CHAPTER FOURTEEN

THE DATABASE SEARCH LIBRARY

The Database Search Library (**web_store_db_lib.pl)** is the Web store's interface to databases. By default, this library provides only an interface to ASCII text file databases. However, because this library acts as a gateway to the text file databases, you can easily change this library to access a real database engine inside the subroutines such as mSQL, Sybase, or Oracle. There are only two routines in this library that are called by the main Web store script: **check_db_with_product_id** and **submit_query**. The rest of the routines in this library are merely supporting routines. If you want to modify the library to access another database engine, the only routines you have to change are **check_db_with_product_id** and **submit_query.**

Global Variables

There is one global variable set inside this library: **$sc_db_lib_was_loaded**. It is set to "yes" so that the main Web store script will know that this library was already loaded once and not to load it a second time if there are multiple references to this script inside the Web store. Since the database library routines are not always used in the Web store, this library is only loaded on an as-needed basis for efficiency. However, because the script is loaded on an as-needed basis, we want to make sure that we do not load the script more than once during the course of running the Web store.

The **Require** command itself keeps track of which libraries were loaded and does not load them twice. However, the subroutine that is used by the Web store script to check the permissions and existence of the required file is inefficient to run multiple times. Thus, the **$sc_db_lib_was_loaded** flag to indicates to the script whether the library of routines is already in use.

```
$sc_db_lib_was_loaded = "yes";
```

check_db_with_product_id Subroutine

The **check_db_with_product_id** subroutine is used by the Web store to make sure that the price and other information about the product in the database matches the information about the item that the customer is ordering. This is really a security check to make sure that the customer has not found a way to hack into the HTML interface and change the pricing and other vital information about the item. This routine merely returns the contents of the database row to the calling subroutine. The Web store itself does the field comparisons once the row has been retrieved from the database matching the product i.d.

The routine is called with two parameters: **$product_id** and ***db_row**. **$product_id** is the unique product i.d. number of the row that needs to be retrieved. The contents of the row are passed back to **@db_row**. Because **@db_row** was passed to the routine by reference using the * operator, changes to **@db_row** inside the subroutine are returned back to the original caller of this routine:

```
sub check_db_with_product_id {
  local($product_id, *db_row) = @_;
  local($db_product_id);
```

The first thing that the script does is open the data file. If the open file call fails, the **file_open_error** routine will record the error:

```
open(DATAFILE, "$sc_data_file_path") ||
    &file_open_error("$sc_data_file_path",
      "Read Database",__FILE__,__LINE__);
```

Each line in the data file is read into the variable **$line**. Then, the line is split into fields that are subsequently placed in **@db_row**. If it turns out that the

product i.d. matches the product i.d. in the database row, the while loop will stop and **@db_row** will contain the row matching the product i.d. The file is then closed:

```
while (($line = <DATAFILE>) &&
         ($product_id ne $db_product_id)) {
    @db_row = split(/\|/,$line);
    $db_product_id = $db_row[0];
  }

  close (DATAFILE);
```

Finally, the subroutine returns the status of the database search. This is done by testing the Boolean expression "is the given product i.d. equal to the product i.d. of the last row that was retrieved from the database." If this statement is true, then the value of "TRUE" is returned from the subroutine. If this statement is false, then the value of "FALSE" is returned from the subroutine. This returned value is used by the script calling this routine to see whether the database row matching the product ID really was found or not:

```
return ($product_id eq $db_product_id);

} # End of check_db_with_product_id
```

submit_query Subroutine

The **submit_query** subroutine is called in order to return the results of a query on the database file. It is called with only one parameter: ***db_rows**. ***db_rows** is actually an array of pipe-delimited (|) rows that satisfy the query results. The * operator means that the variable is passed by reference so that when the subroutine changes the array and populates it with rows, these changes will be reflected back to the original variable that was passed in the calling script:

```
sub submit_query
{
  local(*database_rows) = @_;
```

