CHAPTER 5

Searching the Catalog

"What are you looking for?" There are no places where you'll hear this question more frequently than in both brick-and-mortar stores and e-commerce stores. Like any other quality web store around, your HatShop will allow visitors to search through the product catalog. You'll see how easy it is to add new functionality to a working site by integrating the new components into the existing architecture.

In this chapter, you will

- Analyze the various ways in which the product catalog can be searched.
- Implement a custom search engine that works with PostgreSQL.
- Write the data and business tiers for the searching feature.
- Build the user interface for the catalog search feature using Smarty componentized templates.

Choosing How to Search the Catalog

As always, there are a few things you need to think about before starting to code. When designing a new feature, you should always begin by analyzing that feature from the final user's perspective.

For the visual part, you'll use a text box in which the visitor can enter one or more words to search for. In HatShop, the words entered by the visitor will be searched for in the products' names and descriptions. The text entered by the visitor can be searched for in several ways:

Exact-match search: If the visitor enters a search string composed of more words, this would be searched in the catalog as it is, without splitting the words and searching for them separately.

All-words search: The search string entered by the visitor is split into words, causing a search for products that contain every word entered by the visitor. This is like the exact-match search in that it still searches for all the entered words, but this time the order of the words is no longer important.

Any-words search: At least one of the words of the search string must find a matching product.

This simple classification isn't by any means complete. The search engine can be as complex as the one offered by modern search engines, which provides many options and features and shows a ranked list of results, or as simple as searching the database for the exact string provided by the visitor.

HatShop will support the any-words and all-words search modes. This decision leads to the visual design of the search feature (see Figure 5-1).

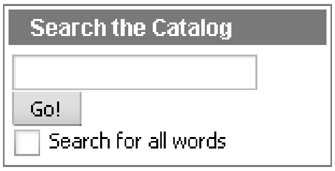
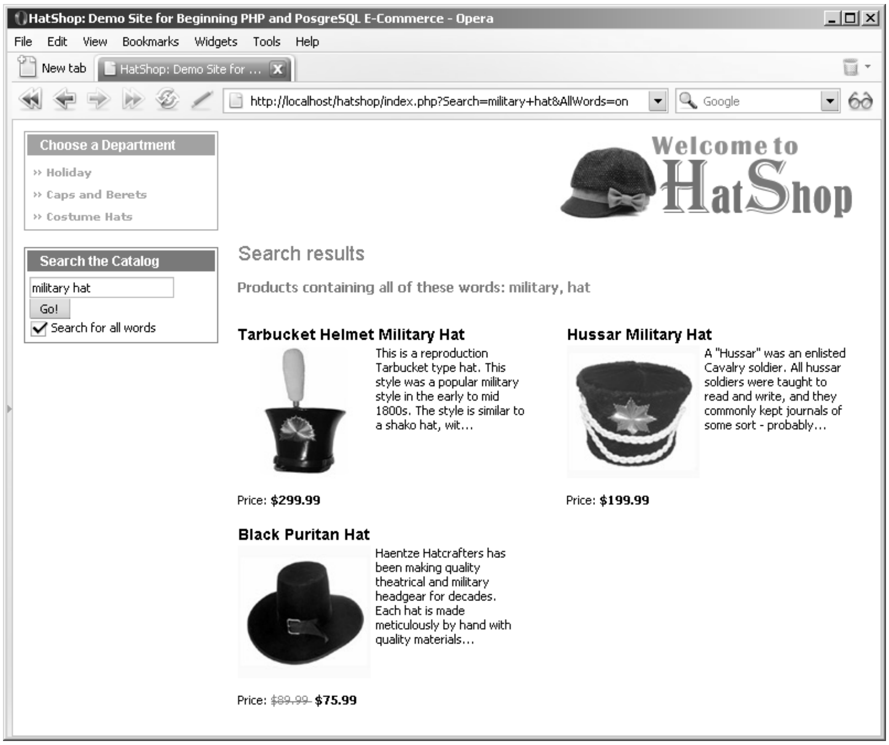


Figure 5-1. *The design of the search feature*

The text box is there, as expected, along with a check box that allows the visitor to choose between an all-words search and an any-words search.

Another decision you need to make here is the way in which the search results are displayed. How should the search results page look? You want to display, after all, a list of products that match the search criteria.

The simplest solution to display the search results would be to reuse the `products_list` componentized template you built in the previous chapter. A sample search page will look like Figure 5-2.


