A Hierarchical Approach to Model Web Query Interfaces for Web Source Integration

Eduard C. Dragut
University of Illinois at Chicago
Computer Science Department
edragut@cs.uic.edu

Thomas Kabisch
Humboldt-Universität zu Berlin
Computer Science Department
kabisch@informatik.hu-berlin.de

Clement Yu
University of Illinois at Chicago
Computer Science Department
yu@cs.uic.edu

Ulf Leser
Humboldt-Universität zu Berlin
Computer Science Department
leser@informatik.hu-berlin.de

ABSTRACT

Much data in the Web is hidden behind Web query interfaces. In most cases the only means to "surface" the content of a Web database is by formulating complex queries on such interfaces. Applications such as Deep Web crawling and Web database integration require an automatic usage of these interfaces. Therefore, an important problem to be addressed is the automatic extraction of query interfaces into an appropriate model. We hypothesize the existence of a set of domain-independent "commonsense design rules" that guides the creation of Web query interfaces. These rules transform query interfaces into schema trees. In this paper we describe a Web query interface extraction algorithm, which combines HTML tokens and the geometric layout of these tokens within a Web page. Tokens are classified into several classes out of which the most significant ones are text tokens and field tokens. A tree structure is derived for text tokens using their geometric layout. Another tree structure is derived for the field tokens. The hierarchical representation of a query interface is obtained by iteratively merging these two trees. Thus, we convert the extraction problem into an integration problem. Our experiments show the promise of our algorithm: it outperforms the previous approaches on extracting query interfaces on about 6.5% in accuracy as evaluated over three corpora with more than 500 Deep Web interfaces from 15 different domains.

1. INTRODUCTION

The Web has evolved into a data-rich repository containing significant structured content. This content resides mainly in Web databases that are also referred to as the *Deep Web*. Recent surveys estimated millions of such sources [6, 16]. In order to obtain the contents of Web databases, a user has to pose structured queries. These queries are formulated by filling in Web query interfaces with valid input values. Common examples are job portals or the search for cheap airline tickets. The interface in Figure 1, on the left, is an example of a query interface for booking airline tickets.

With each application domain hosting a large and increasing number of sources, it is unrealistic to expect the user to probe each source individually. Consequently, significant research effort has been devoted to enable a uniform access to the large amount of data guarded by query interfaces. These approaches include: clustering/classifying of Web databases [2, 20], schema matching across a set of interfaces [10, 23, 4, 21], computation of unified interfaces for a given application domain [8, 12], query translation between query interfaces [11] and surfacing the Deep Web [1, 5, 22].

The success of these applications hinges upon two issues: understanding Web databases and obtaining their data. Both rely on a good *understanding* of Web query interfaces, because a query interface provides a glimpse into the schema of the underlying database and is the main means to retrieve data from the database. Besides identifying all its fields, the understanding of a query interface includes: (1) grouping the fields into semantically connected sets, (2) tagging fields and groups with their semantic roles and (3) annotating fields and groups with additional meta-information (e.g. data type). `Departure Date` is an example of such a group in Figure 1. Groups can form bigger building blocks on the interface, e.g., the `Departure Date` and `Return Date` groups form the block `When Do You Want to Go?`. Such grouping naturally leads to a hierarchical representation of query interfaces. Second, tagging assigns *labels* to fields and groups. For instance, in our running example the text `Adults` is assigned as a label to the field `Adults` and the text `Number of Passengers` is assigned to the group of fields `Adults` and `Children`. Third, information such as data type and unit of measurement must be determined.

The goal of this work is the development of an algorithm that extracts and maps query interfaces into a hierarchical representation.

Such an extraction is challenging because query interfaces are represented in HTML, lack a formal specification and are developed independently. HTML is a markup language, designed for formatting documents and not to express data structures and semantics. HTML has a rather loose grammar and browsers do not enforce the grammar when displaying HTML pages. As a result, ill-written HTML pages can

VLDB '09, August 24-28, 2009, Lyon, France

often be displayed by browsers and used by people. Query interface design seems rather heuristic in nature—there is no clear guidance of how to create an interface. Interfaces follow different design patterns (e.g., the orientation of labels can vary—the fields in Figure 1 have the labels above while the fields in Figure 2 have the labels to the left). Furthermore, similarly looking interfaces can be developed with different HTML constructs.

