QUERY REWRITING FOR EXTRACTING DATA BEHIND HTML FORMS

by

Xueqi Chen

A Thesis Submitted to the Faculty of

Brigham Young University

in Partial Fulfillment of the Requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

March 2004

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Xueqi Chen

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	David W. Embley, Committee Chairman
Date	Stephen W. Liddle, Committee Member
Date	Robert P. Burton, Committee Member
Date	David W. Embley, Graduate Coordinator

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Xueqi Chen in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	David W. Embley Chair, Graduate Committee

Accepted for the Department

David W. Embley
Graduate Coordinator

Accepted for the College

G. Rex Bryce
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

QUERY REWRITING FOR EXTRACTING DATA BEHIND HTML FORMS

Xueqi Chen

Department of Computer Science

Master of Science

Much of the information on the Web is stored in specialized searchable databases and can only be accessed by interacting with a form or a series of forms. As a result, enabling automated agents and Web crawlers to interact with form-based interfaces designed primarily for humans is of great value. This thesis describes a system that can fill out Web forms automatically according to a given user query against a global schema for an application domain and, to the extent possible, extract just the relevant data behind these Web forms. Experimental results on two application domains show that the approach is reasonable for HTML forms.

ACKNOWLEDGMENTS

First of all, I would like to thank Dr. David Embley, my graduate advisor, for patiently guiding me throughout my research.

Secondly, I would like to thank my parents and my brother for supporting me and encouraging me constantly throughout my education.

I thank Dr. Stephen Liddle and Dr. Robert Burton for the time and effort they put on my thesis, and I thank the National Science Foundation for financially supporting this research under grant #IIS-0083127.

Finally, I would like to thank all members of the BYU Data-Extraction Group for generously offering me suggestions and help and providing me their code as a basis for my implementation.

Table of Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 Problem and Related Work

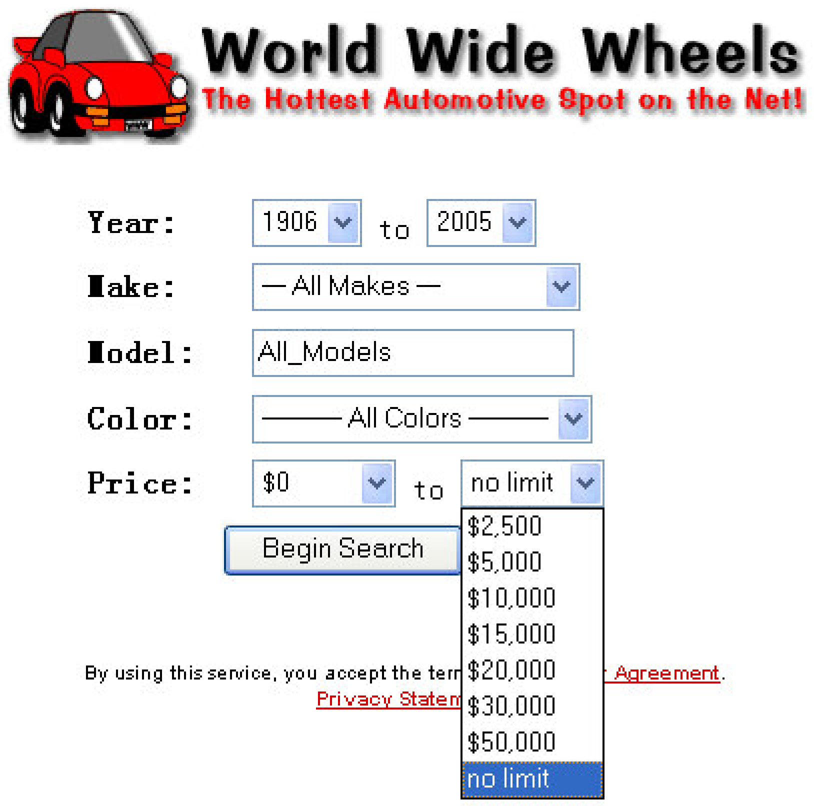
With the enormous amount of information being put on the Internet, databases, which can be accessed by interacting with a form or a series of forms, become a useful and common data management tool for Internet information and service providers. Web forms and dynamically generated pages are helpful to users because users can often get exactly the information they want. It is tedious, however, for users to visit dozens of sites for the same application and fill out different forms provided by each site. As a result, enabling automated agents and Web crawlers to interact with form-based interfaces designed primarily for humans is of great value.

To the best of our knowledge, no other existing form-extraction system considers all the issues mentioned above. The existing BYU form extraction system [LES+02, Yau01], a fully automated system, tries to extract all the information from one Web site (behind one Web form), regardless of what a user wants. The Hidden Web Exposer (HiWE) system [RaG00, RaG01], extends crawlers by giving them the capability to fill out Web forms automatically. HiWE, however, must start with a user-specified list of sources for a particular task, it tends to retrieve all the information behind the sources, and human-assistance is critical to ensure that the Exposer issues queries that are relevant to the particular task. Microsoft's Passport and Wallet system [Mic03] encrypts a user's personal information and then automatically fills out Web forms with the user-provided information whenever it is applicable, but the system makes no attempt to retrieve information behind those forms. The commercial system ShopBot [DEW96] is a general purpose mechanism for comparison shopping. Its form filling process is an automatic but simple process. ShopBot fills each form using a set of domain-specific heuristic rules provided in a domain description. The domain description contains regular expressions encoding synonyms for each attribute. If the regular expression matches the text preceding a field, then the system associates that attribute with the field; if there are multiple matches, the first one listed on the domain description is used; if a match fails, the field is left blank.

1.2 Proposed Solution

There are significant technical challenges in automating the form filling process. First, an automated agent must understand a user's needs by interpreting the user's input or query. Second, an automated agent must understand Web forms, which provide for site queries, and map the user's query to a site query. This is challenging because different Web forms, even for the same application, provide different ways to query their databases.

Figures 1, 2 and 3 show three different Web forms for the same application, used-car searching, from three different information providers. In Figure 1, *Year*, *Make*, *Model*, *Color*, and *Price* are the fields on which a user can query. Figure 2 asks for *Zip Code*, *Make*, and *New or Pre-owned*. In Figure 3, a user must provide values for location (*Zip Code* and *Distance*), but *Price*, *Make*, *Model*, *Year*, and *Key Word* are optional fields.


World Wide Wheels
The Hottest Automotive Spot on the Net!

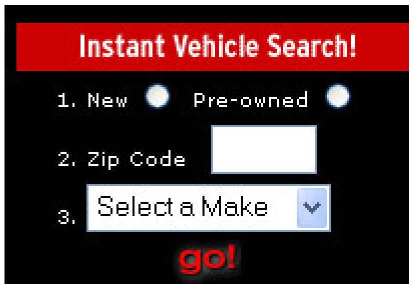
Year:	1906 to 2005
Make:	— All Makes —
Model:	All_Models
Color:	——— All Colors ———
Price:	$0 to no limit

Begin Search

$2,500
$5,000
$10,000
$15,000
$20,000
$30,000
$50,000
no limit

By using this service, you accept the ter Agreement.
Privacy Statem

Figure 1: Web Form for Car Advertisement Search at http://wwwheels.com/cfapps/searchindex.htm, *December, 2003.*


Instant Vehicle Search!

1. New ○ Pre-owned ○

2. Zip Code

3. Select a Make

go!

Figure 2: Web Form for Car Advertisement Search at http://dealernet.com/, *February, 2004.*

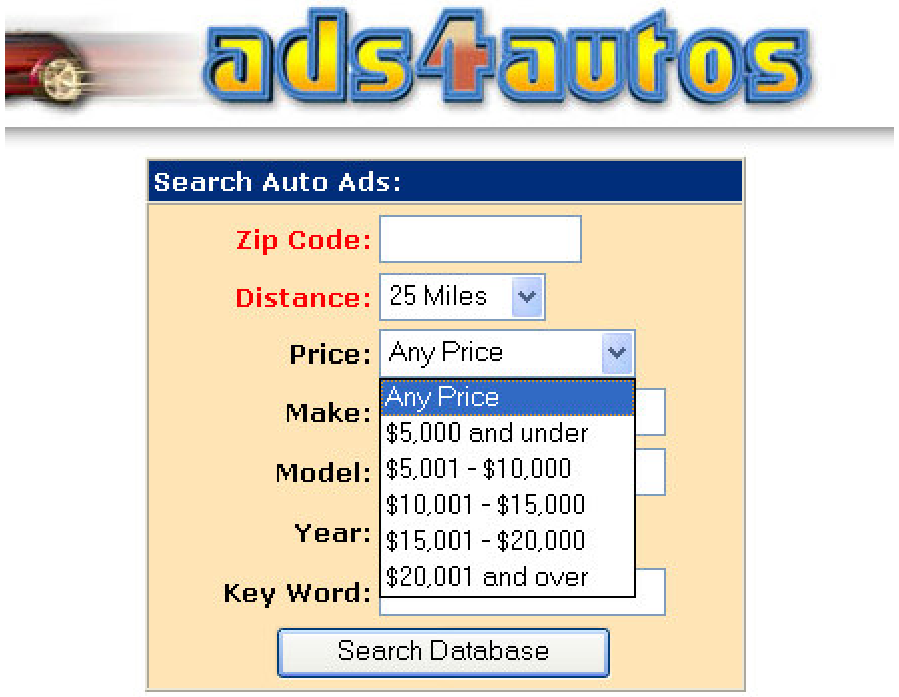


Figure 3: Web Form for Car Advertisement Search at

http://www.ads4autos.com/autos/index.cfm, *December, 2003.*

In addition, information providers can choose to represent their forms using different combinations of radio buttons, checkboxes, selection lists, and text boxes. All of these cause problems in matching a user's query to a site query.

Since Web forms are designed in a variety of ways, handling all kinds of Web forms according to user queries by one automated agent is challenging. Although seemingly simple, direct matches between user-query fields and form fields can be challenging because synonymy and polysemy may make the matching nontrivial. Moreover, problems arise when user queries do not match with form fields. Mismatches occur in the following ways:

(1) Fields specified in a user query are not contained in a Web form, but are in the returned information.

(2) Fields specified in a user query are not contained in a Web form, and are not in the returned information.

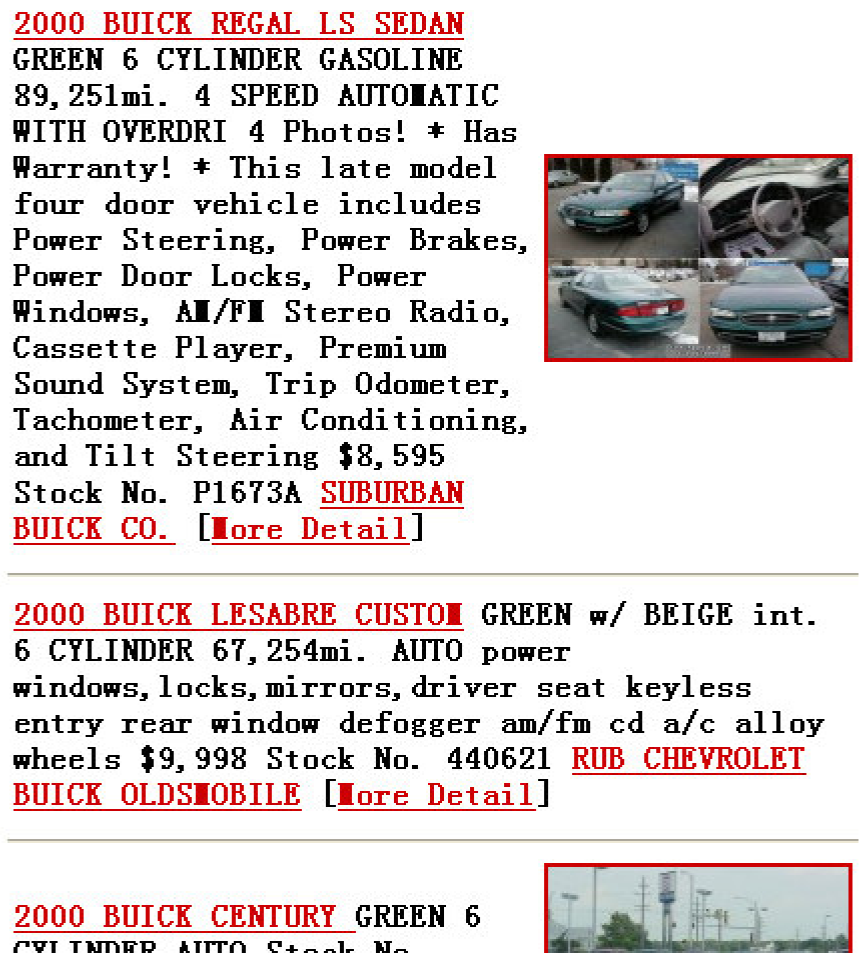
(3) Fields required by a Web form are not provided in a user query, but a general default value, such as “All” or “Any”, is provided by the Web form.

(4) Fields required by a Web form are not provided in a user query, and the default value provided by the Web form is specific, not “All” or “Any”.

(5) Values specified in a user query do not match with values provided in a Web form, which leads to the problem that the desired information cannot be retrieved using a single form query.