HatShop: Demo Site for Beginning PHP and PosgreSQL E-Commerce - Opera
File Edit View Bookmarks Widgets Tools Help
New tab
HatShop: Demo Site for ...
http://localhost/hatshop/index.php?Search=military+hat&AllWords=on
Google
Choose a Department
» Holiday
» Caps and Berets
» Costume Hats
Welcome to HatShop
Search the Catalog
military hat
Go!
Search for all words

Search results

Products containing all of these words: military, hat

Tarbucket Helmet Military Hat

This is a reproduction Tarbucket type hat. This style was a popular military style in the early to mid 1800s. The style is similar to a shako hat, wit...

Price: **$299.99**

Hussar Military Hat

A "Hussar" was an enlisted Cavalry soldier. All hussar soldiers were taught to read and write, and they commonly kept journals of some sort - probably...

Price: **$199.99**

Black Puritan Hat

Haentze Hatcrafters has been making quality theatrical and military headgear for decades. Each hat is made meticulously by hand with quality materials...

Price: ~~$89.99~~ **$75.99**

Figure 5-2. *Sample search results*

You can also see in the figure that the site employs paging. If there are a lot of search results, you'll only present a fixed (but configurable) number of products per page and allow the visitor to browse through the pages using `Previous` and `Next` links.

Let's begin implementing the functionality, by starting, as usual, with the data tier.

Teaching the Database to Search Itself

You have two main options to implement searching in the database:

- Implement searching using `WHERE` and `LIKE`.
- Search using the `tsearch2` module.

Let's analyze these options.

Searching Using WHERE and LIKE

The straightforward solution, frequently used to implement searching, consists of using `LIKE` in the `WHERE` clause of the `SELECT` statement. Let's take a look at a simple example that will return the products that have the word "war" somewhere in their description:

```
 SELECT name FROM product WHERE description LIKE '%war%'
```

The `LIKE` operator matches parts of strings, and the percent wildcard (`%`) is used to specify any string of zero or more characters. That's why in the previous example, the pattern `%war%` matches all records whose description column has the word "war" somewhere in it. This search is case-insensitive.

If you want to retrieve all the products that contain the word "war" somewhere in the product's name or description, the query will look like this:

```
SELECT name FROM product
WHERE description LIKE '%war%' OR name LIKE '%war%';
```

This method of searching has three important drawbacks:

Speed: Because we need to search for text somewhere inside the description and name fields, the entire database must be searched on each query. This can significantly slow down the overall performance of HatShop when database searches are performed, especially if you have a large number of products in the database.

Quality of search results: This method doesn't make it easy for you to implement various advanced features, such as returning the matching products sorted by search relevance.

Advanced search features: These include searching using the Boolean operators (AND, OR) and searching for inflected forms of words, such as plurals and various verb tenses, or words located in close proximity.

So how can you do better searches that implement these features? If you have a large database that needs to be searched frequently, how can you search this database without killing your server?

The answer is using PostgreSQL's `tsearch2` module.

Searching Using the PostgreSQL tsearch2 Module

tsearch2 is the search module we'll be using to implement our site's search feature. This module ships with PostgreSQL, and it allows performing advanced searches of your database by using special search indexes. Read Appendix A for installation instructions.

There are two aspects to consider when building the data tier part of a catalog search feature:

- Preparing the database to be searched
- Using SQL to search the database

Creating Data Structures That Enable Searching

In our scenario, the table that we'll use for searches is product, because that's what our visitors will be looking for. To make the table searchable using the tsearch2 module, you need to prepare the table to be searched in three steps:

1. Add a new field to the product table, which will hold the *search vectors*. A search vector is a string that contains a prepared (searchable) version of the data you want to be searched (such as product names and descriptions). In our case, the command will look like this (don't execute it now, we'll take care of this using an exercise):

   ```
   ALTER TABLE product ADD COLUMN search_vector tsvector;
   ```

2. Update the product table by adding a gist index on the newly added field. gist is the engine that performs the actual searches, and it is an implementation of the Berkeley Generalized Search Tree (find more details about gist at http://gist.cs.berkeley.edu/). The command for adding a gist index on the product table is

   ```
   CREATE INDEX idx_search_vector ON product USING gist(search_vector);
   ```

3. Populate the search_vector field of product with the search vectors. These search vectors are lists of the words to be searchable for each product. For HatShop, we'll consider the words that appear in the product's name and the product's description, giving more relevance to those appearing in the name. This way, if more products match a particular search string, those with matches in the name will be shown at the top of the results list. At this step, we also filter the so-called *stop-words* (also called noise words), which aren't relevant for searches, such as "the," "or," "in," and so on. The following command sets the search vector for each product using the to_tsvector function (which creates the search vector) and setweight (used to give higher relevance to the words in the name):

   ```
   UPDATE product
   SET    search_vector =
            setweight(to_tsvector(name), 'A') || to_tsvector(description);
   ```

The `'A'` parameter of `setweight` gives highest relevance to words appearing in the product's name. For detailed information about how these functions work, refer to *The tsearch2 Reference* at `http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/docs/tsearch2-ref.html`. You can find a `tsearch2` guide at `http://rhodesmill.org/brandon/projects/tsearch2-guide.html` and an excellent article at `http://www.devx.com/opensource/Article/21674`.

