
CHAPTER 15

SQL Database Address Book

OVERVIEW

So far, the chapters in this part of the book have explained how to interface various ASCII text file-based databases on the Web. This provides a useful service, because many companies cannot afford the high cost associated with commercial relational database management systems (RDBMS). However, for those people who have access to a robust RDBMS, this chapter gives an example of how to perform basic database operations on a commercial database using CGI.

An address book is a good application to demonstrate the major functions that have been discussed in the previous chapters. In this example, we will use Sybase as the database server. However, the scripts are applicable to other RDBMSs, such as Oracle and Informix, and require minimal work to be converted to those systems. Basically, the Perl scripts here use the command-line utilities of whatever major RDBMS you happen to be using. The Structured Query Language (SQL) code gets passed to the command-line utility using the Perl scripts. The example discussed here uses Sybase's

ISQL command-line utility to send and receive data to and from a Sybase database. Although database-specific libraries for Perl, such as Sybperl and Oraperl, do exist, we have avoided using them. They are too specific to each database to allow us to program a database-independent script.

INSTALLATION AND USAGE

The SQL address book consists of multiple script files that process the various database operations. The files can be split into two categories: query and maintenance. In addition, other files are used for setup. Figure 15.1 outlines the expanded directory structure and permissions needed for the files belonging to this application.

The root **Address** directory must have permissions that allow the Web server to read and execute and should contain a subdirectory for temporary files called **Temp**.

Address Query Files

addr_query.cgi outputs an HTML form that allows the user to query on the address book. All the scripts prefixed with **addr_query** should be readable and executable.

addr_query_result.cgi processes the form variables from the HTML form output by **addr_query.cgi**. It sends an SQL query to the database server and returns the results to the user.

Address Maintenance Files

addr_maint.cgi outputs the first HTML page for the maintenance side of the address book. It outputs a form with three buttons: **add**, **delete**, and **modify**. All the scripts prefixed with **addr_maint** must be readable and executable.

addr_maint_search.cgi processes the form variable output of **addr_maint.cgi**. It prints an HTML form appropriate to whatever action is being taken: addition, modification, or deletion of an address book entry.

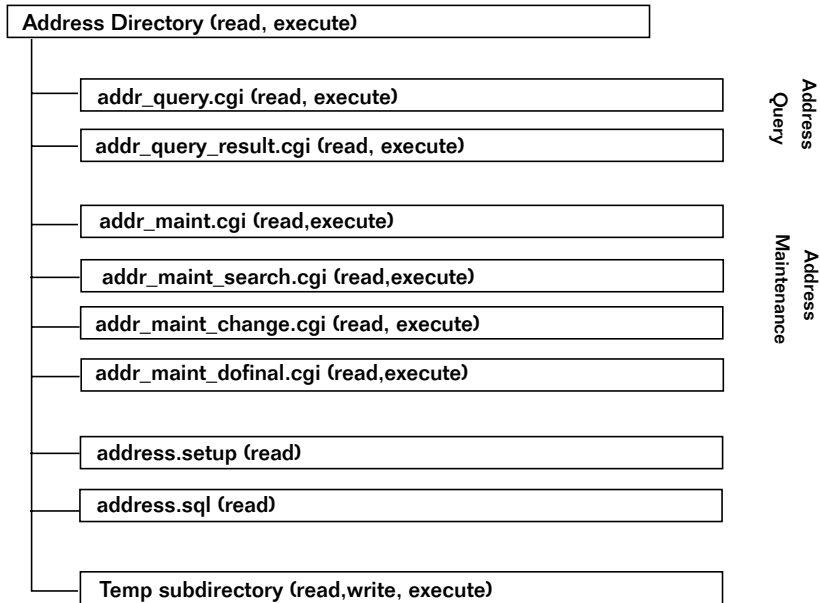


Figure 15.1 Directory structure of the SQL address book.

addr_maint_change.cgi processes the form output from **addr_maint_search.cgi**. If the operation is an addition, it submits the new address information directly to the database server. If the operation is a modification or deletion, it submits the query information to the database server and returns a list of rows that the user can select from to perform the edit or removal.

addr_maint_dofinal.cgi is the final script in the maintenance process for modifications and deletions. It takes the form output of **addr_maint_change.cgi** and submits SQL code to the database server to perform the operation.



NOTE

The address book scripts as they exist here are Sybase-specific. Therefore, you will encounter some Sybase-specific terminology in this chapter. If you own another commercial RDBMS, you should talk to your database administrator about doing the equivalent operations. For example, ISQL is a Sybase command-line utility in which SQL code can be passed. There are similar utilities for other RDBMSes.

Common Address Book Files

address.setup is a common setup file that is shared among the maintenance and query scripts. It must be readable by the Web server. In addition, if you decide to separate the maintenance and query scripts into their own directories, a copy of **address.setup** must exist in both directories.

address.sql is a SQL script that contains the SQL code to create the table in a database.

Server-Specific Setup and Options

The first step is to prepare the database with the information it needs in order to be accessed by the Web. For example, you need to create the table that will contain the addresses, and you need to add a login that has permissions on the table for querying and modification.

The address book scripts in this example use the database “infobase” and the table “address” inside that database. All the scripts here must be able to find this information. If your database cannot be set up in this manner, you need to modify **address.setup** to reference a different area of your database server.

The **address.sql** file contains sample SQL code to create a simple address table inside the infobase database. Simply run this script using the ISQL command-line utility to create the table in the infobase database. The **address.sql** file is discussed in the design discussion later. Here is an example of running the command-line utility with a username of “user” and a password of “pass”:

```
isql -User -Ppass <address.sql
```

Next, you need a login ID and password that has rights to enter the infobase database and perform operations on the address table inside the database. Specifically, this login ID needs to be granted permissions to select, insert, update, and delete items in the address table. The query scripts need only permission to select. The maintenance scripts require the full set of permissions.

In addition, the scripts rely on Sybase having been installed in the default **/opt/sybase** directory on your UNIX server. Thus, the ISQL program should reside in **/opt/sybase/bin**. If Sybase is not installed here, you need to modify the **address.setup** file to change the path where Sybase is installed.

All the address query and maintenance scripts can reside in the same directory. This is the default placement of the files for the address book. However, you may want to separate the maintenance scripts into another directory and password-protect that directory using your Web server if you can. If you separate the scripts, make sure that **address.setup** exists in both areas. Both the maintenance scripts and the regular querying scripts require **address.setup** to work.

THE SETUP FILE

The **address.setup** file contains only a few configuration variables; they are related to the location of Sybase on your server and the way to access the address table within the Sybase RDBMS server.

`$db_dir` is the directory where Sybase is installed. The ISQL program is expected to be installed in a subdirectory under `$db_dir` called **bin**. The interfaces file that tells Sybase where to connect to the running RDBMS server should also be in this directory.

`$db_server` is the server name that ISQL uses to log on with. There can be multiple Sybase servers on a single machine. This name helps differentiate between them. The default server name is simply “sybase.”

`$db_name` is the name of the database where the address table is stored. By default, the address table is stored in **infobase**.



N O T E

If you use a database name other than **infobase**, you will want to modify the **address.sql** file so that the SQL code in it creates the address table in your database instead of a database called **infobase**.

`$db_user` is the name of the login ID that the Web server uses to log on to the Sybase database. Likewise, `$db_password` is the password associated with the login ID.

Here is an example of the **address.setup** file:

```
$db_dir = "/opt/sybase";  
$db_server = "sybase";  
$db_name = "infobase";  
$db_user = "test";  
$db_password = "test";
```

Running the Scripts

Using the address book query script is easy. Merely call the **addr_query.cgi** script through the Web browser. This action takes you to a screen that allows you to query on the various fields of the address book to get a list of employees and their information. Figure 15.2 shows an example of the address book query screen. Here is a sample URL for this script if it is installed in an **Address** subdirectory under a **cgi-bin** directory:

```
http://www.foobar.com/cgi-bin/Address/addr_query.cgi
```

To perform maintenance on the address table through the Web, use the **addr_maint.cgi** script. This action will take you to a screen that gives you a choice of operations: add, modify, or delete. Figure 15.3 shows an example of this screen. A sample URL for this script follows:

```
http://www.foobar.com/cgi-bin/Address/addr_maint.cgi
```

DESIGN DISCUSSION

The address book query and maintenance applications can be thought of as two separate applications, because neither CGI program interacts with the other. The programs that begin with **addr_maint** call only other scripts that begin with **addr_maint**. Similarly, the **addr_query.cgi** script calls only **addr_query_result.cgi**. Figures 15.4 and 15.5 show an example of the logic involved in both sets of programs.