To illustrate these problems, consider the three example forms in Figures 1, 2, and 3 and the user query, “Find green cars that cost no more than $9,000.”

The Web form in Figure 1 illustrates Problems 3 and 5. This form illustrates Problem 3 because for all fields other than *Color* and *Price*, which are specified in the query, general values are provided. This form also illustrates Problem 5 because, for *Price*, “$9,000” is not an option for an upper bound value. Thus, our system needs to choose “$10,000” as the upper bound value when filling out the form. The system then needs proper post processing to filter out cars that cost more than $9,000 from the resulting records. Figure 4 shows the partial results after filling out the form in Figure 1 with *Color*=”Green” and *Price* = “$0” to “$10,000” and submitting the query. Since the second car in Figure 4 costs more than $9,000, the system removes this record from the output in its post processing phase.



2000 BUICK REGAL LS SEDAN
GREEN 6 CYLINDER GASOLINE
89,251mi. 4 SPEED AUTO■ATIC
WITH OVERDRI 4 Photos! * Has
Warranty! * This late model
four door vehicle includes
Power Steering, Power Brakes,
Power Door Locks, Power
Windows, A■/F■ Stereo Radio,
Cassette Player, Premium
Sound System, Trip Odometer,
Tachometer, Air Conditioning,
and Tilt Steering $8,595
Stock No. P1673A SUBURBAN
BUICK CO. [■ore Detail]

2000 BUICK LESABRE CUSTO■ GREEN w/ BEIGE int.
6 CYLINDER 67,254mi. AUTO power
windows,locks,mirrors,driver seat keyless
entry rear window defogger am/fm cd a/c alloy
wheels $9,998 Stock No. 440621 RUB CHEVROLET
BUICK OLDS■OBILE [■ore Detail]

2000 BUICK CENTURY GREEN 6

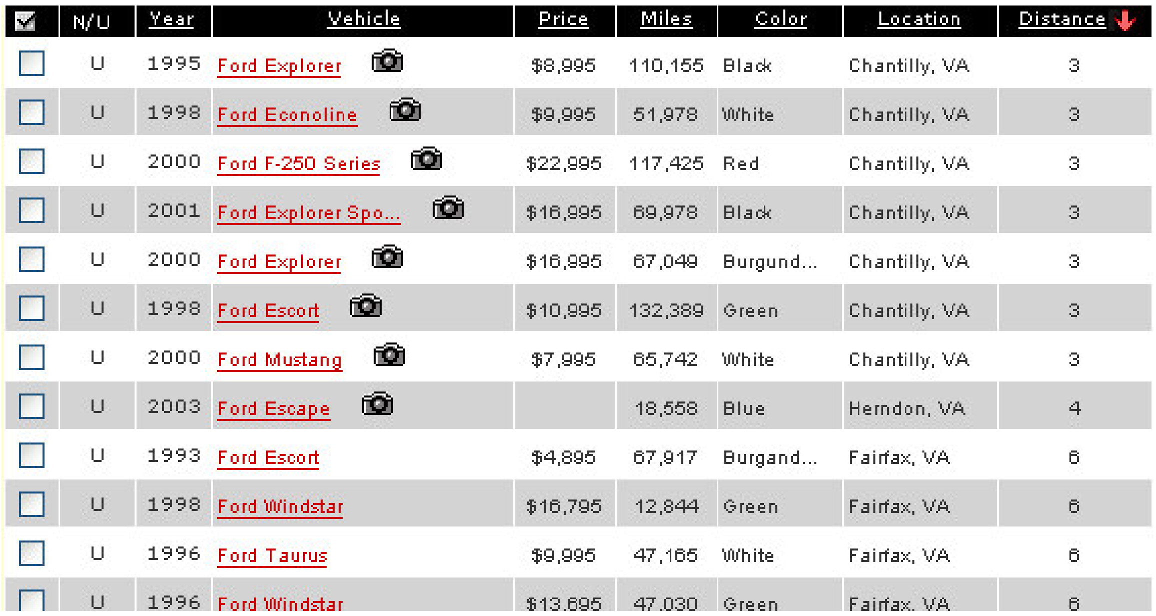
Figure 4: Partial Retrieved Data from http://wwwheels.com/cfapps/searchindex.htm*,*

December, 2003.

The form in Figure 2 illustrates Problems 1 and 4 mentioned above. There is no field about car color, nor about car price in the form, but from the partial retrieved data presented in Figure 5, we notice that both the price and the color information are provided. This demonstrates Problem 1. There is, however, no general value provided for *Make*, so we must search for both "New" cars and "Pre-owned" cars from *Zip Code*

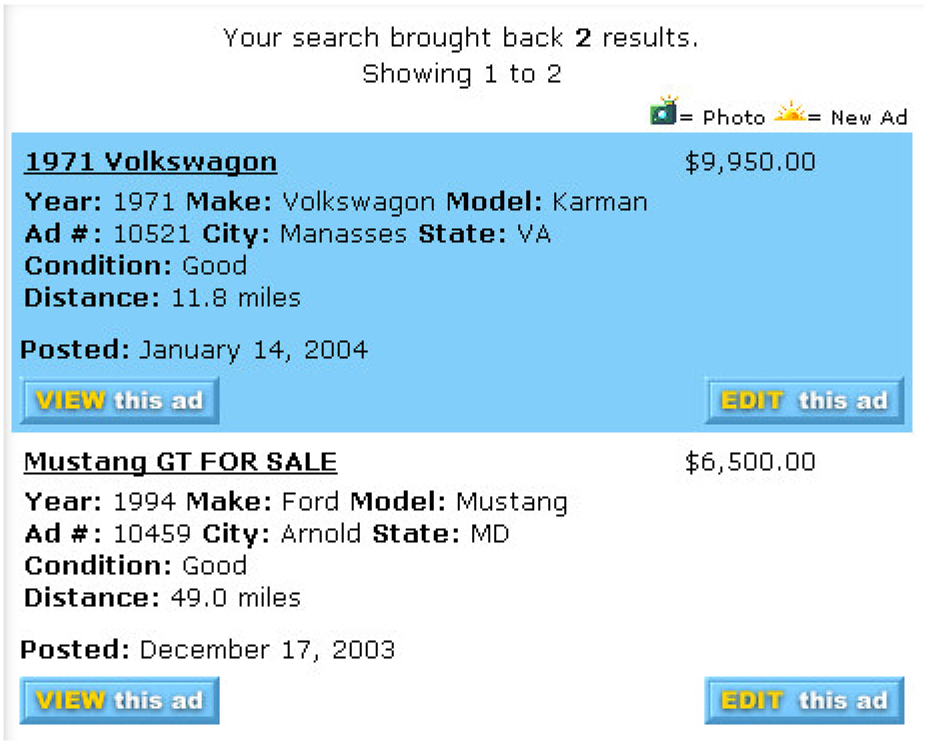
20171 with all *Make* values, one *Make* at a time. This shows Problem 4 listed above. With proper post-processing, we can give a precise and complete set of data to the user.

The form in Figure 3 illustrates Problems 2 and 5. It illustrates Problem 2 since it has no fields for a user to specify car color in the form, and no information about color is provided in the returned information, either. This form leads to Problem 5 because it has a price field, but the field is designed in a way that a user must fill out the form twice in order to get all cars that cost no more than $9,000. Figure 6a is the retrieved data after submitting the query "*Zip Code*= '20171', *Distance*='100 miles', *Price* is between '$5,001 and $10,000'", and Figure 6b is the retrieved data after submitting the query "*Zip Code*= '20171', *Distance*='100 miles', *Price* is '$5,000 and Under'". We should then combine the results from the two submissions and apply post processing in order to get a more precise answer to the user query.



✓	N/U	Year	Vehicle	Price	Miles	Color	Location	Distance
☐	U	1995	Ford Explorer 📷	$8,995	110,155	Black	Chantilly, VA	3
☐	U	1998	Ford Econoline 📷	$9,995	51,978	White	Chantilly, VA	3
☐	U	2000	Ford F-250 Series 📷	$22,995	117,425	Red	Chantilly, VA	3
☐	U	2001	Ford Explorer Spo... 📷	$16,995	69,978	Black	Chantilly, VA	3
☐	U	2000	Ford Explorer 📷	$16,995	67,049	Burgund...	Chantilly, VA	3
☐	U	1998	Ford Escort 📷	$10,995	132,389	Green	Chantilly, VA	3
☐	U	2000	Ford Mustang 📷	$7,995	65,742	White	Chantilly, VA	3
☐	U	2003	Ford Escape 📷		18,558	Blue	Herndon, VA	4
☐	U	1993	Ford Escort	$4,895	67,917	Burgand...	Fairfax, VA	6
☐	U	1998	Ford Windstar	$16,795	12,844	Green	Fairfax, VA	6
☐	U	1996	Ford Taurus	$9,995	47,165	White	Fairfax, VA	6
☐	U	1996	Ford Windstar	$13,695	47,030	Green	Fairfax, VA	6

Figure 5: Partial Retrieved Data from http://dealernet.com/, February, 2004.


Your search brought back **2** results.
Showing 1 to 2

= Photo = New Ad

1971 Volkswagon $9,950.00
Year: 1971 **Make:** Volkswagon **Model:** Karman
Ad #: 10521 **City:** Manasses **State:** VA
Condition: Good
Distance: 11.8 miles

Posted: January 14, 2004

VIEW this ad EDIT this ad

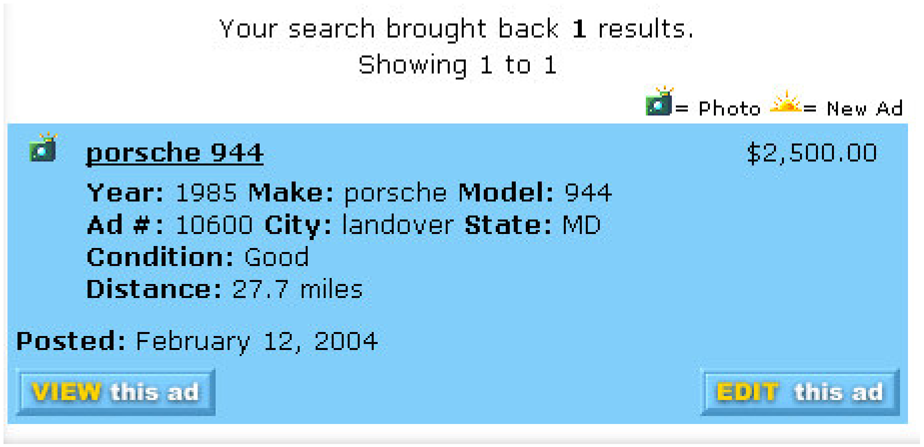
Mustang GT FOR SALE $6,500.00
Year: 1994 **Make:** Ford **Model:** Mustang
Ad #: 10459 **City:** Arnold **State:** MD
Condition: Good
Distance: 49.0 miles

Posted: December 17, 2003

VIEW this ad EDIT this ad

Figure 6a: Partial Retrieved Data for Cars Cost between $5,001 and $10,000 from

http://www.ads4autos.com/autos/index.cfm, *February, 2004.*


Your search brought back **1** results.
Showing 1 to 1

= Photo = New Ad

porsche 944 $2,500.00
Year: 1985 **Make:** porsche **Model:** 944
Ad #: 10600 **City:** landover **State:** MD
Condition: Good
Distance: 27.7 miles

Posted: February 12, 2004

VIEW this ad EDIT this ad

Figure 6b: Partial Retrieved Data for Cars Cost $5,000 and Under from

http://www.ads4autos.com/autos/index.cfm, *February, 2004.*

To solve all the problems mentioned above, we have produced a prototype system designed to fill out Web forms automatically according to a given user query against a global schema. To the extent possible, the system extracts just the relevant data behind Web forms. We have implemented the system as a tool/demo using the Java programming language and Java Servlets technology. We use MySQL as our database management system.

Our prototype system has two central parts, the Input Analyzer and the Output Analyzer. Our Input Analyzer interacts with the user to get a query and a Web site URL with a search form for the chosen domain. Then, it parses the site form, fills in the site form according to the user query, and retrieves relevant Web pages. These Web documents, along with a database schema for the selected domain, are then sent to the Output Analyzer. The Output Analyzer resolves all remaining issues. It retrieves Web data contained in multiple pages using "next" or "more" links, extracts data from all retrieved Web pages and populates our database, removes duplicate and extraneous records, and displays the final results to the user.

1.3 Thesis Overview

We give the details of our solution in the chapters that follow. In Chapter 2, we explain the processes of the Input Analyzer. In Chapter 3, we describe the processes of the Output Analyzer. In Chapter 4, we analyze experimental results and discuss the advantages and disadvantages of our system. Finally, we conclude with summary remarks in Chapter 5, and we mention limitations and possible future work.

Chapter 2

INPUT ANALYZER

Our system starts with a form that allows a user to choose an application from a list. Figure 7 is the form interface. The system obtains the user's selection from the form and sends the extraction ontology of the selected domain to our Input Analyzer.

