CHAPTER FIVE

FOCUS ON THE QUERY-DRIVEN DATABASE STORE

While the HTML version of the store and the product/category-driven store work well for small inventories, both start to break down once a large catalog of items is involved. For example, if you have a catalog of 2500 separate products, maintaining HTML pages for all those products in the HTML-based store can be a nightmare. You could use the database store and let the users select the product category that they wish to see, but then you have to present the user with a lot of categories in order to make any of the lists reasonably short for viewing. Even if 2500 items were broken up into twenty categories, the user would still have to browse through an average of 125 items per category. Thus the natural evolution of Web stores with many items to display is the full Query-driven store, where the user can enter multiple search terms in order to narrow the amount of data retrieved.

The Query-driven store is almost identical to the Database store discussed in the previous chapter. However, in the previous chapter, the query interface was minimal. A Query-driven store allows the user to query on many different fields in many different ways.

Two changes must be done to the Database store configuration to make the Query-driven store possible. First, the frontpage query form must have more search terms added as <INPUT> and <SELECT> tags. Second, the Setup file must be configured to map the new search terms on the front page with the fields in the database. An example of a full query-driven search form appears in Figure 5.1.

Figure 5.1 Query-driven frontpage.

Changes to the Setup File

The changes to the setup file are nearly identical to the changes discussed in Chapter 4 (The standard Database Store). However, with the query driven database store, you must pay more attention to the database query related variables. The setup file **web_store.setup.db.table** contains an example template of a query-driven store. In addition to providing a query interface, this setup file contains minor cosmetic changes designed to make the resulting product list display as a table. A description of the variables that differ in the Query-driven Web store are described below.

Database-Related Changes to the Setup File

@sc_db_query_criteria is an array containing the criteria that defines the logic used to search the database. Each element in this array is a pipe-delimited set of fields. The fields correspond to the possible query search terms. Sample code that defines this variable is shown below from **web_store.setup.db.table.**

```
@sc_db_query_criteria =
("query_price_low_range|2|<=|number",
 "query_price_high_range|2|>=|number",
 "product|1|=|string",
 "keywords|1,2,3,4,5|=|string");
```

The first field of each element in the array is the form variable name that this search term should match against. Remember, search terms are entered by the user on the frontpage form. Thus you will need to come up with a form field name that is used as a particular search term. For example:

```
<INPUT TYPE=text NAME=query_price_low_range>
```

would be a sample form input field corresponding to the first element in the **@sc_db_query_criteria** array shown above.

The second field contains a comma-delimited list of the indexes in the database row that correspond to the database fields the criteria applies to. For instance, in the example above, the values **1,2,3,4,5** would signify that the second, third, fourth, fifth, and sixth fields from the database will be compared against the **keywords** form variable.


NOTE

Remember, fields in PERL start counting at zero. The first field is really field number 0. Thus the second field is referred to as field number 1. The third field is referred to as field number 2. The fourth field is referred to as field number 3. The fifth field is referred to as field number 4 and finally, the sixth field is referred to as field number 5.

In the Setup file, the **%db** associative array can be used to determine which indexes apply to which database field numbers. This use of **%db** was discussed previously in Chapter 2 and Chapter 4.

The third field contains the operator that will be used to compare the form field to the database fields. Possible operators include greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=), equals (=), or not equals (!=). The operators are applied with the form variable on the left-hand side and the database fields on the right hand side. If there is more than one database field specified, then the operator will be applied once to each database field.

For example, if the form variable name was **expiration_date** and the database index for the expiration date was 3, then the equals operator would be applied just as the pseudocode example below shows.

```
if ([EXPIRATION DATE]=[FIELD #3 IN DATABASE)
{
  [WE FOUND A MATCH]
}
```

Finally, the fourth field is the data type of the fields we are comparing. The values for the data type can be date, number, or string—a most important point. For example, if you mistakenly specified a numeric value as a string, and the user types in a number—**30,** for example—while the value in the database is actually **30.00**, a string comparison will reveal that the string **30** is definitely not equal to the string **30.00.** But if you had compared these two strings as numbers, then they would actually match correctly. The number **30** is equal to the number **30.00.** Figure 5.2 graphically illustrates the structure of elements in the query criteria array.

