Understanding Query Interfaces by Statistical Parsing

WEIFENG SU, BNU-HKBU United International College and Shenzhen Key Laboratory of Intelligent Media and Speech, PKU-HKUST Shenzhen Hong Kong Institution
HEJUN WU, Sun Yat-Sen University
YAFEI LI and JING ZHAO, BNU-HKBU United International College
FREDERICK H. LOCHOVSKY, The Hong Kong University of Science and Technology
HONGMIN CAI, South China University of Technology
TIANQIANG HUANG, Fujian Normal University

Users submit queries to an online database via its query interface. Query interface parsing, which is important for many applications, understands the query capabilities of a query interface. Since most query interfaces are organized hierarchically, we present a novel query interface parsing method, StatParser (Statistical Parser), to automatically extract the hierarchical query capabilities of query interfaces. StatParser automatically learns from a set of parsed query interfaces and parses new query interfaces. StatParser starts from a small grammar and enhances the grammar with a set of probabilities learned from parsed query interfaces under the maximum-entropy principle. Given a new query interface, the probability-enhanced grammar identifies the parse tree with the largest global probability to be the query capabilities of the query interface. Experimental results show that StatParser very accurately extracts the query capabilities and can effectively overcome the problems of existing query interface parsers.

Categories and Subject Descriptors: H.3.5 [Online Information Services]: Web-based services—Query Interface; H.5.2 [User Interfaces]: Natural language
General Terms: Algorithms, Performance, Experimentation
Additional Key Words and Phrases: Query interface, maximum entropy

ACM Reference Format:
DOI: http://dx.doi.org/10.1145/2460383.2460387

1. INTRODUCTION

Surveys show that 95% of Web data is located in the deep Web, which is composed of millions of Web databases [Bergman 2001; Chang et al. 2004; Madhavan et al. 2007]. A typical Web database comprises a back-end database and a query interface. The user obtains the data from a Web database by submitting a query via its query interface. Upon receiving the user query, the database retrieves the relevant data and returns it to the user embedded in HTML pages. Hence, the query interface serves
as an intermediary between the user and the Web database. To submit a query, the user first has to understand the query capabilities of the query interface including: (1) the semantics of each element in the form that comprises the query interface, (2) the metadata of each element (such as the data type), and (3) the element organization. Thereafter, the user submits his/her query by filling values into corresponding elements in the query interface.

Many applications require the understanding of query interfaces. These applications include query interface schema matching [Su et al. 2006b], Web database classification/clustering [Barbosa and Freire 2007b; Su et al. 2006a], query interface merging [Dragut et al. 2006], on-the-fly query translation [He et al. 2005a], and record crawling from Web databases [Sheng et al. 2012; Vieira et al. 2008; Barbosa and Freire 2007a; Wu et al. 2006]. In particular, many applications require a hierarchical representation of the query interface. Considering that there are many Web databases available nowadays, and that each domain has a large number of Web databases, understanding the query interface manually is an almost impossible task in many applications that interact with many Web databases. Therefore, automatic query interface understanding is an indispensable component in these applications.

Automatic query interface understanding is challenging because different people create the query interfaces of different Web databases autonomously for different purposes and at different times. While HTML is used for specifying the presentation layout of most web pages, it does not consider the semantics and content in the page. Moreover, it is highly likely that query interfaces with similar functionalities are designed with many differences and in different ways. For example, from Figure 1, which shows two ticket-booking query interfaces from two online airline websites, the following observations can be made.

(1) Different labels are used to describe the same concept. For example, to represent an itinerary with multiple destinations, Figure 1(a) uses the label “Multiple destinations” while Figure 1(b) uses the label “Multi-cities”.

(2) Different kinds of elements are used for the same concept. For example, to denote the number of passengers, Figure 1(a) uses three selection lists while Figure 1(b) uses a text input box.
(3) A different number of elements are used for the same concept as illustrated by the previous example.

(4) Different concepts are presented. Figure 1(a) has the concept “Do you have any preferences?” which is not present in Figure 1(b).

(5) Different concept hierarchies are built as a consequence of the preceding differences.

One can imagine that the presentation of the query interfaces from different domains differ even more. Moreover, some visually similar query interfaces can be very different from each other at the HTML code level.

Existing query interface understanding research can be classified into two categories: rule-based methods and learning-based methods [Dragut et al. 2012; Khare et al. 2010]. A rule-based method usually first creates a set of rules, including textual layout or visual features. In general, the rules are created manually after investigating a large number of query interfaces from different domains. Thereafter, the rules are used to parse a given query interface. The rule-based methods parse a query interface into a parse tree, which is required by many applications. They also usually require human experts to investigate many query interfaces to determine a set of parsing rules, which are often quite large in number. What is more, the rules learned from the query interfaces in one domain usually do not have good performance when applied to another domain.

In contrast, a learning-based method trains a model based on a set of labeled query interfaces and applies the trained model to a new query interface. Learning-based methods effectively capture a domain’s features by learning from labeled query interfaces of the domain. However, to our knowledge, while almost all learning-based methods developed so far are able to divide a query interface into semantically relevant segments, they are incapable of parsing it into a hierarchical parse tree.

In this article, we propose a novel query interface understanding method, Statistical Parser (StatParser), which effectively combines the strengths of rule-based methods and learning-based methods, to automatically parse the query interface into a hierarchical representation. Dragut et al. [2009] provide concrete examples to show that a hierarchical representation is beneficial or very important for many applications, including query interface matching, unified query interface construction, and deep web crawling. Moreover, StatParser does not suffer the shortcomings that exist in rule-based methods or learning-based methods. In StatParser, only a very simple grammar is required initially. Thereafter, given a set of training query interfaces, the grammar is augmented with features and probabilities under the framework of the maximum-entropy learning model. Given a new query interface, the augmented grammar is used to identify the parse tree with the global largest probability as the understanding of the query interface.

The contributions of our approach include the following.

(1) We propose a method that combines the learning-based methods and the rule-based methods to effectively parse a query interface into a hierarchical representation with better performance than either of the state-of-the-art learning-based or rule-based methods.