For an example, see Table 5-1, which shows the search vector for the Santa Jester Hat. Note the vector retains the positions of the words (although we don't really need this), and the "A" relevance factor is added to the words from the product's name. Also note that various forms of the same word are recognized (see "fit" for example) and that the stop-words aren't taken into consideration.

Table 5-1. *The Search Vector for a Product*

Field	Value
`product_id`	6
`name`	Santa Jester Hat
`description`	This three-prong velvet jester is one size fits all and has an adjustable touch fastener back for perfect fitting.
`search_string`	`'fit':13,24 'hat':3A 'one':11 'back':21 'size':12 'prong':7 'santa':1A 'three':6 'touch':19 'adjust':18 'fasten':20 'jester':2A,9 'velvet':8 'perfect':23 'three-prong':5`

■ **Note** When adding new products to the table or updating existing products, you'll need to be sure to also (re)create their search vectors. The index that parses these vectors does its job automatically, but the vector itself must be manually created. You'll take care of this in Chapter 7, where you'll add catalog administration features. Until then, if you change your products manually, just execute the previous SQL command to update the search vectors.

Searching the Database

Now that you've built the search vector for each product, let's see how to use it for searching. For performing the search, once again, there are three steps that you need to take:

1. Build a *search string* that expresses what exactly you are looking for. This can contain Boolean operators; we'll use `&` (AND) when doing all-words searches, and `|` (OR) when doing any-words searches.

2. Apply the `to_tsquery` function on the query string. This prepares the query string into a form that can be used for searching.

3. When performing the search, use the condition `search_vector @@ prepared_search_string`, which returns `TRUE` if there's a match and `FALSE` otherwise. Here, `search_vector` is the one calculated earlier (step 3 of the previous section), and `prepared_search_string` is the result of step 2.

Let's see how this would be applied in practice. The following query performs an all-words search on the "yankee war" search string:

```
SELECT   product_id, name
FROM     product
WHERE    search_vector @@ to_tsquery('yankee & war')
ORDER BY product_id;
```

With the sample products database, this query should have the results shown in Table 5-2.

Table 5-2. *Hats That Match "yankee & war"*

product_id	name
40	Civil War Union Slouch Hat
44	Union Civil War Kepi Cap

To perform an any-words search, you should use `|` instead of `&` in the search string:

```
SELECT   product_id, name
FROM     product
WHERE    search_vector @@ to_tsquery('yankee | war')
ORDER BY product_id;
```

As expected, this time you'll have more matching products as shown in Table 5-3 (because the list is unsorted, you may get these results in different order).

Table 5-3. *Hats That Match "yankee | war"*

product_id	name
26	Military Beret
30	Confederate Civil War Kepi
33	Uncle Sam Top Hat
38	Confederate Slouch Hat
40	Civil War Union Slouch Hat
41	Civil War Leather Kepi Cap
44	Union Civil War Kepi Cap

Sorting Results by Relevance

The previous queries show matching products without ordering them in any particular order. The database engine will simply return the results in whatever order it finds easier. For searches, we're interested in showing the more relevant matches first. Remember that we gave higher priority to matches from the product titles, so everything is set.

The `tsearch2` engine offers the rank function that can be used for ordering the results. The default order is to show the lower ranking matches first, so you'll also need to use the `DESC` option of `ORDER BY` to put the better matches at the top.

The following query performs a ranked any-words search for "yankee war":

```
SELECT   rank(search_vector, to_tsquery('yankee | war')) as rank, product_id, name
FROM     product
WHERE    search_vector @@ to_tsquery('yankee | war')
ORDER BY rank DESC;
```

This time, the results will come ordered. You can also see the search rankings. The products that have matches in the name have significantly higher ranks.

Table 5-4. *Search Results Ordered by Rank*

rank	product_id	name
0.341959	40	Civil War Union Slouch Hat
0.33436	44	Union Civil War Kepi Cap
0.31684	41	Civil War Leather Kepi Cap
0.303964	30	Confederate Civil War Kepi
0.0379954	26	Military Beret
0.0303964	38	Confederate Slouch Hat
0.0303964	33	Uncle Sam Top Hat

You should be ready now to implement your web site's search functionality. To learn more about the inner workings of the `tsearch2` engine, consult its official documentation.

