

Paper 014-2009

Using AJAX and SAS® Stored Processes to Create Dynamic Search-Suggest Functionality

Jeffery A. Fallon, Cardinal Health, Dublin, OH

ABSTRACT

Almost everyone is familiar with the search-suggest functionality that is offered by search engines and other Web applications (for example, a user begins typing into a text box on a Web page and a box fills with suggestions that update as the user types). This paper shows SAS® developers the steps involved in using SAS® Stored Processes to develop this type of functionality for their own Web applications. After a brief introduction to AJAX, the reader is given a guided look at how to call a SAS® stored process from JavaScript using the XMLHttpRequest object. The paper then shows two different approaches to coding the stored processes behind the search-suggest functionality, each with different implications for data preparation, performance, and scalability.

INTRODUCTION

Virtually everyone who has searched the web is familiar with the search suggest functionality offered by Google™ and other search engines. When a user enters text into the search text box, a list of suggested results appears below the search box and updates as they type to try to match what they want to search. This functionality offers numerous benefits to web searchers. It can also add power to intranet based business applications.

In this paper, I will show the reader how to create a similar search suggest utility. Our sample utility will have a simple HTML page as the user interface. Behind the user interface, we will take a very close look at a small amount of JavaScript that will communicate with a SAS Stored Process on the server. The user will be able to type a bit of a name into a textbox and our application will return up to ten names that match what they have typed. These names will be displayed in a list next to the textbox in which they are typing. When the name that they are interested in appears in the list, they can click it and it will be copied into the list box in the center of the page. When the box contains all of the names that they want, they may submit the form and view the results. Annotated copies of the source code for the sample application will be included in appendices A through E of this paper.

This functionality can be a very useful addition to any type of reporting application that requires users to run reports based on the names of the entities that they are interested in. Rather than offering the user a lengthy list of names to choose from, we can just present a textbox for them to begin typing the name of the entity. Within a few keystrokes the name of the entity that they are interested in appears among a list of suggestions.

THE SAMPLE DATA

The data that we will use are in our example are the Firms and Listings support files from the National Drug Code Directory produced by the U.S. Food and Drug Administration. These data were chosen because they satisfy three important criteria for this application.

- They are nonproprietary, free, and publicly available for download from the FDA website (<http://www.fda.gov/Drugs/InformationOnDrugs/ucm142438.htm>).
- They are sizeable enough to illustrate both the usefulness and performance of the utility that we will build.
- They identify entities by both name and number.

The last point may seem a bit unusual. Often, in business data, there are certain fundamental entities, such as Customer or Product, that are identified by both name and number. Inevitably, there are some people who become more familiar with the numbers associated with these entities and others with the names. We want to make our search suggest useful to both. Therefore, we begin by concatenating the identifying names and numbers into a single searchable column.

In the sample data, I have added a Firms_Key and Listings_Key to the corresponding tables. The Firms_Key is the FIRM_NAME concatenated with the LBLCODE in parentheses (i.e. ELI LILLY AND CO (000002)). The Listings_Key is the TRADENAME concatenated with LBLCODE, PRODCODE, and STRENGTH in parentheses and separated by hyphens (i.e. HUMULIN R INJECTION (000002-8215-100)).

THE CLIENT

THE USER INTERFACE

For the sake of simplicity, our sample client consists of a single static HTML page. The search page is a form consisting of a set of radio buttons that allow us to specify the data set that we wish to search and the search suggest stored process that we wish to use. At the top of the page, is a text box into which we will type the text that will be sent to the search suggest. There is a page section off to the right of the textbox that is initially empty, but will fill with our suggestions as we type. When a suggestion appears that we wish to use in our search, we simply click it and it will appear in the list box below the textbox. When we have selected all of the terms that we wish, we click the Submit button and the form will post the terms that we have selected to another web page.