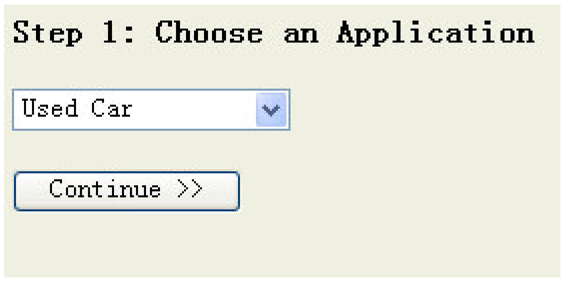


Figure 7: System Starting Page.

The Input Analyzer then parses the extraction ontology (Section 2.1), collects data from the user query (Section 2.2), matches the fields in the user query to the fields in a given site form (Section 2.3), generates a set of one or more queries, and submits the set for processing at the form's site (Section 2.4).

2.1 Extraction Ontology Parsing

An extraction ontology is a conceptual-model instance that serves as a wrapper for a narrow domain of interest such as car ads. The conceptual-model instance consists of two components: (1) an object/relationship-model instance that describes sets of objects, sets of relationships among objects, and constraints over object and relationship sets, and (2) for each object set, a data frame that defines the potential contents of an object set. A data frame for an object set defines the lexical appearance of constant objects for the object set and appropriate keywords that are likely to appear in a document when objects in the object set are mentioned.

Figure 8 is a partial extraction ontology for car ads. An object set in an application ontology represents a set of objects which may either be lexical or nonlexical. Data frames with declarations for constants that can potentially populate the object set represent lexical object sets, and data frames without constant declarations represent nonlexical object sets. *Year* in Figure 8, for example, is a lexical object set whose character representations have a maximum length of 4 characters. *Make*, *Model*, *Mileage*, *Price*, and *PhoneNr* are the remaining lexical object sets in our partial car-ads application ontology; *Car* is the only nonlexical object set.

We describe the `constant` lexical objects and the `keyword`s for an object set by regular expressions using Perl-like syntax. In Figure 8, for example, the constants for *Mileage* are 1-3 digit integers followed by “k” or “K” (plus other possibilities), and the keywords are “miles”, “mi”, “mi.”, and “mileage”. When applied to a textual document, the *extract* clause in a data frame causes a string matching a regular expression to be extracted, but only if the `context` clause also matches the string and its surrounding characters. A `substitute` clause lets us alter the extracted string before we store it in an intermediate file. One of the nonlexical object sets must be designated as the object set of interest, e.g., *Car* for the car-ads ontology, as indicated by the notation “[-> object]” in the first line in Figure 8.

```
Car [-> object];

Car [0:1] has Make [1:*];
Make matches [10]
    constant { extract "\bacura\b"; },
             ...
end;
Car [0:1] has Model [1:*];
Model matches [25]
    constant { extract "\b2.3CL\b"; },
             ...
end;
Car [0:1] has Year [1:*];
Year matches [4]
    constant { extract "\d{2}";
               context "([^\$\d]|^)[4-9]\d[^,\dkK]";
               substitute "^" -> "19"; },
            ...
end;
Car [0:1] has Mileage [1:*];
Mileage matches [8]
    constant { extract "\b[1-9]\d{0,2}k"; substitute "[kK]" -> "000"; },
             ...
    keyword  "\bmiles\b", "\bmi\.", "\bmi\b", "\bmileage\b";
end;
Car [0:1] has Price [1:*];
Price matches [8]
    constant { extract "[1-9]\d{3,5}(\s*(\-+|to)\s*)[1-9]\d{3,5}";
               context "\$[1-9]\d{3,5}(\s*(\-+|to)\s*)\$[1-9]\d{3,5}";},
             ...
end;
Car [0:1] has Color [1:*];
Color matches [20]
    constant { extract "\baqua\s+metallic\b"; },
             ...
end;
Car [0:1] has Transmission [1:*];
Transmission matches [20]
    constant { extract "(5|6)\s*spd\b"; },
             ...
end;
Car [0:*] has Accessories [1:*];
Accessories matches [20]
    constant { extract "\broof\s+rack\b"; },
     ...
end;
Car [0:1] has Engine Characteristics [1:*];
Engine Characteristics matches [10]
   constant { extract "\bv-?(6|8)"; },
             ...
end;
Car [0:1] has Style [1:*];
Style matches [20]
    constant { extract "\b4\s*d(oo)?r\b"; },
             ...
end;
PhoneNr [1:*] is for Car [0:1];
PhoneNr matches [14]
 constant { extract "[1-9]\d{2}-[1-9]\d{2}-\d{4}";
               context "(\b|[^\d])[1-9]\d{2}-[1-9]\d{2}-\d{4}([^\d]|$)"; },
             ...
end;
PhoneNr [0:1] has Extension [1:*];
Extension matches [3]
   constant { extract "\d{1,4}";
               context "(x|ext\.\s+)\d{1,4}\b"; };
    keyword "\bext\b";
end;
```

Figure 8: Partial Car-Ads Ontology.

We denote a relationship set by a name that includes its object-set names (e.g., *Car has Year* and *PhoneNr is for Car* in Figure 8). The *min:max* pairs in the relationship-set name are participation constraints. *Min* designates the minimum number of times an object in the object set can participate in the relationship set, and *max* designates the maximum number of times an object can participate, with * designating arbitrarily many. The participation constraint on *Car* for *Car has Make*, for instance, specifies that a car need not have a listed make and can have at most one make listed, and that there is no specified maximum number of cars that can belong to the same make.

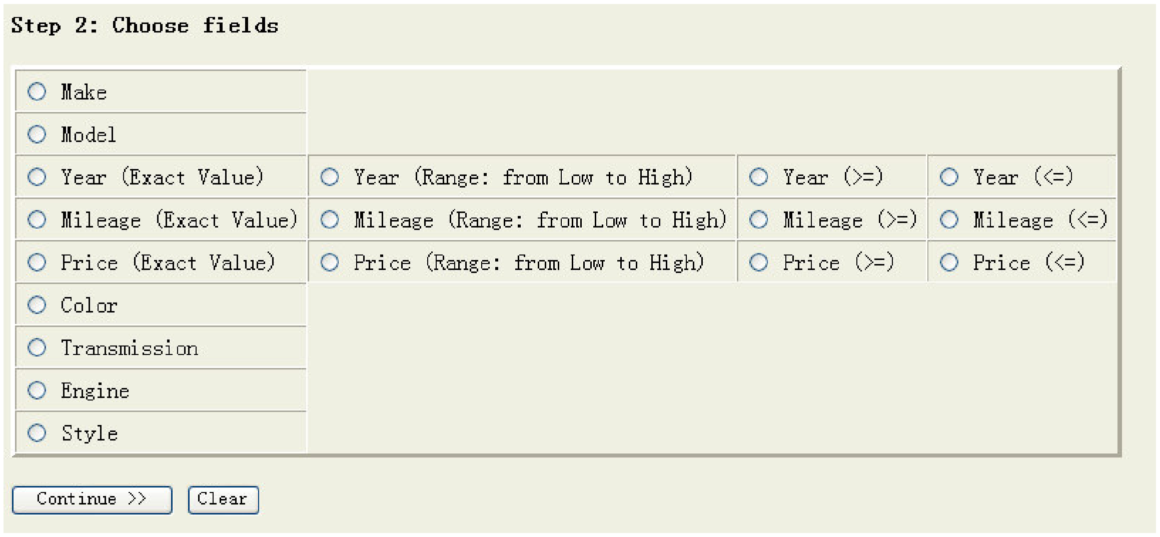
By parsing the ontology, our system obtains a set of object sets, relationship sets, constraints, a set of constant/keyword matching rules, and a database scheme. The system uses the object and relationship sets to acquire a user query; it uses the constant/keyword matching rules to match fields in a user's query to fields in a site form; and it sends the database scheme to the Output Analyzer so that it can be populated with output values.

2.2 User Query Acquisition

Our prototype system provides a user-friendly interface for users to enter their queries[1]. In order to make the interface user-friendly and make query specification easy to understand for our system, we construct an intermediate application-specific form with the lexical object sets of the corresponding application ontology. Figure 9 is a sample intermediate form for the car-ads application. In the intermediate form, our system

[1] There are other ways to obtain a user's query, which may be more conducive to real-world usage. Our prototype system, however, provides what we need for experimenting with automated form filling.

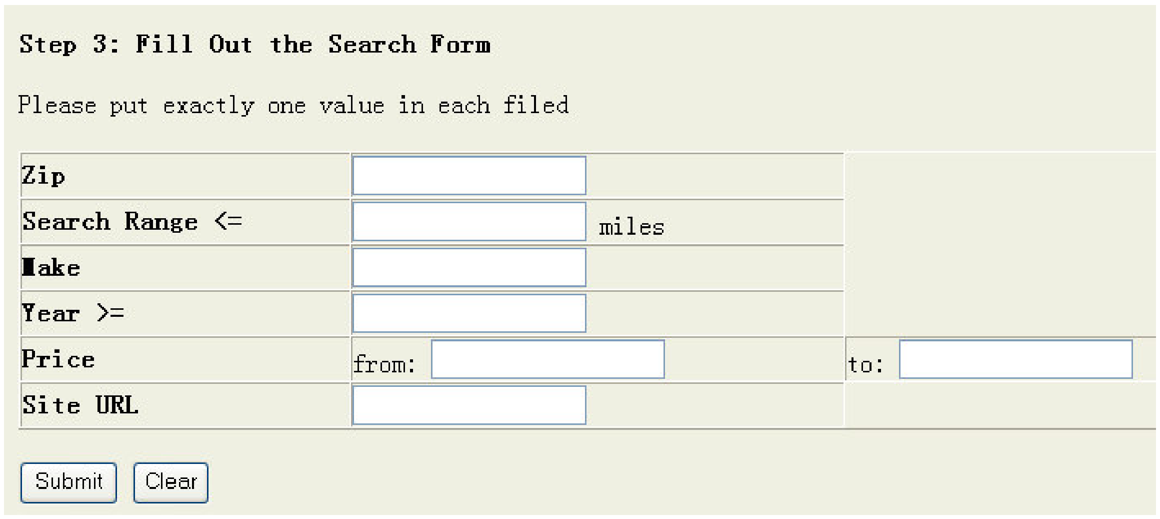
provides the user with a list of fields from which to choose. For fields for which the user can specify ranges, which we call "range fields," our system allows the user to select the type of input — *exact value*, *range values*, *minimum value*, or *maximum value*. After the user selects the desired fields and range types, our system provides a final search form to the user. This search form includes fields selected by the user as well as default fields including *Zip Code*, *Search Range*, and *Site URL*[2]. Figure 10 shows a sample search form after a user chooses *Make*, *Year* with type *minimum value*, and *Price* with type *range value*. Once a user enters a query, our system can parse the query and store each attribute-value pair for later use.


Step 2: Choose fields

○ Make			
○ Model			
○ Year (Exact Value)	○ Year (Range: from Low to High)	○ Year (>=)	○ Year (<=)
○ Mileage (Exact Value)	○ Mileage (Range: from Low to High)	○ Mileage (>=)	○ Mileage (<=)
○ Price (Exact Value)	○ Price (Range: from Low to High)	○ Price (>=)	○ Price (<=)
○ Color			
○ Transmission			
○ Engine			
○ Style			

Figure 9: Sample Intermediate Form for Car-Ads.

[2] Although *Zip Code* and *Search Range* should normally be specified for a car-ads application, the specification of a *Site URL* is for experimenting with our system. There are other ways to specify URLs over which the system can operate. In general, the system is designed to work on multiple URLs so that a user fills in a single query form to query many different sites all having different forms.



Step 3: Fill Out the Search Form

Please put exactly one value in each filed

Zip		
Search Range <=	miles	
Make		
Year >=		
Price	from:	to:
Site URL		

Figure 10: Sample Search Form for Car-Ads with User Selected Fields.

2.3 Site Form Analysis

We assume that the given HTML page has a form that applies to our chosen application. In the case when more than one form is on the page, we consider only the largest form—the one with the largest number of characters between the open and closing `form` tags. We then parse its content into a DOM tree.

Form designers create many fields with `input` tags. For example:

```
<input type="text" size="10" name="zip" maxlength="5" value="">
```

They use `select` and `textarea` to create other fields. An example is:

```
<select name="radius">
      <option value="10">10 miles</option>
      <option value="30">30 miles</option>
      <option value="60">60 miles</option>
      <option value="100" selected>100 miles</option>
      <option value="250">250 miles</option>
      <option value="500">500 miles</option>
</select>
```

Although there are many attributes for the `input` tag, we are only interested in the `type`, `name`, and `value` attributes. After parsing the `input` tag, we store the field name, field type, and field value for fields with type `text`, `hidden`, `checkbox`, `radio`, and `submit`. For the `textarea` tag, we store the field name with field type `textarea`. For the `select` tag, we analyze the content between the opening and closing tags to extract and store the field name, the option values (values inside the `option` tags), and the displayed values (values displayed in the selection list on the Web page).

2.4 Site Form Submission

Our system fills in the site form by generating a query or a set of queries according to the user query and the site form. The form filling process consists of three parts: 1) field name recognition, 2) field value matching, and 3) query generation.

2.4.1 Form Field Recognition

Because site forms vary from site to site, even for the same application domain, site form field recognition is difficult. Because of the way we allow a user to specify a query, field name recognition is essentially a problem of matching the fields in the site form to object sets in our extraction ontology.

