DATABASE ARTICLES RSS

Creating a Search Application

By: Matt Wade

Rating: / 9

2003−07−15

In this tutorial, we will discuss a method of searching web pages for keywords. We will provide methods for exact keyword matching and a type of fuzzy search. Also included in this tutorial is a overview of using databases in PHP and a heavy concentration on classes.

In this tutorial, we will discuss a method of searching web pages for keywords. We will provide methods for exact keyword matching and a type of fuzzy search. Fuzzy searching enables us to find words that are similar to the keyword we searched for. This is extremely useful when the search for an exact keyword does not produce any results. In fact, in our search script we will only utilize fuzzy searching when we do not get results from a normal keyword search.

In order to create the search application, we will need to tackle four distinct tasks. First, we need to create the tables in the database. This is where we will store the information we will search. Second, we will need to gather the keywords and store them in the tables we created. Next, we will develop an exact keyword−matching search. Finally, we will develop a way to do fuzzy searching.

Before we build our search application, however, we need to look an aspect of PHP that we have not touched upon yet. In order to make our search application effective, we will need to utilize some the database functionality of PHP. In this next section will introduce you to these concepts and build a database class that will be used for our search application.

Databases provide us with a way to store data and easily retrieve any that data later. Information is stored in structures called tables. Each table is made up of columns and rows. This can be conceptually visualized as a spreadsheet, though the functionality is very different. Each column is defined to hold a particular data type and is given a name. Rows actually hold the data. For the sake of simplicity, we will only cover the use of the MySQL database server. MySQL is also the preferred database within the PHP community as it has a free license just as PHP does and it is Open Source. MySQL is also a very common feature provided by web hosting companies. Using other database servers is very similar, so the concepts learned here can apply to those as well.

For the purposes of this section, we will assume that you already have a database to work with and an associated username and password for that database. This information should be provided to you by your hosting provider or system administrator. Throughout this section, we will use a database name of test and a username and password of username and password, respectively.

The first step in dealing with databases is to create the tables, or structure, you need on the database server. For this step, we will assume that you have access to a tool like phpMyAdmin where you can issue SQL statements to your database server. It is well beyond the scope of this tutorial to explain how tables are created and designed. Therefore, for our purposes, we will just provide you with SQL statements to create any tables that are necessary.

For this introduction to databases, we will work with a table named introduction that can be created with the following SQL statement. We will assume that this table is created in a database named test.

```
CREATE TABLE 'introduction' (
  'id' INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
  'last_name' VARCHAR( 50 ) NOT NULL ,
  'first_name' VARCHAR( 50 ) NOT NULL
) TYPE=MyISAM COMMENT='Example Table';
```

This table has three columns named id, last_name, and first_name. The column named id is an auto increment column. This means that as data is inserted into this table, the id column will automatically be assigned a number that is unique because it will be assigned an id value that is one greater than that of the current highest−numbered id value in that table. This feature does not exist in all database servers, but it does in MySQL. The other two columns will each hold a string value that is up to 50 characters in length.

MySQL provides several different types of tables. We will be using the MyISAM table type as specified by the TYPE declaration. As you can see, you can also specify a comment about a table when you create it. This simply allows for a description to be placed on the table for later reference.

After this table is created, it will not contain any data, or rows. We will need to populate it with data, but first we need to establish a connection to the database server in PHP.

In order to connect to a database server, you need to know the hostname of the server, a username, and a password. The hostname is generally localhost because the database server normally resides on the same machine you are running your PHP code on. The following statement will make a connection to a database server.

```
<?php
$db = mysql_connect ('localhost', 'username', 'password')
      or die ("Unable to connect to Database Server");
?>
```

The function mysql_connect() makes the connection to the database server and returns a resource id, which we are storing in the variable $db. The resource id is an identifier to PHP that references the database connection. This resource id can be used in other MySQL functions to differentiate between multiple database connections.

One thing you may not recognize is the use of the die function. The die function halts the execution of the script and displays the message passed to it. If we were to translate the above PHP code into English, we would say, "Make a connection to the database server 'localhost', using a username of 'username' and a password of 'password', and store an identifier for that connection in the variable $db. If you are not able to make the connection, stop executing this script and display the message 'Unable to connect to Database Server'.

Now that we have established a connection to the database server, our next task is to select the database we want to work with.

A database server can contain many databases. Each database is a container for tables. The reason we would use multiple databases is to keep data for different applications, or users, separate from one another. As we mentioned earlier, we will assume that you are storing your tables in a database named test.

The function we will use to select the database we want to work with is mysql_select_db(). Its syntax is very simple as

demonstrated by this code.

```
<?php
mysql_select_db ('test', $db)
      or die ("Could not select database");
?>
```

The first parameter is the database we want to use and the second is the resource id from the mysql_connect() function. The second parameter is not mandatory if you are only dealing with one database, but it is good practice to use it. By always specifying the resource id, you can avoid complications when you start programming applications that utilize more than one database.

Now that we have connected to our database server and selected the database we want to work with, it's time to put some data into our table and also examine how to retrieve that data.

At this point, our table, named introduction, is completely empty. The only way it is going to get any data in it is for us to populate it. We will accomplish this by sending an SQL query to the database via the mysql_query() function.