Exercise: Writing the Database Searching Code

1. Load pgAdmin III, and connect to the `hatshop` database.
2. Click Tools ➤ Query Tools (or click the SQL button on the toolbar). A new query window should appear.
3. Write the following code in the query tool, and then execute it by pressing F5. This command prepares the product table to be searched using the `tsearch2` engine, as explained earlier in this chapter.

```
-- Alter product table adding search_vector field
ALTER TABLE product ADD COLUMN search_vector tsvector;

-- Create index for search_vector field in product table
CREATE INDEX idx_search_vector ON product USING gist(search_vector);

-- Update newly added search_vector field from product table
UPDATE product
SET    search_vector =
         setweight(to_tsvector(name), 'A') || to_tsvector(description);
```

4. Use the Query tool to execute this code, which creates the `catalog_flag_stop_words` function into your `hatshop` database:

```
-- Create catalog_flag_stop_words function
CREATE FUNCTION catalog_flag_stop_words(TEXT[])
RETURNS SETOF SMALLINT LANGUAGE plpgsql AS $$
  DECLARE
    inWords ALIAS FOR $1;
    outFlag SMALLINT;
    query   TEXT;
  BEGIN
    FOR i IN array_lower(inWords, 1)..array_upper(inWords, 1) LOOP
      SELECT INTO query
             to_tsquery(inWords[i]);
      IF query = '' THEN
        outFlag := 1;
      ELSE
        outFlag := 0;
      END IF;
      RETURN NEXT outFlag;
    END LOOP;
  END;
$$;
```

5. Use the Query tool to execute this code, which creates the `catalog_count_search_result` function into your `hatshop` database:

```
-- Function returns the number of products that match a search string
CREATE FUNCTION catalog_count_search_result(TEXT[], VARCHAR(3))
RETURNS INTEGER LANGUAGE plpgsql AS $$
  DECLARE
    -- inWords is an array with the words from user's search string
    inWords    ALIAS FOR $1;

    -- inAllWords is 'on' for all-words searches
    -- and 'off' for any-words searches
    inAllWords ALIAS FOR $2;

    outSearchResultCount INTEGER;
    query                TEXT;
    search_operator      VARCHAR(1);
  BEGIN
    -- Initialize query with an empty string
    query := '';
    -- Establish the operator to be used when preparing the search string
    IF inAllWords = 'on' THEN
      search_operator := '&';
    ELSE
      search_operator := '|';
    END IF;

    -- Compose the search string
    FOR i IN array_lower(inWords, 1)..array_upper(inWords, 1) LOOP
      IF i = array_upper(inWords, 1) THEN
        query := query || inWords[i];
      ELSE
        query := query || inWords[i] || search_operator;
      END IF;
    END LOOP;

    -- Return the number of matches
    SELECT INTO outSearchResultCount
           count(*)
    FROM   product,
           to_tsquery(query) AS query_string
    WHERE  search_vector @@ query_string;
    RETURN outSearchResultCount;
  END;
$$;
```

6. Use the query tool to execute this code, which creates the `catalog_ search` function into your `hatshop` database:

```
-- Create catalog_search function
CREATE FUNCTION catalog_search(TEXT[], VARCHAR(3), INTEGER, INTEGER, INTEGER)
RETURNS SETOF product_list LANGUAGE plpgsql AS $$
  DECLARE
    inWords                          ALIAS FOR $1;
    inAllWords                       ALIAS FOR $2;
    inShortProductDescriptionLength ALIAS FOR $3;
    inProductsPerPage                ALIAS FOR $4;
    inStartPage                      ALIAS FOR $5;
    outProductListRow product_list;
    query             TEXT;
    search_operator   VARCHAR(1);
    query_string      TSQUERY;
  BEGIN
    -- Initialize query with an empty string
    query := '';
    -- All-words or Any-words?
    IF inAllWords = 'on' THEN
      search_operator := '&';
    ELSE
      search_operator := '|';
    END IF;

    -- Compose the search string
    FOR i IN array_lower(inWords, 1)..array_upper(inWords, 1) LOOP
      IF i = array_upper(inWords, 1) THEN
        query := query||inWords[i];
      ELSE
        query := query||inWords[i]||search_operator;
      END IF;
    END LOOP;
    query_string := to_tsquery(query);

    -- Return the search results
    FOR outProductListRow IN
      SELECT   product_id, name, description, price,
               discounted_price, thumbnail
      FROM     product
      WHERE    search_vector @@ query_string
      ORDER BY rank(search_vector, query_string) DESC
      LIMIT    inProductsPerPage
      OFFSET   inStartPage
    LOOP
      IF char_length(outProductListRow.description) >
         inShortProductDescriptionLength THEN
        outProductListRow.description :=
          substring(outProductListRow.description, 1,
                    inShortProductDescriptionLength) || '...';
      END IF;
      RETURN NEXT outProductListRow;
    END LOOP;
  END;
$$;
```