(2) We show that only a small and intuitive set of context information is necessary for query interface parsing and understanding while the existing similar work requires a large set of context information during understanding.

We demonstrate that only a simple grammar is required during parsing, while most existing rule-based methods require a complex parsing grammar. The rest of the article is organized as follows. Section 2 defines the query interface understanding problem...
and presents an overview of StatParser. Section 3 presents the grammars required for parsing. Section 4 describes the context information used for parsing. Section 5 shows how the maximum-entropy parser learns from a set of parsed query interfaces and parses a new interface. Section 6 reports the experimental results. Section 7 reviews relevant work. Finally, Section 8 concludes.

2. STATPARSER OVERVIEW

In this section, the query interface understanding problem is first formally defined in Section 2.1, and then an overview of our solution to this problem, the StatParser method, is presented in Section 2.2.

2.1. Problem Definition

A query interface includes a form which is composed of a set of elements. The elements include text edit boxes, selection lists, radio buttons, and checkboxes.

1. A text edit box allows the user to input text, with or without a default value. In Figure 1(b), the element just below “Leaving From” is a text edit box.

2. A selection list provides a set of choices from which the user makes a selection. There are two types of selection lists: single selection list and multiple selection list. A single selection list allows the user to select only one choice while a multiple selection list allows the user to select multiple choices. In Figure 1(a), the element just below the label “Airline” is a single selection list.

3. Radio buttons allow the user to choose only one from a set of predefined options. A set of radio buttons with the same name value forms a radio button group. A radio button group can be seen as another kind of single selection list. In Figure 1(b), the three radio buttons under the label “RESERVE A CHARTER” form a group and they share the same name value.

4. Checkboxes allow the user to choose zero or more values from a set of predefined options. A set of checkboxes with the same name value forms a checkbox group. A checkbox group can be seen as another kind of multiple selection list.

Form designers usually use radio buttons and checkboxes to present the choices explicitly to the users.

In a query interface, the form is usually embedded with labels, which describe the purpose or functionality of the elements in the form. Users usually understand the form according to the labels within the form. For example, in Figure 1(b), the label “Leaving From” is used to describe the departure city which is input in the text edit box just below the label “Leaving From”. Moreover, the elements in a form and the labels that occur within the form usually are organized hierarchically. For example, Figure 2 shows the concept hierarchy for the query interface in Figure 1(a). An italic string in a leaf node corresponds to an element of the query interface for which a value can be specified. The non-root internal nodes are labels in the query interface describing the corresponding element or semantic unit.

To understand a query interface completely, a user has to understand not only a label-element’s description relationship, but also the label-element’s hierarchical combination relationship, that is, the semantic tree.

Definition 1 (Query Interface Understanding). Let $T = \{e_1, e_2, \ldots, e_m\}$ be a form in a query interface in which $e_i$ is an element in the form and $L = \{l_1, l_2, \ldots, l_n\}$ is a set of labels embedded in $T$. Query interface understanding determines the semantic organization between elements and labels using a tree representation, that is, a semantic tree. Each leaf node in the tree represents an element or a label and each internal node
shows the description relationship between a label and an element, or between a label and a semantic unit.

### 2.2. StatParser Workflow

In StatParser, query interface understanding is cast in a statistical parsing framework. Starting from a small grammar, $G$, StatParser uses the maximum-entropy model to augment $G$ with a probabilistic distribution to $G^+$ by learning from a set of parsed query interfaces. Given a new query interface, $I$, the probability augmented grammar, $G^+$, is used to parse $I$ by identifying a parse tree with the maximum probabilistic score. Thereafter, the parse tree is restructured into a semantic tree by removing the redundant nodes in the parse tree.

Figure 3 shows the workflow of StatParser, which consists of two stages: the training stage and the parsing stage. In the training stage, the important features are identified and their corresponding parameters are estimated with a set of parsed query interfaces. In the parsing stage, the features and parameters are used to understand a new query interface. Both stages have two steps: the preprocessing step and the understanding step.

The preprocessing step tries to process the radio buttons and checkboxes by combining each group of radio buttons or checkboxes that share the same name value and their corresponding labels into a single element. The training stage has two components in the preprocessing step: the Feature Selection component and the Parameter Estimation component. Given a set of parsed query interfaces, the Feature Selection component selects a set $E_p$ of features that are supposed to be important for radio-button/checkbox processing. Thereafter, the Parameter Estimation component learns the probability parameters for each of the selected features $E_p$. We handle radio buttons and checkboxes first for two reasons. First, the pattern to combine radio buttons/checkboxes and labels is different from other elements and labels. Second, the radio buttons/checkboxes that have the same name value can be safely merged to simplify the query interface to facilitate further processing.

Correspondingly, the parsing stage also has two components in the preprocessing step: Feature Extraction and Radio-button and Checkbox Processing. Given a new query interface, $I$, the Feature Extraction component identifies all features of $E_p$ that exist in $I$ for each group of radio-buttons/checkboxes and the labels around it. The Radio-button
and Checkbox Processing component combines radio buttons/checkboxes and labels according to the identified features and learned parameters under the maximum-entropy principle. After preprocessing, a preprocessed query interface $I_p$ is generated in which each group of radio-buttons/checkboxes and all of its combined labels are treated as only one element in the understanding step. Hence, we use one leaf node to represent a group of radio-buttons/checkboxes in the semantic tree. In Figure 2, the leaf node flight type is used to represent the radio-button group flight, which includes three radio buttons and the tree labels Roundtrip, One way, and Multiple destinations.

The understanding step generates the parse tree for a preprocessed query interface $I_p$. The training stage has two components in the understanding step: Feature Selection and Parameter Estimation with similar functionalities as those in the preprocessing step, but using different grammars for different purposes. Given the parsed query interfaces, the Feature Selection component selects a set $F$ of features that are supposed to be important for parsing. The Parameter Estimation component learns the probability parameters for the selected features.

The parsing stage has three components in the understanding step. Given a preprocessed query interface, $I_p$, the Feature Extracting component identifies all features in $I$ for each element/label, including label, text edit box, selection list, and simplified radio-button/checkbox group. The Hierarchical Parsing component parses the preprocessed query interface into a parse tree having the maximum probabilistic score according to the identified features and learned parameters. Finally, the Semantic Tree Restructuring component converts the parse tree into a semantic tree.