The mysql_query() function will allow us to send any SQL query to the database. So, not only can we insert data into the database with it, but we can also select, delete, and update information. Before we can do any of the last three operations, we need to insert something into the database. So, in order to insert a single row into the database, we would use the following mysql_query() statement.

```
<?php
$result = mysql_query("INSERT INTO `introduction` "
                      . "VALUES ('', 'Smith', 'John')", $db)
    or die("Invalid query: " . mysql_error());
?>
```

This statement will insert a row into the table with the value of 'Smith' for the last_name column and 'John' for the first_name column. You will notice that we passed an empty string for the id column as the database server will automatically assign this value. The mysql_query() function will return TRUE on success and FALSE on error. We will see in a moment that it can also return a resource identifier under certain circumstances. However, it will always return either TRUE or FALSE for insert, update, and delete statements. Please pay special attention to the fact that the characters on either side of the table name, introduction, are back tick characters. Using single quotes will not work!

There is another new function introduced in the statement above. The mysql_error() function returns the last error message from the database server. In the case that our query fails, the die function will stop execution of the script and display the error message as returned by the mysql_error() function.

That was a simple insert statement that stored one row of data in the table. What if we need to store multiple rows of data in the table at one time? Well, we could run several insert queries like the one we just demonstrated, or we can actually expand the insert statement to store more than one row at a time. This is done by adding more values to the end of the statement. Each row of data is placed within parentheses and added to the end of the insert statement, with each set of values separated by a comma. The following is an example of how we would insert three rows at one time.

```
<?php
$result = mysql_query("INSERT INTO `introduction` "
                      . "VALUES ('', 'Harkey', 'Rob'), "
                      . "('', 'Wade', 'Michelle'), "
                      . "('', 'Gary', 'Amanda')", $db)
    or die("Invalid query: " . mysql_error());
?>
```

Now that we have some data in the table, let's see how we would retrieve it from the database. We will use the mysql_query() function as before, but we will now have to introduce some other functions to work in conjunction with it. First, we will issue a basic select statement to the database like so:

```
<?php
$result = mysql_query ("SELECT * FROM introduction", $db)
    or die ("Invalid query: " . mysql_error());
?>
```

The SQL statement issued will select all rows and columns from the table introduction. This SQL statement will have mysql_query() return a resource identifier associated with the result set of the query which will be stored in the variable $result. We then can use another function named mysql_fetch_assoc() to retrieve a row from the result set as an associative array. The mysql_fetch_assoc() function will return an associative array with the column names serving as the keys of the array. Basic usage of this function would be as follows:

```
<?php
$row = mysql_fetch_assoc ($result);
?>
```

After this statement executes, the variable $row would contain an associative array with three elements. An element would correspond to each of the columns in the table introduction. Now, we can display the data from this row by simply echoing out each element.

```
<?php
echo "{$row['id']}&lt;br /&gt;\n";
echo "{$row['last_name']}, ";
echo "{$row['first_name']}&lt;br /&gt;\n";
?>
```

This would display:

```
1
Smith, John
```

Commonly, you will see the mysql_fetch_assoc() function used as the condition in a while loop. This enables us to process all the rows in a result set. The following example shows how this is done.

```
<?php
while ($row = mysql_fetch_assoc ($result)) {
    echo "{$row['id']}&lt;br /&gt;\n";
    echo "{$row['last_name']}, ";
    echo "{$row['first_name']}&lt;br /&gt;\n";
}
?>
```

It is worth noting that it is not necessary to retrieve the columns in the order in which they are stored in the table. We could have displayed the id, last name, and first name in any order that we wished.

The mysql_fetch_assoc is just one function that you can use to retrieve rows from a result set. Others include mysql_fetch_array, which will return an array with both enumerated and associative indexed, and mysql_fetch_row, which returns only a enumerated array.

Now that we have the basics of working with databases covered, let's build a PHP class to handle our database needs for our search application. So that you may have a better understanding of what a class is, we will define it as a "container for data and functions that work with that data."

Why would we want to write a database class? Well, it can benefit us in a couple of ways. One, it makes other code we write easier to read and follow. By taking care of all the database needs in a class, we can cut down on the amount database code in our search routines. This will allow us to more closely follow what is happening in the search application, without having all the database code getting in the way. Secondly, a database class can be expanded in the future. It gives you a single place to update the database functionality. For an example, if you later wanted to port the search application to another type of database server, you would only need to update the database class. Without a class, you would need to modify all the database code throughout the search application.

Now that we have established why we want to create a database class, let's get started. The name of our database class will be DB_Class. Let's take a look at the functionality the class will provide, and then we will present the class itself.

- **Constructor** − The constructor of the class bears the same name as the class and is called when a new instance of the class is created. In our constructor, we will make the connection to the database server and select the proper database. The constructor will accept a database name, username, and password as parameters.
- **query Function** − This function will be named query and we will be used to run any SQL statement given to it. It will return a resource id. This function will be used by other functions of the class and it will also be called from outside the class.
- **fetch Function** − This function, named fetch, will accept a string containing a SQL statement and will return an array containing the results from the query.
- **getone Function** − The getone function will return the value of the first column on the first row return by the SQL statement passed to it. If the SQL statement does not yield any results, it will return FALSE.