The next few lines of code declare some variables that will be used in this subroutine but will be invisible outsode of it (local variables). The last two variables (**$exact_match** and **$case_sensitive**) are set equal to the form variables of the same name. Chapter 5 discussed how these form variables affect the behavior of queries involving data types equal to "string" with the equals (=) comparison operator.

```
local($status);
local(@fields);
local($row_count);
local(@not_found_criteria);
local($line); # Read line from database

local($exact_match) = $form_data{'exact_match'};
local($case_sensitive) = $form_data{'case_sensitive'};
```

Since there have been no rows found yet, **$row_count** is initialized to zero:

```
$row_count = 0;
```

The data file is opened for reading. If there is an error while performing this operation, the routine **file_open_error** is called:

```
open(DATAFILE, "$sc_data_file_path") ||
    &file_open_error("$sc_data_file_path",
      "Read Database",__FILE__,__LINE__);
```

Then, every line of the file is read into the **$line** variable using a while loop. As long as the row count does not exceed your configured maximum amount of rows to return (as configured in the Setup file), the data file keeps getting read into **$line** so that the query criteria can be applied to each line:

```
while(($line = <DATAFILE> ) &&
        ($row_count< $sc_db_max_rows_returned+1))
  {
```

Before any processing is done on the line, the newline character at the end is stripped off:

```
chop ($line);
```

Each field is split based on the pipe delimiter (|):

```
@fields = split(/\|/, $line);
```

Now the actual query criteria is applied to the line. First, **$not_found** is initialized to zero. This means that there is nothing "not found" yet. This basically means that we are assuming that the query criteria is satisfied by the row (innocent until proven guilty).

Then, for each criteria specified in the **@sc_db_query_criteria** array from the setup file, the script calls a routine to apply the criteria (**flatfile_apply_criteria**). If the criteria is not satisfied, the routine returns the value **1,** which would increment the **$not_found** variable. Thus, **$not_found** will end up being the number of criteria that were not found. By the end of the foreach loop, if **$not_found** is still zero, this means that the criteria was applied successfully:

```
$not_found = 0;
foreach $criteria (@sc_db_query_criteria)
{
$not_found += &flatfile_apply_criteria(
 $exact_match,
 $case_sensitive,
 *fields,
 $criteria);
    }
```

If **$not_found** is zero, and the row count has not exceeded the maximum amount of rows that are supposed to be returned, then the row is pushed into the **@db_rows** array as a pipe-delimited list of fields:

```
if (($not_found == 0) &&
        ($row_count <= $sc_db_max_rows_returned))
    {
      push(@database_rows, join("\|", @fields));
    }
```

Even if the maximum amount of rows being returned has been exceeded, if **$not_found** is zero, the script increments the row count:

```
if ($not_found == 0) {
      $row_count++;
    }
  } # End of while datafile has data
```

Now that the data file has finished being processed, it is closed by the routine:

```
close (DATAFILE);
```

If the row count exceeds the maximum amount of rows that are allowed to be returned, then the status variable is set equal to an error message related to this fact:

```
if ($row_count > $sc_db_max_rows_returned) {
    $status = "max_rows_exceeded";
  }
```

Finally, the status and row count are returned to the calling script and the subroutine ends:

```
return($status,$row_count);

} # End of submit query
```

flatfile_apply_criteria Subroutine

The **flatfile_apply_criteria** subroutine is not actually called by the main Web store script. However, it is the heart of the logic inside this library. It basically does the actual comparison of the query criteria against the rows in the ASCII text flatfile that forms the default database for the Web store. This routine accepts four parameters: **$exact_match**, **$case_sensitive**, ***fields**, and **$criteria**—which will be examined below.

$exact_match and **$case_sensitive** are the values of the corresponding form variables. ***fields** is an array of database fields that are passed by reference.