Fig. 3. StatParser workflow.
3. GRAMMAR
A reasonable grammar is very important for parsing. Section 3.1 first describes some observations over a number of real-life query interfaces. Thereafter, two grammars based on the observations, one for preprocessing and the other for understanding, are presented in Section 3.2. Finally, an algorithm to convert a parse tree into a semantic tree is given in Section 3.3.

3.1. Observations
Although query interfaces are designed autonomously, they are usually designed so that users can very easily understand them. Hence, there are some inherent patterns that almost all designers follow when designing query interfaces. These patterns may not be explicit to the designers, but they are used implicitly to construct query interfaces concisely and to make them easily understood by the users. We validated the nine patterns listed in Dragut et al. [2009] against more than 300 query interfaces from 10 domains both in Chinese and English and found that the first four patterns listed shortly are true in all query interfaces. In addition, we identified two new patterns (Patterns 5 and 6). The histogram in Figure 4 shows how confident each pattern is in our survey.

Pattern 1. Query interfaces are presented in a top-down and left-to-right order.
Pattern 2. Elements and labels in a query interface are organized into semantic groups.
Pattern 3. Each label describes either an element or a semantic unit, but not both.
Pattern 4. The label for an element or a semantic unit is located either above, left, right, or below the element or the semantic unit.
Pattern 5. Each element/semantic unit is described by no more than one label in the semantic tree.
Pattern 6. Each radio button/checkbox is combined with no more than one label.

Patterns 1 and 2 are obvious. Pattern 1 indicates that the query interfaces are usually presented in a way similar to other Web pages: top-down and left-to-right. Pattern 2 corresponds to the fact that it would be easier for the user to understand the query interface if its elements and labels are structured into semantic units. Pattern 3 indicates that each label only plays one role because a multirole label may lead to confusion for the user. Pattern 4 says that a label usually describes an element or a semantic unit that is positioned close to it, as it would be rather peculiar for a label to denote an element that is separated by some other elements or labels.

Since query interfaces are organized semantically hierarchically, if there is an element that is described by two labels, then one of the two labels is first selected to
3.2. Grammar

During query interface parsing, the parsing tree shows how a query interface is generated from the root. A grammar is very important for query interface parsing since it determines what parsing trees are generated. Given a query interface, a good grammar should: (1) be able to correctly generate the parse tree for it and (2) generate as few parse trees as possible for it. In this section, an independent grammar is proposed for each of the preprocessing and understanding steps.

3.2.1. Grammar for Preprocessing. Based on Patterns 1, 3, 4, and 6 in Section 3.1, the grammar in Figure 5 is proposed for radio-button and checkbox processing. RC represents a group of radio buttons/checkboxes, which share the same name value, and their corresponding labels. Each LabelRadio represents a radio button and its corresponding label. Similarly, a LabelCheck represents a checkbox and its corresponding label. The production rule LabelRadio → label radiobutton means a LabelRadio can be decomposed into a label and a radio button and the label is located above/left of the radio button. Similarly, the production rule LabelRadio → radiobutton label denotes a LabelRadio can be decomposed into a label and a radio button and the label is located below/right of the radio button.

In the parsing stage, the Radiobutton and Checkbox Processing component is invoked for each group of radio buttons/checkboxes in the new query interface. The input of the processing is the group of radio buttons/checkboxes and the labels immediately around them. The output of the processing is a three-layer tree rooted in RC. Each child of RC represents a LabelRadio/LabelCheck with child nodes, that is, leaf nodes, which are a radio button/checkbox and a label. In the experiments, we found that more than 99% of radio buttons/checkboxes were correctly combined with their corresponding label.

3.2.2. Grammar for Understanding. Based on Patterns 1 to 5 in Section 3.1, the grammar, $G$, in Figure 6 is proposed. In $G$, there are four nonterminal symbols and 11 production rules. Each Element represents a text edit box, selection list, or a simplified group of radio buttons/checkboxes. Each Condition represents a semantic unit. Each Conditions represents one or several semantic units.

The production rule Condition → label Condition means that a Condition can be decomposed into a label and a Condition and the label is located above/left of the Condition. The production rule Condition → Condition label means that a Condition
can be decomposed into a label and a Condition and the label is located below/right of the Condition.

Compared with the parsing grammar in Zhang et al. [2004], the grammars in Figures 5 and 6 are much simpler in the following ways.

(1) The number of rules is much fewer than that of the grammar in Zhang et al. [2004], which contains more than 80 rules.

(2) The right-hand side of each production rule contains no more than two variables, which means that each internal node in the parse tree contains no more than two children.

A simple grammar has the following advantages compared with a complex grammar.

(1) The number of training instances required to learn a simple grammar is usually smaller than that of a complex grammar because a smaller number of parameters need to be assigned in a simple grammar. A complex grammar usually contains infrequently used rules for which a large number of training instances are required to get reasonable parameters.

(2) The number of possible parse trees for a given query interface generated by a complex grammar is usually much larger than that generated by a simple grammar. Hence, if the simple grammar still can generate the correct parse tree, it is easier to identify the correct parse tree.

3.3. Parse Tree versus Semantic Hierarchical Tree

Given a query interface, the hierarchical parsing component generates a parse tree for it. In the parse tree, each internal node corresponds to a nonterminal and each leaf node corresponds to an element or a label. Each internal node represents a derivation.

A parse tree is different from a semantic tree. Figure 7 shows a fragment of the parse tree for Figure 1(a). Only the fragment that corresponds to “Do you have any preferences?” and its descendants are fully presented. Comparing Figure 7 with the corresponding subtree in Figure 2, we see that the semantic tree is actually the abstract syntax tree of the parse tree and can be generated from the parse tree by removing the nonterminal symbols from it.