We will now take a look at the class itself. We will not spend time examining the class in detail as it is made up of functions and concepts we have already covered.

```
<?php
class DB_Class {

    var $db;

    function DB_Class($dbname, $username, $password) {
        $this->db = mysql_connect ('localhost', $username, $password)
            or die ("Unable to connect to Database Server");

        mysql_select_db ($dbname, $this->db)
            or die ("Could not select database");
    }

    function query($sql) {
        $result = mysql_query ($sql, $this->db)
            or die ("Invalid query: " . mysql_error());
        return $result;
    }

    function fetch($sql) {
        $data = array();
        $result = $this->query($sql);
        while($row = mysql_fetch_assoc($result)) {
            $data[] = $row;
        }
        return $data;
    }

    function getone($sql) {
        $result = $this->query($sql);
        if(mysql_num_rows($result) == 0)
            $value = FALSE;
        else
            $value = mysql_result($result, 0);
        return $value;
    }

}
?>
```

Example use of the database class:

```
<?php
$db = new DB_Class('test', 'username', 'password');
$data = $db->fetch("SELECT * FROM introduction");
?>
```

This concludes our brief introduction to using databases. We strongly recommend that you pick up a book exclusively on the subject of PHP and databases as there is much more to learn on the subject.

By this point, you should have a general understanding of how databases work. This overview of databases was necessary because we will be using them extensively in our search application. You may be wondering why we would use a database rather than a flat file to store our data. The reason is that we will be storing quite a bit of data and will need to perform regular searches on that data. The overhead involved to accomplish this with flat files would overly complicate our search application and likely slow it down.

Now, let's move on to the search application itself and talk a little bit about how it will work. We will continue building this application using classes as we have started to do with the database class. By using classes, we are able to move the logic of the searching out of the main script and provide an interface that anyone can use regardless of whether they know they internal workings of the searching class or not. This allows someone implementing a search using this search application to worry about displaying the results rather than how they were obtained.

Our search application will have two main scripts. The first, called harvest.php, will allow us to specify URLs that we would like to collect keywords from to build our index. In a nutshell, we will grab the source for the supplied URLs and break it into individual words. Each of those words will then be stored in the database along with a reference to what URL it was found on. If a word is found multiple times on a single URL, we will store it multiple times. This will allow us to return search results that have a relevancy factor that correlates to how often the search terms appear on a page.

The second script in our search application will be named search.php. In this script we will provide a form where keywords can be entered for searching. The script will first try and provide exact keyword matching. If exact matching is not possible, it will attempt to find similar words in the database by utilizing the similar_text function of PHP.

As we discussed earlier in the tutorial, we have four tasks we need to accomplish in order to create a search application.

- **Create Database Tables** − Set up the tables where we will store the keywords and URLs.
- **Harvest Keywords** − Before we can search anything, we need to gather the keywords and store them. We will develop a basic script to harvest keywords from given pages.
- **Exact Keyword Search** − The first portion of our search script will provide exact keyword matching. Pages with a greater number of matches will be returned first.
- **Fuzzy Searching** − If the exact keyword search provides no results, we will then search for similar words and return pages that contain them.

For this search application, we will need to have two tables. One will hold each URL that we have gathered keywords from. The other will store each keyword and a reference to the URL it is associated with. Both tables are very simple with only two columns a piece. For the purposes of this tutorial, we will assume you are storing these tables in a database named test.

The urls table

The urls table will have two columns. The first is a unique id that we will use in the keywords table to associate each keyword back to its URL. The second column is the URL itself. We have created a primary key on this table on the id column. This will ensure that this column does not have any duplicate values in it. We have also added an index on the url column named url_idx. This will speed up queries on this column for our search application.

```
CREATE TABLE urls (
  id int(10) unsigned NOT NULL auto_increment,
  url varchar(100) NOT NULL default '',
  PRIMARY KEY  (id),
  KEY url_idx (url)
```

```
) TYPE=MyISAM COMMENT='Table to hold URLs';
```

The keywords table

The keywords table also only has two columns. The first holds the keyword, and the second is the id of the URL it is associated with. We have created two indexes on this table. The first, named keyword_idx, is an index on the keyword column. It will allow for faster searching on that column. The second index, called url_id_idx, will speed up our queries when we later join the two tables together.

```
CREATE TABLE keywords (
  url_id int(11) NOT NULL default '0',
  keyword varchar(100) NOT NULL default '',
  KEY keyword_idx (keyword),
  KEY url_id_idx (url_id)
) TYPE=MyISAM COMMENT='Table to store keywords';
```

Once you have those tables created in your database, you are ready to move on to the script that will gather the keywords.

To gather keywords for our search application, we will provide a simple form that will accept the URLs of the websites to be searched. This will be effective for small sites where all the URLs are known. Large sites with dynamic URLs would be better served by a spider script. Writing an effective spider is well beyond the scope of this document and will be left for an exercise to the reader. For reference, you might want to look at the PHPDig project as they have nice spider built into their application.

The harvest.php script itself will not have much PHP code in it. The reason for this is that the code to perform the harvesting functions will be in a class that is stored in a file named harvestclass.php. Let's take a high level overview of the Harvest_Keywords class and then we will examine each function in detail.

- **Declare class variables** − The first order of business in the class is to declare the variables that will be used to store information that will be shared throughout the class.
- **Constructor** − The constructor of this class will first establish a database connection utilizing the database class we created earlier. Then it will store, as an array, the URL(s) that were entered.
- **_prune** − This function is a private function of the class. A private function is one that is not called from outside the class. PHP does not provide a way to truly make a function private, but we will precede any function we deem to be private with an underscore. This just serves to document them as private for our later reference. We will use this function in conjunction with the array_walk function of PHP to remove unwanted words from the keywords we gather.
- **_checkURL** − Also a private function, this will run a perform a very basic validation of a URL.
- **_getData** − We will use this private function to take care of the details of opening a connection to a URL and obtaining the source.
- **_harvest** − This function is where the real work is done. This private function will call the other functions of this class to gather all the keywords from a single URL and store them in a database.
- **process** − This function is will iterate through each URL provided to it and call the _harvest function to gather keywords.