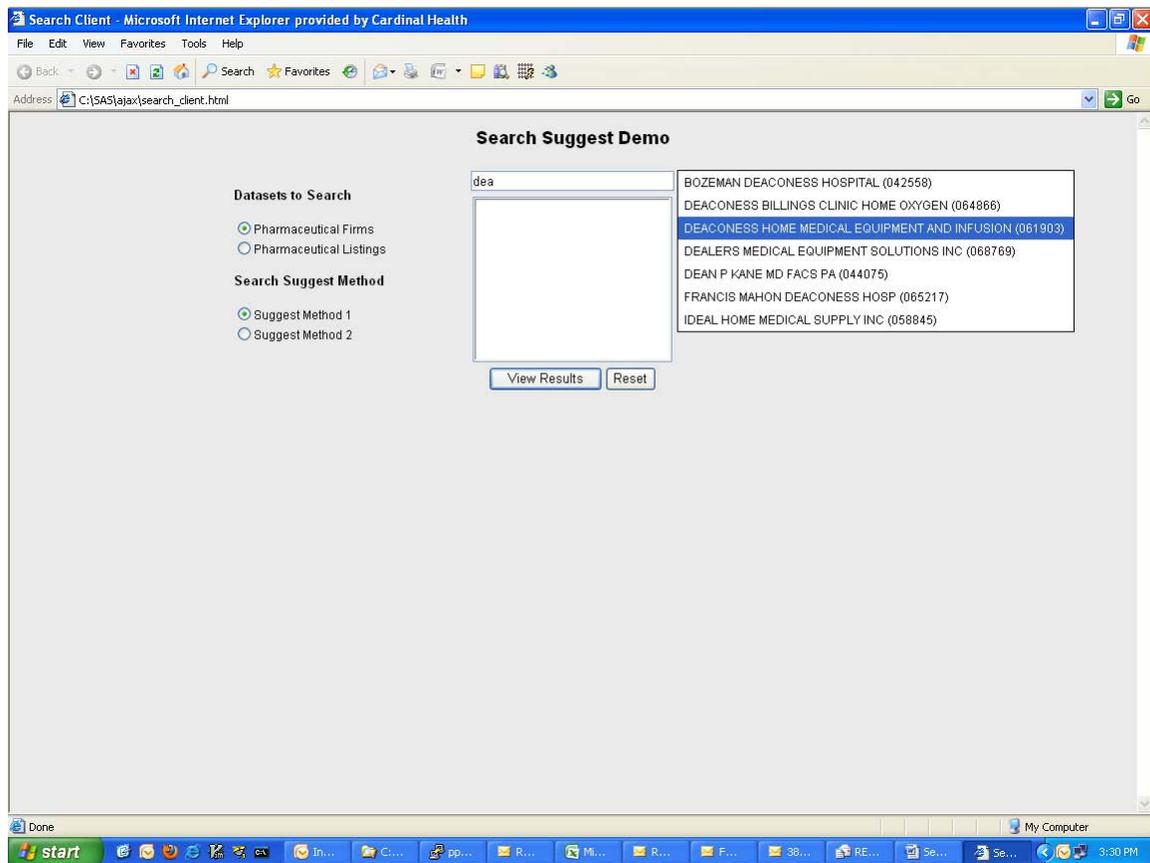


Figure 1. The User Interface for our Sample Application

AJAX

AJAX is an acronym that stands for Asynchronous JavaScript and XML. It is a bit of a misnomer. XML is not only unnecessary, it is often abandoned in favor of a data format that is easier to handle in the web browser, like HTML or even plain text--which we will use in this paper. JavaScript is also not required, although it is most common. The key feature of AJAX is that it is asynchronous. Asynchronous communication with a server is what has ushered in the so-called Web 2.0 era. Among other things, AJAX allows us to build more dynamic pages with input processed on the server side of our applications as soon as it is entered. The relevant portions of the client pages are updated immediately from the server.

Since AJAX has been around for about four years, there are many free web-based resources available to help us to incorporate AJAX into our applications. The client-side JavaScript that I present here is adapted from Ryan Smith's *AJAX Suggest Tutorial* at <http://www.dynamicajax.com> (2005).

Let us look at the JavaScript on the client side of our application. As soon as our page is loaded into the browser, we execute the following code to create the object that makes AJAX possible—the XMLHttpRequest.

```
var suggestRequest = getXmlHttpRequestObject();

function getXmlHttpRequestObject() {
  if (window.XMLHttpRequest) {
    return new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  } else {
    alert("Your browser does not support AJAX.");
  }
}
```

The `getXmlHttpRequestObject()` function either creates the object normally (return new XMLHttpRequest), in a slightly different way for Internet Explorer versions 5 and 6, or not at all if the browser does not support this operation.