The data type, therefore, can be thought of as affecting how the comparison operators perform their job. If the data type is a number, then the form field and database fields are compared as straight numbers. If the data type is a string, then the fields are compared as strings. If the data types are dates, then they are converted to a date format to know whether one date actually occurs before, after, or is equivalent to another date.

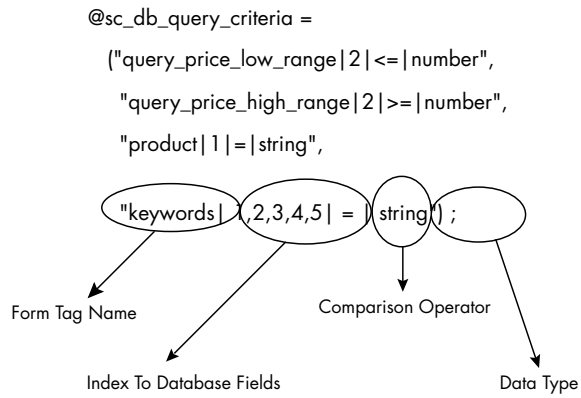

Figure 5.2 Query criteria array structure.

There is one exception to the way operators work. If the data type is a string and the operator is equal (=), then two extra form variables alter the way the string equality is performed: **exact_match** and **case_sensitive.**

By default, string equality matching is *fuzzy*. That is, the match is performed in a case-insensitive manner and the keywords that are typed into the form variable have the ability to satisfy *partial-word matches*—if the keyword matches part of a word in the database, then the match is considered successful.

During string equality matches, the words that are typed as form input are separated by white space (including spaces, tabs, and carriage returns). The Web store script actually considers each word separately and searches on them individually. All the words typed into the form variable must match the database fields in order to consider the match to be successful.

The string equality comparison is case insensitive only if the form variable, **case_sensitive**, is equal to **on**. This generally happens if you have an <INPUT> form variable of TYPE **checkbox** and NAME equal to **case_sensitive** that has been checked on. You can also force case sensitive searches by including a hidden field on the form with a name of **case_sensitive** and the value of on. An example of the HTML code for this tag appears below:

```
<INPUT TYE=checkbox NAME=case_sensitive>
```

Furthermore. the string equality comparison matches only whole words to whole words if the form variable **exact_match** is equal to **on**. Like **case_sensi-**

tive, you will usually place this form variable on the form as an <INPUT> tag of type checkbox such as the following HTML code:

```
<INPUT TYPE=checkbox NAME=exact_match>
```

Finally, it is important to note that the query criteria is additive with regards to narrowing down what the user sees. For example, if there are two separate elements that are being queried on, then a record in the database must successfully satisfy both selection criteria before being considered a totally successful match. In other words, the elements follow the and rule. The first element and any other elements must match together.

Also, only form fields that have values are compared against the query criteria. For example, even if the query criteria has an element that matches the Price field of the database, if the user has not entered a price to match against, then the price will not be considered part of the query. This makes sense because generally we want to give the user the ability to pick and choose what combination of fields on the form will enter into the query results. Furthermore, as the user fills in more and more fields, the query will become additively more and more restrictive.

In summary, the first field is the form variable **NAME**, the second field contains the index to the database fields being compared, the third field is the comparison operator to use, and the fourth field is the data type. In addition, when the query is actually performed on the database, only fields that have been entered by the user enter into the query results. As the user fills in more query fields on the query form, the query becomes more restrictive because more criteria has been entered by the user. The section below goes into the details of possible ways to configure the **@sc_db_query_criteria** variable to suit the most common cases.

Cosmetic Changes to the Setup File

The variables that display the resulting product pages were altered in **web_store.setup.db.table** to display the results of the query using an HTML table view. The three affected variables are **$sc_product_display_header**, **$sc_product_display_footer**, and **$sc_product_display_row**.

$sc_product_display_header is the header of the table The code that configures this variable in the Setup file appears below:

```
$sc_product_display_header = qq!
  <TABLE BORDER = "1">
  <TR>
  <TH>Quantity</TH>
  <TH>%s</TH>
  <TH>%s</TH>
  </TR>!;
```

The table border is set to **1** so that a table border is created. Next, the first table row is defined with **Quantity** plus two other table headers defined as **%s**. The **%s** formats will eventually match the values from **$sc_db_display_fields** (Image (if appropriate), Description).