The main contribution of this paper is the development and evaluation of a method that utilizes mainly the visual content features of a Web query interface as displayed on a browser. Query interfaces are represented as schema trees of arbitrary depth. The extraction of a query interface into a schema tree is guided by two main insights. First, we noticed that certain geometrical patterns (horizontal/vertical lines) can reveal the presence of semantically related fields in a query interface. Second, by drawing a parallel between the layout of documents and the appearance of user interfaces we can further discover meaningful relationships among the elements of a query interface.

Our specific contributions are the following:

- We define a small set of general rules that help in reengeneering query interfaces in a formal model with high accuracy. Previous methods following a similar line used a significantly larger number of heuristics which carries the danger of overfitting (see Section 6).
- We use the visual arrangement of interface elements (labels, fields, etc.) to determine semantically related fields. This approach bypasses the intricacies of coding query interface in HTML. In contrast to previous approaches using the visual layout of Web sites, we use a much more expressive set of rules to exploit layout information for interface extraction (see Section 4).
- We bundle the identified design rules and the knowledge extracted from visual layout into a highly accurate algorithm for automatically transforming query interfaces into schema trees. These schema trees cover much more information for Deep Web integration than previous, "flat" models (see Section 5).

We prove the superiority of our method by an extensive evaluation using more than 500 interfaces from 15 different domains. Our method reaches an accuracy of 90% on average, which is a 6.5% improvement over our closest competitors (see Section 6.4).

Motivation for Hierarchical Representation

A main feature of our method is the representation of query interfaces in a hierarchical format. We provide concrete examples of applications that utilize query interfaces and we show how these applications would benefit from a hierarchical representation of query interfaces.

1. Query Interface Matching It was shown that matching can be significantly improved when interfaces are represented hierarchically [23]. Among others, hierarchical representation helps to avoid false matches due to the *homonymy problem* and to identify complex matchings (e.g., is-a and part-of relationships). An example of the former is provided by the label `From` in Figure 2 (left half) which is used to denote both year and price. Unless the labels of the internal nodes are considered, when matching the interfaces in Figure 2, false matches may be derived, such as `Year.From`=`Price Range.From`. By representing Web query interfaces hierarchically additional schema matching techniques can be applied [17, 7].

2. Building Unified Query Interfaces Recent work [8, 9] has shown that integrated Web query interfaces generated from sources represented hierarchically are qualitatively better than the ones generated from sources having a flat representation [12]. A user survey showed that interfaces generated from hierarchical representations are easy to understand.

3. Deep Web Crawling The Deep Web crawling [1, 3, 5, 19] requires the understanding of query interfaces. A crawler needs to input meaningful values into the fields of query interfaces to retrieve the data from Web databases [3]. We observed that a preorder traversal of the schema tree of a query interface reflects the way a human being parses the interface in order to understand the meaning of the fields. For instance, in Figure 1 before a user reaches the field `From` she first encounters the label `Where Do You Want to Go?`. Thus she has an unambiguous understanding of the meaning of the field `From`. A proper annotation of the nodes of the schema tree of a query interface could help a crawler to automatically understand the meaning of the fields within the interface. The crawler can follow the same preorder traversal that a human uses to parse the interface.

4. Another challenge to automatic understanding of query interfaces is the presence of **inter-related fields** [5]. These fields restrict the kind of queries that can be submitted to a search engine. For example, in the left interface in Figure 2 the fields `Make` and `Model` are related. The selection of a value in the field `Make` restricts the possible values shown in the field `Model`. The problem faced by a crawler is determining those related fields from the cartesian product of fields. The search space can be significantly reduced as such fields are siblings in the hierarchical representation.

The rest of the paper is organized as follows. Section 2 shows related work and underlines the differences between previous work and ours. Section 3 defines our understanding of Web query interfaces. Section 4 exploits the main idea of our approach and lays it down into 9 commonsense design rules. Section 5 describes our extraction algorithm. In Section 6 we present our experimental results. We give a conclusion in Section 7.

2. RELATED WORK

Hierarchical Web interface models are an important step toward a better integration of Web sources. We believe our approach is unique in that it produces structured schema trees which offer a rich set of information for integration systems.