Now that our XMLHttpRequest object (`suggestRequest`) has been created, it is ready for use. Here is the HTML for the search textbox on our client page:

```
<input type="text" id="txtSearch" name="txtSearch" alt="Search Criteria"
tabindex="2" onkeyup="searchSuggest();"/>
```

Notice the function call in the “onkeyup” event attribute. Let’s take a close look at the function.

```
function searchSuggest() {
  if (suggestRequest.readyState == 4 || suggestRequest.readyState == 0) {
    var method = getSuggestMethod();
    var str = '&SRCH=' + escape(document.getElementById('txtSearch').value);
    var table = '&TABLE_ID=' + getTable();
    suggestRequest.open("GET",
      'http://ppdmsas01:8080/SASStoredProcess/do?_PROGRAM=/Search+Suggest/Suggest+Method+' + method + table + str, true);
    suggestRequest.onreadystatechange = processSuggestions;
    suggestRequest.send(null);
  }
}
```

It is always good to keep in mind that we will be calling a SAS Stored Process with each keystroke from the client page. By making sure that the “readyState” of the object is 4 (Done) or 0 (Unsent) helps protect the server by issuing a call to the server only when the object is not waiting on a response to, or processing the results of, a prior request. If it is not waiting or processing, then we may open another request. Otherwise, we can imagine our Stored Process Server crashing under an unrelenting torrent of requests! We get the value of the “Search Suggest Method” radio button and store it in the variable “method”. The next two lines set two macro variables that we will send to the Stored Process. The macro variable SRCH is the text that the user has types into the textbox. The macro variable TABLE_ID is taken from the value of the radio button that the user selected in the “Dataset to Search” section of the client page.

Appropriately enough, we open a request via the “open” method of the object. The open method has two mandatory parameters. The first is the HTTP method of the request (usually GET or POST). The second mandatory parameter is the URL that we will open. In this example, we are building the URL by concatenating a hard-coded string with a few of variables that we set within this function. The finished URL for my environment is: `http://ppdmsas01:8080/SASStoredProcess/do?_PROGRAM=/Search+Suggest/Suggest+Method+One&TABLE_ID=FIRST&SRCH=a`. The third parameter that we pass is an optional parameter “true” that makes explicit the fact that our request is asynchronous.

I want to draw attention to the next line:

```
suggestRequest.onreadystatechange = processSuggestions;
```

This assigns the a function called `processSuggestions()` to the “onreadystatechange” event of the XMLHttpRequest object. That means that this function will execute whenever the “onreadystatechange” event fires. Let us examine that function.

```
function processSuggestions() {
```

```

if (suggestRequest.readyState == 4) {
  var ss = document.getElementById('search_suggest')
  ss.innerHTML = '';
  var str = suggestRequest.responseText.split("\n");
  for(i=0; i < str.length - 1; i++) {
    //Build the divs that contain the search suggestions.
    var suggest = '<div onmouseover="javascript:suggestOver(this);"';
    suggest += 'onmouseout="javascript:suggestOut(this);" ';
    suggest += 'onclick="javascript:setSearch(this.innerHTML);" ';
    suggest += 'class="suggest_link">' + str[i] + '</div>';
    ss.innerHTML += suggest;
  }
}
}

```

We begin by double checking that our object is loaded. In other words, that it has received a complete response from the server. This tells us that we have data. To process the response from the server, we clear the content of the HTML DIV element that will hold the suggestions—which has an ID of “search_suggest”. Then, we process the text of the response to fill it. First, we split it into an array of strings on the newline character (“\n”). Then, we loop through that array concatenating the necessary HTML to each string to make a DIV element for each string. After this is done, we append the new DIV that we have created to the contents of the parent DIV element (“search_suggest”). The newly created DIV elements are coded for three events. In the “onmouseover” event, the suggestOver() function will be called. This simply highlights the text of the element. In the “onmouseout” event, the suggestOut() function is called. As you might guess, this removes the highlighting that suggestOver() added. The “onclick” event calls the setSearch() function which takes the text of the DIV element as a parameter, and appends it to the main list box for the names that we select for our actual search. The code for each of these functions is supplied in appendix B of this paper which contains all of the JavaScript for the client.