$sc_product_display_footer is very simple. All it does is close the table. The code for **$sc_product_display_footer** follows:

```
$sc_product_display_footer = qq!
 </TABLE>!;
```

Finally, **$sc_product_display_row** contains information about every row from the database that will be displayed. The code for $sc_product_display_row is defined as follows:

```
$sc_product_display_row = qq~
<TR>
<TD ALIGN = "center"><INPUT TYPE = "text"
             NAME = "item-%s"
             SIZE = "3" MAXLENGTH = "4"></TD>
<TD ALIGN = "center">%s</TD>
<TD>%s<BR>%s</TD>
</TR>~SCC 3 BOT
```

The table column has an input tag generated with the name of **item-** plus **%s**. The first **%s** format string of any database query is always a pipe-delimited list of values that uniquely identify the item and which will define the fields of the item that will go into the cart. This pipe-delimited list is defined by the Setup variable **@sc_db_index_for_defining_item_id**. The variable above has two

more table columns. The first has one database field displayed in it (one **%s**), and the second has two database fields displayed in it (two **%s**) in the format string. The setup variable **@sc_db_index_for_display** tells the Web store script which database fields map against the **%s** when the row is being displayed. In this case, the three fields are **image_url**, **description**, and **options**.

That is all you have to do to make sure that the product pages are viewed as simple, plain HTML tables.

Creating the Queries

This section covers the basic queries that most Web stores are configured to handle. The first queries that are covered are the general keyword search on the whole database row and the product category search, since they are the most common database query terms. The subsequent cases covered will show how to deal with data types and the different types of comparisons that can arise when dealing with different data types.

The following query cases assume that the database is set up as described below, using the **%db** associative array discussed in Chapter 4.

```
$db{"product_id"}           = 0;
$db{"product"}              = 1;
$db{"expiration_date" }     = 2;
$db{"price"}                = 3;
$db{"name"}                 = 4;
$db{"image_url"}            = 5;
$db{"description"}          = 6;
$db{"options"}              = 7;
```

Query on a General Keyword

The first step of any query that you wish to set up is the configuration of query-related form variables. In this case, because we want to do a keyword search, there are three form variables to consider. First, you need to allow users to enter their keywords to search on. Second, you should give users the option to make the search case-sensitive, and third, you should also give users the

option to make the search match whole words only. The following HTML contains a sample definition for these form variables:

```
<INPUT TYPE = "text" NAME = "keywords"
       SIZE = "40"  MAXLENGTH = "40">
<P>
<INPUT TYPE=checkbox NAME="exact_match">
Exact Match Search (Whole Words Only)
<P>
<INPUT TYPE=checkbox NAME="case_sensitive">
Case Sensitive Search
<HR>
```

The next step is to set up the **@sc_db_query_criteria** so that there is one element of the array that corresponds to the keyword query. Recall that you only need one element for matching against the **keywords** form variable. The other two form variables are not actually compared they merely modify how string searches are performed by **web_store.cgi**. Thus, the first field which corresponds to the form variable name will be **keywords**.

The next field must contain a comma-delimited list of all the database fields that a general keyword search should be performed on. Since this is a general keyword search, you will want to match against all the text fields in the database. Thus, this field value should be **0,1,2,3,4,6**. In this example, field number 5 is left out because it is an image URL that should not be searched because an image is a picture, not text. Remember, fields in Perl start counting at 0.

Finally, the operator is set to equal (=) and the data type is set to **string** for doing string-based keyword searches. The resulting **@sc_db_query_criteria** code appears below:

```
@sc_db_query_criteria
     = ("keywords|0,1,2,3,4,6|=|string)
```

Query on a Single Product Category

Another popular search can be done to let the user choose a specific category of the database to browse. In this case, although only one form variable will be used, there are many different ways to present the choice of several different categories to the user.

One possibility is that you may wish to present the user with a static list of HTML hypertext references to a page representing each product category. This would be represented with the following HTML in the front page and is discussed in Chapter 4:

```
<A HREF=web_store.cgi?cart_id=&product=Vowels>
Vowels</A><BR>
<A HREF=web_store.cgi?cart_id=&product=Letters>
Letters</A><BR>
<A HREF=web_store.cgi?cart_id=&product=Numbers>
Numbers</A>
```

Another possibility is that the user may want to select from a drop-down menu of items as part of a **<SELECT>** tag. This option is presented below:

```
<SELECT NAME=product>
<OPTION VALUE=Vowels>Vowels
<OPTION VALUE=Letters>Letters
<OPTION VALUE=Numbers>Numbers
</SELECT>
```

If neither of these methods will be used, it can suffice to have a standard text **<INPUT>** tag to allow the user to enter a product name just like the keyword search that was discussed above.

