Using PostgreSQL and gnuplot

In this section, you learn the fundamentals of gnuplot and, after some simple examples, move to PostgreSQL and see how it can be combined with gnuplot to generate database-driven graphics. To start gnuplot, use the following:

[hs@duron hs]$ gnuplot -background white`-background white` makes sure that the background of the graphs we want to generate is white. After starting gnuplot, we will be in an interactive shell.

Now we can draw graphs by using the `plot` command:

```
gnuplot> plot tanh(x) - cos(x)
```

We want to see the graph of `tanh(x) - cos(x)` (tanh is tangens hyperbolicus; cos is cosinus). Figure 20.1 shows the result.

Figure 20.1. A simple mathematical function.

gnuplot finds the best interval for the graph so that only the "useful" part is displayed.

gnuplot provides a large number of functions; you explore the most important ones in this section.

Let's get back to PostgreSQL. We have compiled a small table storing incomes for certain years. We will use this table for the next example:

```
mygnuplot=# SELECT * FROM income;
 year | income
------+--------
 1997 |  24000
 1998 |  26300
 1999 |  26000
 2000 |  29000
 2001 |  32100
 2002 |  32000
 2003 |  32700
(7 rows)
```

The data shown does not have the right format to be used for gnuplot directly; we have to use some parameters when retrieving the data from the table. In this example, we want only the first column to be displayed, so we select only the first column from the table:

```
[hs@duron hs]$ psql -c "SELECT income FROM income" -A -F " " -t mygnuplot
24000
26300
26000
29000
32100
32000
32700
```

The data shown already fits our demands. Let's look at the command-line parameters: `-A` tells

PostgreSQL to use the unaligned table output mode. `-F` defines the field separator—this option is useful only when multiple columns are returned by the query. `-t` makes sure that the header is silently omitted.

Now that we have written a shell command, which returns the data in the way we need it, we have to find a way to pass it to gnuplot in order to draw the graph. Luckily, gnuplot supports some easy methods to read the data.

The following shows how we compute the result of the shell command we have shown previously and draw a graph:

```
gnuplot> plot "< psql -c 'SELECT income FROM income' -A -F ' ' -t mygnuplot "
```

The command between the two double quotes is evaluated, and the result is returned to gnuplot, which displays the graph on the screen (see Figure 20.2).

Figure 20.2. gnuplot displays a window containing the graph.

We can see an array of points representing the values in the database.

Up to now, we have created all graphics with the help of gnuplot's shell. This might be a bit of a problem for you when building huge applications. Let's see how the graphics shown previously can be created with the help of one simple Bourne shell command:

```
[hs@duron hs]$ echo "plot \ "< psql -c 'SELECT income FROM income' -A -F ' ' -t
mygnuplot \ " " | gnuplot -background white -persist
```

We simply pipe the command to gnuplot. All double quotes passed to the program have to be masqueraded using a backslash so that the shell does not mix up the double quotes used for the echo command with the double quotes used for gnuplot.

The `-persist` flag is necessary; otherwise, gnuplot will close the X windows when the execution of the script is ready. Using `-persist` makes sure that the window is still alive after gnuplot has terminated.

Having the result displayed in an X window might be nice, but in some cases it is necessary to store the result generated by gnuplot in a file that you can use for further transformation. The following includes a file that contains a short gnuplot script.

First we set the terminal to `pbm`, which means that a `pbm` file is generated instead of X11 output. Then we include the command we showed previously:

```
[hs@duron gnuplot]$ cat config.plot
set terminal pbm color
plot "< psql -c 'SELECT income FROM income' -A -F ' ' -t mygnuplot "
```

We start gnuplot with `config.plot` as the first parameter and redirect the output to a file called `graph.pbm`:

```
[hs@duron gnuplot]$ gnuplot config.plot  > graph.pbm
```

We can use that file now. If `pbm` is not our desired file format, we can easily convert the file to `jgp` or any other format by using a program such as `mogrify`:

```
[hs@duron gnuplot]$ mogrify -format jpg *.pbm
```

Now we want to change the labels of the graphics we have just generated. The first thing we want to do is to eliminate the ugly string containing `psql` on the upper edge of the graph. This can be done by using `set nokey`. The title of the graph should be `Income table`, and the axis should be labeled with `year` and `income in USD per year`. Up to now we have also seen that gnuplot finds the right scale for the graph by itself. In some cases, this might not be what we need, so we define the range of values displayed manually using `xrange` and `yrange`.