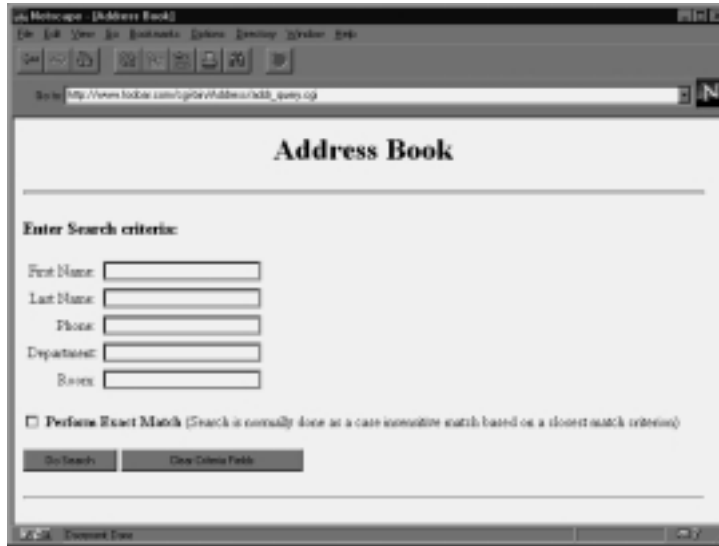


Figure 15.2 The SQL address book query screen.



Figure 15.3 The SQL address book maintenance screen.

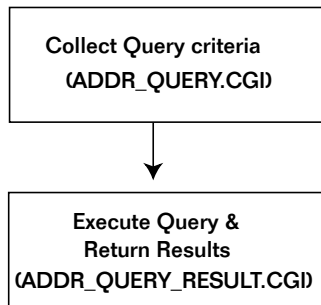


Figure 15.4 Flowchart of the address book query.

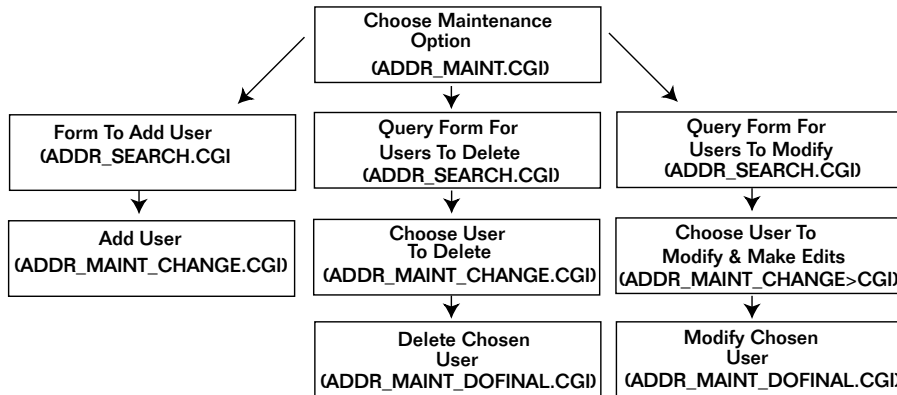


Figure 15.5 Flowchart of the address book maintenance.

In this section, we will discuss SQL basics and then examine the details of the query part of the program. Finally, the maintenance scripts will be explained.

SQL Basics

It is beyond the scope of this book to go into the details of how SQL works. Instead, a brief overview is given here to familiarize you with just enough SQL to understand how the scripts operate. Additionally, your RDBMS should have documentation related to the nature of the SQL

code for your database. The specific SQL code in the address book scripts will also be commented later.



If you do not know SQL, this chapter may seem a bit daunting. Keep in mind that the generated queries are not very complex in the scripts. As you read this overview section, do not feel uncomfortable if you understand only some parts of the SQL language the first time around. The important thing is to get a general idea of SQL before moving on.

The databases that we have dealt with so far in this book have been ASCII text files. These files can be thought of as databases, because, although they are text files, they store information in a structured manner conducive to being searched and updated. Nonetheless, they are only files. To search or manipulate the files, Perl file operations must be used.

Commercial RDBMSes are different. You usually do not have any direct access to the data on the disk. You no longer have an option of opening a file and performing a query yourself by parsing all the lines in a Perl `while` loop. The only way to access tables in an RDBMS such as the one discussed here is to use SQL.

SQL is not a procedural language. Nothing in the SQL specification tells the server how to go about getting the data to you. SQL consists of commands that tell the database what you want the result to be; the database server figures out the “how” of getting it to you.

For example, suppose I wanted to obtain a list of all the employees who live in Texas. If I were dealing with a text file, I would open the file and read in all the lines one by one to see which lines happened to have an employee that lived in Texas. Then I would close the file. With SQL, I would simply send the appropriate SQL code saying, “I want the names of all employees who live in Texas,” and the database server would return the whole result. With the SQL example, you only ask for the information you want instead of defining the procedure for getting that information.

Tables in a database are similar in concept to the ASCII text files discussed earlier. Instead of lines or records in a file corresponding to each employee, a database table is thought of as containing rows of data in which each row corresponds to an employee. Furthermore, whereas the

text file has fields in each line that may be pipe- or comma-delimited, the table has columns that are logically defined. You need not be concerned about how the database server stores the fields (pipe-delimited or not). In addition, SQL does not care about the order of the rows or columns in a table. With a file, you must know the order of the fields, because the Perl script uses procedural code to read the fields in each line one by one.

QUERYING THE DATABASE WITH SQL

The workhorse of the SQL language is the `select` statement. `select` tells the database which field names you are interested in looking at. In addition, the `from` command added to the `select` statement lets the database know which table to get the fields from. Finally, if you do not want the whole table returned, you use the `where` command added to the `select` and `from` commands to tell the database the criteria to use in selecting the fields. The following example SQL statement asks for the first and last names of all employees whose first name is Bill.

```
select first_name, last_name
from address
where first_name = "Bill"
```

The `select` clause tells the database to return the `first_name` and `last_name` fields.



N O T E

The field names in a database normally follow standard variable-naming conventions and tend not to be “pretty.” In our address book script, we generally output different headers for the field names.

The `from` clause tells the database to return the field information from the address table. Finally, the `where` clause tells the database to restrict the rows returned to only those that contain the name Bill.

MODIFYING THE DATABASE WITH SQL

`select` statements are good for searching a table, but frequently we need to perform modification operations such as additions, deletions, and

updates. These operations are done using the `insert`, `delete`, and `update` commands.

The syntax of an `insert` follows this format:

```
INSERT [TABLE_NAME]
([COLUMN_LIST])
VALUES
([VALUE_LIST])
```

Thus, to insert “John Doe” into the address table, we would use the following SQL statement:

```
insert address
(first_name, last_name)
values
("John", "Doe")
```

The syntax for the `delete` command follows this format:

```
DELETE [TABLE_NAME]
WHERE
[EXPRESSION_TO_SEARCH_FOR]
```

If the `where` clause is left out, all the rows in the table will be deleted by default. Thus, it is always important to include a `where` clause so that the database knows which records to remove. To delete “John Doe” from the address table, we would use the following statement:

```
delete address
where
first_name = "John" and last_name = "Doe"
```

The syntax for the `update` command follows this format:

```
UPDATE [TABLE_NAME]
SET [COLUMN_NAME = EXPRESSION]
WHERE [EXPRESSION_TO_SEARCH_FOR]
```

Again, the `where` clause is optional. Remember, though, that omitting it will result in the whole table being updated instead of just a few rows. To

change the name of “John Doe” to “Joe Smith,” we would use the following:

```
update address
set first_name = "Joe", last_name = "Smith"
where first_name = "John" and last_name = "Doe"
```

That’s really all there is to it. SQL consists of a few operators that can do many things depending on how you combine them. The address book is a relatively simple application of SQL, so the SQL code that the scripts generate here is not very complex. SQL can do much more, but let’s move on to discuss the address book, because it does not take advantage of the more advanced features of SQL.

address.sql

Address.sql contains the SQL code for creating the address table. Tables in SQL have variable types associated with them. For the purpose of this application, we stick to VARCHAR (variable-length character field) types of variables for every field. However, commercial databases typically support many datatypes, including date and time, integer, money, and more. The advantage of using these different datatypes is that the SQL language can make certain assumptions about a datatype. For example, with a datatype of money, you can use SQL to gather a sum of all the rows for that field without having to first convert a text string to a number. Because the address book is relatively simple, we use only text-based fields.