We will declare four class variables. Each variable will be a private variable to the class, so we will precede each name with an underscore character. As we touched on a moment ago, PHP does not actually provide functionality to create public and private members of a class. We will name our variables and functions to reflect that they are public or private however to demonstrate good programming technique. By naming your class members appropriately, you create self

documenting code.

We will create a variables to store our database resource, the URLs provided to the class, words that should not be included in the index, and words that should always be included in the index. Below are their declarations.

```
<?php
var $_db;
var $_urlarray;
var $_stopwords = array ('and', 'but', 'are', 'the');
var $_allowwords = array ('c++', 'ado', 'vb');
?>
```

The constructor of the harvesting class will perform two tasks. One, it will connect to the database by utilizing the database class we created earlier. Secondly, it will take a string containing the URLs and separate them and place them in an array. The first step of this is to use the trim function to remove any excess newlines or spaces from the beginning and end of the data. Then, we will use the explode function to break the data down into an array. The resulting array will be stored in the class variable $_urlarray.

```
<?php
function Harvest_Keywords($urls) {
    $this->_db = new DB_Class('test', 'username', 'password');
    $this->_urlarray = trim ($urls);
    $this->_urlarray = explode ("\n", $this->_urlarray);
}
?>
```

It is in this function that you must change the database values to reflect your personal settings.

When gathering keywords in the manner we are doing, there will always be words that you do not want in your index. Any word that is under three characters in length is not normally something we want to search on. We also do not want to include things that contain odd symbols and are not really words.

To eliminate unwanted words, we will use the array_walk() function to examine each element of the $words array. The _prune() function of our class will be supplied to array_walk(). The first thing this function does is to convert the keyword to all lowercase. This is done for consistency so that we always know the case of the words in our index.

Then, the function determines if a word is under three characters, or if it contains odd symbols. We also check to see if the word is in the class variable named $_stopwords. The $_stopwords array contains words that we do not want to index. Common examples would be and, but, are, and the. These are words that would make it through the other checks but are so common that indexing them in most circumstances would be useless.

The function also checks to see if the word exists in the $_allowwords class variable. Words in this array are words that would normally be pruned out, but we would like to include in the index. Examples would be c++, and vb.

If a word does not meet our criteria, we will use the unset() function to remove it from the array. This function destroys a given variable so that it is no longer set.

If the word does pass the tests, we strip any characters that are not alphanumeric, a literal single quote, or a dash. We then run the word through the addslashes() function to escape any single quotes.

```
<?php
function _prune (&amp;$item, $key, $array) {
    $item = strtolower ($item);
    if (((preg_match ("/[^a-z0-9'\?!-]/", $item))
     || (strlen ($item) &lt; 3)
     || (in_array($item, $this->_stopwords)))
     &amp;&amp; (!in_array($item, $this->_allowwords))) {

         unset($array[$key]);
    } else {
        $item = addslashes(preg_replace("/[^a-z0-9'-]/i",
                                        '', $item));
    }
}
?>
```

This function is a very simple check of the URL to see if it appears to have the correct form. We will use this function prior to attempting to connect to any URL.

```
<?php
function _checkURL($url) {
    return preg_match ("/http:\/\/(.*)\.(.*)/i", $url);
}
?>
```

In this function we will take care of obtaining the source from a URL. We will open a connection to the URL with the fopen() function and then read up to 25,000 bytes of the source with the fread() function. Once we have obtained the source from the URL, we will run it through the strip_tags() function to remove any HTML tags. We will also replace any   entities with a literal space character. This is so we can split the words on spaces later on.

If we are unable to open a connection to the URL, the function will return FALSE, otherwise it will return the source.

```
<?php
function _getData($url) {
    $filehandle = @fopen($url, 'r');
    if(!$filehandle) {
        echo "Could not open URL ($url).&lt;br /&gt;\n";
        $return = FALSE;
    } else {
        $data = fread($filehandle, 25000);
        fclose($filehandle);
        $data = strip_tags ($data);
        $data = str_replace('&amp;nbsp;', ' ', $data);
```

```
        $return = $data;
    }
    return $return;
}
?>
```

Here is where all the real work is being done. In this function we will collect the keywords for one URL and store them in the database. We start off by calling the _checkURL() function to determine the validity of the URL, and then get the source of the URL with the _getData() function.

Next we take the string that contains the source and split it into individual words and store them in an array. We can do this fairly easily with the preg_split() function. We will split the string at every occurrence of a white space character, a comma, or a period.

Then, we will use the array_walk() function and have it call the _prune() function for each array element. You may notice that the array_walk() function call is a little different than you have seen it in the past. For the second parameter, we have to pass it an array that contains the $this pointer as the first element and the name of the function as the second element. This is needed because we are calling a class function.

After the array_walk() function completes its task, we then use the sort function on the array of words. We are not so concerned about actually sorting the array, but we want to force our numerical keys to be sequential. After the array_walk() function finishes, it is very likely that we will have gaps in our enumerated array. As an example, we could have keys 0, 1, and 2 and then it might skip to key 6. In order to renumber our numerical keys, we can simply run the array through the sort function.

