Comparison of Approaches for Gathering Data from the Web for Technology Trend Analysis

Ayse Kaya Firat
Wei Lee Woon
Stuart Madnick

Working Paper CISL# 2008-14

August 2008

Composite Information Systems Laboratory (CISL)
Sloan School of Management, Room E53-320
Massachusetts Institute of Technology
Cambridge, MA 02142

Comparison of Approaches for Gathering Data from the Web for Technology Trend Analysis

Ayse Kaya Firat, Stuart Madnick, Wei Lee Woon

August 2008

1. Introduction

We are investigating trend extrapolation using historical data from academic publications to forecast future technology directions. Many sources of academic information on the Web (e.g., Google Scholar, Scirus) provide a wealth of relevant information, yet they are not structured for programmatic access. We can use Web wrappers, programs that can harvest text data from Web pages and present them in a structured format, to overcome this problem. Figure 1, shows an example what is available to us (academic web sources) on the left and what we want to produce on the right:

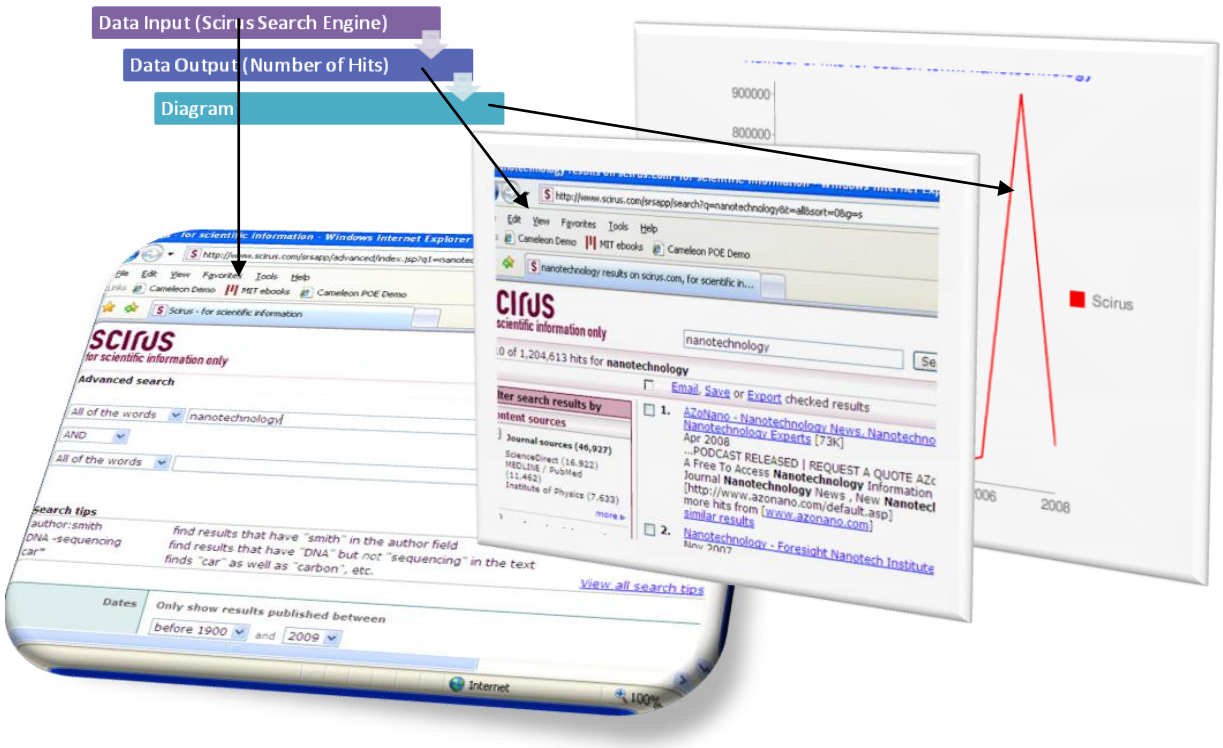

Figure 1: Example of input and output from web source and desired trend analysis

In the example shown in Figure 1 (on the left), using the Scirus search engine a user can specify a search term such as "nanotechnology" and specific years, such as between years 1900 and 2008. In Figure 1 (center), the search engine returns the number of "hits" – the number of academic papers that contain that search term.

But what if we want to create a trend analysis for the number of hits for each year for many years, such as the graph shown in Figure 1 (right). This can be done manually by repetitive use of the search engine to get the hit count year-by-year. But if we are interested in 10 or more years, and 10 or more search engines, and dozens or more search terms (or combinations of search terms), we quickly find that the manual process may have to be repeated thousands or even tens of thousands of times – not a very appealing or practical situation. What we need is the ability to replace the manual process with a programmatic solution.[1]

We have used three different types of wrapper tools/approaches in our project and in this short paper we will compare the pros and cons of each. The first tool, Cameleon#, is a C# based generic wrapper that wraps Web pages by encoding extraction rules in a text file. The second relies on Perl programming and some existing libraries, and the third utilizes Python programming and its libraries. The comparison will be done along the following dimensions:

1) Authoring: The process of creating a wrapper for a web source.
2) Maintainability: The process of updating an existing wrapper.
3) Teachability: The process of teaching wrapper development to a new comer.
4) Capability: The power and flexibility of the wrappers.

2. Background

2.1. Cameleon#

Cameleon# [1] is a generic web wrapper developed by the MIT Context Interchange (COIN) group. Web wrappers in the COIN group are used to treat semi-structured web data as ordinary relational data sources that can be processed using the standard SQL query language (with some capability restrictions) as shown in Figure 2. Wrapper development efforts in the group date back to 1995 and earlier with wrapper development toolkits such as Generic Screen Scraper, Grenouille in Perl, and Cameleon in Java.

Currently, Cameleon#, a reimplementation of Cameleon in .NET, is the toolkit commonly used by the group members. Cameleon# also has a helper tool called Cameleon Studio, which is used to generate wrappers visually (see section 3.1.2). The common element of all of the COIN wrapper development toolkits is that they separate the extraction knowledge from the code (whether in Perl, Java, or C#) by expressing the former in a separate specification file (spec file). The code remains untouched and web sources are wrapped by creating a simple text file expressing extraction rules such as what URL(s) to visit, and what patterns to apply, and so on.

[1] Obviously Google Scholar, Scirus, and the other search engines have actual databases internally, but we generally do not have direct access to these databases and must use of the provided web interfaces.

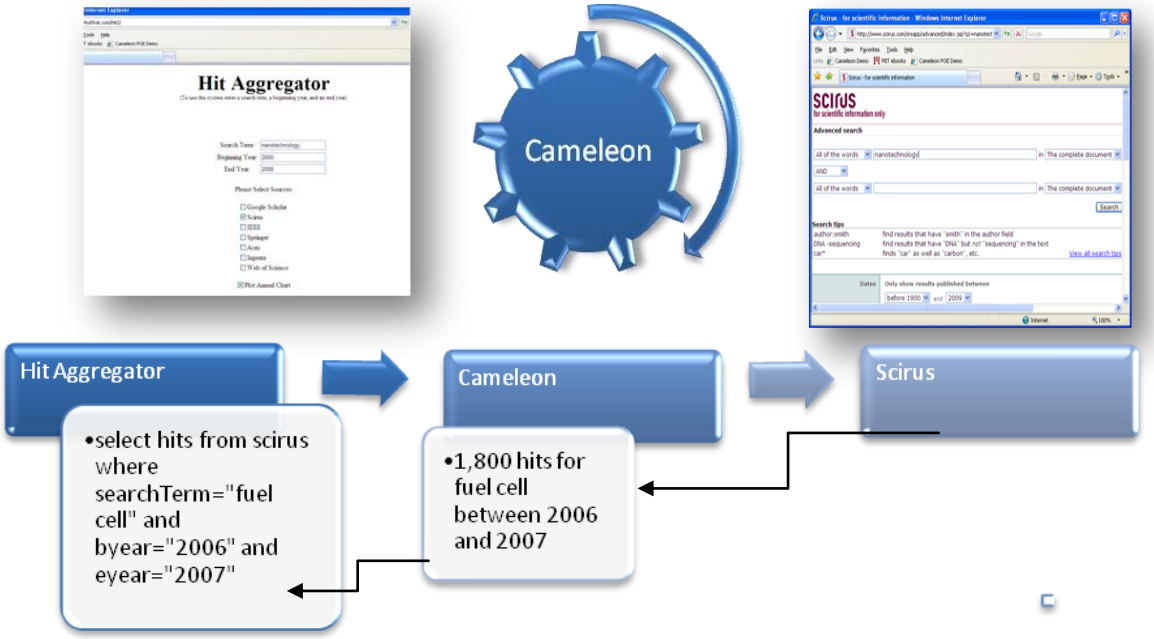

Figure 2: Simple SQL query against the wrapped Scirus Search Engine.