`varchar` is the Sybase keyword for indicating variable-length character data. The number in parentheses is the maximum length of each field. The `go` command is used to submit your SQL code. The `use` command at the beginning of the script tells the ISQL program to switch to the infobase database before the table is created. Note that the `use` command is Sybase-specific; you may need to use syntax appropriate for your own database. In addition, if you are using a database other than infobase to store your address table, you will need to change the reference from infobase to the name you are using for the database.

```
use infobase
go
```

```
create table address
(lname  varchar(15),
 fname  varchar(15),
 depart varchar(35),
 phone  varchar(15),
 room   varchar(15))
go
```

addr_query.cgi

The **addr_query.cgi** script prints an HTML form with fields for searching the address table. Figure 15.2 showed an example of this form. In addition, a check box is included on the form so that the user can select an exact match search or a pattern match–based search. Both searches are performed without regard to upper- and lowercase, but the exact match search requires that all the characters in the search field exactly match the characters in the database column. A CGI script is used to print the HTML (instead of using an HTML file) because some CGI directories on Web servers are set up so that they cannot read HTML files there. In the interest of keeping all the address book files in one place, the HTML code is converted to a CGI script that outputs HTML.

The first thing the script does is to print the HTTP header “Content-type: text/html,” which allows the rest of the CGI program to output HTML code. The remainder of the program prints this HTML code using the HERE DOCUMENT method of Perl. Figure 15.2 showed what the HTML looks like in a Web browser for this screen.

```
#!/usr/local/bin/perl
print "Content-type: text/html\n\n";

# The following is a PERL "Here" document,
# printed out as HTML
print <<__END_OF_HTML__>
<HTML>
<HEAD>
<TITLE>Address Book</TITLE>
</HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">
<CENTER>
<H1>Address Book</H1>
</CENTER>
```

Chapter 15: SQL Database Address Book

```
<HR>
<FORM ACTION=addr_query_result.cgi METHOD=POST>
<H3><STRONG>Enter Search criteria: </STRONG></H3>
<TABLE>
<tr>
<td align=right>First Name:</td>
<td><input name=fname></td>
</tr>
<tr>
<td align=right>Last Name:</td>
<td><input name=lname></td>
</tr>
<tr>
<td align=right>Phone:</td>
<td><input name=phone></td>
</tr>
<tr>
<td align=right>Department:</td>
<td><input name=depart></td>
</tr>
<tr>
<td align=right>Room:</td>
<td><input name=room></td>
</tr>
</TABLE>
<P>
<INPUT TYPE=checkbox NAME=exactmatch><STRONG> Perform Exact
Match</STRONG>
(Search is normally done as a case insensitive match based on a clos-
est
match criterion) <P>
<INPUT TYPE=submit name=doquery value="Do Search">
<INPUT TYPE=reset value="Clear Criteria Fields">
</FORM>

<P><HR>

</BODY></HTML>
__END_OF_HTML__
```

addr_query_result.cgi

addr_query_result.cgi is called when the HTML form that is output by **addr_query.cgi** is submitted to the Web server. It parses the query parameters into SQL code. This SQL code is then sent to the database engine. Finally,

the SQL results are parsed and sent to the user in the form of HTML code. Figure 15.6 shows a sample query result.



Figure 15.6 Sample address book query result.

The first step is to declare a path where libraries can be located. By default, the library files are located in the current directory. This is indicated by the “.” that `$lib` is set to. Then `cgi-lib.pl` is loaded to process the incoming form data. The `address.setup` file is also loaded. It does not use the `$lib` path, because we expect `address.setup` always to be located in the same directory as these CGI scripts. Libraries, on the other hand, may be located in a directory shared by all scripts, or they may remain in the same directory as each of the CGI applications.

```
$lib = ".";
require "$lib/cgi-lib.pl";
require "address.setup";
```

Chapter 15: SQL Database Address Book

The standard Web HTTP header is printed using the `PrintHeader` function from **cgi-lib.pl**. Then the form variables are read into the `%in` associative array using the `ReadParse` function.

```
# Print the magic HTTP header
print &PrintHeader;

# Parse the form variables into the %in associative array.
&ReadParse;
```

The header portion of the HTML code is printed using the `HERE DOCUMENT` method.

```
print <<__END_OF_HTML__>
<HTML>
<HEAD>
<TITLE>Address Book Query Results</TITLE>
</HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">
<CENTER>
<H1>Address Book Information</H1>
</CENTER>
__END_OF_HTML__
```

Before the query results can be printed, they must be formatted. In this case, the address results are sent in the form of an HTML table. `$header_format` contains the HTML code for printing the header. `$format_table` contains the HTML code for printing every row of the table. `$exactmatch` is set to `on` if the user chooses to do an exact match search on the previous HTML form.

```
$format_table = "<TR>";
$header_format = "<TR>";
$exactmatch = $in{"exactmatch"};
```

The `@field_names` array contains a list of the field names as they exist in the address table for performing the query.



The fields appear in the same order that they will be displayed to the user. The names of the fields must match the column names in Sybase as well as the `<INPUT>` tag names in the previous HTML form. If you are customizing this script to your own address book, you must take care to modify `addr_query.cgi` to update the `<INPUT>` tag names there.

Next, `@field_desc` contains the descriptive information about each field; the field names are not pretty enough to display to the user. The order of the field descriptions must be the same as in the `@field_names` array.

In addition, the `@select_list` array is set equal to the `@field_names` array for processing the list of fields to use in the `select` clause of the SQL code the program will generate.

```
@field_names = ("fname", "lname", "phone",  
               "depart", "room");  
  
@field_desc = ("First Name", "Last Name",  
              "Phone #", "Department", "Room");  
  
@select_list = @field_names;
```



If you customize this program, you may want to use a different select list from the field names. For example, you may want to retrieve fewer fields than are queried on with a `where` clause. Each employee record may include a note, which may be very long. In that case, you may want to query on that note but not display the note as part of the returned fields. The default is to display all the fields that we have the capability of querying on.

For each format variable, we need to make a field tag. The `$header_format` tag needs table header tags (`<TH>`) for each field, and the `$format_table` needs normal table column tags for each field (`<TD>`). The `%s` that is in the

Chapter 15: SQL Database Address Book

middle of each table column acts as a placeholder. Later, Perl's `printf` function will be used to map the column values to the list of `%s` symbols.

```
foreach (@field_desc)
{
    $format_table .= "<TD>%s</TD>";
    $header_format .= "<TH>%s</TH>";
}
```

After the `foreach` loop, the formats are ended with a table row closure (`</TR>`).

```
$format_table .= "</TR>";
$header_format .= "</TR>";
```

`@where_list` is an array that contains the expressions used by the `where` part of the SQL code as criteria to determine which address rows to return.

```
@where_list = ();
```

Each field in the `@field_names` array is cycled through, and the value is culled from the form variable corresponding to that field. The value is converted to lowercase, because the `where` clause compares it to the lowercased value of each field in each row.

```
foreach $field (@field_names)
{
    $value = $in{"$field"};
    $value =~ tr/A-Z/a-z/; # Force to lower case
    if ($value ne "")
    {
```

If `$exactmatch` is on and if a value exists for the field, a search expression is generated in the form of `lower(column_name) = 'value_to_search_for'`.

```
    if ($exactmatch eq "on") {
        push(@where_list,
            qq!lower($field) = ! .
            qq!"$value"!);
```



NOTE

The `qq!` is a Perl trick that allows us to change the double-quotation delimiter in a print statement. Normally, double quotations (") are used to delimit the characters in a print statement. However, by replacing the first quotation mark with two `q`'s followed by another character, that final character becomes the new print statement delimiter. Thus, `qq!` tells Perl to use the `!` character to delimit the string. This is explained in Appendix A.