Passing an argument by reference is different from passing by value. *Passing by value* means that the variable that is being passed to the subroutine has a new copy of it allocated in memory that is passed on locally to the subroutine. *Passing by reference,* on the other hand, means that a pointer (location of) the variable is passed to the subroutine. Since the location of the original contents of the variable is passed to the subroutine, it allows the subroutine to make changes to the variable and have the changes take effect on the original contents of the variable. In addition, passing by reference is considered more efficient when dealing with large values such as fields from a database, because making a copy of many bytes of data is time-consuming whereas simply passing a reference to the location of a large block of data is less time consuming to perform.

$criteria is the current criteria from the **@sc_db_query_criteria** array that is being applied to this database row.


NOTE

Before reading about what this subroutine does, it is recommended that you go through Chapter 5 in order to get a firm grasp on the logic that is used when applying the criteria defined in the **@sc_db_query_criteria** array.

```
sub flatfile_apply_criteria
{
  local($exact_match, $case_sensitive,
      *fields, $criteria) = @_;
```

$c_name, **$c_fields**, **$c_op**, and **$c_type** are declared local. They will correspond to the fields in the criteria array. The rest of the variables below are declared to be local to the subroutine as well.

```
local($c_name, $c_fields, $c_op, $c_type);
```

@criteria_fields is an array that will hold the index of which fields in the database this criteria will apply to. Recall from the discussion in Chapter 5, that you can define a form field as being able to be matched against more than one field at a time:

```
local(@criteria_fields);
```

$not_found is a flag indicating whether the subroutine found anything:

```
local($not_found);
```

The value for the form field being compared is stored in **$form_value**:

```
local($form_value);
```

The value for the database field currently being compared is stored in **$db_value**:

```
local($db_value);
```

$month, **$year**, **$day**, **$db_date**, and **$form_date** are all date-related variables that are set up for performing date comparisons:

```
local($month, $year, $day);
local($db_date, $form_date);
```

$db_index is a place marker for the current field in the database row that the routine is examining:

```
local($db_index);
```

@word_list is an entire list of words for matching. Keywords entered by the user as search criteria are split into separate words that are also searched for independantly:

```
local(@word_list);
```

The criteria is split into the appropriate variable defined above:

```
($c_name, $c_fields, $c_op, $c_type) =
   split(/\|/, $criteria);
```

Recall that the criteria can match more than one field in the database. Thus, the routine gets the index values of the fields in each row of the database that the form variable will be compared against and places them in the **@criteria_fields** array.

```
@criteria_fields = split(/,/,$c_fields);
```

The value of the form variable that is being compared in the criteria is assigned to **$form_value**:

```
$form_value = $form_data{$c_name};
```

Case 1: Form Variable Contains No Search Value

There are three cases of comparison that will return a value. In the first case, the form field for the criteria was not filled out, so the match is considered a success. Remember, if the user does not enter a keyword, we want the search to be open-ended. The logic used here is that the search is restricted only if the user chooses to enter a search word into the appropriate query field:

```
if ($form_value eq "")
  {
    return 0;
  }
```

Case 2: A Plain Numeric, Date, or String Comparison

In the second case, the data type is a number or a date or if the data type is a string and the operator is not equals (=), then the operator is matched directly based on the data type. Recall from Chapter 5 that a data type equals "string" and the comparison operator is equals (=), then the application of the query criteria becomes more flexible. This is considered case 3and will be discussed further below:

```
if (($c_type =~ /date/i) ||
     ($c_type =~ /number/i) ||
     ($c_op ne "="))
  {
```

$not_found is set to **yes**. In other words, the routine assumes that the data did not match. If any fields do end up matching the data submitted by the user, then the routine will set **$not_found** to **no** later on:

```
$not_found = "yes";
```

For each data field in the **@criteria_fields** array, the routine needs to run the comparison:

```
foreach $db_index (@criteria_fields)
    {
```

The value of the field that is currently being compared is retrieved from the **@fields** array:

```
$db_value = $fields[$db_index];
```

Now, the comparison operators are actually applied to the data. However, the comparison takes on a slightly different flavor for each type of data that can be compared: date, number, string. The first data type that is coded for comparison below is date. Before dates can be compared in Perl, their variables must be rearranged.