A sample Cameleon# spec file is shown in Figure 3 in XML format. In this spec file, the Web address of Scirus is indicated in the SOURCE tag. The input attributes (searchTerm, bYear, and eYear) are enclosed within # signs, and are expected from the user. The BEGIN and END tags specify (in regular expressions) landmarks preceding and following the data of interest. Finally the pattern specifies the regular expression for the data to be extracted. Figure 3 also shows an actual snapshot from the Scirus web site.

```
<?xml version="1.0" encoding="UTF-8"?>

<RELATION name="scirus">

 <SOURCE

URI="http://www.scirus.com/srsapp/search?sort=0&amp;t=all&amp;q=#searchTerm#&amp

;cn=all&amp;co=AND&amp;t=all&amp;q=&amp;cn=all&amp;g=a&amp;fdt=#byear#&amp;td

t=#eyear#&amp;dt=all&amp;ff=all&amp;ds=jnl&amp;ds=nom&amp;ds=web&amp;sa=all">

  <ATTRIBUTE name="hits" type="string">

   <BEGIN><![CDATA[1-10]]></BEGIN>

   <PATTERN><![CDATA[of\s(.+?)\s]]></PATTERN>

   <END><![CDATA[hits]]></END>

  </ATTRIBUTE>

 </SOURCE>

</RELATION>
```

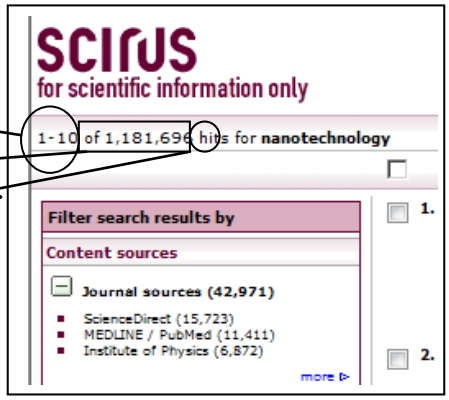

Figure 3. Cameleon# Spec File for Scirus Database

The Cameleon data, then can be used in application programs such as the Hit Aggregator we developed for our technology forecasting project as shown in Figure 4.

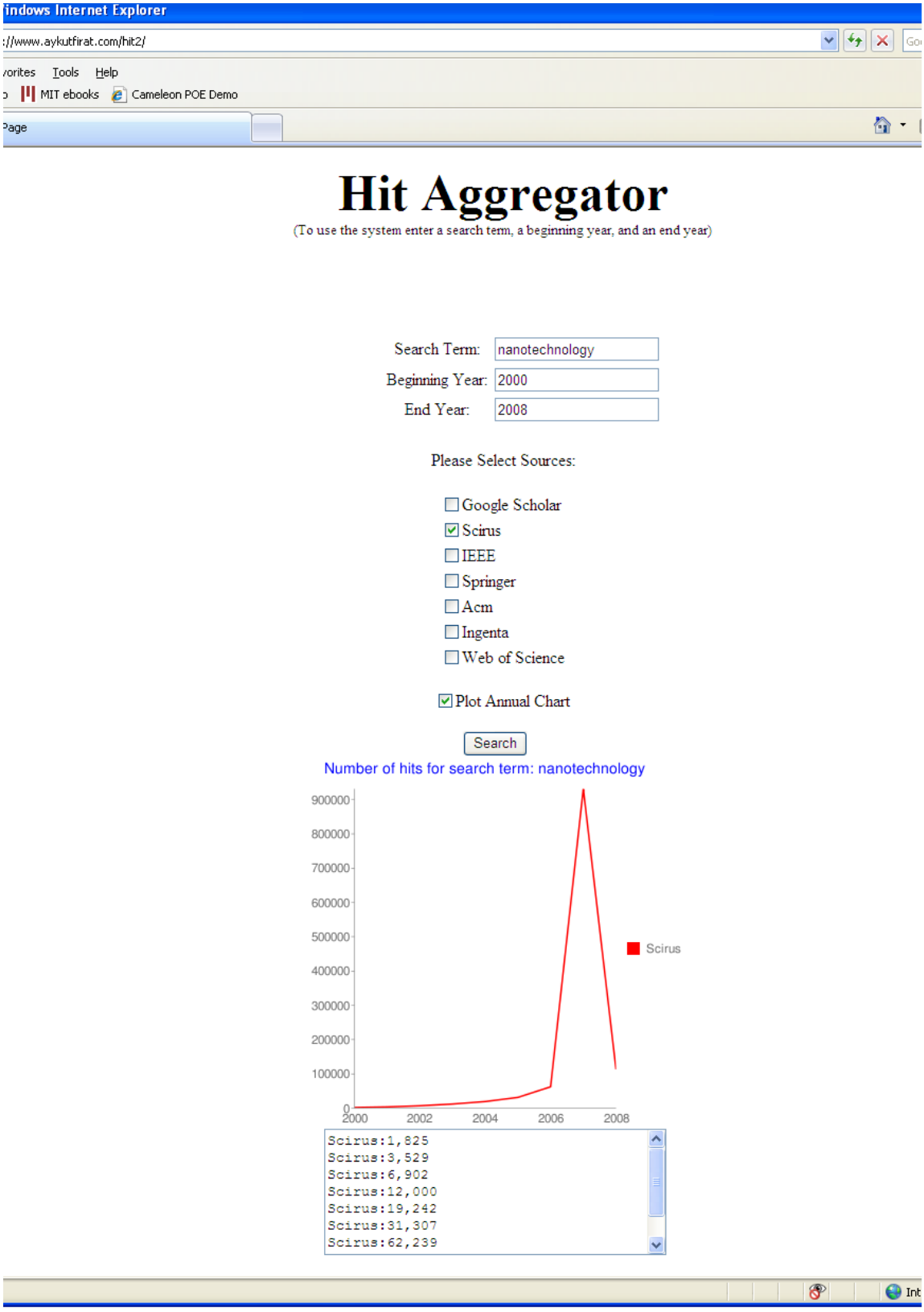

Figure 4. Hit Aggregator we developed for our technology forecasting project.

2.2. Perl based wrapper:

The Perl-based [2] wrapper approach was used by John Baker [3] with the objective of leveraging pre-existing wrapper software to meet our project needs without too much effort. Specifically, John uses an open source Perl tool called WebSearch.pl which leverages the back-end WWW::Search Perl libraries. With these tools he was able to submit queries against the following search engines on the right. Unfortunately, none of these files correspond to the publication databases online.

These .pm files already existed on the web, and each corresponds to a large program dedicated to a single web site. An example file (CraigsList.pm) is shown in the Appendix 1. In order to wrap other web sources of interest, similar programs need to be devised.

An example output of the AltaVista wrapper, only outputting 5 URLs, and in verbose mode gives a title and description as shown below:

- AltaVista.pm
- CraigsList.pm
- Crawler.pm
- Excite/News.pm
- ExciteForWebServers.pm
- Fireball.pm
- FolioViews.pm
- Gopher.pm
- HotFiles.pm
- Livelink.pm
- MetaCrawler.pm
- Metapedia.pm
- MSIndexServer.pm
- NetFind.pm
- Newturfers.pm
- PLweb.pm
- Profusion.pm
- Search97.pm
- SFgate.pm
- Timezone.pm
- Verity.pm
- Yahoo.pm

Request:

```
jab@jabTab][17:41 GMT]=> WebSearch -e AltaVista -m 5 "alternate energy" –verbose
```

Output:

```
2. (title: Alternate Energy Resource Network,
    description: We provide the latest information about alternative energy, solar energy and fuel cells with daily updated industry news, articles and renewable energy resources)
        http://www.alternate-energy.net/
3. (title: Alternate Energy Solutions,
    description: Alternate Energy Solutions Inc. Energy Solutions, Which Work ... photovoltaic modules and alternate energy solutions powering industrial and ...)
```

If the user wants a specific output format, the .pm file needs to be modified.