The following is a file that does exactly what we need:

```
set nokey
set title "Income table"
set xlabel "year"
set ylabel "income in USD per year"

xmax=6
ymin=0
ymax=40000

set xrange [0 : xmax]
set yrange [ymin : ymax]

plot "< psql -c 'SELECT income FROM income' -A -F ' ' -t mygnuplot "
```

We can generate the graph using a simple shell command:

```
[hs@duron gnuplot]$ gnuplot -persist -background white config.plot
```

The result will be the graph shown in Figure 20.3.

Figure 20.3. Redesigning the graph.

Often it is necessary to plot two independent data sources in one graph—especially when you want to compare data. The following script shows how you can plot two data sources (they contain the same data) in two different ways:

```
set nokey
set grid
set title "Income table"
set xlabel "year"
set ylabel "income in USD per year"

xmax=6
ymin=0
ymax=40000

set yrange [ymin : ymax]

plot "< psql -c 'SELECT year, income FROM income' -A -F ' ' -t mygnuplot" \
                with lines linewidth 3, \
        "< psql -c 'SELECT year, income FROM income' -A -F ' ' -t mygnuplot" \
                with boxes linewidth 3
```

We want a grid to be displayed in the background of our graph. The last four lines of code are responsible for plotting. The first data source is displayed as a line, and `linewidth` is set to `3`. The second data source is displayed using boxes. gnuplot makes sure that the two components of the graph are displayed in different colors (see Figure 20.4).

Figure 20.4. Plotting two data sources.

The script plots the graph as we expected it to be, but there are still two problems that have to be solved. We have plotted two data sources, and therefore we have queried the database twice. This leads to two problems: It might happen that the data in the database changes during the execution of the first query, so the result of the second query might differ from the result of the first query. Another problem is that executing two queries takes much longer than executing just one.

Our two problems can be solved easily. To show you how this can be done, we have compiled a table containing the income of males and females:

```
mygnuplot=# SELECT * FROM income;
 year | male  | female
------+-------+--------
 1997 | 24000 |  19200
 1998 | 26300 |  21400
 1999 | 26000 |  21800
 2000 | 29000 |  21000
 2001 | 32100 |  26500
 2002 | 32000 |  26700
 2003 | 33700 |  28200
(7 rows)
```

We use a simple `Makefile` to generate the result:

```
[hs@duron gnuplot]$ cat Makefile
x       :       config.plot
        psql -c 'SELECT year, male, female FROM income' -t -A -F ' ' \
                mygnuplot > file.data
        gnuplot -background white -persist config.plot
        rm -f file.data
```

Let's analyze the `Makefile` first. We select all data from table `income` and store the result of the query in `file.data`. Then we start gnuplot and pass the name of the configuration script to it.

Here is the configuration script:

```
set nokey
set grid
set time
set title "Income table"
set xlabel "year"
set ylabel "income in USD per year"

set yrange[0 : ]
```

```
plot 'file.data' using 1:2 with lines linewidth 2, \
        'file.data' using 1:3 with lines linewidth 2
```

The time the image was created is displayed by using `set time`. We don't know the highest income in the database, so we set the lower limit for `yrange`" to `0` and do not specify the upper value. gnuplot will make sure that a suitable border for the upper limit will be used. In the next step, we tell gnuplot to use the first and the second column in `file.data` as the data source for plotting the first line. The second line will be plotted using the first and the third column. Using one `plot` command is enough, because every additional graph is simply added to the list (see Figure 20.5).

Figure 20.5. Plotting the data in an external file.

The previous examples showed basic tasks that can be accomplished with gnuplot and PostgreSQL. Sometimes it might be necessary to add explanations to the graph so that the reader can easily understand what you want to say. Two components for adding explanations to a graph are essential: arrows and labels.