To recap, we have created an XMLHttpRequest object that will, with each keystroke typed into the textbox, send a request to a SAS Stored Process. The Stored Process will take as parameters the text that we are typing, the table that we have chosen to search, and the stored process that will return search suggestions. When a response is received from the Stored Process, it will parse the results into the HTML of our web page and allows us to select the suggestions for subsequent queries.

Now let us turn our attention server-side, for a look at the SAS Stored Processes that will return the list of search suggestions. We will take a look at two different ways to return search suggestions, each with its own advantages and disadvantages.

THE SERVER

SEARCH SUGGEST METHOD ONE

First, remember that we sent two parameters to the Stored Process as macro variables on the URL. The macro variable TABLE_ID is the name of the table that we will query. The macro variable SRCH contains the search string entered by the user. Our Stored Process begins as follows:

```

libname srchdata "/var/opt/pharma/user/jeff";
data _null_;
  length tmp $100;
  tmp = tranwrd("&SRCH", "'", "'");
  call symput("SRCH", "%'" || strip(tmp) || "%'");
run;

```

After assigning the libref for the data, we replace any single quotes with double single quotes in the user's search string, to avoid any unbalanced quote errors. Then, we enclose the value in nested single quotes and percent signs. The reason for that is in the next segment of code.

```

proc sql noprint outobs=10;
  create table &table_id as
  select distinct &TABLE_ID..&TABLE_ID._key
  from srchdata.&TABLE_ID
  where upcase(&TABLE_ID._key) like upcase(&SRCH);
quit;

```

Here we see that the quotes and percent signs were to make that text into a pattern operand for the LIKE operator. By adding both leading and trailing percent signs, which are wildcards for the LIKE operator, we have made our search very flexible. We will match all strings that contain the text that the user typed—no matter where it is found within the string. Another useful feature is that the user may enter additional percent signs into their data and SAS will interpret them as wildcards. In addition, by converting both the key that we are searching and our pattern to upper case in the WHERE clause of our query, we have added further flexibility by creating a case insensitive search.

Notice the SQL option outobs =10. This will stop query execution after 10 results have been returned, because that is the largest number of suggestions that we want to display on the client in this example.

We process the results with the following code:

```
%macro get_suggestions;
  %if %eval(&sqlobs ne 0) %then %do;
    data _null_;
      file _webout;
      set &TABLE_ID;
      put & TABLE_ID._key;
    run;
  %end;
  %else %do;
    data _null_;
      file _webout;
      put '**No Matches**';
    run;
  %end;
%mend get_suggestions;
%get_suggestions;
```

Checking the value of the automatic macro variable SQLOBS shows us how many results were returned by the query. If it is not equal to zero, we write the results to _webout in a data step with simple put statements. If no results were returned, we alert the user.

PERFORMANCE

That is all there is to it. Now let us consider the performance implications of what we have done. As I mentioned before, putting wildcards around our search term, creates a very flexible search. Unfortunately, it is too flexible to optimize. In order to match all possible values, SAS must scan all of our key values. Even if our data are indexed, the wildcards make it impossible for SAS to gain any benefit from the index. Of course the outobs=10 option on the SQL procedure can cut short these scans, but when the search term is more distinctive and will match only ten or fewer keys, it will still scan the entire table. As a result, on a minimally powerful server, you will begin to notice a slight lag in the search suggest utility when the data volume is near ten-thousand observations. At thirty to forty-thousand the lag is quite pronounced. Such a lag renders the utility largely useless, because the suggestions often don't match the current search string.

SEARCH SUGGEST METHOD TWO

Are we out of luck if we have large tables? Of course not, we just have a little more work to do. Fortunately, we can take another cue from Google and the other search engines. These engines do not search the web directly, they search pre-generated indexes that map the web. What does that mean to us? It means that we need to create pre-generated indexes to map our data. Fortunately, with SAS, that is not as hard as it may sound. Let us now take a look at some SAS code that will prepare these indexes for us.