2.3. Python Based Wrapper

Python [4] based wrapper was undertaken by Prof. Woon [5], and wrappers were developed for Google Scholar, IngentaConnect, Scirus, ACM Guide, SpringerLink and IEEExplore. The Python based wrapper is similar to the Perl-based approach and also utilizes language specific libraries. Like the Perl based wrapper approach, each wrapper is implemented as a program dedicated to a single web site. An example wrapper in Python is shown in Figure 5 for Scirus.

```
# This file contains all functions specific to the individual databases
import re
import urllib
import pdb
import numpy

# Making the bot look like firefox
class myurlopener(urllib.FancyURLopener):
   version="Firefox/2.0.0.7"
urllib._urlopener=myurlopener()
#########################
# Utility functions
#########################
def re_func(result_string,re_string):
   try:
      return int(re.sub("\D","",re.compile(re_string).findall(result_string)[0]));
   except IndexError:
      return 0;
########################
# Functions to generate search terms and regular expressions to extract number of hits
# (database specific bits should be restricted to this part
# Returns [string to pass to urllib,function to extract number of hits from returned webpage]
# Scirus search
#def gen_scirus_search(search_term,search_year=20

#                                                                                    return
["http://www.scirus.com/srsapp/search?t=all&q="+search_term+"&cn=all&co=AND&t=all&q=&cn=all&g=a&fd
t="+str(search_year)+"&tdt="+str(search_year)+"&dt=all&ff=all&ds=jnl&sa=all",lambda
x:re_func(x,"<b>(\S+)\stotal")]

def gen_scirus_search(search_term,search_year=2007):
   if search_year=="":
      return
["http://www.scirus.com/srsapp/search?t=all&q="+search_term+"&cn=all&co=AND&t=all&q=&cn=all&g=a&dt
=all&ff=all&ds=jnl&sa=all",lambda x:re_func(x,"of\s(\S+?)\shits")]
```

```
   else:
      return
["http://www.scirus.com/srsapp/search?t=all&q="+search_term+"&cn=all&co=AND&t=all&q=&cn=all&g=a&fd
t="+str(search_year)+"&tdt="+str(search_year)+"&dt=all&ff=all&ds=jnl&sa=all",lambda
x:re_func(x,"of\s(\S+?)\shits")]
# "Registering" the search functions
search_funcs={};
search_funcs["scirus"]=gen_scirus_search;
def search(search_term,search_year=2007,db="ACM"):
   [search_string,search_re]=search_funcs[db.lower()](search_term,search_year);
   return search_re(urllib.urlopen(search_string).read());
```

Figure 5: An example wrapper in Python is shown for the Scirus online database.

3. Comparison

3.1. Authoring

3.1.1. Authoring wrappers for "difficult pages"

Developing a wrapper in Python/Perl requires basic knowledge of these programming languages, regular expressions, and intimate knowledge of its web related libraries. Wrapping is relatively straight forward, if the data resides in a single page that can be accessed with a static and standard URL. When the developer needs to deal with "a difficult page" involving cookie handling, redirects, form submission, SSL, Javascript interpretation, and passing data from one web page to another, even more code, libraries and external programs are needed. Although these libraries and external programs are available, the final Python/Perl program to wrap a "difficult page" will be complex.

In the case of Cameleon#, the wrapper developer need not know any programming language. With the help of Cameleon Studio (see section 3.1.2), the wrapper developer only needs to learn regular expressions, and the structure of a specification file.

3.1.2. Visual support

One of the most time-consuming aspects of web wrapping is the identification of form elements manually when the target page requires form submission. Without visual tools, the Python/Perl developer needs to use a text editor to identify form elements manually, and potentially introduce errors into the wrapper code. Using Cameleon Studio, a visual application that aids the development of Cameleon spec files, and converting them into web services, this process is more automatic and forms can be added to the spec file with a single click. Furthermore, with a built-in browser Cameleon Studio provides visual support for identifying landmark text and patterns easily.

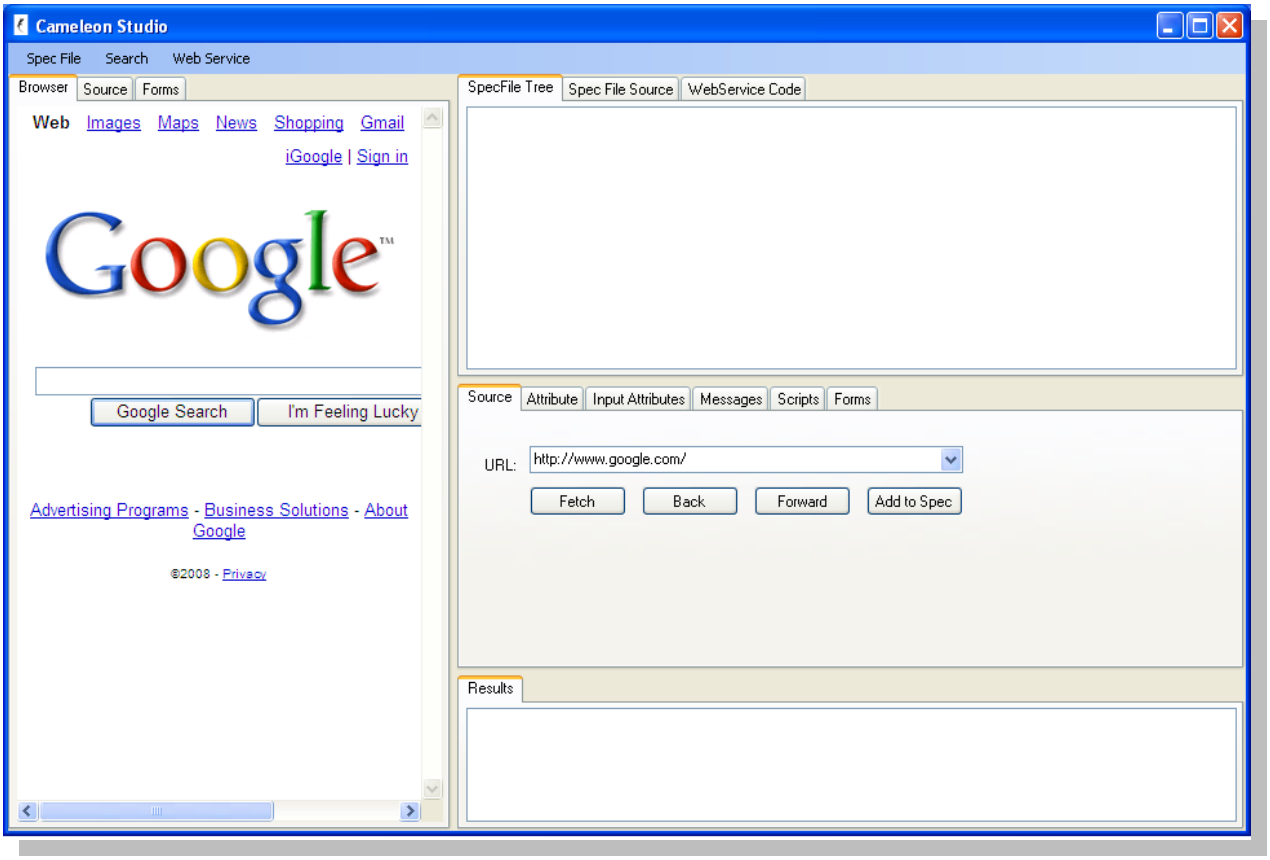

Figure 6: Cameleon Studio Interface.

Cameleon Studio has a built in browser on the left that also shows the source of a web page, and the forms that are in that web page. On the upper right it shows the spec file in tree form, and original form, and the auto produced web service code. On the lower right, it has several tabs for surfing web sites (Sources), defining attributes (Attributes), providing values for input attributes (Input Attributes), displaying messages from the program (Messages), displaying scripts from the web sites, and authoring custom forms (forms). Test results can also be viewed via the Results tab.

3.1.3. Special-purpose debugging

The wrapper developer may sometimes encounter errors when trying to wrap web pages. When the wrapper is developed in Python/Perl, the user is limited to the debugging support of the programming environment. Users of Cameleon Studio, however, are provided with special purpose debugging support. They can, for instance, visually find out what text the regular expression patterns match with a single click, and observe a simulated run of the specification file. This special-purpose debugging support is an important element in speeding up wrapper development.

3.2. Maintenance

One of the fragile aspects of wrapper development is the autonomy of web pages, and their tendency to change their page structure frequently. Wrappers, therefore, need to be updated when the patterns no longer match the desired information in the page. Maintainability of wrappers thus needs to be considered in comparing wrapper development approaches.

3.2.1. Object-Oriented Design Principle: Encapsulate what varies

One of the well-known principles of object oriented software development is separating parts of the code that varies from the parts that stay the same. The primary consequence of applying this principle is better maintainability. In wrapper development, parts that vary from one wrapper to another are extraction knowledge, such as URLs, patterns, form elements, and so on. In Cameleon# these varying parts have been separated from the core code that stays the same for all wrappers. This design enables superior maintenance, because updates are limited to the specification files. Developers cannot introduce any unintended errors to the core hidden code.

This is not the case when both the code and extraction knowledge are lumped together in a Python/Perl program. Maintaining these types of wrappers, thus, becomes a more time-consuming and error-prone task. The convolution of code and knowledge in a Python/Perl program hinders automatic maintenance approaches as well. It is hard to automate maintenance, when there is no fixed structure in a wrapper file. When the extraction knowledge is separated from the code, however, automatic maintenance approaches can be devised utilizing the well-defined structure of the specification file.