The next step we need to take is to insert the URL into the urls table. We should first look to see if it already exists, and if it does delete the keywords associated with it in the keywords table. Doing this will allow us to refresh the information in our database from time to time. If we did not check for the existence of the URL, we could end up with the same URL indexed multiple times.

The final step of the _harvest() function is to insert the keywords into the keywords table. Because we will have a variable number of keywords and those keywords will be ever changing, we need to construct the SQL query dynamically.

We will accomplish this by using the count() function to determine how many words are in the $words array and then adding each word to a variable called $values. We will add the first value to the $values variable outside of the loop so that we can format the SQL query properly with commas in the right places. The $url_id used in the $values variable is taken from the id of the URL in the urls table.

```
<?php
function _harvest($url) {
    if(!$this-&gt;_checkURL($url)) {
        echo "URL is not valid ($url).&lt;br /&gt;\n";
    } elseif ($data = $this-&gt;_getData($url)) {
        $words = preg_split ("/[\s,.]+/", $data);
        array_walk ($words, array($this, '_prune'), &amp;$words);
        sort ($words);
        $url_id = $this-&gt;_db-&gt;getone("SELECT id FROM urls WHERE url='$url'");
        if($url_id) {
```

```
            $this-&gt;_db-&gt;query("DELETE FROM keywords WHERE url_id=$url_id");
        } else {
            $this-&gt;_db-&gt;query("INSERT INTO urls SET url='$url'");
            $url_id = mysql_insert_id();
        }
        $values = "($url_id, '$words[0]')";
        $numwords = count ($words);
        for ($i = 1; $i &lt; $numwords; $i++) {
            $values .= ", ($url_id, '$words[$i]')";
        }
        $this-&gt;_db-&gt;query("INSERT INTO keywords VALUES $values");
    }
}
?>
```

This function iterates through an array of URLs and calls the _harvest() function for each. This is the function we will call from our scripts to access the class.

```
<?php
function process() {
    foreach($this->_urlarray as $url) {
        $this->_harvest($url);
    }
}
?>
```

Now, we will glue all the bits together for you so that you may see the class file in its entirety. At the beginning of the class file, we have used to require statement to include the database class from the earlier section.

```
<?php
require('dbclass.php');

class Harvest_Keywords {

    var $_db;
    var $_urlarray;
    var $_stopwords = array ('and', 'but', 'are', 'the');
    var $_allowwords = array ('c++', 'ado', 'vb');

    function Harvest_Keywords($urls) {
        $this-&gt;_db = new DB_Class('test', 'username', 'password');
    $this-&gt;_urlarray = trim ($urls);
    $this-&gt;_urlarray = explode ("\n", $this-&gt;_urlarray);
    }

    function _prune (&amp;$item, $key, $array) {
```

```
        $item = strtolower ($item);
        if (((preg_match ("/[^a-z0-9'\?!-]/", $item))
           || (strlen ($item) &amp;lt; 3)
           || (in_array($item, $this-&amp;gt;_stopwords)))
           &amp;amp;&amp;amp; (!in_array($item, $this-&amp;gt;_allowwords))) {

             unset($array[$key]);
        } else {
            $item = addslashes(preg_replace("/[^a-z0-9'-]/i",
                                            '', $item));
        }
    }

    function _checkURL($url) {
        return preg_match ("/http:\/\/(.*)\.(.*)/i", $url);
    }

    function _getData($url) {
        $filehandle = @fopen($url, 'r');
        if(!$filehandle) {
            echo "Could not open URL ($url).&amp;lt;br /&amp;gt;\n";
            $return = FALSE;
        } else {
            $data = fread($filehandle, 25000);
            fclose($filehandle);
            $data = strip_tags ($data);
            $data = str_replace('&amp;amp;nbsp;', ' ', $data);
            $return = $data;
        }
        return $return;
    }

    function _harvest($url) {
        if(!$this-&amp;gt;_checkURL($url)) {
            echo "URL is not valid ($url).&amp;lt;br /&amp;gt;\n";
        } elseif ($data = $this-&amp;gt;_getData($url)) {
            $words = preg_split ("/[\s,.]+/", $data);
            array_walk ($words, array($this, '_prune'), &amp;amp;$words);
            sort ($words);
            $url_id = $this-&amp;gt;_db-&amp;gt;getone("SELECT id FROM urls "
                                        . "WHERE url='$url'");
            if($url_id) {
                $this-&amp;gt;_db-&amp;gt;query("DELETE FROM keywords "
                                    . "WHERE url_id=$url_id");
            } else {
                $this-&amp;gt;_db-&amp;gt;query("INSERT INTO urls SET url='$url'");
                $url_id = mysql_insert_id();
            }
            $values = "($url_id, '$words[0]')";
            $numwords = count ($words);
            for ($i = 1; $i &amp;lt; $numwords; $i++) {
```

```
                $values .= ", ($url_id, '$words[$i]')";
            }
            $this-&gt;_db-&gt;query("INSERT INTO keywords VALUES $values");
        }
    }

    function process() {
        foreach($this-&gt;_urlarray as $url) {
            $this-&gt;_harvest($url);
        }
    }
}
?>
```