If `$exactmatch` is not on, the expression takes the same form except that the value to search for has percent (%) symbols appended to the beginning and end of the value. The % symbol is roughly equivalent to asterisks (*) in normal UNIX regular expression matching. When the % symbol is used, the Sybase search will find instances of the value as part of a larger word; the matches found need not be exactly equal to the search value.

```

        } else {
            push(@where_list,
                qq!lower($field) like ! .
                qq!"%$value%!");
        }
    }
}

```

Now that the program has the list of expressions to search on, the SQL code for the `where` statement is generated. To the word `where`, we append the list of expressions separated by `and`. Perl's `join` operator takes every expression in `@where_list` and makes it into one big string separated by the `" and\n"` string.

```

$where_clause = "";
if (@where_list) {
    $where_clause = "where\n" . join(" and\n",
        @where_list);
}

```

As an example, if exact match was not on and the user was querying for the address information of someone named John Doe, the `$where` clause would look like the following:

Chapter 15: SQL Database Address Book

```
where
lower(fname) like "%john%" and
lower(lname) like "%doe%"
```

A `select` statement is generated using the same `join` operator to combine all the elements in the `@select_list` array. The `$sql` variable is used to generate the SQL statement sent to the database.

The `select` list is separated by pipe symbols in the resulting `select` statement. We do this because when ISQL outputs the results, it will be easier to pick individual field values when we know that pipes separate them.

```
$sql = "select\n" .
      join (qq! + "\\|",\n"\\|" +!,
          @select_list);
```

Next, the `from` clause (indicating the database name and table name we are selecting from) is appended to the `$sql` variable. The `$where_clause` is also appended to `$sql`. Finally, the `"order by 1,2"` in the `$sql` variable tells the database to return the results ordered by columns 1 and 2, which are first name and last name, respectively.

```
$sql .= "\nfrom $db_name..address\n";
$sql .= "\n$where_clause\n";
$sql .= "order by 1, 2";
```

In the search for John Doe, `$sql` would now contain the SQL statement shown next. Notice the pipe characters that we discussed. We use two periods between the database name and the table name in the `from` clause, the standard Sybase convention of referring to tables with an absolute database path.

```
select
fname + "|",
"| " +lname + "|",
"| " +phone + "|",
"| " +depart + "|",
"| " +room
from infobase..address
where
```

```
lower(fname) like lower("%john%") and
lower(lname) like lower("%doe%")
order by 1, 2
```

The initial table header is printed. The table header includes the `$header_format` command, which is printed using the `@field_desc` array to supply values to fill in the `%s` characters. The `printf` statement performs the magic of mapping each element of the `@field_desc` array to each `%s` character in the `$header_format` variable. `printf` basically performs a search on all `%s` characters in the string and then replaces them one by one with the elements in the `@field_desc` array.

```
# Print the header for the columns to be returned
print "<TABLE BORDER>\n";
printf $header_format . "\n", @field_desc;
```

A file is generated that contains the SQL code that will be processed by ISQL. `$file` is set to the current process ID plus an `.sql` extension. The first line of the file is the logon password. This is done because ISQL expects a password to be entered right away that matches the username entered as a command-line parameter. Finally, the `$sql` code variable and a `go` command to run the `$sql` are added to the file. Then the file is closed.

```
$file = "$$.sql";
open (QUERYFILE, ">$file");
print QUERYFILE "$db_password\n";
print QUERYFILE "$sql\n";
print QUERYFILE "go\n";
close (QUERYFILE);
```

Sybase expects the environmental variables of `SYBASE` and `DSQUERY` to be set to the directory path of Sybase and the server name respectively. These values are in the **address.setup** file. The command that is used for passing the query is the ISQL program located in the **bin** directory under the path specified by the `$db_dir` variable.

Next, the program stored in the `$command` variable is opened using the special pipe method. Normally, opening a filename in Perl actually opens the file. However, if we append a pipe symbol to the end of the filename,

Perl knows to execute the program instead. Then the output from the running program is sent back as file data to the `ISQLPIPE` file handle. In this case, the program expects the returned data to be the SQL results. ISQL uses the `-U` parameter to indicate the username of the user who is logging on, and the `-w` parameter to specify the width of each returned line (up to 255 characters). `$file` is provided as input to ISQL.

```
$ENV{"SYBASE"} = "$db_dir";
$ENV{"DSQUERY"} = "$db_server";
$command = "$db_dir/bin/isql";
open (ISQLPIPE,
      "$command -U$db_user -w255 <$file |");
```

The `$rowcount` is set to zero to start. Then the `ISQLPIPE` file handle is read line by line to get the results of the SQL query. For each line that contains pipes, `$rowcount` is incremented. Remember that the program has previously embedded pipe symbols as part of the `select` clause of the SQL code sent to the database server. Again, as with the search for John Doe, the sample output from ISQL appears as follows:

Password:

```
_____|_____|_____|_____|_____|
John|   |Doe|   |555-1212|   |Accounting|   |1D
```

In the sample output, the `Password:` line is a password prompt that ISQL expects before logging in to the database server. The rest of the lines consist of the SQL query result.

If the line contains a pipe symbol, further processing is done in the following code. In the example, only the line with John Doe's address information contains pipes.

```
$rowcount = 0;
while (<ISQLPIPE>) {
    if ($_ =~ /\|/) {
```

The regular expressions before `$rowcount` is incremented serve to eliminate the spaces between the pipe symbols. That way, the values that are

parsed out consist only of the values rather than values that are padded with spaces.

```
s/\| *\|/\|\/g; # Clear away excessive spaces
                # between pipes
    s/^ *//;     # Clear away space at the beginning
                # of the select
```

`$rowcount` is incremented, because we have just read a line from the database.

```
$rowcount++;
```

We set `@field_values` to the resulting values by `splitting` the current line by its pipe symbols. Because each field begins and ends with a pipe symbol, fields are separated here using two consecutive pipe symbols. This is why the `split` command takes two pipe symbols instead of one. Also, the regular expressions and the `split` command are operating from the default Perl `$_` variable. This is why those commands do not explicitly state a variable name.

```
@field_values = split(/\|\/);
```

Finally, Perl's `printf` function is used to print the table rows. `printf` takes all the elements in the `@field_values` list and maps them to the `%s` symbols previously defined in the `$format_table` string.

```
    printf($format_table, @field_values);
} # End of IF
} # End of While
```

Now the program closes the pipe to the ISQL program and removes the file containing the generated SQL code using the `unlink` command. Then the closure for the table (`</TABLE>`) is printed.

```
close (ISQLPIPE);
unlink ("$file");

print "</TABLE><P>\n";
```

Chapter 15: SQL Database Address Book

If `$rowcount` has not been incremented, the script returns a message telling the user that the query did not return any results.

```
if ($rowcount == 0) {
    print "<STRONG><P> No Records Found" .
        " That Matched Your Search" .
        " \nCriteria </STRONG>";
}
```

Finally, the program ends by printing the HTML footer using the `HERE DOCUMENT` method.

```
print <<__END_OF_FOOTER__;
</BODY>
</HTML>
__END_OF_FOOTER__
```

addr_maint.cgi

addr_maint.cgi outputs an HTML form that allows a user to choose which maintenance operation to perform. Figure 15.3 showed an example of this. No other libraries are called. Because extensive form processing is not needed, **cgi-lib.pl** is not required.