3.3. Teachability

A wrapper development approach will not gain acceptance if it is not easy to teach and learn. One important criticism against a programming language like Perl is that it allows its developers to author obfuscated code more easily than other languages. The following code from the 2000 Obfuscated Perl Contest is a testimony to the obscurity potential of this language.

```
#:: ::-| ::-| .-. :||-:: 0-| .-| ::||-| .:|-. :||
open(Q,$0);while(<Q>){if(/^#(.*)$/){for(split('-',$1)){$q=0;for(split){s/\|
/:.:/xg;s/:/../g;$Q=$_?length:$_;$q+=$q?$Q:$Q*20;}print chr($q);}}}print"\n";
#.: ::||-| .||-| :|||-| ::||-| ||-:: :|||-| .:|
```

While Python is a more readable language than Perl, it does not fare well in terms of readability when compared to the specification files used in Cameleon#. What is more, specification files are displayed in a tree like structure in Cameleon Studio making it even easier to understand. For novice users, learning Cameleon would be much faster than learning a full fledged programming language.

3.4. Capability

3.4.1. Flexibility

Capabilities of Cameleon# are pre-defined and its code is closed to modification as far as the wrapper developer is concerned. New versions of the code can be released by Cameleon# developers, and its capabilities can be expanded, but the wrapper developer is not expected to undertake this task. This is not the case when developing a wrapper using a programming language such as Python/Perl. Code is open to modification all the time; therefore the developer has the full flexibility of a programming language. If a page with unforeseen intricacies is encountered, the Python/Perl developer can find a way to overcome the problem. For example, certain types of Google search results are divided in multiple pages, and Cameleon# does not offer an easy way to wrap results dispersed over an unknown number of pages. This is, however, quite easy to do in Python/Perl by using a loop. Another example can be given concerning the use of session IDs. While session IDs are extracted from the web page itself when wrapping with Cameleon#, there are no mechanisms to auto generate these Session IDs. A python based wrapper, on the other hand, can easily embed a function to generate legitimate session IDs.

3.4.2. SQL Interface

Cameleon# primary interface accepts simple SQL queries and returns results in table or XML formats. This has several important advantages: (1) Many programmers are familiar with SQL, so writing requests to extract data from web sites is easy to do (assuming that the spec file has already been created for that web site and (2) there are many software systems and tools that have been developed that use the SQL interface (The Excel spreadsheet software is such an example and explained in the next section.) Wrappers developed in Python/Perl usually have custom-designed interfaces (usually simpler than SQL) or would need to implement similar SQL communication patterns, which would add considerable complexity, in order to be compatible with existing systems.

3.4.3. Excel Integration

Microsoft Excel 2007 allows users to retrieve data from the Web and databases in several ways. By using a web query file[2], we can include SQL queries directed to web sites (which have Cameleon# spec files) and import the Cameleon# results into Excel. This enables Cameleon# users to create elaborate Excel based applications similar to the Hit Aggregator (see Appendix II). As Excel is frequently used in business settings and is very familiar to many people who do not view themselves as "programmers," being able to use simple SQL queries to selectively import web data into Excel is an important advantage. Wrappers developed in Python/Perl need to function as a server, and be able to return results in HTML/XML format in order to replicate this capability.

3.4.4. Error Handling

When a page changes, or something goes wrong during the wrapping process, elegant error handling becomes important. Cameleon#, unfortunately, does not have such good error handling routines. Errors like "Specified cast is invalid" does not tell much about what went wrong. The user needs to go through the debugging process to get an idea about the error. Perl and Python based approaches can have custom error handling as each wrapper is a program by itself, but this, of course, needs to be programmed. A typical hastily written wrapper program will not have much error handling.

3.4.5 Java Script Interpretation

Some web pages are based on Javascript, and sometimes the data of interest may be generated during run time via the execution of some Javascript on the web page. In those cases the wrapper needs to be capable of interpreting JavaScript code and utilizing its output. Cameleon# has already some built in JavaScript interpretation support, but it has not been used in many cases. The code takes advantage of .NET framework's ability to mix languages, and can be extended easily. The Perl and Python based wrapper approaches would need to utilize JavaScript libraries that are being made available by the larger Perl and Python community to accomplish the same task.

Conclusion

For our purposes, wrapper development using Cameleon# is likely to be better than Python and Perl based approaches when we consider the ease of **authoring, maintainability,** and **teachability**. On the other hand, both Python and Perl based programming approaches are more **flexible** especially in dealing with

[2] A web query file is a text file where each line of text is separated by a carriage return. Web query files can be created in any text editor, such as Notepad, and they are saved with the .iqy extension.

pages that are idiosyncratic and cannot be currently handled by Cameleon#. At present, our Perl and Python wrappers have not been extended to address such special cases. We do not see a significant difference between Perl and Python based approaches: both use libraries and scripting to wrap web pages. Utilizing preexisting wrappers written in Perl or Python would make the project move faster. Authoring complicated wrappers in these languages, (unless it cannot be done), is not advisable.

A summary of the comparison of between Cameleon# and Python/Perl programming approach can be found in **Table 1**

	Cameleon#	**Python / Perl**
Authoring	Cameleon# separates code and knowledge to extract web data, the wrapper developer need not know any programming language. With the help of Cameleon Studio, the wrapper developer only needs to learn regular expressions, and the grammar of a specification file.	Developing a wrapper in Python or Perl requires basic knowledge of these programming languages, regular expressions, and knowledge of its web related libraries. To deal with "a difficult page" involving cookie handling, redirects, form submission, SSL, Javascript interpretation, and passing data from one web page to another, even more code, libraries and external programs are needed.
Maintanence	In Cameleon#, the parts that vary from one wrapper to another have been separated from the core code. This design enables superior maintenance, because updates are limited to the specification files. Automatic maintenance approaches can be devised utilizing the well-defined structure of the	In a Pytho/Perl program, both the code and extraction knowledge are lumped together, so maintaining wrappers becomes a time-consuming and error-prone task. The convolution of code and knowledge in a Python/Perl program hinders automatic maintenance approaches
Teachability	Specification files are displayed in a tree like structure in Cameleon Studio making it easier to understand. For novice users, it is clear that teaching Cameleon would be much faster than teaching a full fledged programming language.	Perl allows its developers to author obfuscated code more than other languages. While Python is a more readable language than Perl, it does not fare well in terms of readability when compared to the specification files used in Cameleon#.

Capability	Although, Cameleon# has been used to wrap many different web pages, its capabilities are pre-defined and closed to modification until a new version is released.	Code is open to modification, therefore the developer has the full flexibility of a programming language.

Table 1: A summary of the comparison of between Cameleon# and Python/Perl programming

References

[1] Aykut Firat, Nor Adnan Yahaya, Choo Wai Kuan, and Stéphane Bressan, "Information Aggregation using the Caméléon# Web Wrapper," *Proceedings of the 6th International Conference on Electronic Commerce and Web Technologies* (EC-Web 2005)," Copenhagen, Denmark, August 23 - August 26, 2005, also published in Springer *Lecture Notes in Computer Science* (LNCS) 3590, K. Bauknecht et al (Eds.), pp.76-86, Springer-Verlag Berlin, 2005 [SWP # 4562-05, CISL 2005-06],

[2] Official Perl website: http://www.perl.org/

[3] John Baker, **"Madar Tools Report," Internal Memorandum, MIT Sloan School of Management, April 2008.**

[4] Official Python website: http://www.python.org/

[5] Wei Lee Woon, Stuart Madnick, "Technology Forecasting Using Data Mining and Semantics: MIT/MIST Collaborative Research Progress Report for Period 10/1/07 to 3/31/08," April 2008.