Now that we have created the harvesting class, we can code the harvest.php script. This is essentially just an HTML form that posts to back to itself. Because we have placed all the code that does the actual keyword harvesting in a class, this script is clean keeps the presentation separate from the logic.

```
&lt;!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd"&gt;
&lt;html&gt;
&lt;head&gt;
&lt;title&gt;Harvest Keywords&lt;/title&gt;
&lt;/head&gt;
&lt;body&gt;
&lt;?php
require('harvestclass.php');
if(isset($_POST['submit'])) {
    $gather = new Harvest_Keywords($_POST['urls']);
    $gather-&gt;process();
    echo "Keywords Harvested.&lt;br /&gt;\n";
}
?&gt;
&lt;h1&gt;Harvest&lt;/h1&gt;
&lt;p&gt;Enter URLs to harvest keywords from, each on its own line:&lt;/p&gt;
&lt;form method="POST" action="&lt;?php echo $_SERVER['PHP_SELF']; ?&gt;"&gt;
&lt;p&gt;&lt;textarea name="urls" cols=50 rows=10&gt;&lt;/textarea&gt;&lt;/p&gt;
&lt;p&gt;&lt;input type="submit" name="submit" value="Submit"&gt;&lt;/p&gt;
&lt;/form&gt;
&lt;/body&gt;
&lt;/html&gt;
```

So, we now have a method to populate our tables with URLs and keywords. The next thing we need to accomplish is to create a search class. Our search class will provide methods for exact keyword searching and fuzzy searching. Let's start off with an overview of how the class will be built.

- **Class variables** − As with the harvesting class, we will have some variables in the this class to hold data that each function needs access to.
- **Constructor** − The constructor of this search class will connect to the database and then place the search terms into a class variable as an array. It will also record the number of elements in that array in another class variable.
- **doSearch** − This function provides the exact keyword matching search.
- **_highpercent** − This is a private function that will be used along with an array_walk() function call.
- **_slashit** − This private function will also be used for an array_walk() function call.
- **doFuzzy** − In this function we will do a fuzzy search.

Three private class variables will be declared. The will be used to store the database connection identifier, the search terms, and the number of search terms.

```
<?php
var $_db;
var $_searchterms;
var $_numterms;
?>
```

The first task in the constructor is to connect to the database. We will, again, use the database class from earlier in this tutorial. Second, we will add slashes to the search terms provided to this function if the magic_quotes_gpc directive is not turned on. Then, we will separate the search terms and store them in a class variable as an array. If only one word was entered, we will end up with an array of only one element. Last, we will store the number of search terms in a class variable.

```
<?php
function Search($keywords) {
    $this->_db = new DB_Class('test', 'username', 'password');
    if (!get_magic_quotes_gpc()) {
        $keywords = addslashes ($keywords);
    }
    $this->_searchterms = explode(' ', $keywords);
    $this->_numterms = count($this->_searchterms);
}
?>
```

In this function we will do an exact keyword search. This search is relatively straightforward, as it only consists of a single SQL query. Because the number of search terms can change, we need to dynamically create the query. This query will result in URLs being returned along with the number of times those URLs contained the search terms. The results will be ordered from highest to lowest by the number of search term matches.

This function will return the data from the query, or FALSE if there were not any rows in the result set.

```
<?php
function doSearch() {
    $match = "keywords.keyword in ('"
           . $this-&gt;_searchterms[0] . "'";
    for ($i = 1; $i &lt; $this-&gt;_numterms; $i++) {
```

```
        $match .= ", '" . $this-&gt;_searchterms[$i] . "'";
    }
    $match .= ")";
    $query = "SELECT urls.url, count(*) as counter "
           . "FROM urls, keywords "
           . "WHERE $match "
           . "AND keywords.url_id = urls.id "
           . "GROUP BY keywords.url_id "
           . "ORDER BY counter DESC";
    $result = $this-&gt;_db-&gt;fetch($query);
    if (count($result) &gt; 0)
        $return = $result;
    else
        $return = FALSE;

    return $return;
}
?>
```

This private function of the class will be used along with the array_walk() function in our fuzzy search. It will determine if an element of an array is less than 60. If it is, it will unset that element from the array.

```
<?php
function _highpercent(&amp;$item, $key, $array) {
    if ($item &lt; 60)
        unset($array["$key"]);
}
?>
```

Another private function that will be used with array_walk(), this one will add slashes to an element of an array.

```
<?php
function _slashit(&amp;$item, $key) {
    $item = addslashes($item);
}
?>
```

Now it is time to develop a way to do fuzzy searching. What we want to do is find words that are similar to the ones the user searched for and didn't exist in our keyword list. To make this happen, we will use the similar_text() function of PHP.

The similar_text() function provides very good results but at the price of performance. Running a very large amount of keywords through the similar_text() function could create a sluggish application. One alternative is the levenshtein() function which is much better performance wise when compared to the similar_text() function. The drawback is that levenshtein() produces results that are not quite as accurate as similar_text().

The similar_text() function takes three parameters. The first and second are the words to compare, and the third is a variable to store the percentage of how similar they are. A similarity of 100% would be exactly the same, and it goes down from there. The levenshtein() function works in a very similar fashion to similar_text(). The one difference is that it assigns a cost value to the third parameter rather than a percentage. In this case, a lower number means a closer match. If you choose to use the levenshtein() function, keep this in mind when sorting the array later in the script. You will need to sort it in normal order rather than in reverse.

Now that we have covered a little bit of theory behind this function, let's take a look at how it will actually work.