We begin by generating a list of unique key values from our table.

```
%macro search_prep(TABLE_ID= ,KEY=);

proc sql;
  create TABLE_ID distinct_&KEY as
  select distinct &KEY
  from webdata.&TABLE_ID;
quit;
```

Next, we generate a complete map of each KEY value with every possible substring of itself,

```
data substring_xref_1 (keep= substring &KEY);
  length substring $50;
  set distinct_&KEY;
  do i = 1 to length(&KEY);
    do j = 1 to length(&KEY);
      if (i + j - 1) le length(&KEY) then do;
        substring = strip(upcase(substr(&KEY,j,i)));
        substring_length = length(substring);
        if substring_length gt 0 then do;
          output;
        end;
      end;
    end;
  end;
end;
run;
```

You may think that this will create a large number of observations in our data. It will. With the sample data, the number of observations was in the tens of millions for the tables generated from the Listings table. Of course there are two things that we will do to reduce the number of records slightly. The first is to remove all of the duplicate records.

```
proc sort data=substring_xref_1 out=substring_xref_2 noduprecs;
  by substring &KEY;
quit;
```

Next, we count the number of records for each unique substring.

```
data substring_xref_3 (keep=substring &KEY unique_count);
  set substring_xref_2;
  if _N_ eq 1 then do;
    unique_count = 1;
  end;
  retain prev_substring unique_count;
  if _N_ gt 1 and prev_substring eq substring then do;
    unique_count = unique_count + 1;
  end;
  else do;
    unique_count = 1;
  end;
  prev_substring=substring;
run;
```

We only need to keep 10 records associated with any unique substring, because the search only returns 10.

```
data webdata.search_&TABLE_ID (keep=substring &KEY unique_count);
  set substring_xref_3;

  where unique_count le 10;
run;
```

Finally, the data are ready. We have reduced the number of observations somewhat, but they still number in the tens of millions. So how can searching a dataset containing tens of millions of records help us to enhance a utility that was starting to slow down on datasets with tens of thousands of observations? As we saw earlier, the slowness was the result of wildcard searches that could not be optimized. Here we will use direct searches. Direct searches can be optimized, with the creation of an index. We index the new search tables on substring.

```
proc datasets library=webdata;
  modify search_&TABLE;
  index delete _ALL_;
  index create substring;
```

```
quit;
%mend search_prep;
```

Here are the calls to this macro for our sample tables.

```
%search_prep(TABLE_ID=FIRMS, KEY=FIRMS_KEY);
%search_prep(TABLE_ID=LISTINGS, KEY=LISTINGS_KEY);
```

Because we will now query the "SEARCH_" tables that we have created by directly searching on substring, the search suggest Stored Process used will change from what we had done previously.

We replace any single double quotes with a pair of double quotes to avoid any unbalanced quote errors. We initialize a macro variable NUMOBS to zero. This variable will contain the number of observations that our search suggest returned.

```
data _null_;
  call symput("SRCH", upcase(tranwrd("&SRCH", '"', '"')));
  call symput("NUMOBS", 0);
run;
```

Here we execute a data step with a WHERE clause to get the keys with substring values that match our search term. We then set the value of NUMOBS to the number of observations that were returned.

```
data out_results;
  file _webout;
  set webdata.&TABLE_ID end=num;
  where substring eq "&SRCH";
  put &KEY;
  if num then do;
    call symput("NUMOBS", _N_);
  end;
run;
```

Finally, we execute a macro that evaluates NUMOBS to see if the search returned any results. If not, we alert the user.

```
%macro no_suggestions;
  %if (&NUMOBS eq 0) %then %do;
    data _null_;
      file _webout;
      put '**No Matches**';
    run;
  %end;
%mend no_suggestions;
%no_suggestions;
```

CONCLUSION

Here we have taken a detailed look at how to build a search suggest utility with AJAX and SAS Stored Processes. We began by seeing how to create and use an XMLHttpRequest object in JavaScript on the client. Then, we looked at two possible ways to build the server-side Stored Processes. The first method is simple and very flexible, but did not scale well for large datasets. The second method required some data preparation, but yielded a search suggest utility that performs well even for large datasets.

REFERENCES

- Smith, Ryan. "AJAX Suggest Tutorial." Dynamic AJAX 2005
 <http://www.dynamicajax.com/fr/AJAX_Suggest_Tutorial-271_290_312.html >
- Van Kesteren, Anne. "The XMLHttpRequest Object." W3C Working Draft 15 April 2008
 <<http://www.w3.org/TR/2008/WD-XMLHttpRequest-20080415/>>

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Jeffery A. Fallon
Cardinal Health
7000 Cardinal Place
Dublin, OH 43017
614-638-1300
jeffery.fallon@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration.

Other brand and product names are trademarks of their respective companies.

