Writing PostgreSQL Applications

Bruce Momjian

6th February 2002

POSTGRESQL is an object-relational database developed on the Internet by a group of developers spread across the globe. It is an open-source alternative to commercial databases like Oracle and Informix. This article discusses the concepts needed to write applications using POSTGRESQL. It covers the purpose of databases and how to communicate with them from applications.

POSTGRESQL was originally developed at the University of California at Berkeley. In 1996, a group began development of the database on the Internet. They use email to share ideas and file servers to share code. POSTGRESQL is now comparable to commercial databases in terms of features, performance, and reliability. It has transactions, views, stored procedures, and referential integrity constraints. It supports a large number of programming interfaces, including ODBC, Java (JDBC), TCL/TK, PHP, Perl, and Python. POSTGRESQL continues to improve at a tremendous pace thanks a talented pool of Internet developers.

Why Use a Database?

Database-backed applications use databases in very specific ways. They do all the input, processing, and display in the application. They use the database to store information that must be kept after the application exits and information that must be shared with other applications. In summary, the application does its own:

- Input
- Processing
- Display

and relies on the database for:

- Permanent storage
- Sharing information

For example, once an application commits a query, all users can see its changes. Also, when the application exits, all information remains stored in the database.

Of course, a database is not required to permanently store or share information. Information can be stored permanently in flat files and a shared memory area can be used to share information among applications. However, databases make this much easier, and have features like transactions, indexing, joins, aggregates, and a table structure that makes the job of the application programmer easier.

Accessing the Database

The following figure illustrates how applications communicate to the database:

Libpq is the POSTGRESQL C library that allows applications to communicate with the database. Using libpq is a fairly straightforward process:

- Call a libpq function to connect to the database
- Receive a connection handle *(PGconn)*
- Issue a query
  - Use the connection handle to issue a query
  - Receive a result handle *(PGresult)*
  - Access the result
- Issue more queries if desired
- Close the database connection

Examples

As an example, look at the following C application:

```
/*
 *  libpq sample program
 */

#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"                                   /* libpq header file */

int
main()
{
    char        state_code[3];                          /* holds user state code */
    char        query_string[256];                      /* holds constructed SQL query */
    PGconn     *conn;                                   /* holds database connection */
    PGresult   *res;                                    /* holds query result */
    int         i;

    conn = PQconnectdb("dbname=test");                  /* connect to the database */

    if (PQstatus(conn) == CONNECTION_BAD)               /* did the connection fail? */
    {
        fprintf(stderr, "Connection to database failed.\n");
        fprintf(stderr, "%s", PQerrorMessage(conn));
```

```
        exit(1);
    }

    printf("Enter a state code:  ");                    /* prompt user for a state code */
    scanf("%2s", state_code);

    sprintf(query_string,                               /* create an SQL query string */
            "SELECT name \
             FROM statename \
             WHERE code = '%s'", state_code);

    res = PQexec(conn, query_string);                   /* send the query */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)         /* did the query fail? */
    {
        fprintf(stderr, "SELECT query failed.\n");
        PQclear(res);
        PQfinish(conn);
        exit(1);
    }

    for (i = 0; i < PQntuples(res); i++)                /* loop through all rows returned */
        printf("%s\n", PQgetvalue(res, i, 0));          /* print the value returned */

    PQclear(res);                                       /* free result */

    PQfinish(conn);                                     /* disconnect from the database */

    return 0;
}
```

This example is described in the *Interfaces* chapter of my book, *PostgreSQL: Introduction and Concepts*. Please review the application details at http://www.postgresql.org/docs/awbook.html if you are unfamiliar with it.