Algorithm 1 is used to generate the semantic tree from a parse tree. For each nonterminal in Figure 6, there is a corresponding function. The basic idea of the algorithm is
ALGORITHM 1: Semantic tree construction algorithm

**Input:** A parse tree $C$ rooted at $r$

**Output:** A semantic tree $S$

```java

SemNode Root(ParseNode $r$){
    SemNode $n$ = new SemNode();
    $n$.parent = nil;
    if $r$ has two children $c_1$ and $c_2$ /* Root $\rightarrow$ Conditions Condition */
        Conditions($c_1$, $n$);
        Condition($c_2$, $n$);
    else if $r$ has one child $c_1$ /* Root $\rightarrow$ Condition */
        Condition($c_1$, $n$);
    return $n$;
}

void Conditions(ParseNode $r_1$, SemNode $n$){
    if $r_1$ has two children $c_1$ and $c_2$ /* Conditions $\rightarrow$ Conditions Condition */
        Conditions($c_1$, $n$);
        Condition($c_2$, $n$);
    else if $r_1$ has one child $c_1$ /* Conditions $\rightarrow$ Condition */
        Condition($c_1$, $n$);
}

void Condition(ParseNode $r_1$, SemNode $n$){
    if $r_1$ has two children $c_1$ and $c_2$ and $c_1$ is a label /* Condition $\rightarrow$ label Condition */
        SemNode $n_1$ = new node($c_1$);
        AddChild($n$, $n_1$); //Add $n_1$ as a child of $n$
        Condition($c_2$, $n$);
    else if $r_1$ has two children $c_1$ and $c_2$ and $c_2$ is a label /* Condition $\rightarrow$ Condition label */
        Condition($c_1$, $n$);
        SemNode $n_2$ = new node($c_2$);
        AddChild($n$, $n_2$); //Add $n_2$ as a child of $n$
    else if $r_1$ has two children $c_1$ and $c_2$ /* Condition $\rightarrow$ Condition Condition */
        Condition($c_1$, $n$);
        Condition($c_2$, $n$);
    else if $r_1$ has one child $c_1$ /* Condition $\rightarrow$ Element */
        Element($c_1$, $n$)
}

void Element(ParseNode $r_1$, SemNode $n$){
    if $r_1$ has one child $c_1$ /* Element $\rightarrow$ editbox | selectionlist | RC */
        SemNode $n_1$ = new node($c_1$);
        AddChild($n$, $n_1$);
}
```
to perform a depth-first search on the parse tree and to call the corresponding function according to the node’s type in the parse tree. For each production rule in Figure 6, there are lines of code corresponding to it. In each function, there are lines of code corresponding to semantic node generation.

4. CONTEXT INFORMATION EXTRACTION

Given a form for a query interface, StatParser extracts a set of context information for each element and label. According to our experiments, the context information is vital to the success of StatParser. Considering that forms are usually designed to be quickly understood by users, the relevant labels for an element are usually located around the element. To facilitate presentation, we use the term item to represent either an element or a label.

From observing several hundred query interfaces from various domains and various countries, we found that an item usually is most correlated with the items immediately around it (Pattern 4). Hence, we extend the “field scope” concept in Dragut et al. [2009], which states that a label for an element must lie in one of four positions: above the element, below the element, left to the element, or right to the element. As shown in Figure 8, for each item $e_i$, we consider the items in the following four positions around $e_i$ as the features for $e_i$.

(1) *The item just above* $e_i$. If there are multiple items just above $e_i$, only the one that is left aligned to $e_i$ is selected.

(2) *The item just below* $e_i$. If there are multiple items just below $e_i$, only the one that is left aligned to $e_i$ is selected.

(3) *The item just to the left to* $e_i$. If there are multiple items to the left to $e_i$, only the one closest to $e_i$ is selected.

(4) *The item just to the right to* $e_i$. If there are multiple items to the right to $e_i$, only the one closest to $e_i$ is selected.

In a few of the examined query interfaces, we noticed that some examples or comments followed a label. In StatParser, the labels will be ignored if they start with “for example”, “e.g.”, and so on. It is considered a parsing error if a label of this kind is not ignored and is assigned to an element.

Each item generates two pieces of context information: the item type information and the distance information. The item type information represents the type of the item, whose value can be an element of \{*label*, *editbox*, *selectionlist*, *radiobuttons*, *checkboxes*, *null*\}. The *radiobuttons* or *checkboxes* represent a group of radio buttons or a group of checkboxes that has been constructed in the preprocessing step. The item type value is *null* if there is no item in the corresponding position.

The distance information of an item represents the distance between the item and $e_i$. Its value can be an element of \{*far*, *near*, *null*\}. In the experiments, the distance is
Table I. Context Information for Items in Figure 1(a)

<table>
<thead>
<tr>
<th>Item id</th>
<th>Above item</th>
<th>Below item</th>
<th>Left of item</th>
<th>Right of item</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊞</td>
<td>null</td>
<td>radiobuttons</td>
<td>near</td>
<td>null</td>
</tr>
<tr>
<td>⊙</td>
<td>label</td>
<td>far</td>
<td>label</td>
<td>null</td>
</tr>
<tr>
<td>⊞</td>
<td>radiobuttons</td>
<td>far</td>
<td>label</td>
<td>near</td>
</tr>
</tbody>
</table>

far if there is at least one empty table cell or a line between the item and $e_i$ and near otherwise. If the item type is null, its distance value is also null.

Consider the first three items in Figure 1(a), that is, ⊞ the label “What type of flight do you need?”, ⊙ the flight type radio buttons and ⊞ the label “Where and when do you want to travel?” Table I shows the context information of these three items. In the table, the above item type of item ⊞ is null because there is no item above it. The below item type of item ⊞ is item ⊙, which is a radio button and is near item ⊞. Both the left items and right items of item ⊞ are null because there are no items to the left or right of item ⊞.

Besides being used in context information extraction, the type of $e_i$ itself is also used for the parsing task as discussed in Section 5.1. Therefore, altogether nine pieces of information are collected for each item (i.e., the item’s type as well as the type and distance of any items in the four positions around the item as discussed before).