How It Works: The Catalog Search Functionality

In this exercise, you created the database functionality to support the product searching business tier logic. After adding the necessary structures as explained in the beginning of the chapter, you added three functions:

- **`catalog_flag_stop_words`:** As mentioned earlier, some words from the search string entered by the visitor may not be used for searching, because they are considered to be noise words. The `tsearch2` engine removes the noise words by default, but we need to find what words it removed, so we can report these words to our visitor. We do this using `catalog_flag_stop_words`, which will be called from the `FlagStopWords` method of the business tier.
- **`catalog_count_search_result`:** This function counts the number of search results. This is required so that the presentation tier will know how many search results pages to display.
- **`catalog_search`:** Performs the actual product search.

Implementing the Business Tier

The business tier of the search feature consists of two methods: FlagStopWords and Search. Let's implement them first and discuss how they work afterwards.

Exercise: Implementing the Business Tier

1. The full-text search feature automatically removes words that are shorter than a specified length. You need to tell the visitor which words have been removed when doing searches. First, find out which words are removed with the FlagStopWords method. This method receives as parameter an array of words and returns two arrays, one for the stop-words, and the other for the accepted words. Add this method to your Catalog class, located in business/catalog.php:

```
// Flags stop words in search query
public static function FlagStopWords($words)
{
  // Build SQL query
  $sql = 'SELECT *
          FROM catalog_flag_stop_words(:words);';
  // Build the parameters array
  $params = array (':words' => '{' . implode(', ', $words) . '}');
  // Prepare the statement with PDO-specific functionality
  $result = DatabaseHandler::Prepare($sql);

  // Execute the query
  $flags = DatabaseHandler::GetAll($result, $params);

  $search_words = array ('accepted_words' => array (),
                         'ignored_words' => array ());

  for ($i = 0; $i < count($flags); $i++)
    if ($flags[$i]['catalog_flag_stop_words'])
      $search_words['ignored_words'][] = $words[$i];
    else
      $search_words['accepted_words'][] = $words[$i];

  return $search_words;
}
```

2. Finally, add the Search method to your Catalog class:

```
// Search the catalog
public static function Search($searchString, $allWords,
                              $pageNo, &$rHowManyPages)
{
  // The search results will be an array of this form
  $search_result = array ('accepted_words' => array (),
                          'ignored_words' => array (),
                        'products' => array ());

    // Return void result if the search string is void
    if (empty ($searchString))
      return $search_result;

    // Search string delimiters
    $delimiters = ',.; ';
    // Use strtok to get the first word of the search string
    $word = strtok($searchString, $delimiters);
    $words = array ();

    // Build words array
    while ($word)
    {
      $words[] = $word;
      // Get the next word of the search string
      $word = strtok($delimiters);
    }

    // Split the search words in two categories: accepted and ignored
    $search_words = Catalog::FlagStopWords($words);
    $search_result['accepted_words'] = $search_words['accepted_words'];
    $search_result['ignored_words'] = $search_words['ignored_words'];

    // Return void result if all words are stop words
    if (count($search_result['accepted_words']) == 0)
      return $search_result;

    // Count the number of search results
    $sql = 'SELECT catalog_count_search_result(:words, :all_words);';
    $params = array (
      ':words' => '{' . implode(', ', $search_result['accepted_words']) . '}',
      ':all_words' => $allWords);
    // Calculate the number of pages required to display the products
    $rHowManyPages = Catalog::HowManyPages($sql, $params);
    // Calculate the start item
    $start_item = ($pageNo - 1) * PRODUCTS_PER_PAGE;

    // Retrieve the list of matching products
    $sql = 'SELECT *
            FROM   catalog_search(:words,
                                  :all_words,
                                  :short_product_description_length,
                                  :products_per_page,
                                  :start_page);';
    $params = array (
        ':words' => '{' . implode(', ', $search_result['accepted_words']) . '}',
        ':all_words' => $allWords,
        ':short_product_description_length' => SHORT_PRODUCT_DESCRIPTION_LENGTH,
        ':products_per_page' => PRODUCTS_PER_PAGE,
        ':start_page' => $start_item);

      // Prepare and execute the query, and return the results
      $result = DatabaseHandler::Prepare($sql);
      $search_result['products'] = DatabaseHandler::GetAll($result, $params);
      return $search_result;
    }
```

How It Works: The Business Tier Search Method

The main purpose of the `FlagStopWords` method is to analyze which words will and will not be used for searching.