There have been a number of recent suggestions to improve the extraction of Deep Web interfaces. The approaches presented in [14, 19, 15] regard query interfaces as a flat collection of fields. Consequently, the main problem addressed is the prediction of the right label for a field. One frequently used heuristic employed for prediction is the textual similarity between the name of a field and a label. In the example of Figure 1 the name of the field `Adults` is `numAdults` and its label is `Adults`. Since `numAdults` and `Adults` share a significant text portion their similarity score is high enough to suggest that the latter is the label of the field. In our initial effort, we implemented this heuristic as well, but we noticed that its prediction accuracy is rather minute. More-

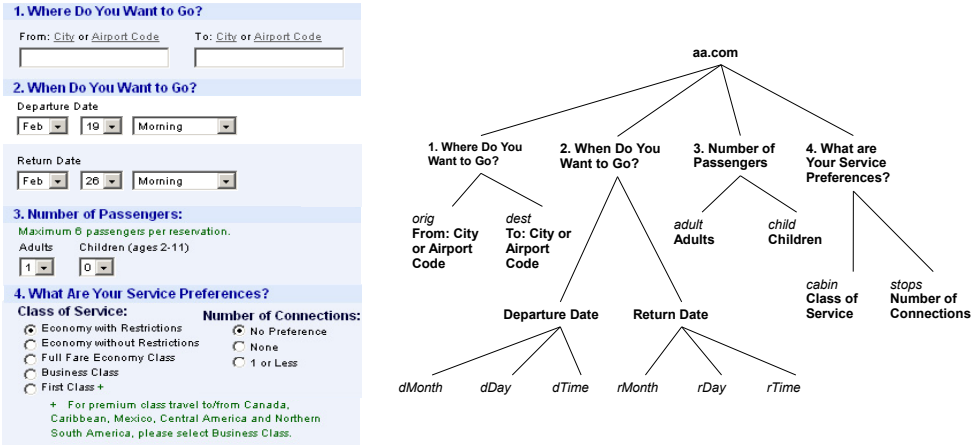

Figure 1: An example of an ordinary query interface in the airline domain along with its schema tree.

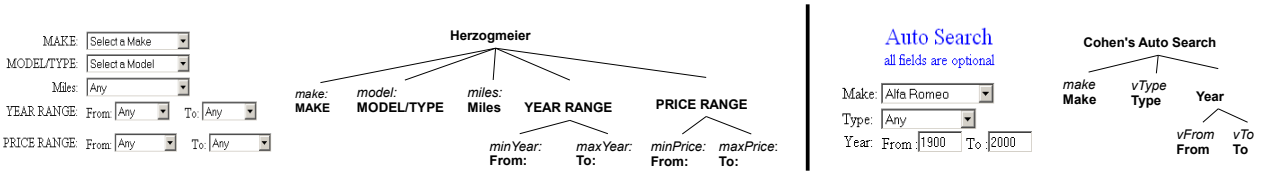

Figure 2: Two query interfaces from the auto domain.

over, it is not able to discover labels for groups of fields (internal nodes) since these do not have names/ids within HTML pages. Subsequently this heuristic was dropped from our implementation. Furthermore, a flat representation of interface fields fails to properly represent the semantic relationships between them.

The approaches most related to ours, in that they also view query interfaces as structured objects, are [24, 13]. [13] uses *attributes* to group sets of related fields. For example, the three fields denoting the departure date in the interface in Figure 1 would be captured as one attribute, and the field `From` on the same interface would be an example of an attribute containing a single field. However, attributes cannot be nested, and therefore this approach would fail to see that the fields `From` and `To` should be grouped together and that they can be semantically characterized by the label `Where Do You Want to Go?`. Similarly, [24] proposes to use a form of grouping fields, but is not capable of representing arbitrarily nested structures. In contrast, our fully hierarchical approach describes Web interfaces as schema trees.

Another feature of our approach is the exploiting of visual layout information for recovering the structure of an interface and for assigning labels to fields. Most existing tools for Web wrapping are based on HTML parsing instead. However, HTML is a rather fuzzy language with many ways to express the same appearance. Especially with newer HTML technologies (e.g. style sheets), the neighborhood of tags and text in HTML is blurred completely and thus does not provide robust clues for interface extraction. In contrast, our method uses visual techniques to analyze Web pages and thus is more robust against HTML particularities and code evolution. The work most similar to our approach is [24] which introduces the notion of viewing query interfaces as visual languages. However, their extraction algorithm uses a grammar with more than 80 productions that was manually derived from a single corpus of 150 interfaces. When using such a high number of very specific rules there is a probability that the resulting system may overfit to the corpus the rules were derived from [18]. Moreover the rules derived in [24] reflect concrete patterns found on Web interfaces at the time the system was developed. But the appearance of Web interfaces evolves with new opportunities of the underlying technology. Therefore, our algorithm is based on only 9 general rules which are completely domain-independent and do not rely on specific specific implementation issues.