5. MAXIMUM-ENTROPY-BASED PARSING

The intuition of the Maximum-Entropy (MaxEnt) model is simple: we choose to model all that is known and do not assume anything we do not know. That is, we find a model that is as uniform as possible while at the same time satisfying all the required constraints. The MaxEnt model has been used to successfully solve problems in natural language processing, such as part-of-speech tagging [Ratnaparkhi 1996], name entity recognition [Borthwick 1999], text categorization [Zhang and Oles 2001], natural language parsing [Charniak 2000], and data extraction [Su et al. 2009].

5.1. MaxEnt Principle

MaxEnt is a modeling technique for approximating an observed distribution. We represent a single observation with $y$, whose value comes from $Y$, and conditional information with $x$, whose value comes from $X$. We assume that the value of $y$ is influenced by the value of $x$. Therefore, the conditional probability $p(y|x)$ is estimated.

In StatParser, the conditional information $x$ refers to context information that is described in Section 4 and $y$ refers to any production in Figure 5 or Figure 6, depending on the processing step. For example, if we observe that in a query interface and its corresponding parse tree a condition is combined with a label on its left, which corresponds to the production rule $\text{Condition} \rightarrow \text{label} \ \text{Condition}$ in Figure 6, then we assign $x$ to be “left item is a label” and $y$ to be $\text{Condition} \rightarrow \text{label} \ \text{Condition}$. As another example, if we observe that an input text box is combined with a near-right label, which corresponds to the production rule $\text{Condition} \rightarrow \text{Condition label}$ in Figure 6, then we assign $x$ to be “near right item is a label” and $y$ to be $\text{Condition} \rightarrow \text{Condition label}$. We observe all the parse trees and training query interfaces and collect all training pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.

Suppose that a particular pair $(x, y)$ occurs $c_e(x, y)$ times in the training samples. Thereafter, the empirical probability is summarized for pair $(x, y)$ in the training
samples as
\[ p_e(x, y) = \frac{c_e(x, y)}{\sum_x c_e(x, y)}. \] (1)

For a pair \((x, y)\), which is assumed to be important to model the process, we would like to create a feature corresponding to it as
\[ f_i(x, y) = \begin{cases} 1, & \text{if the feature is expressed in case } (x, y) \\ 0, & \text{otherwise} \end{cases} \] (2)

Hence, the expected value of feature \(f_i\) for the empirical distribution \(p_e(x, y)\) is
\[ p_e(f_i) = \sum_{x, y} p_e(x, y) f_i(x, y), \] (3)
while the expected value of feature \(f_i\) for the model \(p(y|x)\) is
\[ p(f_i) = \sum_{x, y} p_e(x) p(y|x) f_i(x, y), \] (4)
in which \(p_e(x)\) is the empirical distribution of \(x\) in the training samples.

The maximum-entropy model tries to maximize the entropy \(H(p)\) in which \(p \equiv p(y|x)\) under the constraints \(p_e(f_i) \equiv p(f_i)\) for all features \(i = 1, \ldots, n\). To maximize \(H(p)\), we generate a parameter \(\lambda_i\) (the Lagrange multiplier) for each feature \(f_i\) using an iterative algorithm, such as generalized iterative scaling or conjugate gradient ascent [Minka 2003].

5.2. Features
The context and distance information in a query interface provide useful hints which are used as features for query interface understanding. Each piece of context information, or the combination of several pieces of context information listed in Section 4, can be encoded as a feature for the maximum-entropy model. For example, a useful feature is
\[ f(x, y) = \begin{cases} 1, & \text{if } x = \text{“left item is a label” and } y = \text{“Condition } \rightarrow \text{ label Condition”} \\ 0, & \text{otherwise} \end{cases} \]

This feature reveals the fact that the left label usually describes a condition with a left label. We ignore whether the left item is far or near to the element. Hence, the parameter corresponding to this feature increases the probability \(p(\text{left})\) of an edit box with a left label.

Another feature is
\[ f(x, y) = \begin{cases} 1, & \text{if } x = \text{“near left item is a label” and } y = \text{“Condition } \rightarrow \text{ label Condition”} \\ 0, & \text{otherwise} \end{cases} \]

This feature is similar to the previous feature except that it requires the left item label to be close to the element.

The \(x\) in a feature may be a combination of several items. Consider the following feature:
\[ f(x, y) = \begin{cases} 1, & \text{if } x = \text{“left item is a label and right item is a label” and } y = \text{“Condition } \rightarrow \text{ label Condition”} \\ 0, & \text{otherwise} \end{cases} \]

This feature states that the left label usually describes an element with a left and right label.
Some features occur seldom, so the estimates of their expectation based on counts are not likely to be accurate. Therefore, only features with occurrence larger than a threshold are used. In our experiments, only features that occur more than three times are selected in the training stage and extracted in the parsing stage.

5.3. Scoring Parse Tree

Suppose that each feature $f_i$ has a weight $\lambda_i, i = 1 \ldots k$. Given an internal node $n$, the probability of a derivation $a$ and its relevant features $h = \{f_1, \ldots, f_m\}$, is set as

$$p(h, a) = \prod_{j=1}^{k} \lambda_j^{f_j(h,a)}.$$  \hspace{1cm} \text{(5)}

Hence, the conditional probability of a derivation $a$ given $h$ and $n$ is

$$p(a|h) = \frac{p(h, a)}{\sum_{a' \in A} p(h, a')} \hspace{1cm} \text{(6)}$$

in which $A$ represents the set of all possible derivations of $n$.

Let $T$ be a parse tree for a query interface $I$, and $\text{Der}(T) = \{a_1, \ldots, a_m\}$ be the set of derivations contained in $T$. The score of $T$ is defined to be the product of the conditional probability

$$P(a_1, \ldots, a_m | d_1, \ldots, d_m) = \prod_{i=1}^{m} p(a_i | h_i).$$ \hspace{1cm} \text{\text{(7)}}

Therefore, the query interface understanding problem is reduced to searching for the parse tree with the largest probability. That is, we find the best parse tree $T^*$, defined as

$$T^* = \arg \max_{T \in \text{tree}(I)} \text{score}(T)$$

in which $\text{tree}(I)$ represents the set of all possible complete parse trees for query interface $I$.