The script first prints the header that tells the Web server that this CGI program is printing HTML code.

```
#!/usr/local/bin/perl
print "Content-type: text/html\n\n";
```

Then the `HERE DOCUMENT` method is used to print the HTML page.

```
print <<__END_OF_HTML__;
<HTML><HEAD><TITLE>Address Book Maintenance
</TITLE></HEAD>

<BODY BGCOLOR="FFFFFF">
<CENTER>
<H1>Address Book Maintenance</H1>
<HR>
<P>
```

```
<FORM ACTION=addr_maint_search.cgi METHOD=POST>
<input type=submit
name=new_address_op value="New Address">
<input type=submit
name=mod_address_op value="Modify Address">
<input type=submit
name=del_address_op value="Delete Address">
</form>
</center>
<hr>
</BODY></HTML>
__END_OF_HTML__
```

The HTML code uses an interesting technique that will be used throughout our discussion of the address book maintenance scripts. Several different `<INPUT TYPE=SUBMIT>` HTML buttons make up the form. Each button has a different name and value. The submit buttons have different names so that when the user clicks on one of them, only the one that was clicked will have its value transferred to that form variable name for the subsequent CGI script to process. This technique allows us to provide a kind of menu to the user. Users can press any desired button, and the next CGI script will process the correct choice.



Another interesting trick is to place different forms in the same HTML page. If you ever use this technique, though, remember that information that is input into one form on an HTML page cannot be submitted with information on another form on that same page.

addr_maint_search.cgi

addr_maint_search.cgi is called from the HTML form that is output by **addr_maint.cgi**. It takes the input from the **addr_maint.cgi** form and sends a different HTML form based on whether the operation specified is an addition, deletion, or modification to the address book table.

The first step is to declare a path to the directory where libraries will be defined. By default, the library files are located in the current directory, as is indicated by the “.” that `$lib` is set to. Then **cgi-lib.pl** is required for processing the previous form.

Chapter 15: SQL Database Address Book

```
#!/usr/local/bin/perl

$lib = ".";
require "$lib/cgi-lib.pl";
```

The standard CGI HTTP header is printed using the `PrintHeader` function. Then the form variables are read into the `%in` associative array using `ReadParse`.

```
print &PrintHeader;
&ReadParse;
```

Next, the script determines which buttons were pressed. If a button is pressed, there will be a value for it. If the button was not pressed, the name of the button will not have a value.

```
$new_address_op = $in{'new_address_op'};
$mod_address_op = $in{'mod_address_op'};
$del_address_op = $in{'del_address_op'};
```

`$address_op` is set up as a header title based on the value of the button that was pressed. Because any one of the three buttons may have been pressed, all three variables are checked for a value to impart to `$address_op`.

```
$address_op = $new_address_op;
if ($address_op eq "") {
    $address_op = $mod_address_op;
}
if ($address_op eq "") {
    $address_op = $del_address_op;
}
```

The HTML header is then printed.

```
print <<__HTMLHEADER__>
<HTML><HEAD>
<TITLE>Address Book Maintenance</TITLE>
</HEAD>
<BODY BGCOLOR="FFFFFF">
<H1>$address_op</H1>
<HR>
<P>
```

```
<FORM ACTION=addr_maint_change.cgi METHOD=POST>
__HTMLHEADER__
```

Information is printed based on what was entered on the previous form. The script checks the `$new_address_op`, `$mod_address_op`, and `$del_address_op` variables for a value to determine which operation the client has requested. Appropriate instructions are given for the different operations.

The new address operation results in HTML that tells the user to enter a new address. Figure 15.7 shows an example of the add address book entry screen. The modify and delete operations result in HTML code that tells the user to enter criteria to search for a list of users to modify or delete. Figure 15.8 shows an example of a modification search screen.

```
if ($new_address_op ne "") {
print "Enter The New Information In The Form Below\n";
} elsif ($mod_address_op ne "") {
print "Enter Criteria To Query On In The Form Below.  You will then be
able to choose entries to modify from the resulting list.\n";
} else {
print "Enter Criteria To Query On In The Form Below.  You will then be
able to choose entries to delete from the resulting list.\n" }
```



The screenshot shows a Netscape browser window titled "Netscape - [Address Book Maintenance]". The address bar displays "http://www.toc&le.com/cgi-bin/Address/addr_maint_addch.cgi". The main content area is titled "New Address" and contains the instruction "Enter The New Information In The Form Below". The form consists of five text input fields: "First Name" with the value "John", "Last Name" with "Smith", "Phone" with "555-1233", "Department" with "Purchasing", and "Room" with "41". Below the fields are two buttons: "Add This New Address" and "Clear Form". The browser's status bar at the bottom shows "Document Done" and "127".

Figure 15.7 An example of the add address book entry screen.



Figure 15.8 Example of a search screen for modifying the address book.

The `__HTMLMIDDLE__` HERE DOCUMENT is used to print the HTML code related to a form where the user can enter search criteria or new address book information depending on the context this script is running in.

```
print <<__HTMLMIDDLE__>>
<HR>
<P>

<TABLE>
<tr>
<td align=right>First Name:</td>
<td><input name=fname></td>
</tr>
<tr>
<td align=right>Last Name:</td>
<td><input name=lname></td>
</tr>
<tr>
<td align=right>Phone:</td>
<td><input name=phone></td>
</tr>
```

```
<tr>
<td align=right>Department:</td>
<td><input name=depart></td>
</tr>
<tr>
<td align=right>Room:</td>
<td><input name=room></td>
</tr>
</TABLE>
```

__HTMLMIDDLE__

Based on the operation being performed, different `<INPUT TYPE=SUBMIT>` HTML buttons will be generated. Like the buttons printed on the previous HTML form, they have names that are appropriate to the operation being performed. In addition, for the modify and delete search operations, an input variable for choosing an exact match search is output.

```
# Use different buttons for different operations
# just as before
if ($new_address_op ne "") {
    print "<p><input type=submit " .
        "name=new_address_op" .
        " value=\"Add This New Address\"><p>\n"; }
elseif ($mod_address_op ne "") {
    print "<INPUT TYPE=checkbox " .
        "NAME=exactsearch><STRONG>" .
        "Perform Exact Search</STRONG>";
    print "<p><input type=submit " .
        "name=mod_address_op" .
        " value=\"Query For Modification\"><p>\n"; }
else {
    print "<INPUT TYPE=checkbox " .
        "NAME=exactsearch><STRONG>" .
        "Perform Exact Search</STRONG>";
    print "<p><input type=submit " .
        "name=del_address_op" .
        " value=\"Query For List To Delete\"><p>\n"; }
```

The HTML footer is printed using the `HERE DOCUMENT` method, which ends this script.

```
print <<__HTMLFOOTER__>
<input type=reset value="Clear Form">
</FORM>
```

```
</BODY></HTML>  
__HTMLFOOTER__
```

addr_maint_change.cgi

addr_maint_change.cgi takes the client-defined input from the HTML form generated by **addr_maint_search.cgi** and performs operations based on it. If the operation from the previous screen was an addition, this script performs the insertion into the database, ending the maintenance process. If the operation is a deletion or modification, it prints a list of users that satisfies the previous screen's search criteria. Each returned row has a radio button to allow the user to choose the address book entry on which to perform the operation. The modify operation has additional data entry fields to allow the user to enter the new data. Figure 15.9 displays a sample modification screen. If the operation is not an addition, the HTML form is submitted to **addr_maint_dofinal** for the final change or deletion processing.



Figure 15.9 An example of the address book modification screen.