The full-text feature of PostgreSQL automatically filters the words that are less than four letters by default, and you don't interfere with this behavior in the business tier. However, you need to find out which words will be ignored by PostgreSQL so you can inform the visitor.

The `Search` method of the business tier is called from the presentation tier with the following parameters (notice all of them except the first one are the same as the parameters of the data tier `Search` method):

- `$searchString` contains the search string entered by the visitor.
- `$allWords` is "on" for all-words searches.
- `$pageNo` represents the page of products being requested.
- `$rHowManyPages` represents the number of pages.

The method returns the results to the presentation tier in an associative array.

Implementing the Presentation Tier

The catalog-searching feature has two separate interface elements that you need to implement:

- A componentized template named `search_box`, whose role is to provide the means to enter the search string for the visitor (refer to Figure 5-1).
- A componentized template named `search_results`, which displays the products matching the search criteria (refer to Figure 5-2).

You'll create the two componentized templates in two separate exercises.

Creating the Search Box

Follow the steps in the exercise to build the `search_box` componentized template, and integrate it into HatShop.

Exercise: Creating the search_box Componentized Template

1. Create a new template file named `search_box.tpl` in the `presentation/templates` folder, and add the following code to it:

```
{* search_box.tpl *}
{load_search_box assign="search_box"}
{* Start search box *}
<div  class="left_box" id="search_box">
  <p>Search the Catalog</p>
  <form action={"index.php"|prepare_link:"http"}>
    <input maxlength="100" id="Search" name="Search"
           value="{$search_box->mSearchString}" size="23" />
    <input type="submit" value="Go!" /><br />
    <input type="checkbox" id="AllWords" name="AllWords"
    {if $search_box->mAllWords == "on" } checked="checked" {/if}/>
      Search for all words
  </form>
</div>
{* End search box *}
```

2. Create a new Smarty function plugin file named `function.load_search_box.php` in the `presentation/smarty_plugins` folder with the following code in it:

```
<?php
// Plugin functions inside plugin files must be named: smarty_type_name
function smarty_function_load_search_box($params, $smarty)
{
  // Create SearchBox object
  $search_box = new SearchBox();

  // Assign template variable
  $smarty->assign($params['assign'], $search_box);
}

// Manages the search box
class SearchBox
{
  // Public variables for the smarty template
  public $mSearchString = '';
  public $mAllWords = 'off';

  // Class constructor
  public function __construct()
  {
    if (isset ($_GET['Search']))
      $this->mSearchString = $_GET['Search'];
      if (isset ($_GET['AllWords']))
        $this->mAllWords = $_GET['AllWords'];
    }
  }
  ?>
```

3. Add the following styles needed in the `search_box` template file to the `hatshop.css` file:

```
#search_box
{
  border: 1px solid #0583b5;
}
#search_box p
{
  background: #0583b5;
}
form
{
  margin: 2px;
}
input
{
  font-family: tahoma, verdana, arial;
  font-size: 11px;
}
```

4. Modify the `index.tpl` file to load the newly created template file:

```
...
      {include file="departments_list.tpl"}
      {include file="$categoriesCell"}
      {include file="search_box.tpl"}
...
```

5. Load your project in a browser, and you'll see the search box resting nicely in its place (refer to Figure 5-1).

How It Works: The search_box Componentized Template

By now, you're used to the way we use function plugins in conjunction with Smarty templates. In this case, we use the plugin to maintain the state of the search box after performing a search. When the page is reloaded after clicking the `Go!` button, we want to keep the entered string in the text box and also maintain the state of the `AllWords` check box.

The `load_search_box` function plugin simply saves the values of the `Search` and `AllWords` query string parameters, while checking to make sure these parameters actually exist in the query string. These values are then used in the `search_box.tpl` Smarty template to recreate the previous state.

Note that we could have implemented this functionality by reading the values of the `Search` and `AllWords` query string parameters using `$smarty.get.Search` and `$smarty.get.AllWords` instead of a plugin. However, having a plugin gives you more control over the process and also avoids generating warnings in case the mentioned parameters don't exist in the query string.

Displaying the Search Results

In the next exercise, you'll create the componentized template that displays the search results. To make your life easier, you can reuse the `product_list` componentized template to display the actual list of products. This is the componentized template that we have used so far to list products for the main page, for departments, and for categories. Of course, if you want to have the search results displayed in another format, you must create another user control.