- Select all keywords from the keywords table that start with the same character as one of our search terms.
- Build a list of matches for each search term
- Merge the search term matches
- Build query
- Run query and display results

Select all keywords from table

The first thing we need to do is query the database and select all the keywords from the keywords table that start with the same character as one of our search terms. To only match against the first letter of the keyword, we will utilize the LEFT function of MySQL in our query. We will also use the DISTINCT SQL function to only retrieve unique keywords. The query will be built dynamically as we have done several other times. We will take the results from this query and store them in an array.

```
<?php
$match = "LEFT(keyword,1) in ('"
       . substr($this-&gt;_searchterms[0], 0, 1) . "'";
for ($i = 1; $i &lt; $this-&gt;_numterms; $i++) {
    $match .= ", '" . substr($this-&gt;_searchterms[$i], 0, 1) . "'";
}
$match .= ")";
$query = "SELECT DISTINCT(keyword) FROM keywords WHERE $match";
$keywords = $this-&gt;_db-&gt;fetch($query);
?>
```

Build a list of matches for each search term

Now, we need to take each search term and each keyword and let the similar_text() function determine how alike they are. We will use foreach loops to iterate through each search term and keyword. Before we begin, we will use the reset() function to return the array pointer back to the first element of the $_searchterms class variable.

Once we determine how similar each keyword is in relation to a search term, we use the array_walk() function to call the _highpercent() function for each element to remove any keywords that are found to be less than 60% similar.

```
<?php
reset($this-&gt;_searchterms);
foreach ($this-&gt;_searchterms as $term) {
    foreach ($keywords as $keyword) {
        $word = $keyword['keyword'];
        similar_text($term, $word, $matches["$term"]["$word"]);
    }
```

```
    array_walk ($matches["$term"],
                array($this, '_highpercent'),
                &amp;$matches["$term"]);
}
?>
```

Merge the search term matches

At this point, we will have an associative array named $matches that has as an element for each search term with that search term as the key. Each of these elements is also an associative array. What we now need to do is to combine each of these inner associative arrays into one. To do this we will need to use the array_merge() function. Because we don't know how many search terms we are dealing with, we will need to also use the eval() function to accomplish this task.

The eval() function takes a string and evaluates it as PHP code. What we are going to do is build a string that contains a PHP statement, much as we did for the queries earlier. Once we build the string, we will pass it to the eval() function and let it evaluate it. Upon evaluation, the inner arrays will be merged and stored in the $merged variable.

If we only have one search term, we don't need to go through all this so we will just assign the contents of the only inner array to the variable $merged. Once we have the data in the $merged array we will sort it in reverse order, and maintain key relationships, with the arsort() function. Then we will extract the names of keys, as they are the similar keywords, and store them in an array called $search.

```
<?php
if($this-&gt;_numterms &gt; 1) {
    $merge = '$merged = array_merge($matches["'
           . $this-&gt;_searchterms[0] . '"]';
    for ($i = 1; $i &lt; $this-&gt;_numterms; $i++) {
        $merge .= ', $matches["'
                . $this-&gt;_searchterms[$i] . '"]';
    }
    $merge .= ');';
    eval ($merge);
} else {
    $merged = $matches[$this-&gt;_searchterms[0]];
}
arsort($merged);
$search = array_keys($merged);
?>
```

Build query

Now, we have a list of similar keywords that exist in our keywords table. We now must run a query to determine what URLs they are associated with and with what frequency they occur. Before we run the query, however, we will escape any character in the $search array by passing it through the array_walk() function and specifying the function _slashit() that will call the addslashes() function for each array element.

Then, we will build the query as we have in the past. If we have rows in the result set, we will return them. Otherwise we will return FALSE.

```
<?php
array_walk($search, array($this, '_slashit'));
$match = "keywords.keyword in ('"
       . $search[0] . "'";
$numwords = count ($search);
for ($i = 1; $i &lt; $numwords; $i++) {
    $match .= ", '" . $search[$i] . "'";
}
$match .= ")";
$query = "SELECT urls.url, keywords.keyword, count(*) as counter "
       . "FROM urls, keywords "
       . "WHERE $match "
       . "AND keywords.url_id = urls.id "
       . "GROUP BY keywords.url_id, keywords.keyword "
       . "ORDER BY counter DESC";
$result = $this-&gt;_db-&gt;fetch($query);
if (count($result) &gt; 0)
    $return = $result;
else
    $return = FALSE;

return $return;
?>
```