In the next example, we added two labels and one arrow to the scenery:

```
set nokey
set grid
set time

set arrow from 1999, 15000 to 2000, 20000
set label "significant changes" at 1998.35, 13000

set label "(c) SAMS" at 2002.3, 2200

set format y "%g$"
set title "Income table"
set xlabel "year"
set ylabel "income"

set yrange[0 : ]

plot 'file.data' using 1:2 with lines linewidth 2, \
        'file.data' using 1:3 with lines linewidth 2
```

First we add the arrow to the graph. As you can see, the coordinates can simply be defined. Then we add the labels to the scenery. The first component that we define for the label is the text we want to be displayed. Then we tell gnuplot where to place it, as we did for the arrow. We start gnuplot with the file shown.

The result can be seen in Figure 20.6.

Figure 20.6. A plot including arrows and comments.

For many applications, it is necessary to produce three-dimensional plots. Because gnuplot is a highly developed software, it is also capable of generating three-dimensional graphics.

To show you how such plots can be created, we compiled a table that contains two additional columns, which store the number of people we used to compute the average income (`wf` is an abbreviation for workforce):

```
mygnuplot=# SELECT * FROM income;
 year | avg_male | wf_male | avg_female | wf_female
------+----------+---------+------------+-----------
 1997 |    24000 |     732 |      19200 |       932
 1998 |    26300 |    1412 |      21400 |      1054
 1999 |    26000 |    1930 |      21800 |      1320
 2000 |    29000 |    2065 |      21000 |      1150
 2001 |    32100 |    2163 |      26500 |      1259
 2002 |    32000 |    2254 |      26700 |      1292
 2003 |    33700 |    2620 |      28200 |      1721
(7 rows)
```

The following is the config file for our three-dimensional plot:

```
set nokey
set time

set format y "%g$"
set title "Income table"
set xlabel "year"
set ylabel "income"

splot 'file.data' using 1:2:3 with lines linewidth 2, \
        'file.data' using 1:4:5 with lines linewidth 2
```

Here, we have to use `splot` instead of `plot`. Because we have one additional dimension, we have to define three columns containing the data—we select the numbers of the columns according to the order they can be found in the input file.

The screenshot of the plot is shown in Figure 20.7.

Figure 20.7. A simple 3-D plot.

Geometric Datatypes and gnuplot—A Simple Example

PostgreSQL supports a powerful set of geometric datatypes. Displaying geometric data stored in a PostgreSQL database for many applications might be useful. Drawing database-driven images can be a difficult task, although there is a lot of Unix software available to do the job. Gimp, for instance, provides a powerful ASCII interface to generate and manipulate images from the command line (which is useful for Web applications). You also can use a scripting language in combination with gnuplot to do the job. This section shows you some basic ideas for visualizing geometric data stored in a PostgreSQL database. Let's start with a simple example.

We have compiled a small table containing some points:

```
mygnuplot=# SELECT * FROM mypoints;
 mypoints
----------
 (1,0)
 (6,4)
 (-3,-4)
 (3,-4)
(4 rows)
```

With the help of a trivial shell script, we can generate a simple plot. Before we get to the gnuplot code, here is the shell script that we will use to transform the data to the desired format:

```
#!/bin/sh

psql -c 'SELECT * FROM mypoints' -t mygnuplot | sed -e 's/,/ /gi' -e 's/)/ /gi'
-e 's/(/ /gi'
```

First we select the data from table `gnuplot`. `-t` makes sure that only the data is returned by the query. In the next step, we use `sed` (stream editor) to parse the data. We eliminate all brackets and commas so that gnuplot can easily read the input. Then we start gnuplot and use some simple commands:

```
gnuplot> set nokey
gnuplot> set xrange [-10 : 10]
gnuplot> set yrange [-10 : 10]
gnuplot> plot '< /home/hs/geo/plot.sh'
```

The first three lines define some basic properties of the graph, such as the range of values displayed on the x-axis. The fourth line starts our shell script and sends the data to gnuplot. Figure 20.10 shows the display when you run gnuplot.

Figure 20.10. Drawing points.

The next example demonstrates how polygons can be plotted with the help of gnuplot. We have already dealt with polygons in Chapter 3, "An Introduction to SQL;" this time we generate a simple prototype of a script that prints all polygons stored in the table in a graph.