You'll need to modify the templates-logic file of the products list (`products_list.php`) to recognize when it's being called to display search results, so it calls the correct method of the business tier to get the list of products.

Let's create the `search_result` template and update the templates-logic file of the `products_list` componentized template in the following exercise:

Exercise: Creating the search_results Componentized Template

1. Create a new template file in the `presentation/templates` directory named `search_results.tpl`, and add the following to it:

```
{* search_results.tpl *}
<p class="title">Search results</p>
<br />
{include file="products_list.tpl"}
```

2. Modify the `presentation/smarty_plugins/function.load_products_list.php` file by adding the following lines at the end of the constructor method of the `ProductList` class (`__construct`):

```
    // Get search details from query string
    if (isset ($_GET['Search']))
       $this->mSearchString = $_GET['Search'];

    // Get all_words from query string
    if (isset ($_GET['AllWords']))
       $this->mAllWords = $_GET['AllWords'];
```

3. Add the `$mSearchResultsTitle`, `$mSearch`, `$mAllWords`, and `$mSearchString` members to the `ProductsList` class, located in the same file:

```
class ProductsList
{
  // Public variables to be read from Smarty template
  public $mProducts;
  public $mPageNo;
  public $mrHowManyPages;
  public $mNextLink;
  public $mPreviousLink;
  public $mSearchResultsTitle;
  public $mSearch = '';
  public $mAllWords = 'off';
  public $mSearchString;
```

```
// Private members
private $_mDepartmentId;
private $_mCategoryId;
```

4. Modify the `init` method in `ProductsList` class like this:

```
public function init()
{
  /* If searching the catalog, get the list of products by calling
     the Search busines tier method */
  if (isset ($this->mSearchString))
  {
    // Get search results
    $search_results = Catalog::Search($this->mSearchString,
                                      $this->mAllWords,
                                      $this->mPageNo,
                                      $this->mrHowManyPages);
    // Get the list of products
    $this->mProducts = $search_results['products'];
    // Build the title for the list of products
    if (count($search_results['accepted_words']) > 0)
      $this->mSearchResultsTitle =
        'Products containing <font class="words">'
        . ($this->mAllWords == 'on' ? 'all' : 'any') . '</font>'
        . ' of these words: <font class="words">'
        . implode(', ', $search_results['accepted_words']) .
        '</font><br />';
    if (count($search_results['ignored_words']) > 0)
      $this->mSearchResultsTitle .=
        'Ignored words: <font class="words">'
        . implode(', ', $search_results['ignored_words']) .
        '</font><br />';
    if (!(count($search_results['products']) > 0))
      $this->mSearchResultsTitle .=
        'Your search generated no results.<br />';
  }
  /* If browsing a category, get the list of products by calling
     the GetProductsInCategory business tier method */
  elseif (isset ($this->_mCategoryId))
    $this->mProducts = Catalog::GetProductsInCategory(
          $this->mCategoryId, $this->mPageNo, $this->mrHowManyPages);
...
```

5. Add the following lines in the beginning of `presentation/templates/products_list.tpl`, just below the `load_products_list` line:

```
{* products_list.tpl *}
{load_products_list assign="products_list"}
{if $products_list->mSearchResultsTitle != ""}
  <p class="description">{$products_list->mSearchResultsTitle}</p>
{/if}
```