Appendix I – Example of Python .pm file

Craigslist.pm

Here is an example .pm file for Craigslist. Similar .pm files such as AltaVista.pm and Yahoo.pm already exist on the web, and each corresponds to a large program dedicated to a single web site. Unfortunately, none of these files correspond to the publication databases online. In order to wrap new pages of interest, similar programs need to be devised.

```
# $Id: CraigsList.pm,v 1.4 2008/02/01 02:43:31 Daddy Exp $

=head1 NAME

WWW::Search::CraigsList - backend for searching www.craigslist.com

=head1 SYNOPSIS

  use WWW::Search;
  my $oSearch = new WWW::Search('CraigsList');
  my $sQuery = WWW::Search::escape_query("rolex gold");
  $oSearch->native_query($sQuery);
  while (my $oResult = $oSearch->next_result())
    print $oResult->url, "\n";

=head1 DESCRIPTION

This class is a craigslist.com specialization of L<WWW::Search>.  It
handles making and interpreting searches on the infamous Craig's List
website F<http://www.craigslist.com>.

This class exports no public interface; all interaction should
be done through L<WWW::Search> objects.

=head1 NOTES

=head1 SEE ALSO

To make new back-ends, see L<WWW::Search>.

=head1 BUGS

Please tell the maintainer if you find any!

=head1 AUTHOR

Robert Nicholson

=head1 LEGALESE

THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

=head1 VERSION HISTORY
```

```
See the Changes file

=cut

#####################################################################

package WWW::Search::CraigsList;

use strict;
use warnings;

use Carp ();
use Data::Dumper;  # for debugging only
use HTML::TreeBuilder;
use WWW::Search;
use WWW::SearchResult;
use URI;
use URI::Escape;

use vars qw( $VERSION $MAINTAINER @ISA );

@ISA = qw( WWW::Search );
$VERSION = do { my @r = (q$Revision: 1.4 $ =~ /\d+/g); sprintf "%d."."%03d" x
$#r, @r };
$MAINTAINER = 'Martin Thurn <mthurn@cpan.org>';

sub native_setup_search
  {
  my ($self, $sQuery, $rhOptsArg) = @_;
  # print STDERR " +     This is CraigsList::native_setup_search()...\n";
  # print STDERR " +       _options is ", $self->{'_options'}, "\n";

  #$self->http_method('POST');
  $self->user_agent('non-robot');
  # $self->agent_email('user@timezone.com');

  $self->{_next_to_retrieve} = 1;
  $self->{'search_base_url'} ||= 'http://www.craigslist.com';
  $self->{'search_base_path'} ||= "/cgi-bin/search.pl";
  $self->{'_options'} = {
                         areaID => 11,
                         subAreaID => 0,
                         catAbbreviation => 'fur',
                         minAsk => '',
                         maxAsk => '',
                         addTwo => '',
                         query => $sQuery,
                        };
  my $rhOptions = $self->{'_options'};
  if (defined($rhOptsArg))
    {
    # Copy in new options, promoting special ones:
    foreach my $key (keys %$rhOptsArg)
      {
      # print STDERR " +   inspecting option $key...";
```

```
      if (WWW::Search::generic_option($key))
        {
        # print STDERR "promote & delete\n";
        $self->{$key} = $rhOptsArg->{$key} if defined($rhOptsArg->{$key});
        delete $rhOptsArg->{$key};
        }
      else
        {
        # print STDERR "copy\n";
        $rhOptions->{$key} = $rhOptsArg->{$key} if defined($rhOptsArg-
>{$key});
        }
      } # foreach
    # print STDERR " + resulting rhOptions is ", Dumper($rhOptions);
    # print STDERR " + resulting rhOptsArg is ", Dumper($rhOptsArg);
    } # if
  # Finally, figure out the url.
  $self->{'_next_url'} = $self->{'search_base_url'} . $self-
>{'search_base_path'} .'?'. $self->hash_to_cgi_string($rhOptions);

  $self->{_debug} = $self->{'search_debug'} || 0;
  $self->{_debug} = 2 if ($self->{'search_parse_debug'});
  } # native_setup_search


sub preprocess_results_page
  {
  my $self = shift;
  my $sPage = shift;
  if ($self->{_debug} == 77)
    {
    # For debugging only.  Print the page contents and abort.
    print STDERR $sPage;
    exit 88;
    } # if
  return $sPage;
  } # preprocess_results_page

sub parse_tree
  {
  my $self = shift;
  my $oTree = shift;
  print STDERR " + ::CraigsList got tree $oTree\n" if (2 <= $self->{_debug});
  my $hits_found = 0;
  my $oTD = $oTree->look_down(_tag => 'td',
                              width => '50%',
                              align => 'center',
                             );
  if (ref($oTD))
    {
    my $s = $oTD->as_text;
    print STDERR " DDD approx TD is ==$s==\n" if (2 <= $self->{_debug});
    if ($s =~ m!Found: ([0-9,]+) Displaying: [0-9,]+ - [0-9,]+!)
      {
      my $s1 = $1;
```

```
      $s1 =~ s!,!!g;
      $self->approximate_hit_count(0 + $s1);
      } # if
    } # if
  my @aoA = $oTree->look_down(_tag => 'a',
                              sub { defined($_[0]->attr('href')) &&
                                    ($_[0]->attr('href') =~ m!\d+.html\Z!) },
                              );
 A_TAG:
  foreach my $oA (@aoA)
    {
    # Sanity checks:
    next A_TAG unless ref($oA);
    my $oP = $oA->parent;
    next A_TAG unless ref($oP);
    next A_TAG unless ($oP->tag eq 'p');
    my $sTitle = $oA->as_text || '';
    my $sURL = $oA->attr('href') || '';
    next A_TAG unless ($sURL ne '');
    print STDERR " DDD URL is ==$sURL==\n" if (2 <= $self->{_debug});
    my $s = $oP->as_text;
    my $sDate = 'unknown';
    my $p = $oP->as_text;
    if ($p =~ /(\w\w\w-\s?\d*)/)
      {
      $sDate = $1;
      } # if
    my $sPoster = 'unknown';
    my $hit = new WWW::SearchResult;
    $hit->add_url($sURL);
    $hit->title($sTitle);
    $hit->change_date($sDate);
    $hit->source($sPoster);
    push(@{$self->{cache}}, $hit);
    $hits_found++;
    } # foreach A_TAG
  return $hits_found;
  } # parse_tree

sub strip
  {
  my $self = shift;
  my $s = &WWW::Search::strip_tags(shift);
  $s =~ s!\A\s+  !!x;
  $s =~ s!  \s+\Z!!x;
  return $s;
  } # strip

1;

__END__
```

Appendix II - Pulling Data into MS Excel using Web Queries

MS Excel 2007 allows users to use a Web query to retrieve refreshable data that is stored on the Internet, such as a single table, multiple tables, or all of the text on a Web page. For example, you can retrieve and update stock quotes from a public Web page or retrieve and update a table of sales information from a company Web page.

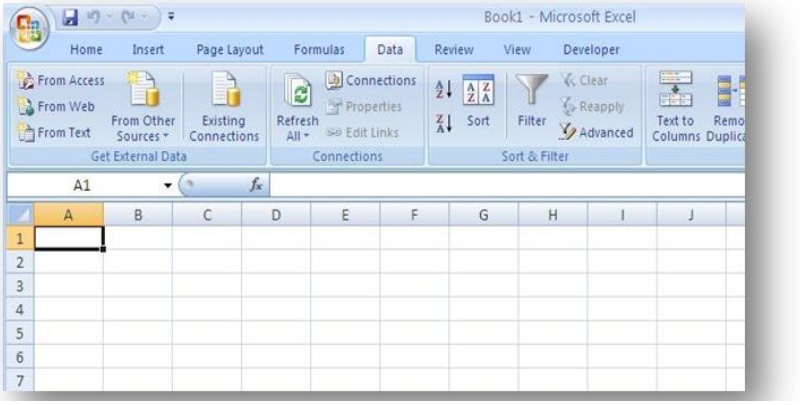

Figure 1: Excel allows users to pull data from Access, Web, Text & other sources.

Web queries are especially useful for retrieving data that is in tables or preformatted areas. (Tables are defined with the HTML <TABLE> tag. Preformatted areas are often defined with the HTML <PRE> tag.) The retrieved data does not include pictures, such as .gif images, and does not include the contents of scripts.To create a Web query, the user needs access to the World Wide Web (WWW). Below is a step by step guide to import data into Excel from Yahoo! Finance (http://finance.yahoo.com).