We have compiled a small table containing three polygons:

```
mygnuplot=# SELECT * FROM mypolygons;
          mypolygons

 ((-4,2),(3,9),(6,10),(13,13))
 ((4,2),(5,7),(6,12),(9,10))
 ((-4,2),(3,9),(6,0),(3,10))
(3 rows)
```

The following source code is a small script generating a config file for gnuplot:

```
#!/usr/bin/perl

use DBI;

# connecting
$connect="dbi:Pg:dbname=mygnuplot;port=5432";
$dbh=DBI->connect("$connect","hs")
     or die "cannot connect to the database\ n";

# getting lines
```

```
$sql="SELECT * FROM mypolygons";
$sth=$dbh->prepare("$sql")
     or die "cannot prepare statement\ n";
$sth->execute()
     or die "cannot execute query\ n";

# open gnuplot config-file
open(PLOT,"> config.plot")
     or die "cannot open config file for gnuplot\ n";
print PLOT "set xrange [-5 : 15]\ n";
print PLOT "set yrange [-5 : 15]\ n";

# setting variables and retrieving data
$files=0;
$plotstring="plot";

while   (@row = $sth->fetchrow_array)
{
     $line=$row[0];
     $line=~s/\ ),\ (/\ n/gi;
     $line=~s/\ (|\ )//gi;
     $line=~s/,/ /gi;

     # open file that contains data
     open(DATA,"> $files.data")
             or die "cannot open file $files.data\ n";
     print DATA "$line\ n";
     close(DATA);

     # generating plot command
     $plotstring.=" '$files.data' with lines linewidth 3,";
     $files++;
}

$plotstring=~s/,$//gi;
print PLOT "$plotstring\ n";
close(PLOT);

# drawing data and cleanups
system("gnuplot -background white -persist config.plot");
system("rm -f [0-9].data");
```

First we connect to the database and send a simple SQL query to the server that retrieves all records in table `mypolygons`. As you learned in Chapter 9, "Extended PostgreSQL—Programming Interfaces," the SQL query has to be prepared and executed to receive the required result. Then we open the file, called `config.plot`, that we need to store the gnuplot code:

```
$files=0;
$plotstring="plot";
```

These two lines initialize two important variables. `$files` contains the number of polygons our graph will contain. `$files` is also used for generating the temporary file. `$plotstring` contains the `plot` command for our graph. Every polygon added to the scenery requires its own entry in the `plot` command.

After initializing the variables, we start retrieving the data from the query that we have processed before. Every line returned will be transformed to the required format and stored in a temporary file. After that, the polygon is added to the scenery: We extract the line from the result of our query and perform some basic operation such as substituting),(for a linefeed. This is necessary so that gnuplout can distinguish the various points of our polygon. We also eliminate all other brackets by using simple regular expressions.

After adding all components to the graph, we substitute the comma at the end of $plotstring for nothing and write the string to the file. Our config file is now ready and we can execute gnuplot using the file. Before we have a look at the graph, we have included the file config.plot. You can see that the `plot` command consists of three components, each having its own datafile:

```
set xrange [-5 : 15]
set yrange [-5 : 15]
plot '0.data' with lines linewidth 3, '1.data' with lines linewidth 3, '2.data'
with lines linewidth 3
```

The most important component of our program is the converter that we use for reformatting the output of the database to a format that we can use for gnuplot. Recall that you do this with regular expressions. Lines such as the next one are converted to a file containing multiple lines:

```
((-4,2),(3,9),(6,10),(13,13))
```

Every node of the polygon is transformed to a separate line:

```
-4 2
3 9
6 10
13 13
```

Keep in mind, that the columns of the file should be separated by a blank or a tab; if you use commas, gnuplot will not display the result correctly (if you don't define an input format). Figure 20.11 shows the graph.

Figure 20.11. Drawing the polygons stored in the database.

Three polygons are displayed, because three records have been retrieved from the database.

As you can see, drawing database-driven geometric objects is an easy task with gnuplot. Of course it is not only possible to implement converters for points and geometric objects. Because gnuplot is flexible and powerful, it is possible to draw almost anything—just try it.

Online em: http://books.google.com.br/books?id=LjXBeAWM-KgC&pg=PA692&lpg=PA692&dq=Using+PostgreSQL+and+gnuplot&source=web&ots=fj-TQl-gBA&sig=2A_cqDUSVqrcsbFuo4sP2RNIPto&hl=pt-BR&sa=X&oi=book_result&resnum=8&ct=result#PPA694,M1