5.4. Best Tree Search

Suppose a query interface $I$ has $n$ items. This means that there are $n$ leaf nodes in the parse tree for $I$. According to the grammar in Figure 6, each internal node generates at most one leaf node. Hence, there are at least $n$ internal nodes in the parse tree for $I$. For example, the query interface in Figure 1(a) has 25 items after preprocessing and its parse tree has 43 internal nodes. Since each internal node has at least two possible actions, the number of possible parse trees is at least $2^n$. Hence, it is impossible to enumerate all parse trees to identify the best parse tree unless $n$ is very small.

An observed property is that the partial tree of a high-probability full parse tree is also likely to have a high probability. Based on this observation, a beam search algorithm [Feiner et al. 2003] is employed to find the maximum probability parse tree. We do not enumerate all possible parse trees. Instead, only the top $K$ scoring partial trees are explored in the frontier. Algorithm 2 shows the beam search employed for the understanding step. $K$ empty trees $C_1, \ldots, C_K$ are initialized (line 1). From the first element, an iteration is executed (lines 2–7). In each step of the iteration, a new item is considered and each tree $C_i$ is extended to generate $n$ new trees to include the new item (line 4). A new set of $K$ trees is selected from the set of newly generated trees (line 6). The iteration stops when all elements are considered. Finally, the tree with the maximum probability is returned (line 8).
ALGORITHM 2: Beam search

Input: Query interface with \( m \) items \( \langle I_1, I_2, \ldots, I_m \rangle \)  
Conditional probabilities \( P \).  
Integer \( k \).

Output: A parse tree \( S \) with the largest probability.

Called functions: Generate: \( T, I \rightarrow \langle D_1, D_2, \ldots, D_n \rangle \)  
\( /* \) Input: A partial parse tree \( T \)  
\( /* \) A query interface item \( I \)  
\( /* \) Integer \( k \)  
\( /* \) Output: A set of \( n \) parse trees with the maximum probabilities which are generated from the parse tree \( T \) by including item \( I \) */

Merge: \( S_1, S_2, k \rightarrow S \)  
\( /* \) Input: Two sets of parse trees \( S_1, S_2 \)  
\( /* \) Integer \( k \)  
\( /* \) Output: A set of \( k \) parse trees from \( S_1 \cup S_2 \) with the maximum probabilities */

Method:
1: \( C_0 \leftarrow \langle \rangle \)  
\( // C_i \) is the set of \( k \) parse trees with maximum probabilities generated from the first \( i \) items
2: \( \text{for } i = 1..k \text{ do} \)
3: \( \text{Add}(C_0, \langle \text{empty tree} \rangle) \)
4: \( \text{end for} \)
5: \( \text{for } i = 1..m \text{ do} \)
6: \( C_i \leftarrow \langle \rangle \)
7: \( T = \text{First}(C_{i-1}) \)
8: \( \text{while } T \neq \text{NIL} \text{ do} \)
9: \( C_i \leftarrow \text{Merge}(C_i, \text{Generate}(T, I_i), k) \)
10: \( T \leftarrow \text{next}(T) \)
11: \( \text{end while} \)
12: \( \text{end for} \)
13: \( \text{return } \text{First}(C_m) \)

6. EXPERIMENTS

We evaluate the performance of our approach and compare it with some state-of-the-art methods using query interface forms extracted from multiple domains. The cross-validation between query interfaces from different domains and the use of different languages is also reported.

6.1. Datasets

Four datasets were used to perform the evaluation.

(1) TEL-8 is first used in Zhang et al. [2004]. It contains 487 query interfaces from eight domains: airlines, autos, books, car rentals, hotels, jobs, movies, and music records. Each domain contains 20 to 80 query interfaces.
(2) ICQ [Wu et al. 2004] contains 100 query interfaces from five domains: airlines, autos, books, jobs, and real estate. Twenty query interfaces were extracted for each domain.
(3) WISE is used in He et al. [2005b] and consists of 147 query interfaces collected from seven domains: books, electronics, games, movies, music, toys, and watches.
(4) CNW is a dataset containing 200 query interfaces in Chinese from four domains: books, movies, airlines, and autos. Fifty query interfaces are extracted for each domain.
Table II. Characteristics of CNW Dataset Semantic Trees

<table>
<thead>
<tr>
<th>Domain</th>
<th>Leaves</th>
<th>Internal Nodes</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>books</td>
<td>4.96</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>movies</td>
<td>4.26</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>airlines</td>
<td>10.28</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>autos</td>
<td>4.2</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Overall</td>
<td>5.93</td>
<td>1</td>
<td>21</td>
</tr>
</tbody>
</table>

Table III. The Performance of StatParser

<table>
<thead>
<tr>
<th></th>
<th>parse precision</th>
<th>tree metric</th>
<th>condition metric</th>
<th>element labeling correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEL-8</td>
<td>96.1%</td>
<td>95.4%</td>
<td>95.4%</td>
<td>96.5%</td>
</tr>
<tr>
<td>ICQ</td>
<td>94.5%</td>
<td>93.6%</td>
<td>95.1%</td>
<td>96.4%</td>
</tr>
<tr>
<td>WISE</td>
<td>96.4%</td>
<td>96%</td>
<td>96.4%</td>
<td>97.3%</td>
</tr>
<tr>
<td>CNW</td>
<td>92.6%</td>
<td>90.8%</td>
<td>90.4%</td>
<td>92.9%</td>
</tr>
</tbody>
</table>

In all, 734 query interfaces in English and 200 query interfaces in Chinese are used. Table II shows some statistics of CNW. The statistics of WISE and ICQ are shown in Dragut et al. [2009].

6.2. Evaluation Metrics

Four metrics are used to evaluate StatParser and compare it with existing work.

The first metric, parse precision, is the number of correct nonterminal semantic units divided by the number of nonterminal semantic units in the semantic tree.

The second metric, tree metric [Dragut et al. 2009], measures the correctness of the generated semantic tree. A tree edit distance, which counts the minimum number of insert, delete, and relabeling operations needed to convert one tree into another, is used. For an interface, its precision is $P_t = (N_g - D_e)/N_g$, in which $N_g$ is the number of nodes in the generated semantic tree and $D_e$ is the edit distance to the gold standard tree. Its recall is $R_t = (N_g - D_e)/N_s$, in which $N_s$ is the gold standard tree. Finally, the F-score $F_t = 2P_tR_t/(P_t + R_t)$ is calculated.