The first step is to declare a path to the directory where libraries will be defined. Then **cgi-lib.pl** is loaded for processing the incoming form data. The **address.setup** file is also loaded.

```
#!/usr/local/bin/perl
$lib = ".";
require "$lib/cgi-lib.pl";

require "address.setup";
```

The header to tell the Web server that HTML code is about to be printed is output using the `PrintHeader` function. Then the form variables are read into the `%in` associative array using `ReadParse`.

```
# Prints out the magic HTML Header
print &PrintHeader;

&ReadParse;
```

Next, the script determines which buttons were pressed. If a button is pressed, there will be a value for it. If the button was not pressed, the name of the button will not have a value. `$address_op` is set up as a header title based on the value of the button that was pressed. Because any one of the three buttons could have been pressed, all three variables are checked for a value to impart to `$address_op`.

```
$new_address_op = $in{'new_address_op'};
$mod_address_op = $in{'mod_address_op'};
$del_address_op = $in{'del_address_op'};

$address_op = $new_address_op;
if ($address_op eq "") {
    $address_op = $mod_address_op;
}
if ($address_op eq "") {
    $address_op = $del_address_op;
}
```

Now the HTML header is printed using the `HERE DOCUMENT` method.

Chapter 15: SQL Database Address Book

```
print <<__HTMLHEADER__;  
<HTML><HEAD>  
<TITLE>Address Book Maintenance Change</TITLE>  
</HEAD>  
<BODY BGCOLOR="FFFFFF">  
<FORM ACTION=addr_maint_dofinal.cgi METHOD=POST>  
__HTMLHEADER__
```

Information is printed based on what was entered on the previous form. The script checks the `$new_address_op`, `$mod_address_op`, and `$del_address_op` variables for a value to determine which operation is being performed. `$op` is also set to be equal to a three-letter abbreviation of the current operation. Different instructions are given for the different operations.

The new address operation results in HTML telling the user that the address was inserted, and SQL code for inserting gets sent to the database. The modification and deletion operations result in SQL code sent to the server asking for a list of rows that satisfy the previously entered criteria. The modification operation also adds data entry fields for each address book field, allowing the user to enter the modified information. Recall that Figure 15.9 showed an example of this.

```
if ($new_address_op ne "") {  
    $op = "new";  
    $head = "Inserting New Address";  
} elsif ($mod_address_op ne "") {  
    $op = "mod";  
    $head = "Modifying Address Book";  
} else {  
    $op = "del";  
    $head = "Deleting From Address Book";  
}  
  
print "<H1>$head</H1>\n<HR>";
```

Just as in the other scripts, an array of field names is used to match the form variables to the database fields.



N O T E

.....
The `@field_names` array must have the field names spelled exactly as they are labeled in the address table and the previous HTML input form.
.....

```
@field_names = ("fname", "lname", "phone",  
               "depart", "room");
```

The following section processes the addition operation, assuming that the `$op` variable was set to `new` as a result of addition being chosen on the previous HTML form. A few variables are set up in preparation for this processing. `$filledin` is set to 1. If any variables are left out, `$filledin` indicates this and refuses to enter the address. `$insertlist` and `$value` contain the list of field names and values, respectively, to insert.

```
if ($op eq "new") {  
  
    $filledin = 1;  
  
    $insertlist = "";  
    $value = "";
```

The array of field names is iterated. The `$insertlist` is generated from the field names, and the `$value` is generated through the values that correspond to the form variables on the previous HTML form. If any value does not exist in the form variables, `$filledin` is set to zero.

```
foreach $n (@field_names)  
{  
    $value= $in{$n};  
    if ($value ne "")  
    {  
        $value = $value . "\\"$value\\"";  
        $insertlist = $insertlist . "," . $n ;  
    }  
    else  
    {  
        $filledin = 0;  
    }  
}
```

`$insertlist` and `$value` are then surrounded by parentheses. Also, the first character of both variables is stripped off using the `substr` command, because the addition of the values separated the value and insert list with commas at the beginning of each item and the first comma

Chapter 15: SQL Database Address Book

needs to be removed. Strings and arrays in Perl begin counting at zero, so using an index of 1 with the `substr` command returns the string starting at the second character.

```
$insertlist = "(" . substr($insertlist,1) . " ";
$valueulist = "(" . substr($valueulist,1) . " ";
```

If the `$filledin` flag is equal to zero, an error message is displayed and the program exits.

```
if ($filledin == 0) {
    print "<STRONG> Some Fields Were Not " .
        "Entered! Address" .
        " Not Inserted. </STRONG>";
    exit;
}
else
{
```

The `$sql` code variable is generated as an insert statement referencing the database name stored in `$db_name` and the address table in that database. The `$insertlist` of field names is appended to `$sql` along with the values in `$valueulist`.

```
$sql = "insert $db_name..address\n" .
    $insertlist .
    "\nvalues\n" . $valueulist;
```

If the address of John Doe was inserted and the location was Accounting with a phone number of 555-1222 in room 12B, then `$sql` would contain the following SQL code:

```
insert infobase..address
(fname,lname,phone,depart,room)
values
("John","Doe","555-1222","Accounting","12B")
```

A file is generated that contains the SQL code to be processed by the ISQL program. `$file` is set to the current process ID plus an `.sql` extension. The first line of the file is the logon password. ISQL expects a password to be

entered right away that matches the username entered as a command-line parameter. Finally, the `$sql` code variable and a `go` command to run the `$sql` are added to the file. Then the file is closed.

```
$file = "$$.sql";
open (QUERYFILE, ">$file");
print QUERYFILE "$db_password\n";
print QUERYFILE "$sql\n";
print QUERYFILE "go\n";
close (QUERYFILE);
```

Sybase expects the environmental variables `SYBASE` and `DSQUERY` to be set to the directory path of Sybase and the server name, respectively. These values are in **address.setup**. The command that is used for passing the query is the `ISQL` program located in the **bin** directory under the path specified by the `$db_dir` variable.

Next, the program stored in the `$command` variable is opened using the special pipe method. Normally, opening a filename in Perl opens the file. However, if we append a pipe symbol to the end of the filename, Perl knows to execute the program instead. Then the output from the running program is sent back as file data to the `ISQLPIPE` file handle. In this case, the program expects the returned data to be the SQL results. `ISQL` uses the `-U` parameter to indicate the username of the user who is logging on, and the `-w` parameter to specify the width of each returned line (up to 255 characters). `$file` is provided as input to `ISQL`.

The script then takes the pipe and reads all the results into the `@rows` array. This array is defined for further trouble-shooting and is not actually used. If you wanted to see the output of the `ISQL` program, you could print the contents of the `@rows` array. Because the SQL code being passed to `ISQL` is insert code, we do not expect any query rows to be returned.

If the insertion is successful, `@rows` will contain the following information:

```
Password:
(1 row affected)
```

`Password:` is the password prompt that `ISQL` is expecting. And `(1 row affected)` means that the insertion statement created one row.

Chapter 15: SQL Database Address Book

Finally, the `ISQLPIPE` file handle is closed and the query file containing the SQL code is removed using the `unlink` command. A message is printed telling users that the address was inserted.

```
    $ENV{"SYBASE"} = "$db_dir";
    $ENV{"DSQUERY"} = "$db_server";
    $command = "$db_dir/bin/isql";
    open (ISQLPIPE,
         "$command -U$db_user -w255 <$file |");
    @rows = <ISQLPIPE>;
    close (ISQLPIPE);
    unlink($file);
    print "<STRONG> Address Was Inserted " .
         "Successfully! </STRONG>";
}
```

The rest of the program is dedicated to the modification and deletion operations. For a modification, users are told to select an entry to modify from a generated list and then enter to the new information in data entry fields provided below the list. For a deletion, they are told to select a row to delete. Figure 15.10 is an example of the deletion screen.

```
} elsif ($op eq "mod") {
    print "Select Entry To Modify And Enter " .
         "New Information Below.\n";
} else {
print "Select Entry To Delete.\n" }

print "<P>";
```

If the operation is not an addition, the `where` clause needs to be generated. Then the SQL can be sent to the database server to get a list of addresses to choose from for modification or deletion.