Once the HTML form has been coded using the above tags, the **@sc_db_query_criteria** is easy to construct. The form variable name is product. There is only one database field that corresponds to the product category. In our example, the database index of the product category is **1**. Finally, just like the keyword search, the data type is **string** and the comparison operator is equals (=). The code below shows a sample of what **@sc_db_query_criteria** will look like:

```
@sc_db_query_criteria
     = (product|1|=|string);
```

Query on an Exact Value

Querying on one specific value, such as the price of the product, is similar to the previous query on the product category. First, you need to create a form input tag in order to allow the user to enter a price. This tag is demonstrated below:

```
<INPUT TYPE = "text" NAME = "product_price"
       SIZE = "10"  MAXLENGTH = "10">
```

Next, the **@sc_db_query_criteria** element for matching the **product_price** is created. The first field in the query criteria element is **product_price**. The second field is **3** because that is the index number of the price field in this database. The third field is equals (=) and the fourth field is **number**, since this query involves a numeric comparison. The code for the **@sc_db_query_criteria** follows:

```
@sc_db_query_criteria
     = (product_price|3|=|number);
```

Query on Price Range

Querying on a price range is a little more complex than checking for one exact value match. Since a range is being compared, this actually involves two separate queries criteria elements. The first checks for the low end of the range, and the second checks for the high end of the range. Thus for checking a price range, you would need to have two form variables as shown below:

```
Lowest Price To Search For:
<INPUT TYPE = "text" NAME = "query_price_low_range"
       SIZE = "10"  MAXLENGTH = "10">
Highest Price To Search For:
<INPUT TYPE = "text" NAME = "query_price_high_range"
       SIZE = "10"  MAXLENGTH = "10">
```

For each of these form variables, there needs to be a separate element in the query criteria array—one for comparing the low range and another for comparing the high range. The sample **@sc_db_query_criteria** is presented below.

```
@sc_db_query_criteria
     = ("query_price_low_range|3|<=|number",
        "query_price_high_range|3|>=|number");
```

For the first element, the first field is **query_price_low_range** and the second field is **3** to match the price. The comparison operator is **<=** and the data type is **number**. Basically, this corresponds to the statement that the value entered into

the **query_price_low_range** form field must be less than or equal to whatever price is in the record for the product in order to produce a successful match. For example, if the user has entered **10.00** into the low range field, then a product with a price of **5.00** will not match successfully but a product with a price of **15.00** would match successfully. This is correct behavior because the user wanted to make sure that the lowest price that matched was **10.00**.

For the second element, the first field is **query_price_high_range,** and the second field is **3** to match the price. The comparison operator is **>=** and the data type is **number**. This corresponds to the statement that whatever the user has entered into the **query_price_high_range** form field should be greater than or equal to the price for the record in the database in order to produce a successful match.

Query on Date Range

Producing a query on the date range follows the same basic logic as producing a query on a price range except that the data type will be **date** instead of **number**. An example of a query date range search would be to allow the user to search on the low and high range of an expiration date. Sample HTML to produce the form variables is described as follows:

```
Lowest Expiration Date To Search For:
<INPUT TYPE = "text" NAME = "query_exp_date_low_range"
       SIZE = "10"  MAXLENGTH = "10"> <BR>
Highest Expiration Date To Search For:
<INPUT TYPE = "text" NAME = "query_exp_date_high_range"
       SIZE = "10"  MAXLENGTH = "10"> <BR>
```

The **@sc_db_query_criteria** would be assigned using the code below:

```
@sc_db_query_criteria
     = ("query_exp_date_low_range|2|<=|date",
     "query_exp_date_high_range|2|>=|date");
```

The first field of each element corresponds to the form variables **query_exp_date_low_range** and **query_exp_high_range**. The second field is **2**, which corresponds to the expiration date. Finally, the comparisons are done

based on the **<=** and **>=** operators with a data type of **date**. One example, is that the user could fill in **12/15/96** as the low range for the expiration date. If the database has a record that has an expiration date of **12/1/96**, then this will not produce a successful match because **12/15/96** is not less than or equal to **12/1/96**. However, if the expiration date for a record in the database is **12/17/96**, then this will produce a successful match because **12/15/96** is less than **12/17/96**.