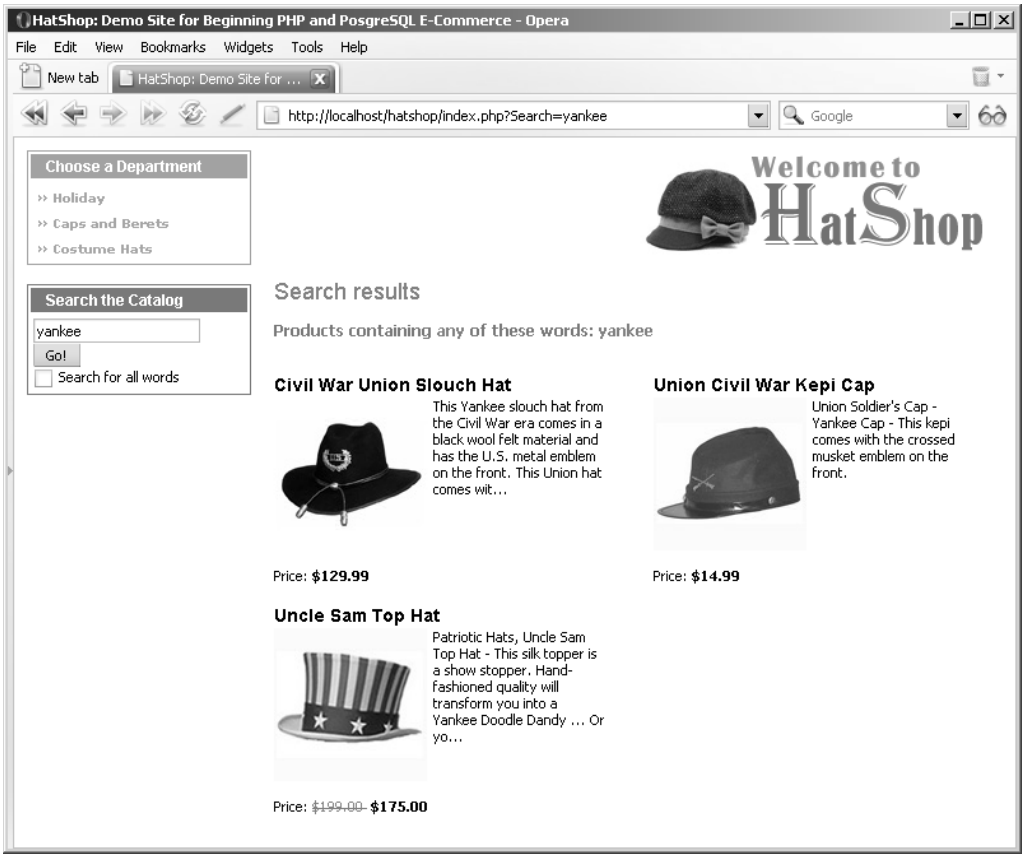
6. Modify the `index.php` file to load the `search_results` componentized template when a search is performed by adding these lines:

```
...
// Load department details if visiting a department
if (isset ($_GET['DepartmentID']))
{
  $pageContentsCell = 'department.tpl';
  $categoriesCell = 'categories_list.tpl';
}

// Load search result page if we're searching the catalog
if (isset ($_GET['Search']))
  $pageContentsCell = 'search_results.tpl';

// Load product details page if visiting a product
if (isset ($_GET['ProductID']))
  $pageContentsCell = 'product.tpl';
...
```

7. Add the following style to the `hatshop.css` file:

```
.words
{
  color: #ff0000;
}
```

8. Load your project in your favorite browser and type **yankee** to get an output similar to Figure 5-3.


HatShop: Demo Site for Beginning PHP and PosgreSQL E-Commerce - Opera
File Edit View Bookmarks Widgets Tools Help
New tab
HatShop: Demo Site for ...
Google
Choose a Department
» Holiday
» Caps and Berets
» Costume Hats
Search the Catalog
Go!
Welcome to HatShop

http://localhost/hatshop/index.php?Search=yankee

yankee

Search for all words

Search results

Products containing any of these words: yankee

Civil War Union Slouch Hat

This Yankee slouch hat from the Civil War era comes in a black wool felt material and has the U.S. metal emblem on the front. This Union hat comes wit...

Price: **$129.99**

Union Civil War Kepi Cap

Union Soldier's Cap - Yankee Cap - This kepi comes with the crossed musket emblem on the front.

Price: **$14.99**

Uncle Sam Top Hat

Patriotic Hats, Uncle Sam Top Hat - This silk topper is a show stopper. Hand-fashioned quality will transform you into a Yankee Doodle Dandy ... Or yo...

Price: ~~$199.00~~ **$175.00**

Figure 5-3. *Sample HatShop search results page*

How It Works: The Searchable Product Catalog

Congratulations, you have a searchable product catalog! There was quite a bit to write, but the code wasn't very complicated, was it?

Because you've used much of the already existing code and added bits to the already working architecture, there weren't any surprises. The list of products is still displayed by the `products_list` template you built earlier, which is now updated to recognize the `Search` element in the query string, in which case it uses the `Search` method of the business tier to get the list of products for the visitor.

The `Search` method of the business tier returns a `SearchResults` object that contains, apart from the list of returned products, the list of words that were used for searching and the list of words that were ignored (words shorter than a predefined number of characters). These details are shown to the visitor.

Summary

In this chapter, you implemented the search functionality of HatShop by using the full-text searching functionality of PostgreSQL. The search mechanism integrated very well with the current web site structure and the paging functionality built in Chapter 4. The most interesting new detail you learned in this chapter was about performing full-text searches with PostgreSQL. This was also the first instance where the business tier had some functionality of its own instead of simply passing data back and forth between the data tier and the presentation tier.

In Chapter 6, you'll learn how to sell your products using PayPal.