APPENDIX A

HTML for the User Interface.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Search Client</title>
  <style type="text/css">
body {
  font: 78%/1.5 arial, helvetica, serif;
  background-color: #EAEAEA;
  text-align: center;
  padding: 0;
  margin: 1em;
}
submit {
  margin-top: 5em;
}
.suggest_link {
  background-color: #FFFFFF;
  padding: 2px 6px 2px 6px;
}
.suggest_link_over {
  color: #FFFFFF;
  background-color: #3366CC;
  padding: 2px 6px 2px 6px;
  white-space: pre;
}
#search_suggest {
  position: absolute;
  background-color: #FFFFFF;
  text-align: left;
  border: 1px solid #000000;
  display:inline;
}
h2 {
  font-size: large;
  margin:0em 0em 1em 0em;
}

input,#frmSrch {
  margin:0px;
  padding:0px;
}
h5 {
  margin:1em 0em 1em 0em;
  padding:0px;
  font-size: small;
}
.formpart {
  margin:5px 0px 5px 0px;
  padding:0px;
  clear: both;
}
#lstBox {
  width: 15em;
}
#txtSearch {
  margin:0px;
  width: 15em;

```

```

}
#optGrp {
  position: absolute;
  top:0;
  left:0;
  margin-top: 60px;
  margin-left: 225px;
  width: 13em;
  text-align: left;
}
div {
  margin-left: 0px;
}
</style>
<script type="text/javascript" src="./search_suggest.js"></script>
</head><body>
<h2>Search Suggest Demo</h2>
<form id="frmSearch" method="post" action="Your Stored Process URL Goes Here!"
onsubmit="selectList();" >
<div id="optGrp">
  <h5>Datasets to Search</h5>
  <input type="radio" name="table_id" value="FIRMS" checked="checked"
onclick="clearIt();" />Pharmaceutical Firms<br/>
  <input type="radio" name="table_id" value="LISTINGS" onclick="clearIt();"
/>Pharmaceutical Listings<br/>

  <h5>Search Suggest Method</h5>
  <input type="radio" name="suggest_method" value="One" checked="checked"
onclick="clearIt();" />Suggest Method 1<br/>
  <input type="radio" name="suggest_method" value="Two" onclick="clearIt();"
/>Suggest Method 2<br/>
</div>
<div class="formpart">
  <input type="text" id="txtSearch" name="txtSearch" alt="Search Criteria"
tabindex="2" onkeyup="searchSuggest();"/>
  <div id="search_suggest">
  </div>
</div>
<div class="formpart">
  <select id="lstBox" name="PARM" size="10" tabindex="-1" multiple="multiple"
ondblclick="deleteIt();"><option /></select>
</div>
<div class="formpart">
  <input type="submit" value="View Results" alt="Run Search" />
  <input type="reset" value="Reset" alt="Reset" onclick="clearIt();" />
</div>

</form>
</body>
</html>

```

APPENDIX B

The JavaScript.

```

/*****
 * search_suggest.js
 *
 * This file contains all of javascript for the Google-like Search Suggest
 * The file begins with the AJAX code needed to make the requests for the
 * search suggestions. Following that, I have put in some additional code
 * for the UI.
 *****/
/** Copyright Statement from the original AJAX Suggest JavaScript File */
/*
    This is the JavaScript file for the AJAX Suggest Tutorial

    You may use this code in your own projects as long as this
    copyright is left in place. All code is provided AS-IS.
    This code is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

    For the rest of the code visit http://www.DynamicAJAX.com

    Copyright 2006 Ryan Smith / 345 Technical / 345 Group.
*/
//Gets the browser specific XmlHttpRequest Object.
//This allows us to make requests to the server between the loading of the
//search form and the user hitting submit.
//The "XML" is irrelevant--all we are getting is plain text delimited
//by newlines.
function getXmlHttpRequestObject() {
    if (window.XMLHttpRequest) {
        //All modern browsers except IE 5 & 6.
        return new XMLHttpRequest();
    } else if(window.ActiveXObject) {
        //IE 5 & 6.
        return new ActiveXObject("Microsoft.XMLHTTP");
    } else {
        //old browsers.
        alert("Your browser does not support AJAX.");
    }
}

//Call the code to create the XmlHttpRequest object.
var suggestRequest = getXmlHttpRequestObject();