We first group all `radio` fields and `checkbox` fields that have the same value for the `name` attribute and consider each group as one field. Then, for fields with values provided, i.e., `select` fields and grouped `radio` fields and `checkbox` fields, we apply our constant/keyword matching rules to determine the field names. If more than

50% of the values in a field belong to the same object set, we conclude that the field corresponds to that object set. For all `input` fields of type "text" and all `textarea` fields in the site form, we compare the field-tag names to the object-set names using similarity measures from zero (least similar) to one (most similar), and we choose the object set with the highest similarity as long as the similarity is above a certain match threshold.

When the field tag names and the object set names are exactly the same, we assign a one to the similarity measure and conclude that there is a match between the two fields. Otherwise, we calculate the similarity between the two strings that represent the names of the object set and the field using heuristics based on WordNet [Mil95, Fel98]. WordNet is a readily available lexical reference system that organizes English nouns, verbs, adjectives, and adverbs into synonym sets, each representing one underlying lexical concept. We use the C4.5 decision tree learning algorithm [Qui93] to generate a set of rules based on features we believe would contribute to a human's decision to declare a potential attribute match from WordNet, namely (f0) same word (1 if A = B and 0 otherwise), (f1) synonym (1 if "yes" and 0 if "no"), (f2) sum of the distances of A and B to a common hypernym ("is kind of") root, (f3) the number of different common hypernym roots of A and B, and (f4) the sum of the number of senses of A and B. We calculate the similarity between an object-set name and a field name based on the set of rules generated by the C4.5 decision tree. If the similarity between the object-set name and the field name reaches a certain threshold, we match the two.

If there is still no match, we calculate the similarity between an object-set name and a field name by a combination of four character-based string matching techniques.

First, we apply standard information-retrieval-style stemming to get a root for each name [Por80]. Then, we combine variations of the Levenshtein edit distance [Lev65], soundex [HD80], and longest common subsequence algorithms to generate a similarity value.

The Levenshtein edit-distance algorithm calculates the number of characters that need to be added, deleted, or changed to transform one string into another. To convert edit distance to a similarity measure, we first normalize the edit distance by dividing it by the length of the object-set name and cap the result at one. Next, we subtract from one since smaller edit distance denoted more similar strings:

$$sim_{Lev} = 1 - min(1, \frac{edit\_dist(name_{os}, name_{df})}{length(name_{os})})$$

The soundex algorithm was developed for automatically recognizing alternate spellings of the same surname in genealogy applications. The algorithm generates a four-character code for a string according to the following rules: 1) The first character in the code is the first letter of the string, and 2) the remaining characters in the code correspond to the next three letters of the string, excluding *A*, *E, I*, *O*, *U, H*, *W*, and *Y*. The letters are divided into six groups of letters that are considered similar, and all the letters in a group generate the same code (i.e., *M* and *N* both generate code 5). Soundex codes are generally compared with an all-or nothing matching approach, but we find this to be too restrictive. For example, *Phone* and *PhoneNumber* have codes *P500* and *P555*, respectively, so the typical matching approach yields zero percent similarity. Instead of all-or-nothing matching, we base our similarity measure on the length of the common prefix for the two four-character codes, so *Phone* and *PhoneNumber* have 50% similarity.

Since any two soundex codes have a common prefix length from zero to four, multiplying by 0.25 yields a similarity between zero and one:

$$sim_{soundex} = common\_prefix\_length(soundex(name_{os}), soundex(name_{df})) * 0.25$$

The longest common subsequence (LCS) algorithm finds the length of the longest (not necessarily contiguous) sequence of characters that appears in both strings. As for edit distance, we normalize the LCS length by dividing by the length of the object-set name to get a similarity measure between zero and one. We find LCS to be more useful than longest common substring, which does not allow for non-contiguous sequences; however, for strings containing common letters, it is possible to recognize completely unrelated sequences. For example, *BusinessEmail* has an LCS length of 5 with both *Email* and *SizeArea*. Our solution is to penalize characters skipped by the LCS in one string or the other by subtracting the number of skipped characters from the LCS length, and we use 0 for negative result, so when compared with *BusinessEmail*, *Email* still has a LCS length of five because 0 character is skipped, while *SizeArea* has an LCS length of zero because 7 characters are skipped:

$$sim_{LCS} = \frac{\begin{gathered}LCS\_len(name_{os}, name_{df}) - \\ min(skipped\_chars(LCS(name_{os}, name_{df})), LCS\_len(name_{os}, name_{df}))\end{gathered}}{length(name_{os})}$$

To calculate the combined similarity between an object-set name and a field name, we combine the similarities from the Levenshtein edit distance, soundex, and LCS calculations. We use a weighted average, giving each of the three components a weight in the combined similarity:

$$sim_{comb} = (sim_{lev} * weight_{lev}) + (sim_{soundex} * weight_{soundex}) + (sim_{LCS} * weight_{LCS})$$

2.4.2 Form Fields Matching

Because fields in a site form do not always match exactly with fields in a user query, we treat different cases in different ways. In this process, we offer solutions to Case 0, which is the direct match between user-query fields and site-form fields and two of the five issues raised in the Chapter 1 (Cases 3 and 4). We also offer a partial solution to Case 5 and leave Cases 1 and 2 and the other part of Case 5 to the Output Analyzer.

Case 0: Fields specified in user query have a direct match in a site form, both by field name and by field value (or values). For example, a user searches for cars around a certain *Zip Code*, and *Zip Code* is a field of type `text` in the site query.

Solution: We simply pair and store the user-provided value with the *Zip Code* attribute.

Case 3: Fields required by a site form are not provided in a user query, but a general default value, such as “All”, “Any”, “Don’t care”, etc. is provided by the site form. For example, a user does not specify any particular *Make* of cars of interest, and *Make* is a field with a `select` tag in a site form with a list of option values including a general default value as a selected default value.

Solution: We find the general display value and pair and store the corresponding option value with the field name.

Case 4: Fields that appear in a site form are not provided in a user query, and the default value provided by the site form is specific, not "All", "Any", "Don't care", etc.. For example, a user does not specify any particular *Make* of cars of interest, and *Make* is a field with a `select` tag in a site form with a list of option values. Unfortunately, no general value is provided in the option list.

Solution: We pair and store the field name with each of the option values provided for the fields by the site form. Later in the submission process, we submit the form once for each name/value pair.

Case 5: Values specified in a user query do not match with values provided in a site form. For example, a user searches the form in Figure 3 for "cars that cost no more than $9,000." The HTML source code for the field *Price* is as follows:

```
<select name="myprice" size=1>
      <option value="">Any Price
      <option value="5000">$5,000 and under
      <option value="10000">$5,001 - $10,000
      <option value="15000">$10,001 - $15,000
      <option value="20000">$15,001 - $20,000
      <option value="20001">$20,001 and over
</select>
```

Solution: This case happens only for range fields. As human beings, we know that we should find the least number of ranges that cover the user's request. For our example, we should submit the form for "$5,000 and under" and "$5,001 - $10,000". For our system, however, the process is not as direct as for a human. The next several pages explain the details for the solution.

Figures 1, 3 and 11 show the three common range structures we can find for range fields in site forms. The *Price* fields in Figure 1 show one kind of range structure (Type 1). In this form, *Price* is actually represented by two form fields, the lower bound value field and the upper bound value field. In this structure, both lower bound and upper bound values are provided explicitly, but they are independent of each other. A user can choose one of the values from the lower bound value set, and choose any value in the upper bound value set to construct a range. The *Price* field in Figure 3 is a different kind of range structure (Type 2). It provides both lower and upper boundary values explicitly in one field. That is, each boundary value in the structure is paired with another value. When a user selects one value for the field, both the lower bound and the upper bound values are specified. In Figure 11, the *Distance* field is a third kind range field (Type 3). It provides one set of boundary values explicitly and implicit indicates the other boundary value. When parsing fields like this one, our system determines whether the explicit values are lower bound values or upper bound values and then assigns an appropriate opposite bound which is a default value supplied by a data frame. As human beings, we can tell directly from looking at the boundary values provided in the selection list whether the values are upper bound or lower bound values, but for our system, this is not obvious.

Even though range fields can be presented in the three different ways mentioned above, they are ultimately the same since they all have both lower bound values and upper bound values. Thus, our system justifies all range fields so that each field contains one set of lower bound values and one set of upper bound values explicitly.

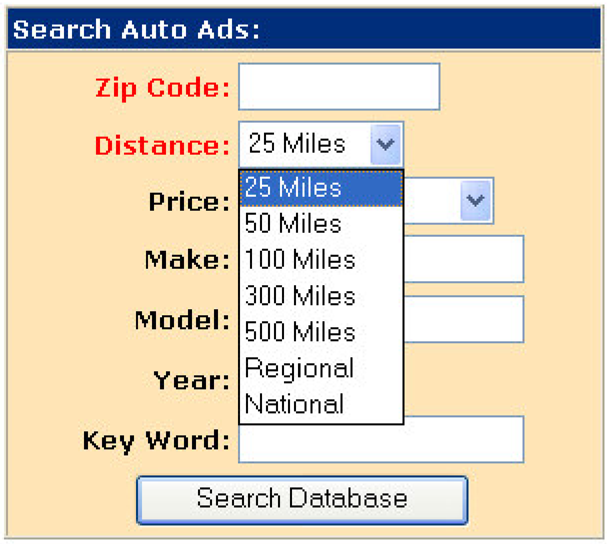


Figure 11: Web Form for Car Advertisement Search at

http://www.ads4autos.com/autos/index.cfm, *December, 2003.*

For Type 1 fields such as the *Price* fields in Figure 1, our system creates two lists, an ordered lower bound value list and an ordered upper bound value list, and sets a flag (Paired) to "false" to indicate that the two ordered value lists are independent of each other. During this process, if our system sees keywords such as "any", "all", "don't care", etc., it sets the value to a default boundary value supplied by a data frame. (All data frames in our system that can participate in ranges always have a default minimum and maximum value.) The following two ordered lists are results from the process:

Lower value list: [0, 1, 5000, 10000, 15000, 20000, 30000];

Upper value list: [2500, 5000, 10000, 15000, 20000, 30000, 50000, 999999].

The "999999" is the default upper bound in the *Price* data frame, which has been substituted for "no limit" in Figure 1. The lower bound value set is fully specified (why the site designers included both “0” and “1” as lower bounds is not clear, but that’s the way the site for Figure 1 was coded).

For Type 2 fields such as the *Price* field in Figure 3, our system separates the two values from each option in the list, puts the first value in an ordered lower bound value list and the second value in an ordered upper bound value list, and sets a flag (Paired) to “true” to indicate that each value in this structure is paired with a corresponding value in the opposite boundary list. Our system also replaces keywords found in the option values with corresponding default boundary values specified in a data frame. The following two ordered lists are results from the process:

Lower value list: [0, 0, 5001, 10001, 15001, 20001];

Upper value list: [999999, 5000, 10000, 15000, 20000, 999999];

The first “0” in the lower bound list is for “Any Price” in Figure 3, and the second is for “and under.” The initial “999999” in the upper bound list is for “Any Price”, and the second is for “and over”. Thus, the pair for “Any Price”, for example, is “0-999999”.

For Type 3 fields such as the *Distance* field in Figure 11, our system first parses each option value to see if the option value is a single number. If more than half of the option values are single numbers, our system examines the list to determine whether the option values contain the default lower bound value from the matching data frame. If so, our system infers that the number values are lower bound values. Otherwise, our system checks to see whether the option values are in ascending order or in descending order. By design convention, when the values are in ascending order, the values are lower bound values. When the values are in descending order, the values are upper bound values. If most of the option values are not single numbers, our system looks for keywords in the option values. If more than 50% of the option values contain keywords such as “or more”, “and above”, “or above”, “or newer”, “at most”, etc., our system infers that the number values are lower bound values. If more than half of the option values contains keywords such as “or less”, “and under”, “or under”, “or older”, “at least”, etc., our system infers that the number values are upper bound values. If our system infers that the value are upper (lower) bound values, it creates a lower (upper) bound value list of the same length with all default minimum (maximum) values. Our system also sets a (Paired) flag to “true”. The following two ordered lists are results from the process for the upper bound values in the *Distance* field in Figure 11:

Lower value list: [25, 25, 25, 25, 25, 25, 25];

Upper value list: [25, 50, 100, 300, 500, 500, 500].

Because the car-ads ontology does not have a data frame for *Distance*, the system creates the lower value list with, 25, the smallest value given in the field. The “Regional” and “National” values are both 500, the maximum value given in the field.

Our system always produces a lower and an upper bound value for range fields in a user query—either because both are given by the user or because one is given by the user and the other is obtained using default values from a data frame. Thus for the query, “cars that cost no more than $9,000,” we obtain “0” as the minimum value for *Price*. Given a lower and upper bound in a query, the field-matching process for range fields becomes simple: match lower to lower and upper to upper by finding the largest value in the “minimum” ordered list that is less than or equal to the query’s low range value and by finding the smallest value in the “maximum” ordered list that is greater than or equal to the query’s high range value.