In other words, the dates to be compared need to be rearranged so that they form a string of the format "YYYYMMDD," where "YYYY" is a four-digit year, "MM" is a two-digit month, and "DD" is a two-digit day. All of these are padded with zeros if the number does not fill the space. The key trick here is that a date in the form of "YYYYMMDD" can be compared against another date of the same format, using standard numerical operators. Notice that when the date is rearranged this way, numerical comparisons of the dates correspond directly to chronological comparisons. In addition, two-digit years are converted to four-digit years so that this script will not be subject to the "year 2000" problem:

```
if ($c_type =~ /date/i)
      {
```

The first date that is converted is **$db_value**. First, it is split into **$month**, **$day**, and **$year** based on the forward-slash (/) character:

```
($month, $day, $year) =
  split(/\//, $db_value);
```

$month is padded with a zero if it is only one digit long:

```
$month = "0" . $month
  if (length($month) < 2);
```

$day is padded with a zero if it is only one digit long:

```
$day = "0" . $day
  if (length($day) < 2);
```

The year is converted to a four-digit year. Specifically, if the year is greater than 50 but less than 1900, 1900 is added to the year because it is assumed that a year greater than 50 corresponds to the period 1950 to 1999. Otherwise, if the year is still less than 1900, it is assumed that the other options for two-digit years are 2000 through 2049. Of course, if the user entered a year that is already four digits, it is kept the same:

```
if ($year > 50 && $year < 1900) {
  $year += 1900;
}
if ($year < 1900) {
  $year += 2000;
}
```

The two if tests presented above are used separately instead of using a catch-all else statement to add 2000 to the year in place of the second if. This is because there is a possibility that the year has already been entered by the user as a four-digit year. In this case, both if tests should fail and the year will be kept the same.

$db_date is then assigned as the year plus the month plus the day in the "YYYYMMDD" format that was described earlier:

```
$db_date = $year . $month . $day;
```

The form value is processed in the exact same way as the database field value. In the end, **$form_date** is set equal to the form value that has been processed to comply with the "YYYYMMDD" format:

```
($month, $day, $year) =
  split(/\//, $form_value);
$month = "0" . $month
  if (length($month) < 2);
$day = "0" . $day
  if (length($day) < 2);
if ($year > 50 && $year < 1900) {
  $year += 1900;
}
if ($year < 1900) {
  $year += 2000;
}
$form_date = $year . $month . $day;
```

Now, the dates can be compared with simple Perl numeric comparison operators. If any of the operators matches, a zero is returned to let the **submit_query** routine know that a match was found:

```
if ($c_op eq ">") {
  return 0 if ($form_date > $db_date); }
if ($c_op eq "<") {
  return 0 if ($form_date < $db_date); }
if ($c_op eq ">=") {
  return 0 if ($form_date >= $db_date); }
if ($c_op eq "<=") {
  return 0 if ($form_date <= $db_date); }
if ($c_op eq "!=") {
```

```
  return 0 if ($form_date != $db_date); }
if ($c_op eq "=") {
  return 0 if ($form_date == $db_date); }
```

If the data type is a number, then straight numeric comparisons are performed with no additional work being done to the values. A zero is returned if a match was found:

```
} elsif ($c_type =~ /number/i) {
  if ($c_op eq ">") {
    return 0 if ($form_value > $db_value); }
  if ($c_op eq "<") {
    return 0 if ($form_value < $db_value); }
  if ($c_op eq ">=") {
    return 0 if ($form_value >= $db_value); }
  if ($c_op eq "<=") {
    return 0 if ($form_value <= $db_value); }
  if ($c_op eq "!=") {
    return 0 if ($form_value != $db_value); }
  if ($c_op eq "=") {
    return 0 if ($form_value == $db_value); }
```