`$where_clause` is a list of the expressions used by the `where` part of the SQL code as criteria to determine which address rows to return. Each field in the `@field_names` array is cycled through. The value is culled from the form variable corresponding to those fields.

```
if ($op ne "new") {
    $where_clause = "";
    $exactsearch = $in{"exactsearch"};
```

```

foreach $n (@field_names)
{
    $value = $in{"$n"};
    if ($value ne "")
    {

```

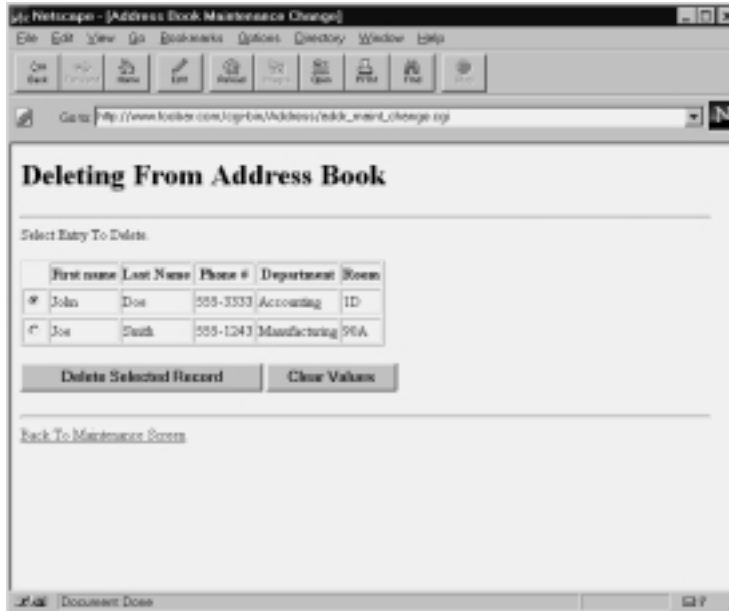


Figure 15.10 An example of the delete address book entry screen.

If `$exactmatch` is not on, the expression takes the same form except that percent (%) symbols are appended to the beginning and end of the search value. The % symbol is roughly equivalent to an asterisk (*) in normal UNIX regular expression matching. When the % symbol is used, the Sybase search will find instances of the value as part of a larger word; the matches found need not be exactly equal to the search value. The lowercase keyword is used to make sure that the SQL query search is case-insensitive.

```

if ($exactsearch ne "on")
{
    $where_clause = qq!$where_clause and ! .
        qq!lower($n) like lower("%$value%")!;
}

```

Chapter 15: SQL Database Address Book

If `$exactmatch` is on and if a value exists for the field, a search expression is generated in the form of `lower(column_name) = lower('value_to_search_for')`.

```
else
{
$where_clause = qq!$where_clause and ! .
    qq!lower($n) = lower("$value")!;
}
}
```

The `$where_clause` has the first five characters stripped off using the `substr` command. This gets rid of the extra " and " appended in the preceding routine.

```
$where_clause = substr($where_clause,5);
```

A `select` statement is generated using the same `join` operator to combine all the elements in the `@field_names` array.

```
$sql = "select ";
$sql .= join(qq! + "\|", "\|" + !,
    @field_names) . qq! + "\|", "\|" + !;
$sql .= join(qq! + "\|", "\|" + !,
    @field_names) . " \n";
```

Notice that the field names are appended twice. In the format variables used later, a duplicate set of values will be mapped to the `%s` symbols in the format variable. One set of values is used to display to the user, and the other set is used to compose the value of the radio button that the users click to select a row.

The `select` list is separated by pipe symbols in the resulting `select` statement. When ISQL outputs the results, it will be easier to pick individual field values when we know that pipes separate them.

Next, the `from` clause (indicating the database name and table name we are selecting from) is appended to the `$sql` variable. The `$where_clause` is also appended to `$sql`.

```
$sql .= "from $db_name..address \nwhere " .
    $where_clause;
```

The header for the table is printed in preparation for the returned rows that result from the SQL code (stored in the `$sql` variable) being sent to the server.

```
print "<TABLE BORDER>\n";
print <<__TABLEHEADER__>;
<TABLE BORDER>
<TR><TH></TH>
<TH>First name</TH>
<TH>Last Name</TH>
<TH>Phone #</TH>
<TH>Department</TH>
<TH>Room</TH>
</TR>
__TABLEHEADER__
```

As with the **addr_query_result.cgi** program, format variables are set up with `%s` as a placeholder for field values. The `$format` variable consists of a definition of the radio button used to select an item plus the display of the fields inside `<TD>` HTML table field tags. Because there are only five fields, a `for` loop from 1 to 5 generates this list of values. The definition of `$format` ends with table row `<TR>` tags at its beginning and end.

```
$format = "<TR>\n";
$format .= qq!<TD><INPUT TYPE=radio NAME=rec !;
$format .= qq!VALUE="%s| %s| %s| %s"></TD>\n!;
for (1..5) {
    $format .= "<TD>%s</TD>";
}
$format .= "\n";

$format = "<TR>" . $format . "</TR>\n";
```

A file is generated that contains the SQL code to be processed by ISQL. `$file` is set to the current process ID plus an **.sql** extension. The first line of the file is the logon password. ISQL expects a password to be entered right away that matches the username entered as a command-line parameter. Finally, the `$sql` code variable and a `go` command to run the `$sql` are added to the file. Then the file is closed.

```
$file = "Temp/$$.sql";
open (QUERYFILE, ">$file");
print QUERYFILE "$db_password\n";
```

Chapter 15: SQL Database Address Book

```
print QUERYFILE "$sql\n";
print QUERYFILE "go\n";
close (QUERYFILE);
```

The next few lines of code, as before, set up the call to the database server command-line utility ISQL.

```
$ENV{"SYBASE"} = "$db_dir";
$ENV{"DSQUERY"} = "$db_server";
$command = "$db_dir/bin/isql";
open (ISQLPIPE,
      "$command -U$db_user -w255 <$file |");
```

The `$rowcount` is set to zero to start. Then the `ISQLPIPE` file handle is read line by line to get the results of the SQL query. For each line that contains pipes, `$rowcount` is incremented. Remember, previously the program embedded pipe symbols as part of the `select` clause of the SQL code being sent to the database server.

Before `$rowcount` is incremented, the regular expressions serve to eliminate the spaces between the pipe symbols. That way, the values that are parsed out will consist only of the values rather than values that are padded with spaces. We set `@field_values` to the resulting values by splitting the current line by its pipe symbols. Because each field begins and ends with a pipe symbol, fields are separated here using two pipe symbols. This is why the `split` command takes two pipe symbols instead of one. Also, the regular expressions and the `split` command are operating from the default Perl `$_` variable. This is why those commands do not explicitly state a variable name.

```
$rowcount = 0;
while (<ISQLPIPE>) {
    if ($_ =~ /\|/) {
        s/\| *\\|\\|\\|/g;
        s/^ *//;
        $rowcount++;
        @field_values = split(/\|\\|/);
```

Finally, Perl's `printf` function is used to print the table rows. `printf` takes all the elements in the `@field_values` list and maps them to the `%s` symbols previously defined in the `$format` string.

```
    printf ($format, @field_values);
} # End of if
} # End of while
```

Now the program closes the pipe to the ISQL program and removes the file containing the generated SQL code using the `unlink` command. Then the closure for the table (`</TABLE>`) is printed.

```
close (ISQLPIPE);
unlink ("file");

print "</TABLE><P>\n";
} #End of IF OP is not insert a new record
```

If the operation is a modification, then a table of `<INPUT>` fields to modify is displayed. The user can then enter new information into any field related to the chosen row in the list of address book entries.

```
if ($op eq "mod")
{
print <<__ENDOFTABLE__>;
<TABLE>
<tr>
<td align=right>First Name:</td>
<td><input name=fname></td>
</tr>
<tr>
<td align=right>Last Name:</td>
<td><input name=lname></td>
</tr>
<tr>
<td align=right>Phone:</td>
<td><input name=phone></td>
</tr>
<tr>
<td align=right>Department:</td>
<td><input name=depart></td>
</tr>
<tr>
<td align=right>Room:</td>
<td><input name=room></td>
</tr>
</TABLE>
__ENDOFTABLE__

} # End of IF op eq mod
```

Finally, the HTML footer is printed along with submit buttons appropriate to the operation being performed. If the operation is a modification, a submit button with the value **Update Selected Record** is displayed. If the operation is a deletion, a button that states **Delete Selected Record** is shown. No submit buttons are displayed for the addition operation; this script has already performed the addition, and there is no further CGI script to submit information to.