3. REPRESENTATION OF QUERY INTERFACES

In this section, we present our way of modeling a query interface. Methods for mapping a given Web query interface into such a model are described in the rest of the paper. Data in searchable databases are accessible through form-based search interfaces (mostly HTML forms). The basic building blocks for these forms are: *text input boxes, selection lists, radio buttons* and *check boxes*. We will generically call them fields. A *text input box* is rendered as a empty box with or without a default value. The Field `From` on the interface in Figure 1 is such a field. A *selection list* presents the user with a set of choices to select from. There are two types of selection lists: single selection list (e.g. combo box) and multiple selection list (e.g. listbox). *Radio buttons* and *check boxes* are employed by designers to explicitly display the choices to the user. For example, in Figure 1 the `Class of Services` are shown as radio buttons. The difference between radio buttons and check boxes is that choices are exclusive in a radio button group, whereas multiple check boxes may be selected at the same time. A radio button group can be regarded as a single selection list and a group of check boxes as a multiple selection list. For example, the set of radio buttons denoting the class of service is treated as a field (single selection list) whose label is `Class of Service`. The labels attached to each individual radio button (e.g. `Economy with Restrictions`) become the values of the se-

lection list field. To summarize, we have two types of fields: fields with predefined sets of values and fields without predefined values.

A field has a *name*, which identifies the field in the HTML script (for programming purposes). Fields may also have *labels* that describe to users the meaning of the field. Fields may not have their own labels, rather they share a group label with other fields. For instance, the three fields denoting the departure date in Figure 1 do not have their own labels but they share a group label. In some cases the label may be entirely left out as the designer relies on the set of values of the field to convey the semantics of the field. While names are readily available from HTML, the assignment of labels requires substantial work, but is necessary for the correct understanding of the semantics of a field or group of fields.

An important aspect of user interfaces is a sort of *spatial locality property* among the fields. That is, semantically related fields are usually *grouped* together in an interface. For example, in the interface in Figure 1 the fields denoting the types of passengers traveling are next to each other. Moreover, several related groups can further be grouped together. In our interface the two groups of fields denoting departure date and return date, respectively, are put together (`When Do You Want to Go?`). Thus, this bottom-up characterization gives rise to a hierarchical structure for interfaces. In addition, each group of fields may have labels that describe to the user what the group is about.

The hierarchical structure of query interfaces was first observed by [23]. More in detail, a query interface is an ordered tree of elements so that leaves correspond to the fields in the interface, internal nodes correspond to groups of fields in the interface, and the order among the sibling nodes within the tree resembles the order of fields in the interface (this is from left-to-right for documents in the western world). This *schema tree* captures both the order semantics and the nested grouping of the fields in a query interface. Figure 1 shows a typical example of a query interface in the airline domain and its corresponding schema tree. Observe that the schema tree has four levels and that each level except for the root refines the information in the level above. The first level is a generic root node (usually the root has the name of the Web site).

Three more types of meta-information are attached to the leaves (fields) and internal nodes (groups of fields): *domain type*, *default value* and *unit of measurement*. Domain types are of two kinds: simple and complex. Examples of the former type are integer or string. Examples of the latter type are date, time, datetime and currency. Complex domain types can be associated with a group of fields (e.g. `Departure Date` in Figure 1 has the datetime domain type) Many fields have default values, e.g., the fields denoting month and day of the departure date in Figure 1. Frequently a default value may appear in a text box. As shown in [5] the default value of a text box may be a valuable indicator of the kind of input the field expects, since the domain type of text boxes is difficult to determine in general. Units of measurements such as "miles" and "square feet" are important pieces of information that need to be properly extracted and attached to the right field (group of fields). They frequently appear abbreviated in query interfaces (eg. "mi" stands for "miles"). The abbreviations are recognized by consulting certain Web sites, e.g. `www.abbreviations.com`.

4. EXPLOITING DESIGN RULES

In this section, we describe our observation that almost all real-life Web interfaces obey to a small set of rules that partly determine their appearances. Although they appear trivial in first place, we shall show in Sections 5.2, 5.3 and 5.4 that exploiting these rules enormously help in re-engineering Web interfaces in a formal model.

Automatic extraction of query interfaces is challenging because interfaces are created autonomously and with languages (e.g., HTML) obeying a loose grammar. The question arises whether there is an inherent set of rules that designers of query interfaces intuitively follow. Our investigation of a reasonable large number of query interfaces in various domains showed that a *small* set of *commonsense design rules* emerges from heterogeneous query interfaces. We first enumerate the rules and then motivate them by drawing a parallel between documents and query interfaces.

Except for Rule 0 and 6, all the other are new, and not encountered in any of the previous extraction techniques. This is of no surprise since none of them have the concepts of groups and subgroups.