If the data type is a string, then the Perl string operators are used to compare the values instead of the numeric operators. For example, instead of **>**, the Perl **gt** string operator is used. A zero is returned if any of the comparisons returned a successful match:

```
} else { # $c_type is a string
  if ($c_op eq ">") {
    return 0 if ($form_value gt $db_value); }
  if ($c_op eq "<") {
    return 0 if ($form_value lt $db_value); }
  if ($c_op eq ">=") {
    return 0 if ($form_value ge $db_value); }
  if ($c_op eq "<=") {
    return 0 if ($form_value le $db_value); }
  if ($c_op eq "!=") {
    return 0 if ($form_value ne $db_value); }
}
    } # End of foreach $form_field
```

Case 3: Full String–Based Keyword Search

Now, consider Case 3. Recall that Case 3 handles situations in which the data type is a string and the operator is **=**. This is more complex because a fuzzy search is done based on the rules disclosed in Chapter 5. This fuzzy search checks to see whether we need to do case sensitive matches or whether the comparisons should match against whole words. In other words, the variables **$case_sensitive** and **$exact_match** actually affect the search. In addition, each word entered in the form variable is applied to the database field independently. All the words that the user enters must match against the database fields in order for the search to return a success:

```
} else { # End of case 2, Begin Case 3
```

@word_list is the array of words that the user entered into the form variable:

```
@word_list = split(/\s+/,$form_value);
```

Each field defined for comparison is checked against the keywords entered into the form variable:

```
foreach $db_index (@criteria_fields)
    {
```

The database field value that the routine is currently comparing is placed in **$db_value:**

```
$db_value = $fields[$db_index];
```

Initially, **$not_found** is set equal to **yes**, under the assumption that the search failed. If the search finds a match with the fields, this variable will later be set to **no:**

```
      $not_found = "yes";
```

$match_word acts as a place marker for the words that are going to be looked up in the database row. **$x** is used as a place marker inside the for loops that iterate through the keywords that the user entered into the form variable:

```
local($match_word) = "";
local($x) = "";
```

The following routine is the same code that exists for keyword searching in the **FindKeywords** subroutine discussed in Chapter 13:

```
if ($case_sensitive eq "on") {
    if ($exact_match eq "on") {
        for ($x = @word_list; $x > 0; $x--) {
      # \b matches on word boundary
            $match_word = $word_list[$x - 1];
            if ($db_value =~ /\b$match_word\b/) {
                splice(@word_list,$x - 1, 1);
            } # End of If
        } # End of For Loop
    } else {
        for ($x = @word_list; $x > 0; $x--) {
            $match_word = $word_list[$x - 1];
            if ($db_value =~ /$match_word/) {
                splice(@word_list,$x - 1, 1);
            } # End of If
        } # End of For Loop
    } # End of ELSE
} else {
    if ($exact_match eq "on") {
        for ($x = @word_list; $x > 0; $x--) {
# \b matches on word boundary
            $match_word = $word_list[$x - 1];
            if($db_value =~ /\b$match_word\b/i) {
                splice(@word_list,$x - 1, 1);
            } # End of If
        } # End of For Loop
    } else {
        for ($x = @word_list; $x > 0; $x--) {
            $match_word = $word_list[$x - 1];
            if ($db_value =~ /$match_word/i) {
                splice(@word_list,$x - 1, 1);
            } # End of If
        } # End of For Loop
    } # End of ELSE
}
```

After the keyword search routine, the **foreach** loop goes back to check every database field that was specified in the search criteria:

```
} # End of foreach $db_index
```

If there was nothing left in the keyword list that was being compared, then the routine set **$not_found** to **no**, indicating that there were no "not found" words. In other words, the routine found all the words in the word list. This ends Case 3:

```
if (@word_list < 1)
    {
      $not_found = "no";
    }

  } # End of case 3
```

Finally, if **$not_found** is still equal to **yes**, the routine returns a **1**, indicating that the criteria was not satisfied. Otherwise, a **0** is returned, indicating that the routine found a successful match:

```
if ($not_found eq "yes")
  {
    return 1;
  } else {
    return 0;
  }
} # End of flatfile_apply_criteria
```