Now, we present the search class script as a whole.

```
<?php
require('dbclass.php');

class Search {

    var $_db;
    var $_searchterms;
    var $_numterms;

    function Search($keywords) {
        $this-&gt;_db = new DB_Class('test', 'username', 'password');
        if (!get_magic_quotes_gpc()) {
            $keywords = addslashes ($keywords);
        }
        $this-&gt;_searchterms = explode(' ', $keywords);
        $this-&gt;_numterms = count($this-&gt;_searchterms);
    }

    function doSearch() {
        $match = "keywords.keyword in ('"
               . $this-&gt;_searchterms[0] . "'";
        for ($i = 1; $i &lt; $this-&gt;_numterms; $i++) {
```

```
            $match .= ", '" . $this-&gt;_searchterms[$i] . "'";
        }
        $match .= ")";
        $query = "SELECT urls.url, count(*) as counter "
               . "FROM urls, keywords "
               . "WHERE $match "
               . "AND keywords.url_id = urls.id "
               . "GROUP BY keywords.url_id "
               . "ORDER BY counter DESC";
        $result = $this-&gt;_db-&gt;fetch($query);
        if (count($result) &gt; 0)
            $return = $result;
        else
            $return = FALSE;

        return $return;
    }

    function _highpercent(&amp;$item, $key, $array) {
        if ($item &lt; 60)
            unset($array["$key"]);
    }

    function _slashit(&amp;$item, $key) {
        $item = addslashes($item);
    }

    function doFuzzy() {
        $match = "LEFT(keyword,1) in ('"
               . substr($this-&gt;_searchterms[0], 0, 1) . "'";
        for ($i = 1; $i &lt; $this-&gt;_numterms; $i++) {
            $match .= ", '" . substr($this-&gt;_searchterms[$i], 0, 1)
                    . "'";
        }
        $match .= ")";
        $query = "SELECT DISTINCT(keyword) FROM keywords "
               . "WHERE $match";
        $keywords = $this-&gt;_db-&gt;fetch($query);
        reset($this-&gt;_searchterms);
        foreach ($this-&gt;_searchterms as $term) {
            foreach ($keywords as $keyword) {
                $word = $keyword['keyword'];
                similar_text($term, $word,
                $matches["$term"]["$word"]);
            }
            array_walk ($matches["$term"],
                        array($this, '_highpercent'),
                        &amp;$matches["$term"]);
        }
        if($this-&gt;_numterms &gt; 1) {
            $merge = '$merged = array_merge($matches["'
```

```
                . $this-&gt;_searchterms[0] . '"]';
            for ($i = 1; $i &lt; $this-&gt;_numterms; $i++) {
                $merge .= ', $matches["'
                        . $this-&gt;_searchterms[$i] . '"]';
            }
            $merge .= ');';
            eval ($merge);
        } else {
            $merged = $matches[$this-&gt;_searchterms[0]];
        }
        arsort($merged);
        $search = array_keys($merged);
        array_walk($search, array($this, '_slashit'));
        $match = "keywords.keyword in ('"
               . $search[0] . "'";
        $numwords = count ($search);
        for ($i = 1; $i &lt; $numwords; $i++) {
            $match .= ", '" . $search[$i] . "'";
        }
        $match .= ")";
        $query = "SELECT urls.url, keywords.keyword, "
               . "count(*) as counter "
               . "FROM urls, keywords "
               . "WHERE $match "
               . "AND keywords.url_id = urls.id "
               . "GROUP BY keywords.url_id, keywords.keyword "
               . "ORDER BY counter DESC";
        $result = $this-&gt;_db-&gt;fetch($query);
        if (count($result) &gt; 0)
            $return = $result;
        else
            $return = FALSE;

        return $return;
    }
}
?>
```

Now, we can create a script called search.php and actually do some searching. In this script, we will display a text box to allow for search terms to be entered. When the form is submitted, we will first do an exact keyword match. If that does not yield any results, we will perform a fuzzy search.

```
<?php
&lt;!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd"&gt;
&lt;html&gt;
&lt;head&gt;
&lt;title&gt;Search&lt;/title&gt;
```

```
&lt;/head&gt;
&lt;body&gt;
&lt;h1&gt;Search&lt;/h1&gt;
&lt;p&gt;Enter keywords to search for:&lt;/p&gt;
&lt;form method=POST action="&lt;?= $_SERVER['PHP_SELF'] ?&gt;"&gt;
&lt;p&gt;&lt;input type="text" name="search_term" size="20"&gt;
&lt;input type="submit" name="submit" value="Submit"&gt;&lt;/p&gt;
&lt;/form&gt;
&lt;?php

require('searchclass.php');

if (isset ($_POST['submit'])) {
    $search = new Search($_POST['search_term']);
    $results = $search−&gt;doSearch();
    if($results) {
        echo "&lt;p&gt;&lt;b&gt;Your search results:&lt;/b&gt;&lt;/p&gt;\n";
        echo "&lt;p&gt;";
        foreach($results as $row) {
            echo "&lt;a href=\"{$row['url']}\"&gt;"
                 . "{$row['url']}&lt;/a&gt;&lt;br /&gt;\n";
        }
        echo "&lt;/p&gt;\n";
    } else {
        $results = $search−&gt;doFuzzy();
        echo "&lt;p&gt;&lt;b&gt;No matches! ";
        if($results) {
            echo "These pages contain similar words "
                 . "to what you searched for:&lt;/b&gt;&lt;/p&gt;\n";
            echo "&lt;p&gt;&lt;i&gt;(Similar terms are in parentheses)&lt;/i&gt;&lt;/p&
            echo "&lt;p&gt;";
            foreach($results as $row) {
                echo "&lt;a href=\"{$row['url']}\"&gt;{$row['url']}"
                     . "&lt;/a&gt;&amp;nbsp;({$row['keyword']})&lt;br /&gt;\n";
            }
            echo "&lt;/p&gt;\n";
        } else {
            echo "No similar words either.&lt;/b&gt;&lt;/p&gt;\n";
        }
    }
}
?&gt;
&lt;/body&gt;
&lt;/html&gt;
?>
```

While you now have a search application that you can use on your web site, it was not our intention to only provide a script that you can use. We hope that by learning to develop this search application you have a greater understanding of how PHP works. It is also our desire that the foundations learned for databases in this tutorial will pave the way for you to

learn more about them.

There are many ways you could extend this search application. For instance, we mentioned earlier that you could develop a spider to harvest the keywords. While this is not a simple undertaking, it would be an excellent learning tool. You could also extend this application to harvest keywords from data already stored in a database and display the relevant content when a match was made. Use your imagination and then start coding. With a little determination and a lot of study, you will be amazed at the applications you can create with PHP.