//Set the components of the URL for the request.
var server = 'http://ppdmsas01:8080';
var stpSrv = '/SASStoredProcess/do?_PROGRAM=';
var stp = '/Search+Suggest/Suggest+Method+';

//Makes the AJAX request.
//Called from keyup on the search textbox.
function searchSuggest() {
    if (suggestRequest.readyState == 4 || suggestRequest.readyState == 0) {
        var method = getSuggestMethod();
        //Build the parameters to our Stored Processes.
        var table = '&TABLE_ID=' + getTable();
        var str = '&SRCH=' + escape(document.getElementById('txtSearch').value);
    }
}

```

```

        suggestRequest.open("GET", server + stpSrv + stp + method + table + str ,
true);
        suggestRequest.onreadystatechange = processSuggestions;
        suggestRequest.send(null);
    }
}

//Called when the AJAX response is returned.
//Parse the result into the Search Suggest Div.
function processSuggestions() {
    if (suggestRequest.readyState == 4) {
        var ss = document.getElementById('search_suggest')
        ss.innerHTML = '';
        var str = suggestRequest.responseText.split("\n");
        for(i=0; i < str.length - 1; i++) {
            //Build the divs that contain the search suggestions.
            var suggest = '<div onmouseover="javascript:suggestOver(this);" ' +
            suggest += 'onmouseout="javascript:suggestOut(this);" ' +
            suggest += 'onclick="javascript:setSearch(this.innerHTML);" ' +
            suggest += 'class="suggest_link">' + str[i] + '</div>';
            ss.innerHTML += suggest;
        }
    }
}

//Mouse over function
function suggestOver(div_value) {
    div_value.className = 'suggest_link_over';
}
//Mouse out function
function suggestOut(div_value) {
    div_value.className = 'suggest_link';
}

//Click event for the suggestion list.
function setSearch(value) {

// document.getElementById('txtSearch').value = value;
var combo = document.getElementById("lstBox");

    var option = document.createElement("option");
    //These are important, because the ampersands get escaped in process.
    //It does not appear that other characters are escaped.
    option.text=value.replace("&","&");
    option.value=value.replace("&","&");
    combo.add(option,combo.last);
    document.getElementById('search_suggest').innerHTML = '';
}

/*****
* This is the rest of the UI event code.
*****/

//Read the value that is selected in the option group to put on the URL.
function getTable() {

    chosen = "";
    var table = document.getElementById('frmSearch').table_id;

    var len = table.length;

    for (i = 0; i <len; i++) {
        if (table[i].checked) {

```

```

        chosen = table[i].value;
    }
}
return chosen;
}
//Read the value that is selected in the option group to put on the URL.
function getSuggestMethod() {

    chosen = "";
    var table = document.getElementById('frmSearch').suggest_method;

    var len = table.length;

    for (i = 0; i <len; i++) {
        if (table[i].checked) {
            chosen = table[i].value;
        }
    }
    return chosen;
}

//The items in the list are not submitted unless selected.
//This function selects them. It is called from the onsubmit event
//of the form.
function selectList() {

    var list = document.getElementById('lstBox');

    for (i=0; i<list.length; i++) {
        list.options[i].selected = true;
    }
}

//Delete a single item from the list.
//this will be the click event for the list.
function deleteIt() {
    var contSearch = 1;
    var selIndex;
    var list = document.getElementById('lstBox');

    // check to see if there is at least one item selected
    if (list.selectedIndex == -1) {
        alert("No entries selected for deletion");
        return true;
    }

    // loop through all selected items and delete them
    while (contSearch > 0) {
        selIndex =list.selectedIndex;
        if (selIndex >= 0) {
            list.options[selIndex] = null;
            --contSearch;
        }
        else {
            contSearch = 0;
        }
    }
    return true;
}

//Reset the list box, called from the reset button click
//event, because it does not clear list boxes by default.
function clearIt() {

```

```
var list = document.getElementById('lstBox');  
list.length = 0;  
numItems = 0;  
list.value = "";  
document.getElementById('search_suggest').innerHTML = '';  
document.getElementById('txtSearch').value = '';  
document.getElementById('txtSearch').focus();  
return true;  
}
```

APPENDIX C

The source code for the first suggest method.