The third metric, condition metric, measures how well the query conditions are captured. A condition consists of three aspects of information: the name/label of the condition, the set of domain elements, and the set of constraint elements. This metric has been applied in He et al. [2005b] and Zhang et al. [2004]. Given a set of query interfaces, let $Q_g$ be the gold standard query conditions and $Q_a$ be the automatically extracted query conditions. The precision, recall, and F-score are defined, respectively, as $P_c = (Q_g \cap Q_a)/Q_a$, $R_c = (Q_g \cap Q_a)/Q_g$, and $F_c = 2P_cR_c/(P_c + R_c)$.

The fourth metric, Element Labeling Correctness (ELC), measures the correctness of assigning a label to each element. It is defined as the ratio of the number of correctly labeled elements to the total number of elements.

For each query interface in the preceding datasets, the gold standard semantic tree and query conditions are constructed manually.

6.3. Experiment Results

Table III shows the performance of StatParser on the four datasets that are trained using 20 randomly selected query interfaces from the corresponding dataset. It can be seen that StatParser has excellent performance on all of the four datasets.

6.3.1. StatParser Compared with Other Methods. We first compare StatParser’s performance with that of Wise-Extractor [He et al. 2005b], BEP [Zhang et al. 2004], and HA [Dragut et al. 2009]. WISE-Extractor uses some learning, while BEP and HA do.
not use learning. Since Wise-Extractor and BEP only extract query conditions, they cannot be evaluated using the tree metric. For each dataset, StatParser is first trained with 20 query interfaces randomly selected from different domains.

Figure 9 shows the tree metric F-scores of StatParser and HA over ICQ and WISE. StatParser has at least 5% better performance than HA on both datasets. Figure 10 presents the condition metric F-scores of different query interface understanding methods over datasets TEL-8, ICQ, and WISE. The F-scores for Wise-Extractor and HA are taken directly from Dragut et al. [2009] and the F-scores for BEP over TEL-8 only are from Zhang et al. [2004]. It can be seen that StatParser consistently has the best performance among all the three datasets. Its performance is at least 4% better than any of the other three methods.

We also compared StatParser with OPAL [Furche et al. 2012] and LabelEx [Nguyen et al. 2008] on ICQ and TEL-8. StatParser has an Element Labeling Correctness (ELC) of 96.4% and 96.5% on ICQ and TEL-8, respectively. OPAL is reported to have an ELC of more than 98% and 96% on ICQ and TEL-8, respectively [Furche et al. 2012]. Nguyen et al. [2008] report that LabelEx has an ELC of 89% to 95% in four domains on TEL-8. While OPAL has a slightly better performance than StatParser, it requires domain knowledge and does not model the query interfaces hierarchically.

6.3.2. Validation among Different Domains. Table IV shows the performance of StatParser on CNW in which it is trained with 20 query interfaces from one domain and then used to parse other new query interfaces in all domains. The following observations are made from the table.
Table IV. The Cross-Domain Tree Metric F-Scores of StatParser

<table>
<thead>
<tr>
<th>Training with interfaces from</th>
<th>books</th>
<th>movies</th>
<th>airlines</th>
<th>autos</th>
</tr>
</thead>
<tbody>
<tr>
<td>books</td>
<td>95.7%</td>
<td>93.8%</td>
<td>92.2%</td>
<td>91.6%</td>
</tr>
<tr>
<td>movies</td>
<td>92.2%</td>
<td>94.1%</td>
<td>89.7%</td>
<td>91.5%</td>
</tr>
<tr>
<td>airlines</td>
<td>84.6%</td>
<td>84.2%</td>
<td>88.7%</td>
<td>86.6%</td>
</tr>
<tr>
<td>autos</td>
<td>87.8%</td>
<td>87.2%</td>
<td>86.5%</td>
<td>91.4%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>89.5%</td>
<td>89.4%</td>
<td>89.3%</td>
<td>90.2%</td>
</tr>
</tbody>
</table>

Fig. 11. The tree metric F-scores with different number of training interfaces.

(1) The performance on query interfaces from the books and movies domains are consistently the best because the structures of these query interfaces are more regular and are easier to parse correctly. The query interfaces from the airlines domain have the most complex structure and have the worst performance in general.

(2) The query interfaces are usually parsed most correctly when StatParser is trained with the query interfaces from the same domain because the query interfaces from the same domain usually share similar structures.

6.3.3. Effect of the Number of Training Web sites. Figure 11 lists the tree metric F-scores of StatParser over the TEL-8, ICQ, WISE, and CNW datasets given different numbers of training query interfaces. In the experiments, we ran a cross-validation on each dataset to make full use of it. That is, each dataset is divided into several subsets with an equal number of query interfaces. One of the subsets is selected as the training set to train StatParser and all other subsets are used as the test sets to evaluate the trained StatParser. In different experiments, the number of query interfaces in each subset is 5, 10, 15, 20, 25, or 30. On the one hand, the tree metric F-score of label assignment initially increases rapidly as the number of training Web sites increases to 15 because more features are identified and the probability parameters approach the real distribution. On the other hand, the tree metric F-score is fairly stable thereafter as most features have been identified and the probability parameters are very close to the real distribution.

6.3.4. Chinese Form vs. English Form. Besides the cross-validation between query interfaces of different domains, we also cross-parsed query interfaces in different languages. That is, we trained a parser with query interfaces in Chinese and used it to parse query interfaces in English, and vice versa.