We illustrate the matching process for the form in Figure 1 with the sample user query, “cars that cost no more than $9,000.” The HTML source for the two *Price* fields is as follows:

```
<select name="PriceLow">
<option value="0">$0
<option value="1">$1
<option value="5000">$5,000
<option value="10000">$10,000
<option value="15000">$15,000
<option value="20000">$20,000
<option value="30000">$30,000
</select>

<select name="PriceHigh">
<option value="2500">$2,500
<option value="5000">$5,000
<option value="10000">$10,000
<option value="15000">$15,000
<option value="20000">$20,000
<option value="30000">$30,000
<option value="50000">$50,000
<option value="no limit" SELECTED>no limit
</select>
```

After interpreting “no limit” in the HTML source code for “PriceHigh” as the default upper bound value for *Price*, 999999, our system translates both the “PriceLow” and the “PriceHigh” fields’ displayed options to the following ordered lists, respectively:

Lower value set: [0, 1, 5000, 10000, 15000, 20000, 30000];

Upper value set: [2500, 5000, 10000, 15000, 20000, 30000, 50000, 999999].

When a human fills in “<= 9,000” for *Price* in the query, “0” for “PriceLow” and “10000” for “PriceHigh” is selected. The system therefore sends the name/value pairs (PriceLow, 0) and (PriceHigh, 10000) to the Web site in its form submissions.

In this example, the matching process is only partially complete because our system chooses ranges broader than what the user wants. In the Output Analyzer, our system filters out irrelevant returned records, i.e. cars with prices higher than $9,000.

2.4.3 Form Query Generation

Once our system has Web form fields paired with values, it can "fill out" the form. In the system, "fill out" the form means generate queries for the site form. Our system selects one name/value pair from each form field, concatenates all selected pairs together, and appends them to the URL constructed from the meta information of the page and the `action` attribute of the `form` tag.

Using the form in Figure 12 as an example, the URL of the Web site is:

```
http://www.carbuyer.com
```

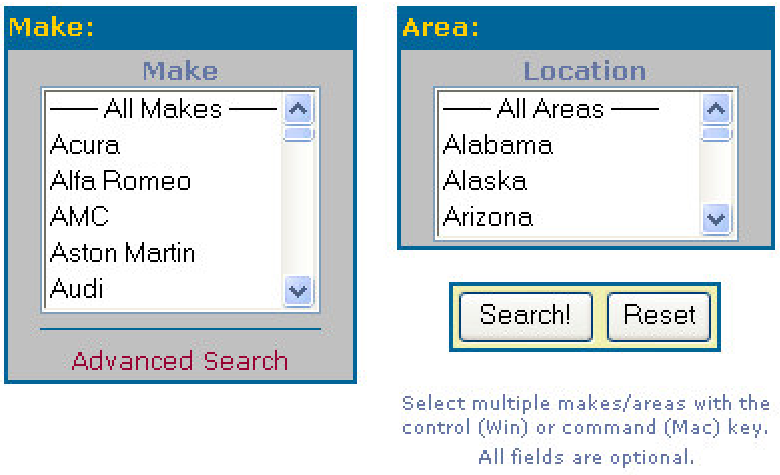
and the source code for the target form is as follows:

```
<FORM ACTION=vehicles.html METHOD=POST>
    <INPUT TYPE=HIDDEN NAME="s" VALUE="">
    <INPUT TYPE=HIDDEN NAME=type VALUE=1>
    ...
    <SELECT multiple name="makeid[]" size="6">
      <OPTION value="-1">------ All Makes ------
      <OPTION VALUE=1>Acura
      <OPTION VALUE=37>Alfa Romeo
      <OPTION VALUE=32>AMC
      ...
      <OPTION VALUE=63>[Other]
    </SELECT>
    ...
    <SELECT multiple name="state[]" size="4">
      <OPTION value="-1">------ All Areas ------
      <OPTION VALUE=AL>Alabama
      <OPTION VALUE=AK>Alaska
      <OPTION VALUE=AZ>Arizona
    </SELECT>
    ...
    <INPUT TYPE=HIDDEN NAME=adv VALUE="">
</FORM>
```

For our sample query, "Find green cars that cost no more than $9000," our output analyzer generates the following string:

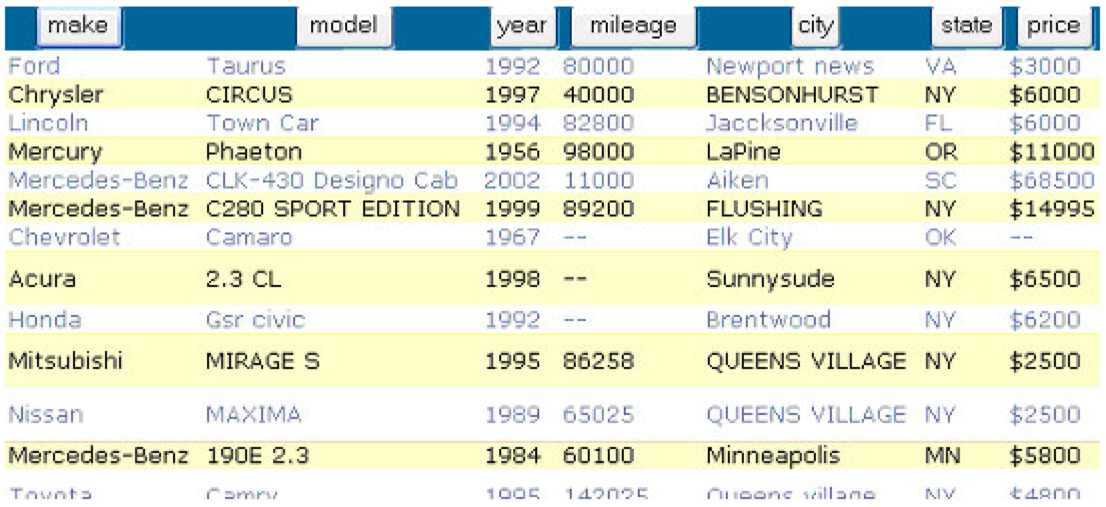
```
http://www.carbuyer.com/vehicles.html?s=&type=1&makeid[]=-
1&state[]=-1&adv=
```

Figure 13 shows partial results for this query. In cases where more than one query can be generated, we submit a query for the site form for every combinations of pairs for the form fields. We collect all Web pages obtained by the set of queries and send them to our Output Analyzer, which we will explain in detail in Chapter 3.



Select multiple makes/areas with the control (Win) or command (Mac) key.
All fields are optional.

Figure 12: Web form for Car Advertisement Search at http://www.carbuyer.com/*, December, 2003.*



make	model	year	mileage	city	state	price
Ford	Taurus	1992	80000	Newport news	VA	$3000
Chrysler	CIRCUS	1997	40000	BENSONHURST	NY	$6000
Lincoln	Town Car	1994	82800	Jaccksonville	FL	$6000
Mercury	Phaeton	1956	98000	LaPine	OR	$11000
Mercedes-Benz	CLK-430 Designo Cab	2002	11000	Aiken	SC	$68500
Mercedes-Benz	C280 SPORT EDITION	1999	89200	FLUSHING	NY	$14995
Chevrolet	Camaro	1967	--	Elk City	OK	--
Acura	2.3 CL	1998	--	Sunnysude	NY	$6500
Honda	Gsr civic	1992	--	Brentwood	NY	$6200
Mitsubishi	MIRAGE S	1995	86258	QUEENS VILLAGE	NY	$2500
Nissan	MAXIMA	1989	65025	QUEENS VILLAGE	NY	$2500
Mercedes-Benz	190E 2.3	1984	60100	Minneapolis	MN	$5800
Toyota	Camry	1995	142025	Queens village	NY	$4800

Figure 13: Partial Initial Results for "Find Green Cars That Cost No More Than $9000" at http://www.carbuyer.com*, December, 2003.*

Chapter 3

OUTPUT ANALYZER

Our system stores the Web result pages the Input Analyzer collects and sends them to the Output Analyzer. The Output Analyzer examines each page and extracts the information relevant for the user query (Section 3.1). The Output Analyzer then filters the extracted information with respect to the user query and displays the final results in HTML format to the user (Section 3.2). At this point, the records displayed to the user are, to the extent possible, just the data relevant to the user's original query.

3.1 Form Results Processor

Sometimes, the results for one query come in a series of pages, but by submitting the query, we can only retrieve the first page of the series. To obtain all results, our system iteratively retrieves consecutive *next* pages. It may need to follow the value of the `href` attribute of an anchor node with the keyword "next" or "more" or a sequence of consecutive numbers that appear as a text child node of the anchor node:

```
<a href=…>…next…</a>
```

or
```
<a href=…>…more…</a>
```

or
```
<a href=…>…n…</a>,
```

where `n` represents a number. Or it may need to submit a form with the keyword "next" or "more" appearing in the `value` attribute of an `input` node with `type` "submit" in the form node:

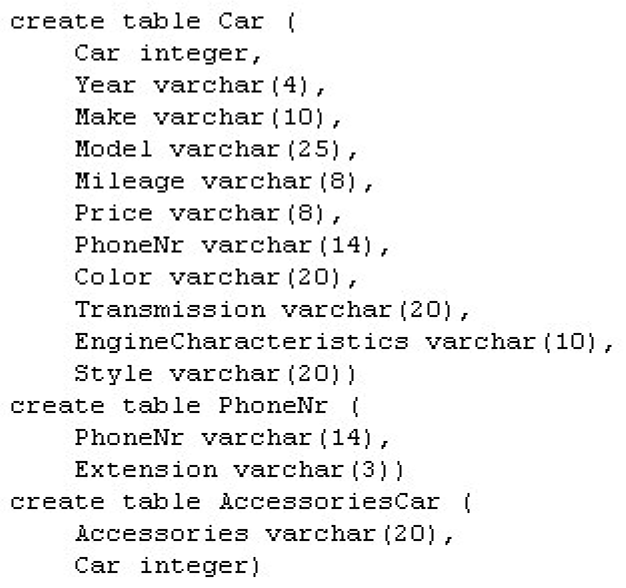
```
<form …>
…
…
…
```

```
       <input value=…next… type=submit …>
       …

or     <form …>
       …
       …
       …
       <input value=…more… type=submit …>
       …
```

To get a complete set of results, our system continues in this way until the last page of the series is reached.

In the next step, the Output Analyzer takes one page at a time, runs it through a record separator [EJN99, SIG01] and then through BYU Ontos [Deg04, ECJ+99], a data extraction system, to populate a database. To extract information from Web pages using Ontos requires recognition and delimitation of records. By "record", we mean a group of information relevant to some entity, e.g. an individual car ad in a car-ads application. Our record separator captures the structure of a page as a DOM tree, locates the node containing the records of interest, identifies candidate separator tags within the node, selects a consensus separator tag, removes all other HTML tags, replaces the separator tag with five pound signs, and outputs the modified page in a "record file". Given the output from the record separator, the next step is to invoke Ontos, which is an ontology-based data extraction engine developed by the Data Extraction Group in the Computer Science department at Brigham Young University [Deg03]. For each document, Ontos produces a data-record table containing a set of descriptor/string/position tuples for the constants and keywords recognized in the record, resolves all conflicts found in the data-record table, and constructs SQL `insert` statements for database tuples from the modified data-record table. Figure 14 shows the database scheme produced by the ontology parser for the ontologies in Figure 8. After the system processes all pages obtained from the Web site, we obtain a fully populated database.



```
create table Car (
    Car integer,
    Year varchar(4),
    Make varchar(10),
    Model varchar(25),
    Mileage varchar(8),
    Price varchar(8),
    PhoneNr varchar(14),
    Color varchar(20),
    Transmission varchar(20),
    EngineCharacteristics varchar(10),
    Style varchar(20))
create table PhoneNr (
    PhoneNr varchar(14),
    Extension varchar(3))
create table AccessoriesCar (
    Accessories varchar(20),
    Car integer)
```

Figure 14: Car-ads Database Scheme.

3.2 Final Results Generator

Even if our system fills in the site form the best it can according to the user query, the results returned may not be the best we can do. Among the cases introduced in Chapter 1, Cases 1 and 5 may lead to extraneous records. Case 1 may produce extraneous records because the input form may not allow us to constrain a field whose values nevertheless appear in the output. Case 5 may produce extraneous records because ranges in the user's query may overlap, rather than coincide, with ranges in the site form.

To prevent extraneous records introduced by Cases 1 and 5 from being displayed to the user, our output analyzer executes an SQL statement corresponding to the user's given query over the records returned to the database where we store our intermediate results. The system generates the SQL statement as follows. First, our system generates a `select` statement for each field specified in the original user query that did not match any field in the site form with a `count` statement to check whether the information for the field is in the results. A sample statement is:

```
select count (*)
from <table name>
where <field name> != "".
```

Our system compares the result C_f, which is the number of records selected, to the number of records C_t selected by the statement

```
select count (*) from <table name>.
```

If the value C_f / C_t is less than or equal to 50%, we claim that information for the field is not provided in the results and output an HTML message to the user that this field is not considered in our search. Otherwise, our system generates another `select count` statement for the field to check if there are any records in our database that have the value the user specified. A sample statement is:

```
select count (*)
from <table name>
where <field name> <operator> <field value>.
```

If the result is zero, our system outputs an HTML message to the user that no record is found and terminates the program.