```

/*****
*
* Suggest Method One
* SAS Stored Process to return search suggestions as the user enters their
* search string.
*****/
options mlogic mprint symbolgen;

libname srchdata "/var/opt/pharma/user/jeff";

/* Preprocess the substring that we will use in our search. */
data null;
  length tmp $1000;
  *replace single quotes with a pair of single quotes to prevent errors.;
  tmp = tranwrd("&SRCH","'", "'");
  /*strip the substring, upcase it and put the percent signs and parentheses
  around it
  this will allow it to be used as a pattern operand of the LIKE operator. */
  call symput("SRCH","%" || strip(tmp) || "%");
run;

proc sql noprint outobs=10;
  create table &table_id as
  select distinct &TABLE_ID..&TABLE_ID._key
  from srchdata.&TABLE_ID
  where upcase(&TABLE_ID._key) like upcase(&SRCH);
quit;

%macro get_suggestions;
  %if %eval(&sqllobs ne 0) %then %do;
    data _null_;
      file _webout;
      set &TABLE_ID;
      put &TABLE_ID._key;
    run;
  %end;
%else %do;
  data _null_;
    file _webout;
    put '**No Matches**';
  run;
%end;
%mend get_suggestions;
%get_suggestions;

```

APPENDIX D

The source code to create pre-generated indexes for the data.

```

rsubmit;
options mlogic mprint symbolgen compress=yes;
libname webdata "/var/opt/pharma/user/jeff";

%macro search_prep(TABLE_ID= ,KEY=);
* Get the list of unique KEYS ;
proc sql;
  create TABLE_ID distinct_&KEY as
  select distinct &KEY
  from webdata.&TABLE_ID;
quit;

* Generate a complete map of each KEY value with every possible substring of
itself;
data substring_xref_1 (keep= substring &KEY);
  length substring $50;
  set distinct_&KEY;

  do i = 1 to length(&KEY);

    do j = 1 to length(&KEY);

      if (i + j - 1) le length(&KEY) then do;

        substring = strip(uppercase(substr(&KEY,j,i)));

        substring_length = length(substring);

        if substring_length gt 0 then do;

          output;
        end;
      end;
    end;
  end;
run;

* Eliminate duplicates ;
proc sort data=substring_xref_1 out=substring_xref_2 noduprecs;
  by substring &KEY;
quit;

* Count the number of records for each unique substring;
data substring_xref_3 (keep=substring &KEY unique_count);
  set substring_xref_2;

  if _N_ eq 1 then do;

    unique_count = 1;
  end;
  retain prev_substring unique_count;

  if _N_ gt 1 and prev_substring eq substring then do;

    unique_count = unique_count + 1;
  end;
  else do;

```

```
        unique_count = 1;
    end;

    prev_substring=substring;
run;

/* Make sure that we only keep 10 records associated with any unique substring,
   because the search only returns 10 */
data webdata.search_&TABLE_ID (keep=substring &KEY unique_count);
    set substring_xref_3;

    where unique_count le 10;
run;

/* Index the new TABLE_ID on substring. Even though these
   TABLE_IDs have a large number of observations, the index allows fast execution.
*/
proc datasets library=webdata;
    modify search_&TABLE_ID;
        index delete _ALL_;
        index create substring;
quit;

%mend search_prep;

%search_prep(TABLE_ID=FIRMS, KEY=FIRMS_KEY);
%search_prep(TABLE_ID=LISTINGS, KEY=LISTINGS_KEY);

endrsubmit;
```

APPENDIX E

The source code for the second suggest method.

```

/*****
***
* Suggest Method Two
* SAS Stored Process to return search suggestions as the user enters their
* search string.

*****/
*/
options mlogic mprint symbolgen;

libname srchdata "/var/opt/pharma/user/jeff";

/* Preprocess the substring that we will use in our search. */
data null;
  call symput("SRCH",upcase(tranwrd("&SRCH",' ','')));
  call symput("NUMOBS",0);
run;

data out_results;
  file _webout;
  set srchdata.search_&TABLE_ID end=num;
  where substring eq "&SRCH";

  put &TABLE_ID._key;
  if num then do;

      call symput("NUMOBS",_N_);
  end;
run;

%macro no_suggestions;
  %if (&NUMOBS eq 0) %then %do;
    data _null_;
      file _webout;
      put '**No Matches**';
    run;
  %end;
%mend no_suggestions;
%no_suggestions;

```