The date query mechanism built into the Web store relies on the fact that dates must be in the format MM/DD/YY where MM is the month, DD is the day, and YY is the year. The Web store also supports four-digit years as well.

Query on Multiple Text (String) Fields

Querying on many string fields is different from the keyword search because instead of querying all the fields at once, the user is allowed to enter specific keywords to match individual fields. The form is more complex as well because there must be multiple <INPUT> tags corresponding to each database field that is being searched. The HTML below contains sample input tags for allowing the user to search separately on **product_category**, **product_name**, and **product_description**. Also included are field to allow the user to select whether or not the matches in these fields are exact or case sensitive:

```
Product Category:
<INPUT TYPE = "text" NAME = "product_category"
       SIZE = "20"  MAXLENGTH = "20"> <BR>
Product Name:
<INPUT TYPE = "text" NAME = "product_name"
       SIZE = "20"  MAXLENGTH = "20">  <BR>
Product Description:
<INPUT TYPE = "text" NAME = "product_description"
       SIZE = "20"  MAXLENGTH = "20">
<P>
<INPUT TYPE=checkbox NAME="exact_match">
Exact Match Search (Whole Words Only)
<P>
<INPUT TYPE=checkbox NAME="case_sensitive">
Case Sensitive Search
<HR>
```

The **@sc_db_query_criteria** array must have a separate element for each database field being searched. The sample code for assigning **@sc_db_query_criteria** appears below:

```
@sc_db_query_criteria
     = ("product_category|1|=|string",
     "product_name|4|=|string",
     "product_description|6|=|string");
```

Query for Multiple Fields of Different Types

In the final example of query criteria, all the elements of querying are brought together. This example will include a date range, price range, and a multiple-field keyword search. The HTML form must include the capability to allow the user to enter all this criteria. Sample HTML appears below:

```
Product Category:
<INPUT TYPE = "text" NAME = "product_category"
       SIZE = "20"  MAXLENGTH = "20"> <BR>
Product Name:
<INPUT TYPE = "text" NAME = "product_name"
       SIZE = "20"  MAXLENGTH = "20">  <BR>
Product Description:
<INPUT TYPE = "text" NAME = "product_description"
       SIZE = "20"  MAXLENGTH = "20">
<P>
<INPUT TYPE=checkbox NAME="exact_match">
Exact Match Search (Whole Words Only)
<P>
<INPUT TYPE=checkbox NAME="case_sensitive">
Case Sensitive Search<P>
Lowest Expiration Date To Search For:
<INPUT TYPE = "text" NAME = "query_exp_date_low_range"
       SIZE = "10"  MAXLENGTH = "10"> <BR>
Highest Expiration Date To Search For:
<INPUT TYPE = "text" NAME = "query_exp_date_high_range"
       SIZE = "10"  MAXLENGTH = "10"> <BR>
Lowest Price To Search For:
<INPUT TYPE = "text" NAME = "query_price_low_range"
       SIZE = "10"  MAXLENGTH = "10">
```

```
Highest Price To Search For:
<INPUT TYPE = "text" NAME = "query_price_high_range"
       SIZE = "10"  MAXLENGTH = "10">
```

Finally, an **@sc_db_query_criteria** array must be constructed to allow all these form variables to be matched. This job is easy because it involves setting up all the elements discussed previously. The final **@sc_db_query_criteria** array code appears below:

```
@sc_db_query_criteria
= ("product_category|1|=|string",
   "product_name|4|=|string",
   "product_description|6|=|string",
   "query_exp_date_low_range|2|<=|date",
   "query_exp_date_high_range|2|>=|date",
   "query_price_low_range|3|<=|number",
   "query_price_high_range|3|>=|number");
```

Summary

In conclusion, the examples above show that **@sc_db_query_criteria** is a highly flexible variable that allows for a variety of querying to take place on an ASCII text database file. This query mechanism is not available with the HTML version of the Web store. By merely adjusting the form variables and the array, many different ways of viewing the data can be presented to the user—from simple product category matches to full-blown queries based on the data type of each database field. In addition, we introduced how adjusting the **$sc_product_display_row–**related variables in the Setup file allows you to change the look and feel of the Database store so that the products are viewed in a tabular format.