The “Amazon effect” states that Web databases within a domain tend to be influenced by their peers as the number of Web databases grows [Chang et al. 2004].
Table V. The Tree Metric F-Scores with Training Datasets from Different Datasets

<table>
<thead>
<tr>
<th>Training with interfaces from</th>
<th>TEL-8</th>
<th>ICQ</th>
<th>WISE</th>
<th>CNW</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEL-8</td>
<td>95.4%</td>
<td>93.2%</td>
<td>94.6%</td>
<td>84.6%</td>
</tr>
<tr>
<td>ICQ</td>
<td>94.7%</td>
<td>93.6%</td>
<td>93.2%</td>
<td>84.8%</td>
</tr>
<tr>
<td>WISE</td>
<td>93.5%</td>
<td>92.2%</td>
<td>96%</td>
<td>86.7%</td>
</tr>
<tr>
<td>CNW</td>
<td>88.3%</td>
<td>89.3%</td>
<td>93%</td>
<td>90.8%</td>
</tr>
</tbody>
</table>

The performance of StatParser trained with 20 query interfaces from one dataset and then used to parse other new query interfaces in all datasets is shown in Table V. It should be noted that CNW contains query interfaces in Chinese while all other datasets contain query interfaces in English. It can be observed that the parser trained with English query interfaces does not have a satisfactory performance on Chinese query interfaces and vice versa. This observation demonstrates that, while English and Chinese query interfaces share lots of commonalities, they also have some minor differences in their design.

It also should be noted that learning from WISE, which has mostly flat interfaces (e.g., it does not have the airlines domain), still has good performance on ICQ and TEL8, which have more complex query interfaces. StatParser is not sensitive to the depth of the query interface. Instead, it is sensitive to the regularity of the query interface. That is, a query interface is more likely to be correctly parsed if its layout conforms to the patterns of most query interfaces.

7. RELATED WORK
Existing approaches focusing on automatic understanding of query interfaces can be categorized into three different types: rule-based methods, learning-based methods, and approaches that use a combination of rule-based and learning-based methods. A thorough survey is presented in Khare et al. [2010, 2012].

7.1. Rule-Based Methods

The Hidden Syntax Parser (HSP) [Zhang et al. 2004] assumes that there exists a hidden syntax during query interface creation. Hence, a set of rules is created manually based on extensive observations on a large number of query interfaces. These rules focus on the visual layout of labels and elements in the query interface. A conflict resolution strategy is also proposed to resolve rule conflicts. HSP suffers the following two drawbacks: (1) an element may be grouped into two different conditions and (2) an element may be missed from a condition.

In Raghavan and Garcia-Molina [2001], the form analysis component creates a set of candidate mappings between elements and labels them according their pixel distances. Thereafter, the mappings are ranked according to position, number of words, font size, etc.

In He et al. [2007], the HTML structure is used to associate the elements and labels. The textual layout of a query interface is represented as an interface expression, which consists of three items \( t, e, \) and \( | \), where \( t \) represents a label, \( e \) represents an element, and \( | \) denotes a new row HTML tag, such as \(<\texttt{p}>\) or \(<\texttt{br}>>. The interface expression is used to assign labels for elements using a set of predefined rules.

In Kaljuvee et al. [2001], a combination of textual, styling, and layout features is used to assign a label for one or several elements. In DEQUE [Shestakov et al. 2005],
an element is assigned with candidate labels that are visually adjacent to it. If there are multiple labels for an element, some heuristics are used to remove some labels until only one label is left for an element. In OPAL [Furche et al. 2011, 2012; Guo et al. 2012], a set of domain-independent rules combined with a domain-dependent ontology is used to understand real-estate forms.

All the preceding rule-based methods do not model the query interface hierarchically, which is very important for many applications. In HA [Dragut et al. 2009], a query interface is modeled as a hierarchical tree. This is the first work that tries to model a query interface hierarchically. Based on a set of rules, two trees are created: a tree representing the label organization according to the visual features and another tree representing the element organization. Thereafter, the two trees are iteratively integrated into a single tree. Compared with HA, StatParser has the following advantages.

(1) StatParser requires less context information and has weaker assumptions than HA, which make it less vulnerable to irregular query interfaces.
(2) StatParser also has a global optimization mechanism which further allows it to be more robust to irregular query interfaces.

### 7.2. Learning-Based Methods

The learning-based methods include LabelEx [Nguyen et al. 2008] and two-layered HMM [Khare and An 2009]. In a learning-based method, a set of labeled query interfaces is required. A learning-based method comprises two stages: the learning stage and the understanding stage. Parameters are learned from the labeled query interfaces in the learning stage and then are used to assign labels for the elements in the understanding stage.

**LabelEx** first generates a set of candidate mappings between labels and elements. Thereafter, a set of trained classifiers is combined to select a most possible mapping for each element. **LabelEx** only assigns labels to elements and does not group related elements into a hierarchically meaningful unit.

In Khare and An [2009], a layered HMM that includes two layers, T-HMM and S-HMM, is used to segment the query interface into meaningful units. The T-HMM layer (where T stands for Tagging) tags each component with a semantic label (attribute-name, operator, or operand) and the S-HMM layer (where S stands for Segmentation) creates groups of related components. Similar to **LabelEx**, a group of related elements is not organized into a hierarchically meaningful unit.

There are also some methods, such as ExQ [Wu et al. 2009] and Benslimane et al. [2007], which use a combination of rule- and learning-based methods. ExQ first employs a spatial clustering method to extract the schema structure of a query interface according to its visual representation. Then, ExQ matches the discovered schema with labels from the query interface using a set of rules.

Benslimane et al. [2007] use a set of rules to create groups of segments, called structural units. Each structural unit corresponds to a logical entity in the database schema. At the same time, a machine learning technique that learns from examples is used to extract relationships between two structural units and the constraints of a query interface.

Neither of the aforesaid methods models the query interface using a hierarchical representation.

### 8. CONCLUSIONS

In this article, we present a novel query interface understanding algorithm, StatParser, which effectively parses a query interface into a hierarchical representation. StatParser
uses a simple grammar enhanced by probabilities that are learned from a set of parsed query interfaces using the maximum-entropy model. The grammar with probabilities is then used to parse a new query interface into parse trees that depict the concept relationships in the query interface. The parse tree with the largest probability is identified as the one that represents the query capabilities of the query interface. StatParser has the advantages of both rule- and learning-based methods. Experimental results show that StatParser is very precise in capturing the element relationships in a query interface and is very effective at extracting the query conditions.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments.

REFERENCES


Received March 2012; revised October 2012; accepted January 2013.