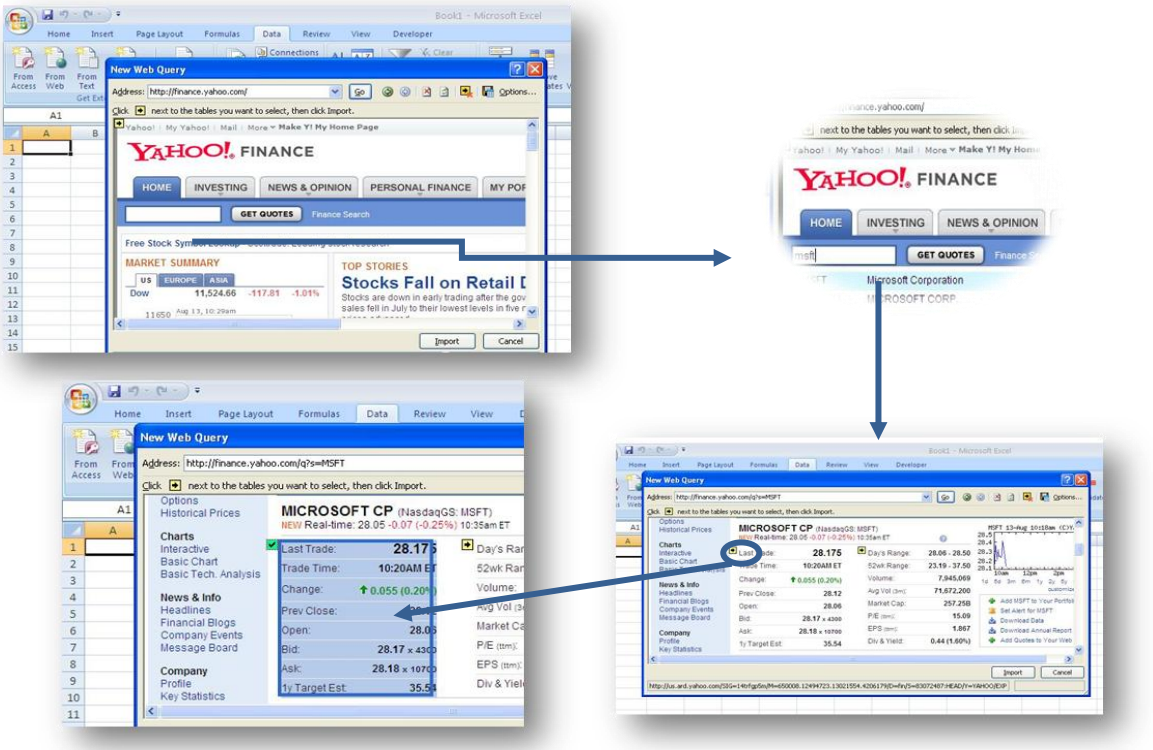

Figure 2: Importing data into Excel from Yahoo Finance.

Step 1: Go to Data > From Web > New Web Query… Enter the Web address for the page you want to use in the Address box (such as Yahoo! Finance - http://finance.yahoo.com), hit Enter or click Go to load the page (in the Yahoo example, enter a company name to get quotes) as in Figure 2.

Step 2: Select the table you want to extract: When the page appears in the New Web Query window, Excel adds yellow arrow boxes next to every table you can import. As you hover over each arrow box with the mouse, Excel draws a bold blue outline around the related table.Once you find the table you want to extract, click the arrow box (which then changes into a green checkmark). To deselect a table, just click it again.

Step 3: When you've finished selecting all the tables you want, click the Import button at the bottom of the New Web Query window. Select where you want to put the data. You can do this for any static web site with tables in it. Once you click OK, Excel begins to fetch the information it needs. During this time, you'll see an information message appear in your worksheet (…Getting data…) Excel then replaces this message with the downloaded data, as shown in Figure 3.

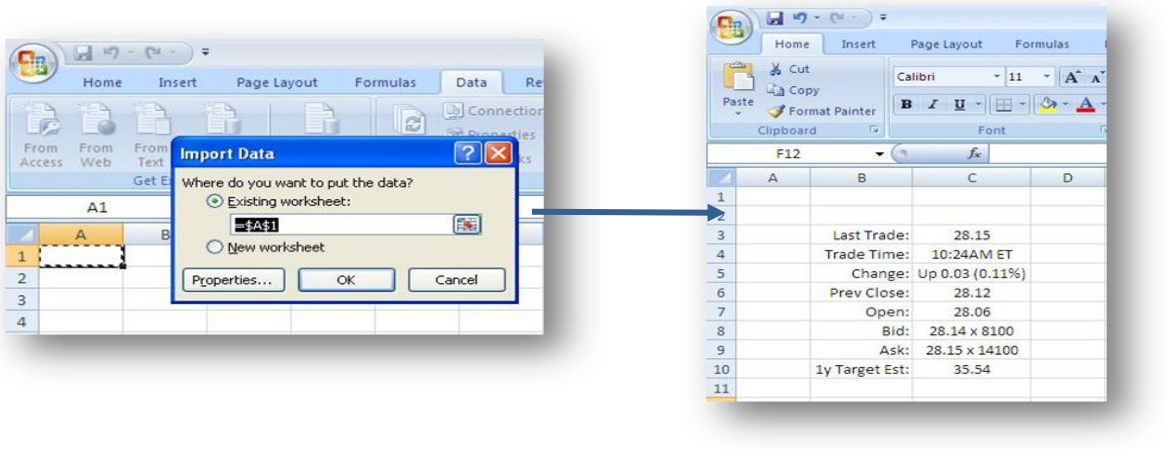

Figure 3: Retrieving data that is in tables or preformatted areas.

Working with XML Files

Similarly, a user can also make query against XML Files. Excel will create a schema based on the XML source data. As in the example in Figure 4, by clicking Data > From Web > New Web Query, and entering the address of the XML file on the left, the data can easily be imported to MS Excel.

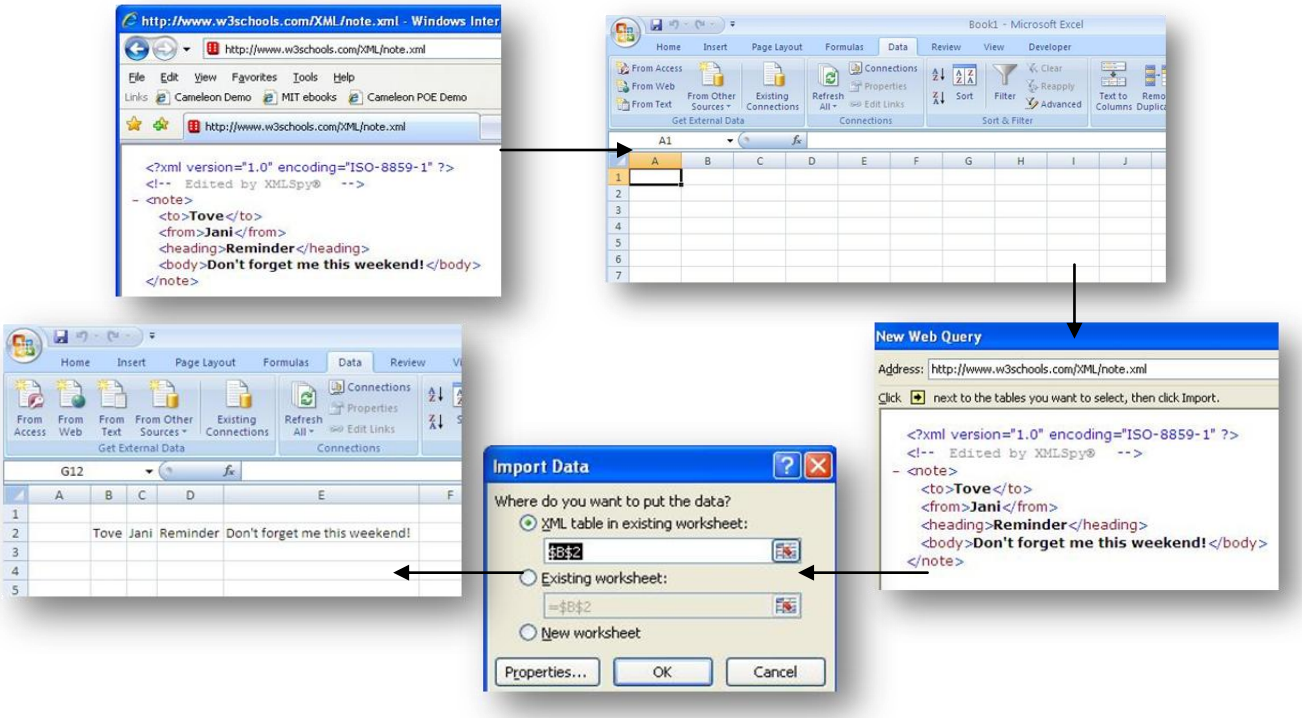

Figure 4: A user can also make queries against XML files.

Calling Cameleon from Excel:

First step calling Cameleon in Excel is to download the web query file for Cameleon (cameleon.iqy) (Figure 5). A web query file is a text file where each line of text is separated by a carriage return. Web query files can be created in any text editor, such as Notepad, and they are saved with the .iqy extension.

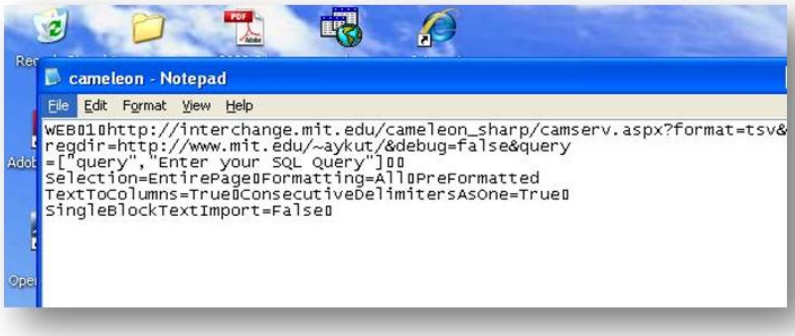

Figure 5: Cameleon.iqy

The rest will be explained with an example. For our Technological Forecasting project, we've created Excel Hit Aggregator, a tool that allows users to get the number of hits (the number of academic papers) for a specific search term from several academic search engines. Below is a screen shot of Excel Hit Aggregator v1 and its query sheet .