Then, if there is one non-empty table, our system concatenates all fields with values presented in our database to form the `where` clause. To eliminate the possibility of duplicate records in the final results, our system uses the keyword `distinct` in the SQL statement. The final statement is as follows:

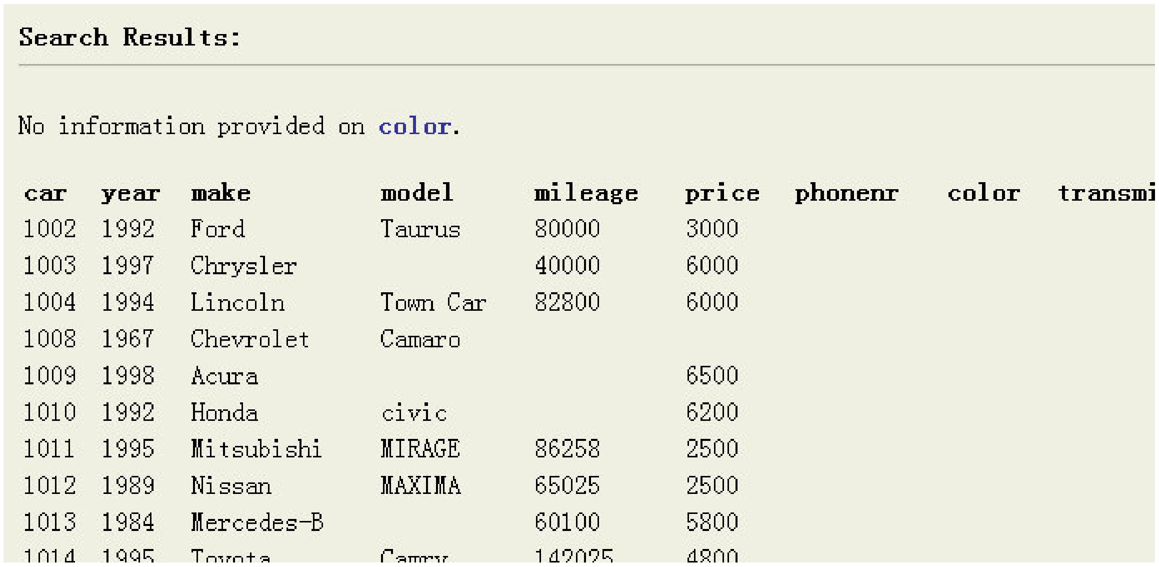
```
Select distinct *
from   <table name>
where  <field1 name> <operator1> <field1 value>
       and  <field2 name> <operator2> <field2 value>
       …
       and  <fieldn name> <operatorn> <fieldn value>
```

where <field$_n$ name> is the nth field the user specified in the original, and `<field`$_n$ `value>` and <operator$_n$> are the value and the operator the user specified for the nth field in the query. The <operator> can be "<=", ">=", or "=".

Finally, we display the final results to the user in HTML format. Figure 15 shows partial final results our system displays to the user after applying the SQL statement

```
select distinct * from car where price <= 9000
```

for the search results shown in Figure 13.


Search Results:

No information provided on color.

car	year	make	model	mileage	price	phonenr	color	transmi
1002	1992	Ford	Taurus	80000	3000			
1003	1997	Chrysler		40000	6000			
1004	1994	Lincoln	Town Car	82800	6000			
1008	1967	Chevrolet	Camaro					
1009	1998	Acura			6500			
1010	1992	Honda	civic		6200			
1011	1995	Mitsubishi	MIRAGE	86258	2500			
1012	1989	Nissan	MAXIMA	65025	2500			
1013	1984	Mercedes-B		60100	5800			
1014	1995	Toyota	Camry	142025	4800			

Figure 15: Partial Final Results Displayed to User for Search Results in Figure 13.

If there is more than one non-empty table for the domain in the database, our system joins all tables and selects distinct qualified records from the database. According to the database scheme generated by Ontos, the first table from the database scheme is always the primary table, which means that all other tables generated from the same process have common fields with this table. If the database has three non-empty tables A, B, and C with the following scheme:

A: [a, b, c, d, e];

B: [a, f];

C: [d, g],

where all fields have values present, and the user query is:

b>=n_1 and f<=n_2 and g=”xx”,

the final SQL statement would be the following:

```
select   distinct A.a, A.b, A.c, A.d, A.e, B.f, C.g
from     A left join B on (A.a=B.a) left join C on (A.d=C.d)
where     b>n1 and f<n2 and g=”xx”
```

Chapter 4

EXPERIMENTAL RESULTS AND ANALYSIS

In this project, we experimented on seven Web sites for each of two applications: car ads and digital camera ads. The approach, however, is not limited to the two applications on which we experiment. It can work with other applications as long as those applications have Web sites with forms, and we have ontologies for those applications. The process of rewriting queries in terms of site forms is the same.

4.1 Experimental Results

We are interested in three kinds of measurements: field-matching efficiency, query-submission efficiency, and post-processing efficiency.

To know if we properly matched the fields in a user query with the fields in a site query, we measure the ratio of the number of correctly matched fields to the total number of fields that could have been matched (a recall ratio R_{fm} for field matching, *fm*), and we measure the ratio of the number of correctly matched fields to the number of correctly matched fields plus the number of incorrectly matched fields (a precision ratio P_{fm}):

$$R_{fm} = \frac{number\_of\_correctly\_matched\_fields}{total\_number\_of\_fields\_that\_should\_have\_been\_matched}$$

$$P_{fm} = \frac{number\_of\_correctly\_matched\_fields}{total\_number\_of\_matched\_fields}$$

To know if we submitted the query effectively, we measure the ratio of the number of correct queries submitted to the number of queries that should have been submitted (a recall ratio R_{qs} for query submission, *qs*), and we measure the ratio of the number of correct system queries submitted to the number of correct queries submitted plus the number of incorrectly submitted queries (a precision ratio P_{qs}):

$$R_{qs} = \frac{number\_of\_correct\_queries\_submitted}{total\_number\_of\_queries\_that\_should\_have\_been\_submitted}$$

$$P_{qs} = \frac{number\_of\_correct\_queries\_submitted}{total\_number\_of\_queries\_submitted}$$

We also conduct an overall efficiency measurement which we obtain by multiplying the three recall measurements and the three precision measurements together:

$$R_{overall} = R_{fm} * R_{qs}$$

$$P_{overall} = P_{fm} * P_{qs}$$

Because the two kinds of metrics measure two stages of one single process, we use the products to calculate the overall performance of the process with respect to our extraction ontology.

4.1.1 Car Advertisements

We experimented on seven Web sites containing search forms for car ads. We issued five queries to each of the sites and obtained the following results. (Appendix A lists the Web sites, and Appendix B lists the queries.) We found 31 fields in the seven forms. Among them, there were 21 fields that are recognizable with respect to our application extraction ontology. The system correctly matched all 21 of them. There were no false positives. According to the five queries, the system should have submitted 146 original queries and 1858 queries for retrieving all next links. Since our submission process deals with all form fields (not just those applicable to the ontology), the system actually submitted 372 original queries and 1863 queries for retrieving next links. If we ignore nonapplicable fields, the system should have submitted 249 original queries and 1847 queries for next links. For just the applicable fields, the system actually submitted 301 original queries and 1858 queries for next links, which includes the 249 original queries and 1847 queries for next links the system should have submitted. Because we do not want to measure the effectiveness of the existing application extraction ontologies and plug in programs, such as the Record Separator and Ontos, and outside the scope of our work, we do not measure their effectiveness (see [ECJ+99, ETL03] for measures of their effectiveness). Finally, we do not measure the effectiveness of the post-processing part of our system because, by itself, it cannot fail. Table 1 shows the precision and recall ratios calculated with respect to recognizable fields for the measurements we made, and it also shows the overall efficiency.

Number of Forms: 7 Number of Fields in Forms: 31 Number of Fields Applicable to the Ontotlogy: 21 (67.7%)			
	Field Matching	**Query Submission**	**Overall**
Recall	100% (21/21)	100% (249/249)	100%
Precision	100% (21/21)	82.7% (249/301) [97.1% ((249+1847)/(301+1858))]*	82.7% [97.1%]*

* Numbers in square brackets are calculated including queries submitted for retrieving next links.

Table 1: Experimental Results for Used-Cars Search.

4.1.2 Digital Camera Advertisements

We experimented on seven Web sites containing search forms for digital camera advertisements. We issued four queries to each of the site and obtained the following results. (Appendix A lists the Web sites, and Appendix B lists the queries.) We found 41 fields in the seven forms. Among them, there were 23 fields that were applicable to our application extraction ontology. The system correctly matched 21 of them. There were no false positives. According to the four queries and the 21 matched fields, the system should have submitted 31 original queries and 85 queries for retrieving all next links. It actually submitted 31 original queries and 85 queries for retrieving next links. Table 2 shows the precision and recall ratios for the three measurements we made, and it also shows the overall efficiency.

Number of Forms: 7

Number of Fields in Forms: 41

Number of Fields Applicable to the Ontology: 23 (56.1%)

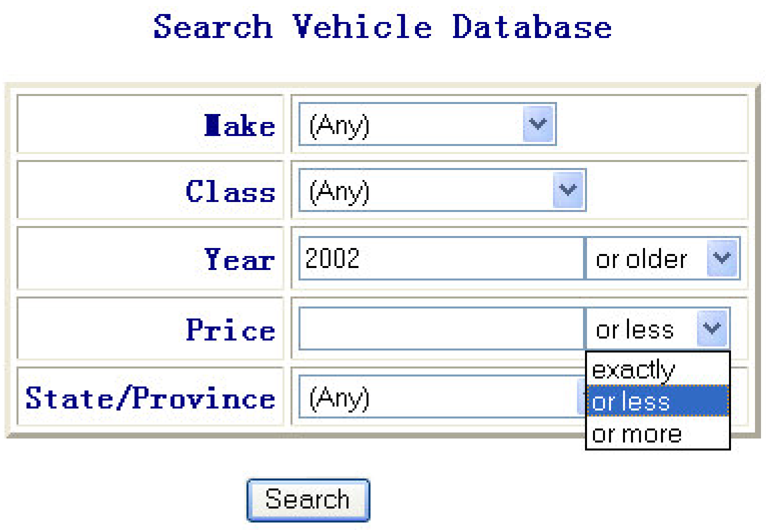
	Field Matching	**Query Submission**	**Overall**
Recall	91.3% (21/23)	100% ((31+85)/(31+85))	91.3%
Precision	100% (21/21)	100% ((31+85)/(31+85))	100%

Table 2: Experimental Results for Digital-Cameras Search.

4.2 Results Analysis and Discussion

Field-matching efficiency is a measurement for field-name matching. This matching is affected by the value for the name attribute and the type attribute the site form designer chose for each field. For fields in the site form with no values provided or fields having less than half their values recognized, our system depends only on the values of name attributes. If the site form designer assigns meaningful names to each tag, our field-matching efficiency is high. In our experiment, we found respectively 7 and 6 such fields from the two domains, and our system recognized 95.7% of the fields for the two domains. We found respectively 14 and 17 fields from the two domain with values provided, and the result was 100% for both precision and recall for the two domains tested. We have no way of recognizing fields that are not described in our extraction ontology, so we did not consider those fields when calculating name matching efficiency.

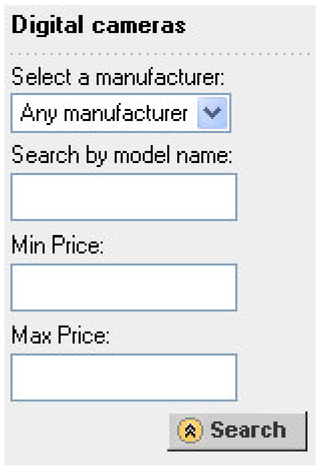
Query-submission efficiency is a measurement of field-value matching. When calculating query-submission efficiency, we consider only the fields where names matched correctly. This efficiency is greatly affected when fields are designed in a way that our system cannot handle. For the form in Figure 16, we found two fields that are for *Price*; together, they form a range. Range fields, if are formed by two independent fields, normally are of the same type, i.e., both fields are text fields (*Price* in Figure 17), or both fields are selection lists (*Price* in Figure 1). The range in Figure 16, however, is formed by a text field and a selection list. Our system does not recognize these two fields as a range. So, when it fills out the *Price* fields, it puts both the lower value and the upper value a user specifies in the first *Price* field. For the second *Price* field, which is a selection list, our system chooses all three values in the selection list. This generates six queries instead of one query — properly chosen, one query would be sufficient. For the particular form in Figure 16, our system always generate 6 queries if either *Price* or *Year* is specified by a user, among which 5 of the queries are not necessary. When a user specifies both fields, our system submits 36 queries, among which 35 of the queries are not necessary. This result greatly affects the precision of query submission.. The recall, however, is 100% because all queries that could have been submitted are submitted correctly.


Search Vehicle Database

Make	(Any)	
Class	(Any)	
Year	2002	or older
Price		or less
State/Province	(Any)	

Figure 16: Web form for Car-Ads Search at http://www.autointerface.com/vfs.htm*,*

February, 2004.