In the above program, connection/disconnection to the database is processed by the lines in red, a query is issued and the result cleared in blue, and the result is accessed in green. The *conn* structure holds connection information, and *res* holds result information. As you can see from the colors, *conn* is used to obtain *res,* and *res* is used to access results. You can see the following TCL program follows the same pattern. It even uses *conn* and *res* in the same way:

```
#!/usr/local/pgsql/bin/pgtclsh
#
#   pgtclsh sample program
#

set conn [pg_connect -conninfo "dbname=test"]           ;# connect to the database

puts -nonewline "Enter a state code:  "                 ;# prompt user for a state code
flush stdout
gets stdin state_code

                                                        ;# send the query
set res [pg_exec $conn \
        "SELECT name \
         FROM statename \
         WHERE code = '$state_code'"]

set ntups [pg_result $res -numTuples]

for {set i 0} {$i < $ntups} {incr i} {                  ;# loop through all rows returned
        puts stdout [lindex [pg_result $res -getTuple $i] 0]    ;# print the value returned

}
pg_disconnect $conn                                     ;# disconnect from the database
```

Fancier configurations are possible. For example, you can create multiple connection handles by issuing multiple connection requests, even to different databases or as different users. You can also skip clearing results and use them later in your application. Of course, results are static. They represent the result at the time the query was executed. *LIbpq* even has an asynchronous set of library calls that allow multiple queries to be sent and retrieved simultaneously.

Finer Details

This section describes each step in more detail.

Connection

You will notice a string is used to specify the database in the connection function call. Connection parameters include:

- host
- hostaddr (host address)
- port
- dbname
- user
- password

There are other options available too. The connection string format is *option=value option=value* .... For example, *host=billing.bigco.com dbname=finance user=sam* connects as user *sam* to database *finance* on host *billing.bigco.com*. The connection handle returned should be checked to make sure it succeeded before sending a query.

Query

You will notice queries are passed to the database as ordinary strings. This makes programming very easy. You can easily create queries by creating query strings:

```
char query_string[500];
char name[50];
int unique_id;
sprintf(query_-
string, "SELECT * FROM tab WHERE col = %d", unique_id);
res = PQexec(conn, query_string);
```

or

```
sprintf(query_-
string, "INSERT INTO tab VALUES (%d, '%s')", unique_id, name);
res = PQexec(conn, query_string);
```

You can see more examples in the programs in the previous section.

The result handle should be checked to make sure the query succeeded before continuing to access the result. In the last query, *name* is a character string variable. If it contains single quotes, you have to convert them to backslash-quote (\') or two single-quotes (''). If *name* contains backslashes, you have to convert them to double backslashes (\\).

Result

Accessing result information is straightforward. First, only SELECT returns a result set. Other commands like INSERT and DELETE return simple status information.

SELECT's result set is a table made up of rows and columns. Of course, sometimes the result is only one row or one column, but it still needs to be accessed as a table. This accesses the first column in the first row of the result:

```
printf("%s\n", PQgetvalue(res, 0, 0));
```

The first row and first column are numbered zero. In libpq, *PQntuples()* returns the number of rows in the result, and *PQnfields()* returns the number of columns. Using this information, any result value can be retrieved by specifying its row and column. Here is a more complicated example that displays all values returned in a result.

```
int i, j;

for (i = 0; i < PQntuples(res); i++)                /* loop through all rows returned */
    for (j = 0; j < PQnfields(res); j++)                /* loop through all columns
        printf("%s\n", PQgetvalue(res, i, j));          /* print the value returned */
```

Unless a binary cursor is used, all values are returned as ASCII strings. If you need them in a different format for your application, the values must be converted in the application. POSTGRESQL interfaces have many more functions that return useful information:

- Result status
  - error messages strings associated with connection and result handles
- SELECT result information
  - number of rows returned
  - number of columns returned
  - column names associated with result column numbers
  - column numbers associated with result column names

  - column data types
  - column data type modifiers
  - column length
  - cursor binary status

- SELECT values
  - value null status
  - value length

- Non-SELECT result information
  - command status strings
  - OID of inserted rows

Conclusion

This article illustrates the steps needed use POSTGRESQL in applications:

- Issue a connection request and get a connection handle
- Issue a query and get a result handle
- Access the result

This article focuses on writing applications using only two programming languages. However, writing applications in other languages uses the same concepts. A chapter showing the same application written in various programming languages can be found in my book, *PostgreSQL: Introduction and Concepts* at http://www.postgresql.org/docs/awbook.html. More information about each programming interface is available in the *PostgreSQL Programmer's Manual,* http://developer.postgresql.org/docs/postgres/index.html. It covers the connection, query, and result functions specific to each interface.