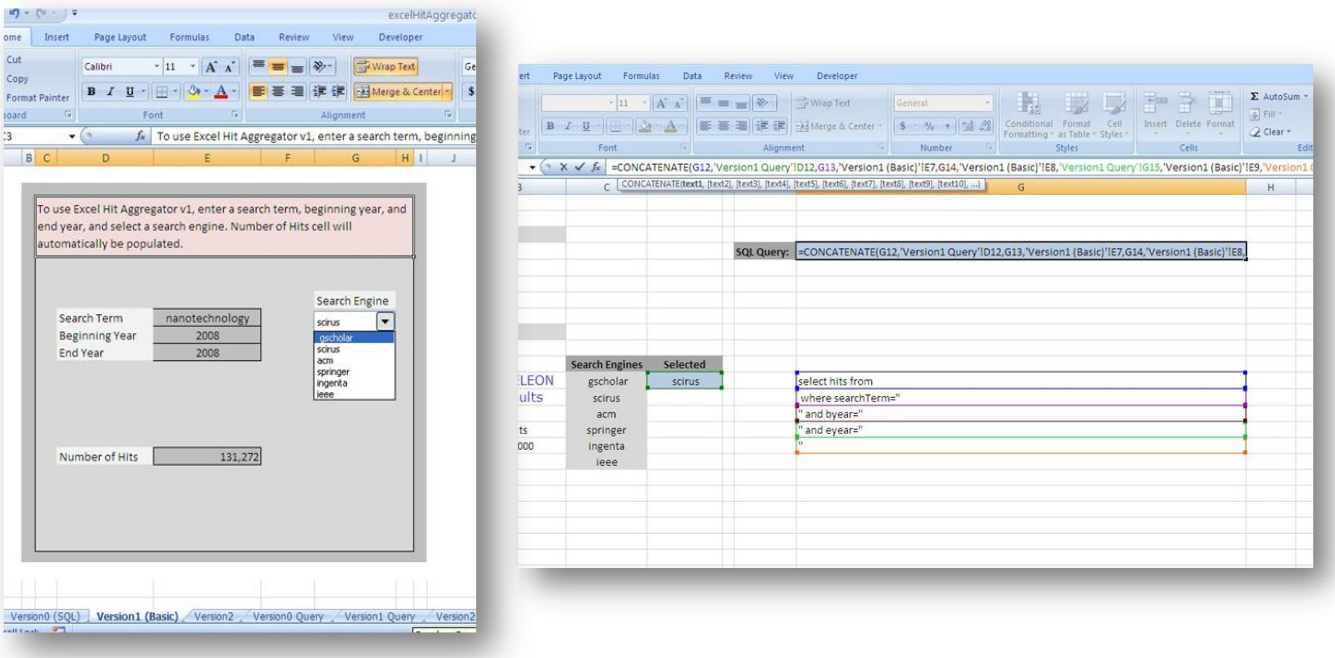

Figure 6: Excel Hit Aggregator v1 and its query sheet.

To create Excel Hit Aggregator v1:

Step 1: Creating a dynamic query: As in Figure 6, write the static portions of your SQL query in Excel cells (such as "select hits from", "where searchTerm=", etc). Then use the concatenate function in Excel, to combine the static portions of the query with the parameters the user will enter (Figure 6, left).

Step 2: Calling Cameleon: Use Data > Existing Connections > Browse for More. Choose cameleon.iqy. Choose the cell you put your dynamic query in Step 1. Click "Use this value/reference for future refreshes" and "Refresh automatically when cell value changes". You will have Cameleon Results in Excel (Figure 7.)

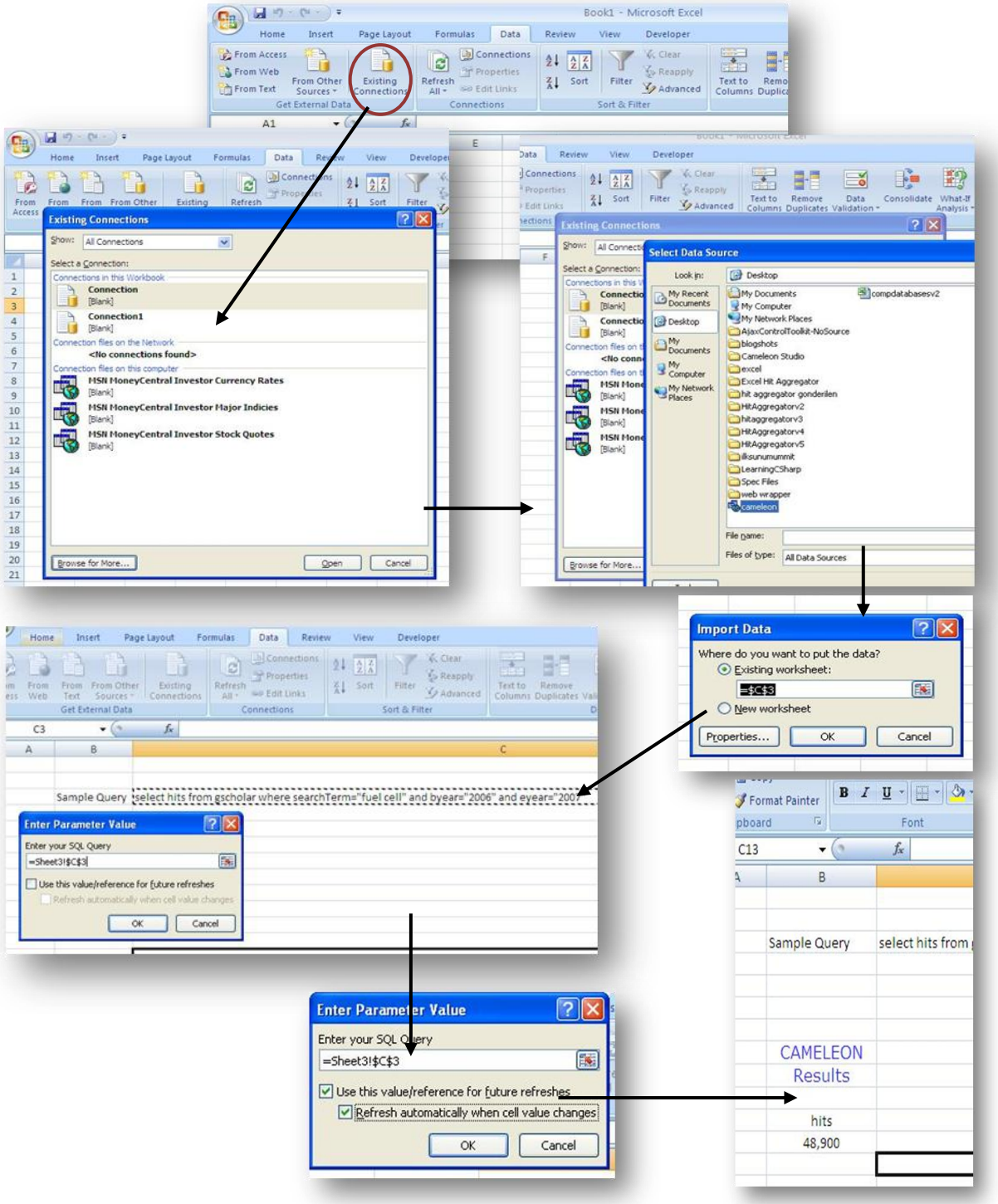

Figure 7: Calling Cameleon from Excel

We used Excel developer form controls to add more functionality to our tool. (Figure8)

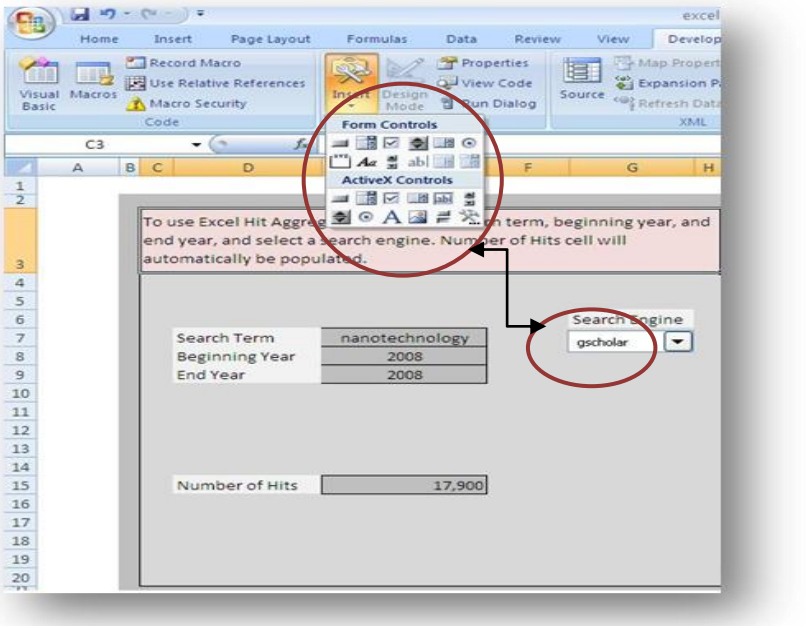

Figure 8: Using Form Controls in Excel Developer

We have also created Excel Hit Aggregator v2 which uses of the Excel graph capability (Figure 9) .

