn9      pic x.
    10 sqlwarnA      pic x.
    10 sqlstate      pic x(5).

```

## The SQLCODE

TODO

## C routines reference

```

int sql_connect(PGconn **conn,char *db);
int sql_disconnect(PGconn **conn);

int sql_error_text(PGconn **conn,char *text);

int sql_query(PGconn **conn,PGresult **res,struct sqlca *ca,char *sql);
int sql_query_free(PGresult **res);

int sql_get_item(PGresult **res,int *lin,int *col,char *value);
int sql_get_item_by_name(PGresult **res,int *lin,char *col-
name,char *value);
int sql_get_line(PGresult **res,int *lin,char *value);
int sql_get_info(PGresult **res,int *lin,int *col,char *value);

int sql_make_update(PGconn **conn,struct sqlca *psqlca,char *data,struct for-
mat_camp *format);

```

```
int sql_make_insert(PGconn **conn,struct sqlca *psqlca,char *data,struct format_camp *format);
int sql_create_field_table(PGconn **conn,struct format_camp *format);

int sql_make_create(PGconn **conn,struct sqlca *psqlca,struct format_camp *format);

int sql_escape(char *sql, int len);
int sql_exec_file(PGconn **conn,struct sqlca *psqlca,char *filename);
```