Digital cameras

Select a manufacturer:

Any manufacturer

Search by model name:

Min Price:

Max Price:

Figure 17: Web form for Digital-Camera Search at

http://www.netbuyer.co.uk/categories/englishcameracrawlzd.html, *February, 2004.*

Chapter 5

CONCLUSION

In this research, we designed and implemented a system that can fill out and submit Web forms automatically according to a given user query against a corresponding application extraction ontology. From the returned results, the system extracts information from the pages, puts the extracted records in a database, and queries the database with the original user query to get, to the extent possible, just the relevant data behind these Web forms.

We tested our system on two applications: car advertisements and digital camera advertisements. In average, there were 61.9% fields in the site forms that were applicable to the extraction ontologies. The system correctly matched 95.7% of them. Considering only the fields that were applicable to the extraction ontologies and were correctly matched, the system correctly sent out all queries that should have been submitted to the Web sites we tested. It, however, also sent out some additional queries that are not necessary according to the original user query. Among all queries our system submitted for our experiments, only 91.4% of them are necessary. Further, for the Web sites we tested, our Output Analyzer correctly gathered all linked pages. Finally, of the records correctly extracted by Ontos, our system always correctly returned just those records that satisfied the user-specified search criteria.

Even though our experimental results turned out to be working well, the results can be adversely affected if

- text fields come with poor internal tag names,

- extraction ontologies are poorly written.

Poor tag names can make the name-matching process for text fields impossible, and poorly written extraction ontologies would make the name-matching process for fields with values inaccurate. Both cases would decrease the field-matching efficiency dramatically. With considerably more work, which we did not do because others have already solved this problem [RGa00, Rga01, MGJ01], it would be possible to resolve the first problem by locating the displayed names, which should be human readable, rather than the internal tag names, which need not be human readable. The solution for the second problem is to improve the quality of a poorly written extraction ontology.

In addition, our system is not designed to handle all kinds of forms on the Web. It does not try to handle

- multiple forms (one form lead to another),
- forms whose actions are coded inside scripts.

As future work, we could program our system to handle multiple forms. The field-matching process for multiple forms is like separating the field-matching process for single forms into several parts. After matching and submitting, we repeat the process until we have exhausted all forms in the chain of forms. Query-submission and post-processing processes are the same for both multiple-forms and single-form cases. Note that our system must know the submission paths (a form's action attribute) in order to submit queries. It is possible to write script code that submits queries independent of explicit action attributes. So our system will not always be able to submit queries to such sites. Therefore, our experiements ignore those pathological cases.

Bibliography

[Deg04] Brigham Young University Data Extraction Group Homepage. http://www.deg.byu.edu.

[DEW96] Robert B. Doorenbos, Oren Etzioni, and Daniel S. Weld. "A Scalable Comparison-Shopping Agent for the World-Wide Web," In *Proceedings of the First International Conference on Autonomous Agents*, Marina del Rey, California, USA, February 5-8, 1997, pp. 39-48.

[ECJ+99] David W. Embley, Douglas M. Campbell, Yuan S. Jiang, Stephen W. Liddle, Yiu-Kai Ng, Dallan Quass, and Randy D. Smith. "Conceptual-Model-Based Data Extraction from Multiple-Record Web Pages," In *Data & Knowledge Engineering*, Volume 31, Number 3, November, 1999, pp. 227-251.

[ECS98] David W. Embley, Douglas M. Campbell, Randy D. Smith. "Ontology-Based Extraction and Structuring of Information from Data-Rich Unstructured Documents," In *Proceedings of Conference on Information and Knowledge Management (CIKM 1998)*, Bethesda, Maryland, November 3-7, 1998, pp. 52-59.

[EJN99] David W. Embley, Yuan S. Jiang, Yiu-Kai Ng. "Record-Boundary Discovery in Web Documents," In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania, May 31 – June 3, 1999, pp. 467-478.

[EJX01] David W. Embley, David Jackman, and Li Xu. "Multifaceted Exploitation of Metadata for Attribute Match Discovery in Information Integration," In *Proceedings of The Vienna Institute for Comparative Economic Studies (WIIW01)*, Rio de Janeiro, Brazil, April 9-11, 2001, pp. 110-117.

[Emb89] David W. Embley. "NFQL: the natural forms query language," In *Proceeding of the ACM Transactions on Database Systems*, Volume 14, Number 2, June, 1989, pp. 168-211.

[ETL02] David W. Embley, Cui Tao, and Stephen W. Liddle. "Automatically Extracting Ontologically Specified Data from HTML Tables with Unknown Structure," In *Proceeding of the 21st International Conference on Conceptual Modeling (ER 2002)*, Tampere, Finland, October 7-11, 2002, pp. 322-337.

[Jav04] Java Technology from Sun Microsystem. http://java.sun.com.

[LDE+02] Deryle Lonsdale, Yihong Ding, David W. Embley, and Alan Melby. "Peppering Knowledge Sources with SALT: Boosting Conceptual Content for Ontology Generation," In *Processing of the Eighteenth National Conference on Artificial Intelligence Workshop*, Edmonton, Alberta, Canada, July, 28, 2002, pp. 30-36.

[LES+02] Stephen W. Liddle, David W. Embley, D.T. Scott, and Sai Ho Yau. "Extracting Data Behind Web Forms," In *Proceedings of the Workshop on Conceptual Modeling Approaches for e-Business*, Tampere, Finland, October 7-11, 2002, pp. 402-413.

[LYE01] Stephen W. Liddle, Sai Ho Yau, and David W. Embley. "On the Automatic Extraction of Data from the Hidden Web," In *Proceedings of the International Workshop on Data Semantics in Web Information Systems (DASWIS-2001)*, Yokohama, Japan, November 27-30, 2001, pp. 212-226.

[MGJ01] Giovanni Modica, Avigdor Gal, and Hasan Jamil. "The Use of Machine-Generated Ontologies in Dynamic Information Seeking, " In *Proceedings of the 9th International Conference on Cooperative Information Systems (CoopIS 2001)*, Trento, Italy, September 5-7, 2001, pp. 433-448.

[Mic04] Microsoft Passport and Wallet Services. http://memberservices.passport.com.

[Mys04] MySQL. http://www.mysql.com.

[Qui93] J. Ross Quinlan. "*C4.5: Programs for Machine Learning*." San Mateo, CA: Morgan Kaufmann, 1993.

[RGa00] Sriram Raghavan and Hector Garcia-Molina. "Crawling the Hidden Web," Technical Report 2000-36, Department of Computer Science, Stanford University, Stanford, California, December 2000.

[RGa01] Sriram Raghavan and Hector Garcia-Molina. "Crawling the Hidden Web," In *Proceedings of the 27th Very Large Data Bases (VLDB) Conference*, Rome, Italy, September 11-14, 2001, pp. 129-138.

[Tom04] The Apache Jakarta Project. http://jakarta.apache.org/tomcat/index.html.

[Yau01] Sai Ho Yau. “ Automating the Extraction of Data Behind Web Forms,” Master’s Thesis, Department of Computer Science, Brigham Young University, Provo, Utah, 2001.

Appendix A

The Web Forms Used in Experiments

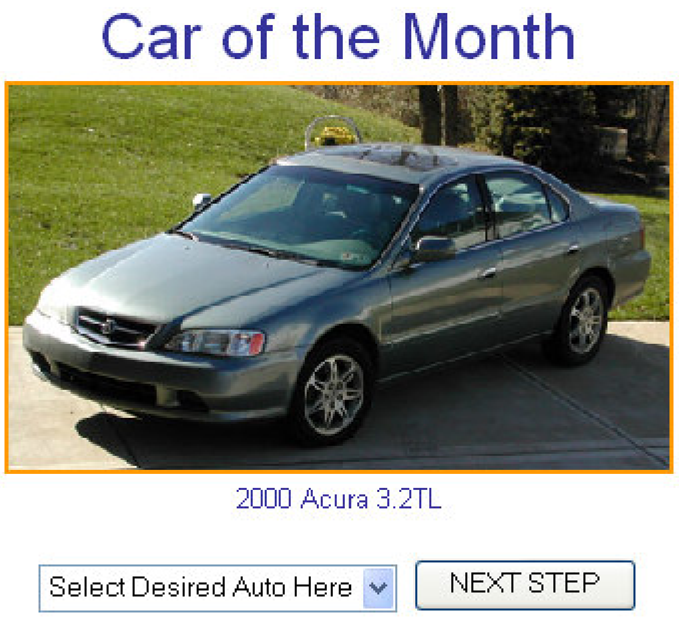


Figure A1: Web form for Car-Ads Search at http://2see.com/buy.asp*, February, 2004.*

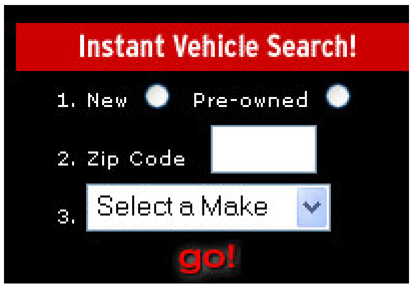
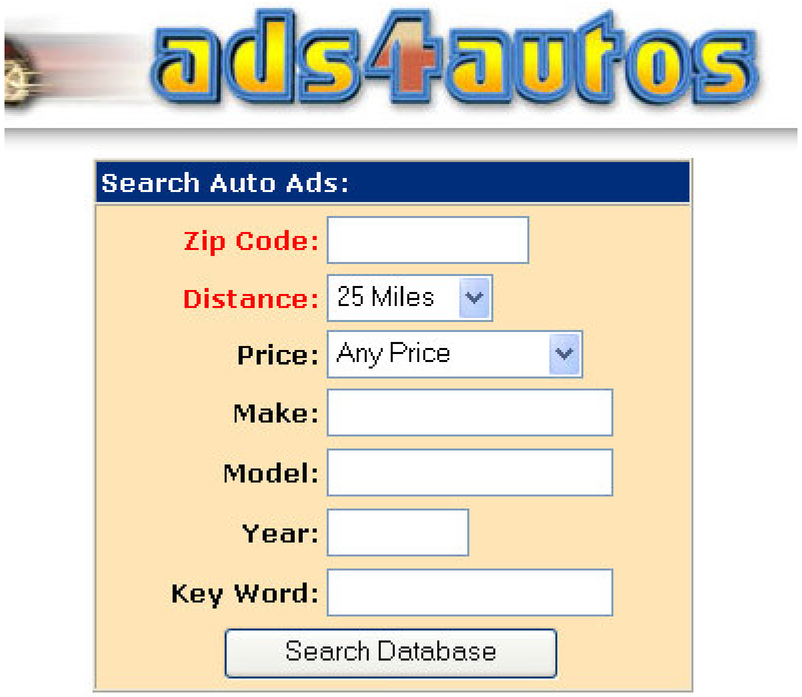


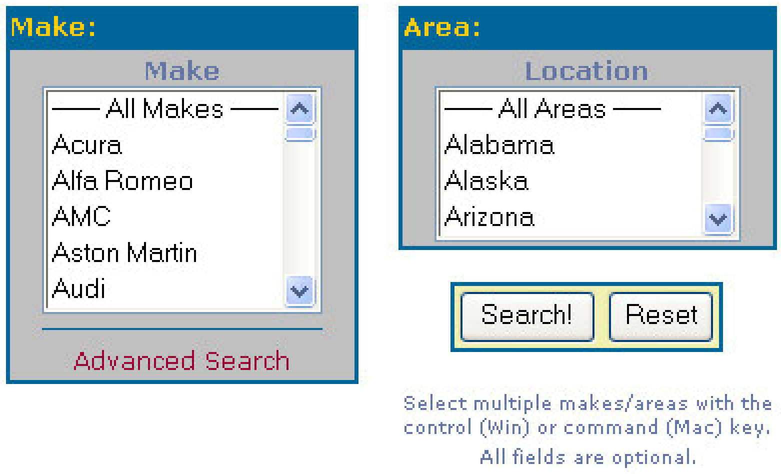
Figure A2: Web form for Car-Ads Search at http://dealernet.com/*, February, 2004.*



Search Auto Ads:

Zip Code:	
Distance:	25 Miles
Price:	Any Price
Make:	
Model:	
Year:	
Key Word:	

Figure A3: Web form for Car-Ads Search at http://www.ads4autos.com/autos/index.cfm, *February, 2004.*


Make:

Make

—— All Makes ——
Acura
Alfa Romeo
AMC
Aston Martin
Audi

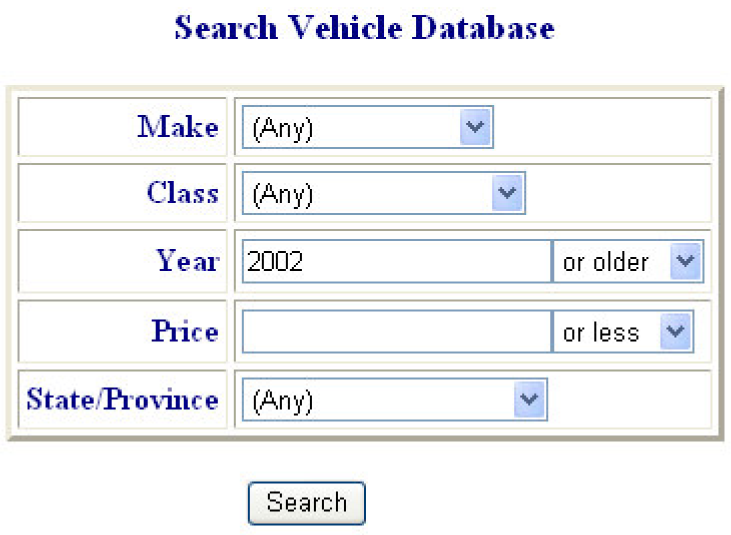
Area:

Location

—— All Areas ——
Alabama
Alaska
Arizona

Select multiple makes/areas with the control (Win) or command (Mac) key.
All fields are optional.