The modification and deletion operations have a **Clear Values** reset button. In addition, the footer contains a reference to the main address maintenance script.

```
if ($op eq "mod") {
    print "<p><input type=submit " .
        "name=mod_address_op" .
        " value=\"Update Selected Record\">\n";
} elsif ($op eq "del") {
    print "<p><input type=submit " .
        "name=del_address_op" .
        " value=\"Delete Selected Record\">\n";
}

if (($op eq "mod") || ($op eq "del")) {
    print qq!<input type=reset ! .
        qq!value="Clear Values"><p>!; }

# print the HTML footer.
print <<__HTMLFOOTER__>>
<HR>
<A HREF=addr_maint.cgi>
Back To Maintenance Screen</A>
<P>
</FORM>
</BODY></HTML>
__HTMLFOOTER__
```

addr_maint_dofinal.cgi

addr_maint_dofinal.cgi is the final script that is called in the maintenance part of this application for modifications and deletions to the database. If the operation being performed is deletion, simple delete SQL code is generated. If the operation is modification, update SQL code is generated.

The following code declares the path to the shared library files for the application. **cgi-lib.pl** is loaded from the library area. By default, this is the current directory.

```
$lib = ".";
require "$lib/cgi-lib.pl";
```

Then the **address.setup** configuration parameters are read into the script. `PrintHeader` is called to tell the Web server that the CGI program is outputting HTML. `ReadParse` reads the incoming form variables, using the `%in` associative array to store the values.

```
require "address.setup";
print &PrintHeader;

&ReadParse;
```

If the user pressed the button on the HTML form corresponding to the `<INPUT>` field name `mod_address_op`, `$mod_address_op` contains a value. Similarly, `$del_address_op` will have a value if the user pressed the button corresponding to the deletion of the address book row.

```
$mod_address_op = $in{'mod_address_op'};
$del_address_op = $in{'del_address_op'};
```

The HTML header is printed using the `HERE DOCUMENT` method. It informs the user that an update to the database is about to occur.

```
print <<__HTMLHEADER__;
<HTML><HEAD>
<TITLE>Address Book Final Update</TITLE>
</HEAD>
<BODY BGCOLOR="FFFFFF">
__HTMLHEADER__
```

If `$mod_address_op` contains a value, then `$op` is set to `mod` for modification and the `$head` variable is set to a message indicating that we are updating an entry in the address book. Otherwise, the script assumes that the user is performing a deletion. In this case, `$op` is set to `del` and `$head` is set to a

Chapter 15: SQL Database Address Book

message stating that an entry is being deleted from the address book. The header is then printed as HTML code.

```
if ($mod_address_op ne "") {
    $op = "mod";
    $head = "Updating Entry In Address Book";
} else {
    $op = "del";
    $head = "Deleting From Address Book";
}

print "<H1>$head</H1>\n<HR>";
```

As in `addr_query_result.cgi`, the `@field_names` array contains a list of the field names for the address table as it exists on the database server. These field names must also match the field names in the various HTML forms that process the address book.

```
@field_names = ("fname", "lname", "phone",
                "depart", "room");
```

If the operation is a modification, the update query is generated. In preparation for this, two variables are set up. `$filledin` keeps track of whether any fields were entered for updating. `$updatelist` keeps track of the expression used to update the table.

The `@field_names` array is iterated. If a field has a value from the previous form, `$updatelist` is added to and the `$filledin` variable is set to 1, indicating that at least one field was entered. When the iteration is complete, `$updatelist` is chopped to get rid of the last comma in the values that are entered for `$updatelist`.

Next, if the `$filledin` variable is still zero, the program knows that none of the fields had a value entered into it for updating. In this case, the program prints an error and the `$erroroccurred` flag is set to 1, preventing the update query from being sent to ISQL later.

```
if ($op eq "mod") {

    $filledin = 0;
    $updatelist = "";
```

```
foreach $n (@field_names)
{
$value= $in{"$n"};
  if ($value ne "")
  {
    $updatelist .= qq!$n = "$value",!;
    $filledin = 1;
  }
}
# Get rid of the last comma from the
# appending of the values above
chop($updatelist);

  if ($filledin == 0) {
    print "<STRONG> No Fields Were Entered!" .
      " Address" .
      " Not Updated. </STRONG><p>";
    $erroroccured = 1;
  }

} # end of if operator is update
```

The value in the `rec` form variable contains the list of values that are used as criteria to determine which row to update or delete. Remember that the `rec` form variable is a radio button that the client uses to choose the row to delete or update. The value of the `rec` form variable is a pipe-delimited set of values for the various database fields that we filled in previously using `printf` with an extra set of `%s` symbols for the radio button definition.

The value list is then `split` via the pipes into an `@value_list` array. If the `@value_list` is not generated, an error message is sent and the `$erroroccured` flag is set to 1.

```
$value = $in{"rec"};

@value_list = split(/\|/, $value);

if (@value_list == 0)
{
  print "<STRONG> No Record Was Selected! " .
    "</STRONG><p>";

$erroroccured = 1;
}
```

Chapter 15: SQL Database Address Book

Next, the `$where_clause` is generated. In this program, there are five fields in the `@field_names` array. The script uses a `for` loop to process these fields from element number zero through 4. Remember that arrays start counting at zero. The `where` clause is generated by taking the field name values and matching them to the values in the `rec` form variable.

```
$where_clause = "";
# Make the where clause based on the number of fields
for (0..4)
{
    $value = $value_list[$_];
    $key = $field_names[$_];
    $where_clause =
        qq!$where_clause and $key = "$value"!;
}
```

At the end of the `where` clause generation, an extra `"` and `"` exists in the `$where_clause` variable. This is stripped off using the `substr` operator.

```
$where_clause = substr($where_clause,5);
```

If the operation is a `mod` for modification, the `$sql` code variable is set to an update clause referencing the database name in `$db_name` and the address table. The `$updatelist` is also appended to set the values in the update statement.

If the operation is not a `mod`, the script assumes that a deletion is occurring and makes `$sql` equal to a delete clause, referencing the database name in `$db_name` and the address table in that database.

To `$sql`, we append the `where` clause, restricting which row will be updated or deleted based on the radio button that the user selected on the previous HTML form.

```
if ($op eq "mod") {
    $sql = "update $db_name..address\n set " .
        $updatelist . "\n ";
} else {
    $sql = "delete $db_name..address ";
}

$sql .= "where $where_clause\n";
```

If no errors have occurred, the program will generate the SQL code file. The filename is specified as the current process ID with an `.sql` extension. This arrangement makes the file unique in case other instances of this script are running at the same time.

```
if ($erroroccured == 0) {
    $file = "Temp/$$$.sql";
    open (QUERYFILE, ">$file");
```

The first line of the file is the logon password. ISQL expects a password to be entered right away that matches the username entered as a command-line parameter. Finally, the `$sql` code variable and a `go` command to run the `$sql` are added to the file. Then the file is closed.

```
print QUERYFILE "$db_password\n";
print QUERYFILE "$sql\n";
print QUERYFILE "go\n";
close (QUERYFILE);
```

The following block of code has been used before (in `addr_maint_change.cgi`) to insert a new record. The discussion of that script explains in detail why the code is used.

```
$ENV{"SYBASE"} = "$db_dir";
$ENV{"DSQUERY"} = "$db_server";
$command = "$db_dir/bin/isql";
open (ISQLPIPE,
    @rows = <ISQLPIPE>;
close(ISQLPIPE);
unlink($file);
}
```

Finally, the script prints the HTML footer. A reference to the main address book maintenance script is printed as part of the footer.

```
print <<__HTMLFOOTER__>>;

<A HREF=addr_maint.cgi>
Back To Maintenance Screen</A>
<P>
</BODY></HTML>
__HTMLFOOTER__
```