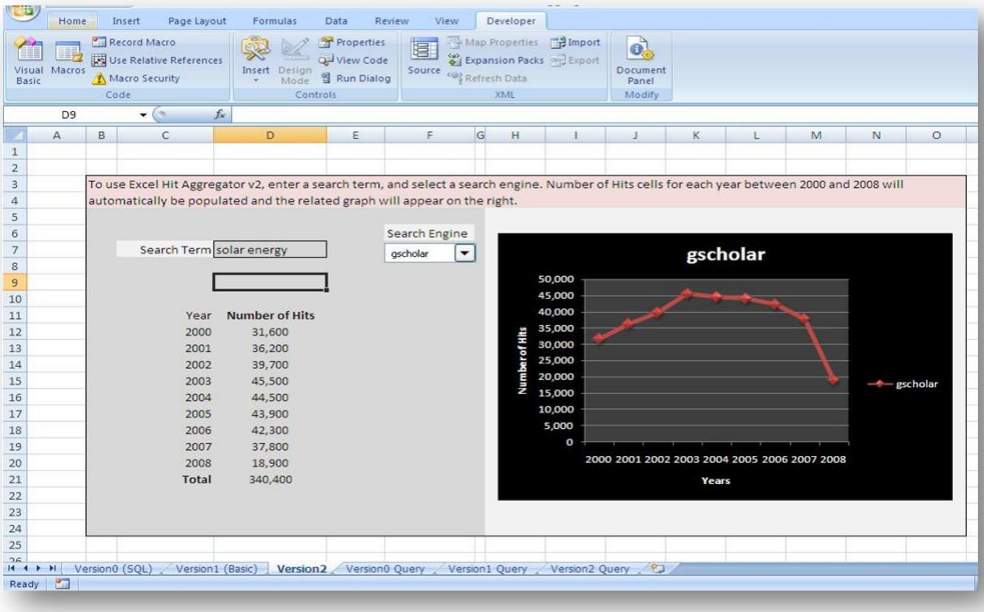

Figure 9: Excel Hit Aggregator v2

Figure 10 shows the v2 SQL query sheet that drives the tables and diagram shown above in Figure 9.

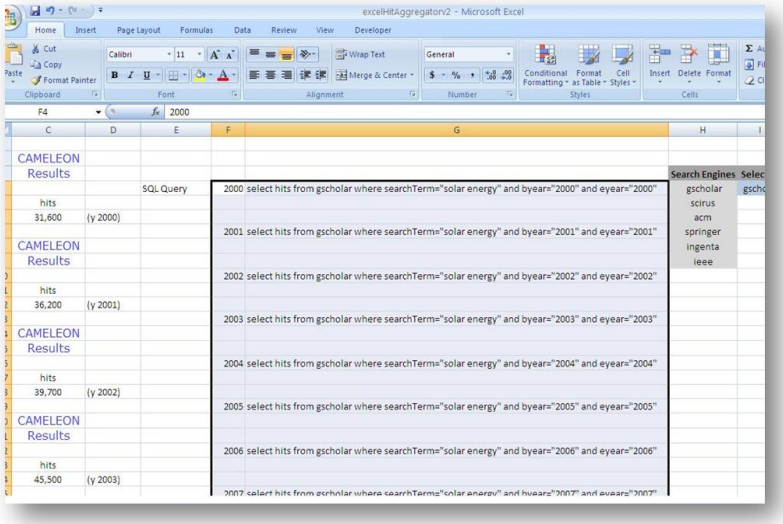

Figure 10: Excel Hit Aggregator v2 query sheet

To help understand the basic working and usage of SQL queries within an Excel spreadsheet, we also created the v0 shown in Figure 11. It shows a single SQL query in the area labelled "Enter the sql query here" that, when executed, displays the results in the area labelled "Number of Hits."

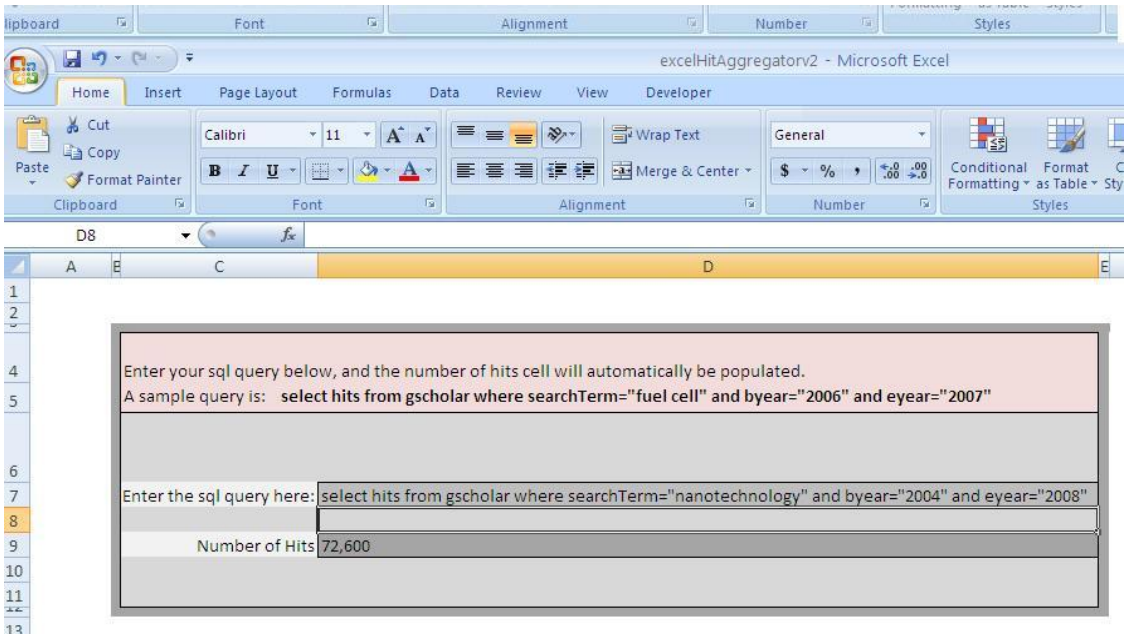

Figure 11: Excel Hit Aggregator v0

Changing Registry Directory: You can create dynamic queries by replacing the value of the parameter in the web query file with:["paramname","Enter the value for paramname:"]. In cameleon.iqy, instead of using (**regdir=**http://www.mit.edu/~aykut/), you can write (regdir=[“regdir”, “Enter registry directory”]. When this modified .iqy file is used, Excel will ask the user to enter registry directory (Figure 12)

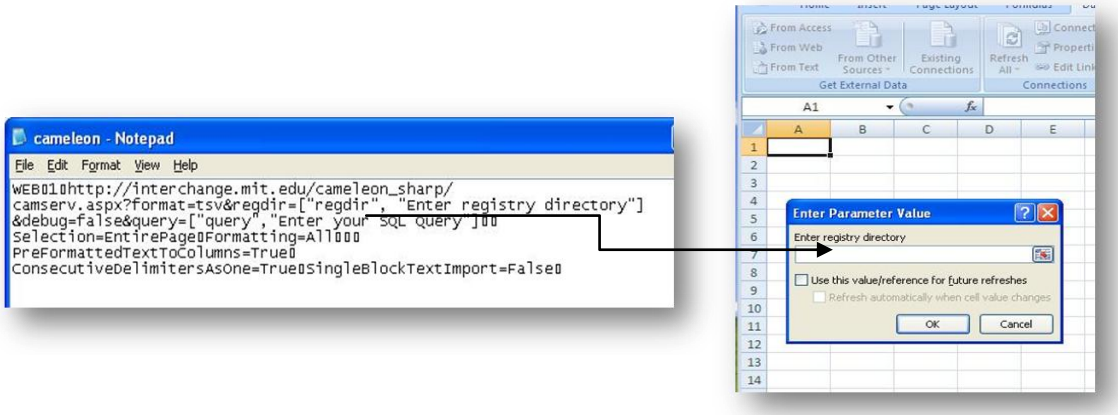

Figure 12: Changing Registry Directory