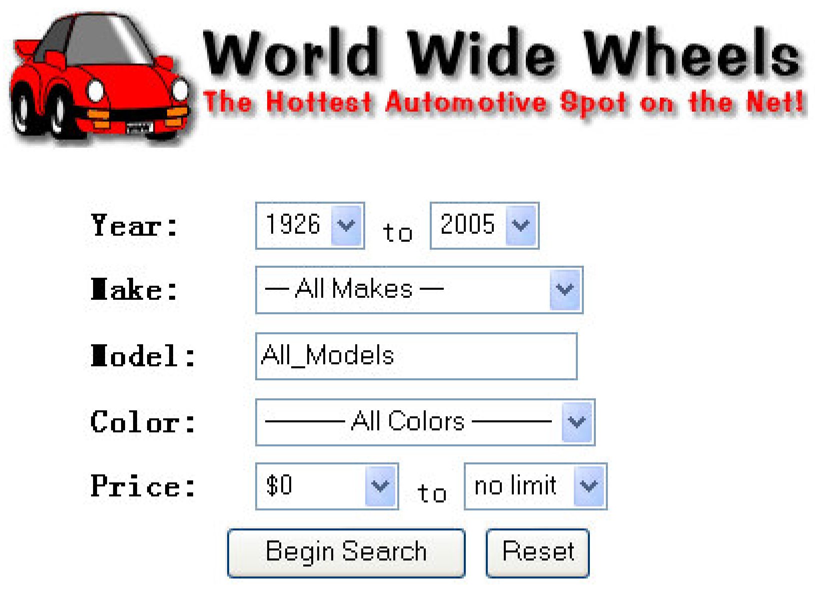
Figure A4: Web form for Car-Ads Search at http://www.carbuyer.com/, *February, 2004.*



Search Vehicle Database

Make	(Any)
Class	(Any)
Year	2002 or older
Price	or less
State/Province	(Any)

Figure A5: Web form for Car-Ads Search at http://www.autointerface.com/vfs.htm*,*
February, 2004.



Year:	1926 to 2005
Make:	— All Makes —
Model:	All_Models
Color:	——— All Colors ———
Price:	$0 to no limit

Figure A6: Web form for Car-Ads Search at http://wwwheels.com/cfapps/autosearch.cfm*,*
February, 2004.

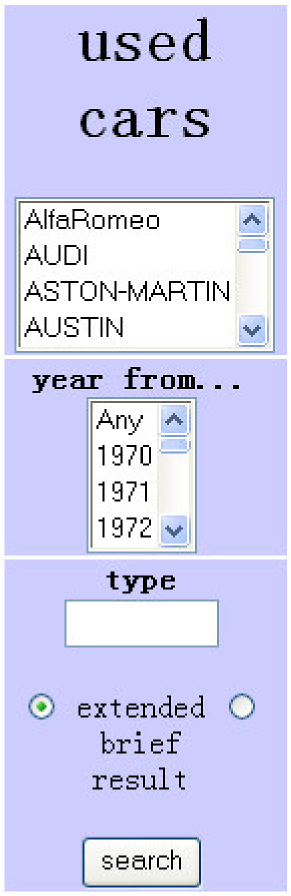


Figure A7: Web form for Car-Ads Search at http://www.belcyber.net/cars/, *February, 2004.*

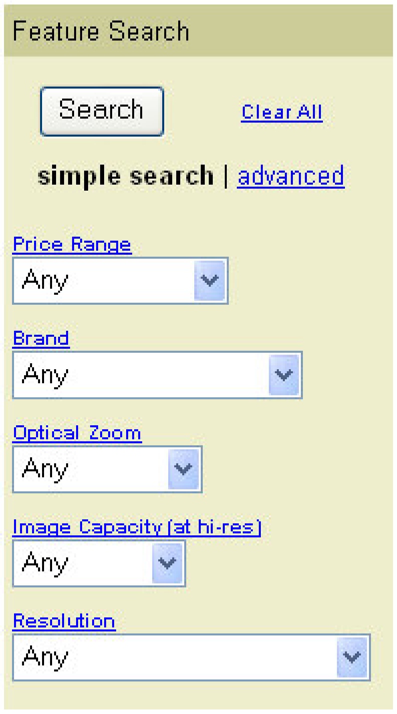


Figure A8: Web form for Digital-Camera Search at *http://www7.activebuyersguide.com/abg/nav/StartPageHandler.cfm?PID=12X7X97X557 36X5f6e0&option=search&CatID=2&RefID=12&check=0, February, 2004.*

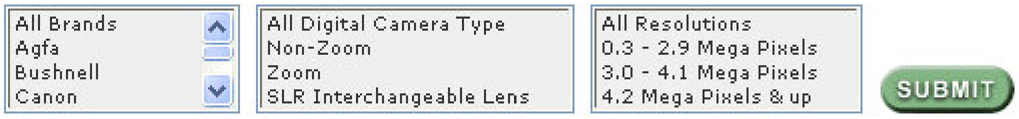
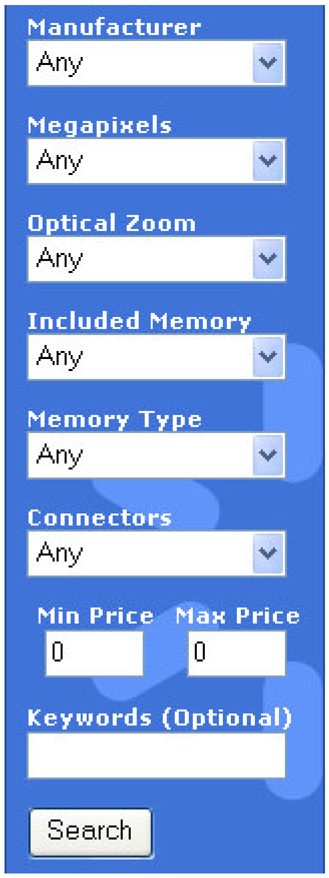


Figure A9: Web form for Digital-Camera Search at *http://www.bhphotovideo.com/bnh/controller/home;jsessionid=A9808MtKaz!121158313 9?O=NavBar&A=search&Q=&ci=1082&sb=bs%2Cupper%28ds%29&sq=asc&pn=1 &si=filter, February, 2004.*



Manufacturer: Any

Megapixels: Any

Optical Zoom: Any

Included Memory: Any

Memory Type: Any

Connectors: Any

Min Price: 0

Max Price: 0

Keywords (Optional):

Figure A10: Web form for Digital-Camera Search at http://dealcam.com/?ref=ov-digital_camera&sess=c6a1dd6e4adf00108692ae0f0d50b515, February, 2004.

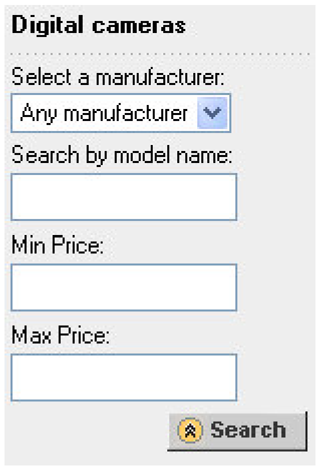


Figure A11: Web form for Digital-Camera Search at http://www.netbuyer.co.uk/categories/englishcameracrawlzd.html, February, 2004.

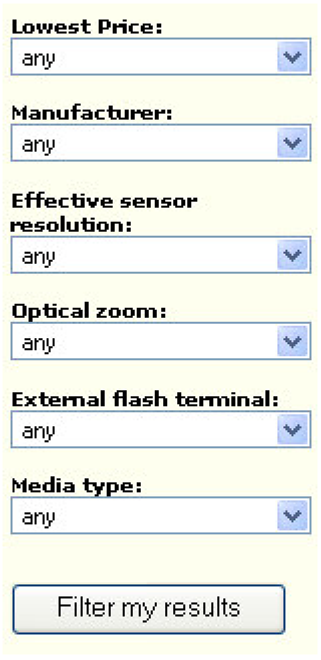
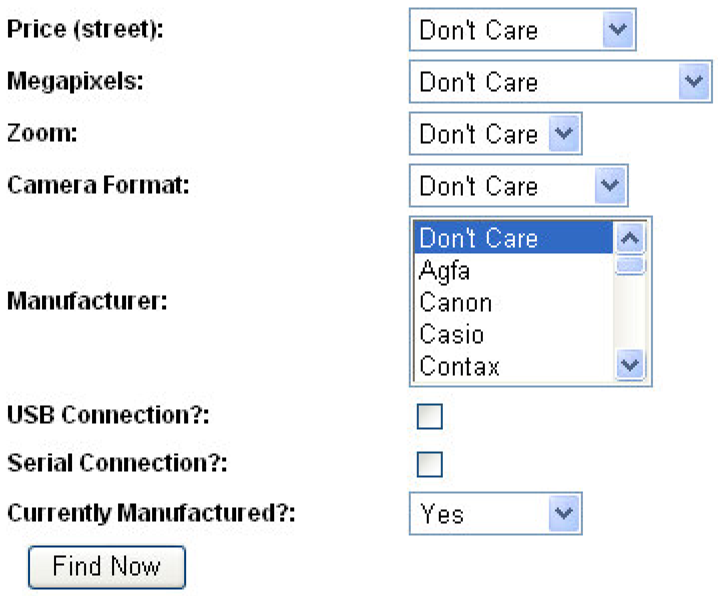
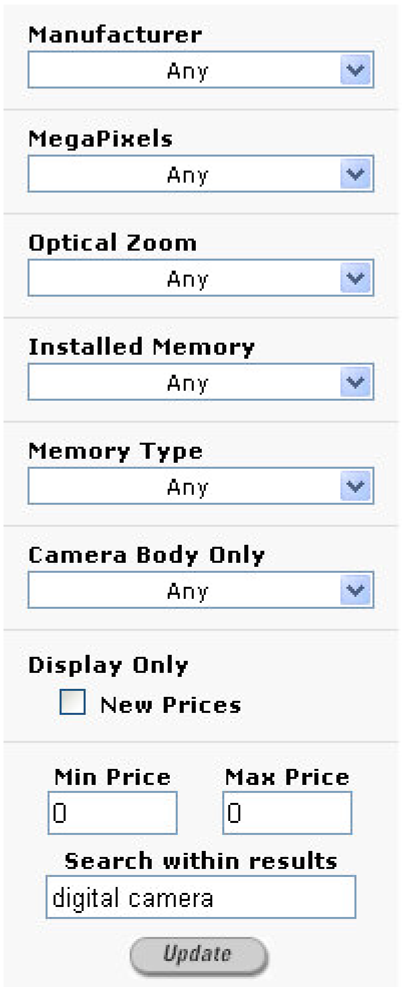


Figure A12: Web form for Digital-Camera Search at http://overture1-cnet.com.com/Digital_Cameras/4007-6501_9-0.html?&part=overture1-cnet&subj=digital_camera&tag=ref, February, 2004.



Price (street):	Don't Care
Megapixels:	Don't Care
Zoom:	Don't Care
Camera Format:	Don't Care
Manufacturer:	Don't Care Agfa Canon Casio Contax
USB Connection?:	☐
Serial Connection?:	☐
Currently Manufactured?:	Yes

Figure A13: Web form for Digital-Camera Search at http://www.imaging-resource.com/CAMDB/camera_finder.php, February, 2004.



Manufacturer
Any

MegaPixels
Any

Optical Zoom
Any

Installed Memory
Any

Memory Type
Any

Camera Body Only
Any

Display Only
☐ New Prices

Min Price	Max Price
0	0

Search within results
digital camera

Figure A14: Web form for Digital-Camera Search at *http://www.pricegrabber.com/search_attrib.php/page_id=48/topcat_search=1/form_keyword=digital+camera/mode=gotoph/ut=97c8eb606d9cf20c, February, 2004.*

Appendix B

The Queries Used in Experiments

B1. Car advertisement queries:

1. zip=20171, search range=100 miles, make=ford
2. zip=20171, search range=100 miles, make=ford, price<=10000
3. zip=20171, search range=100 miles, make=ford, model=mustang, year>=1995
4. zip=20171, search range=100 miles, price>=4000, price<=10000
5. zip=20171, search range=100 miles, make=ford, price<=10000, year>=1995, color=green

B2. Digital cameras queries:

1. zip=20171, search range=100 miles, Manufacturer=canon
2. zip=20171, search range=100 miles, Manufacturer=canon, CCD resolution>=3
3. zip=20171, search range=100 miles, price<=800
4. zip=20171, search range=100 miles, Manufacturer=canon, CCD resolution=3, optical zoom>=3.0