- **Rule 0:** Query interfaces are organized top-down and left-to-right.
- **Rule 1:** Fields within an interface are organized in semantic units of information, i.e. groups.
- **Rule 2:** A label is used to denote either the semantics of a field or of a group of fields, but not both.
- **Rule 3:** If a field f with a label l_f belongs to a group g with label l_g then the *text-style* of the label l_f is different from the text-style of label l_g.
- **Rule 4:** If a group g with a label l_g is a subgroup of a group G with label l_G then the text-style of the label l_g is different from the text-style of label l_G.
- **Rule 5:** The labels of all the members of a group have the same text-style.
- **Rule 6:** The orientation of a label of a field is either to the left, above, right or below of the field. The label of a group is either above or to the left of the group.
- **Rule 7:** The labels of all the members of a group have the same orientation.
- **Rule 8:** Let G be a group and g be one of its subgroups. Suppose a label with text-style FS_1 is assigned to G and a label with a different text-style FS_2 is assigned to g, then for any group H and its subgroup h the label assigned to H cannot have the text-style FS_2 when the label assigned to h has the text-style FS_1.

The first two rules phrase rather obvious observations. First, ordinary people expect the content of documents (Web pages) to be laid out in a predicted pattern—i.e. top-down and left-to-right. Second, the content of such a document must be structured in some organic units so that it is easy to understand—e.g., it would be rather peculiar to have the fields denoting the departure date separated by some other fields, such as the passenger fields. Rule 2 says that a label cannot play multiple roles as this would confuse an ordinary user.

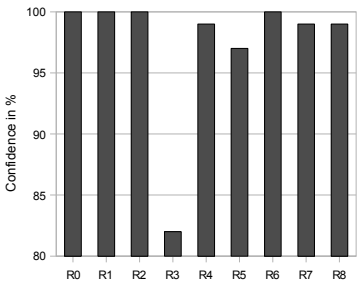

Figure 3: Histogram of rule confidence.

In a text document, such as this paper, headings are employed to organize the content of the document. Headings are located at the top of sections and subsections which they delimit. Headings serve several important roles in documents: they preview and succinctly summarize upcoming content and they show subordination. They naturally lead to a hierarchical structure for a document.

In many ways a Web query interface can be regarded as a document: its fields along with their labels are the content and the labels of the groups are the headings. A label of a group of fields, similarly to a heading, should succinctly summarize the upcoming set of fields. For example, the label `Where Do You Want to Go?`, in Figure 1, describes the purpose of the fields in the section it introduces. A user, thus, learns that the fields `From` and `To` represent the departure and arrival information, respectively.

A set of heuristics emerges from the parallel between query interfaces and documents. The text-style of a heading is different from the text-style of the content. Likewise, the label of a field has a distinct text-style than that of a label of a group of fields (Rule 3). The text-style of a heading is different than that of its subheadings; similarly, the text-style of the label of a group is distinct from that of the label of its subgroup (Rule 4). Headings at the same depth in the document hierarchy have the same text-style and labels denoting sections at the same depth of a query interface have the same text-style (Rule 5). The subheadings of a heading have all the same alignment (e.g., left, center). Likewise, the labels of the members of a group have the same orientation (Rule 7). If a text-style is chosen for a heading H and a different text-style is chosen for its subheading h, there must be no other subheading having the text-style of H and moreover there must not be a sibling heading of H having the same text-style as h. If this rule is translated to the labels in a query interface, Rule 8 is reached.

We conducted an informal survey of these rules. The ICQ dataset was used. (The datasets used in this work are described in Section 6.) The histogram in Figure 3 depicts the outcome of our study. The survey reveals that there are many similarities between the organization of a document and that of a query interface. All the rules, except for Rule 3, are satisfied by almost all query interfaces in the dataset. Rule 3 is violated in 18% of all the interfaces. The reason is that these interfaces use the same text-style for the labels of fields as well as for the labels of groups. An example is shown in Figure 2, the field-label `Make` utilizes the same text-style as that of `Year Range`. We introduce a heuristic to cope with such cases in Section 5.3.

The survey also shows that this set of rules holds across diverse domains (ICQ has interfaces from five domains). It implies that there are implicit conventions that influence the

Figure 4: Fields and texts with bounding boxes.

design of Web query interfaces. Although there is not a common wisdom how to build an ideal query interface, the creative process is guided by the way most humans expect documents to be laid out. People in the western world read from left to write and top to bottom. Furthermore, objects referred throughout a document are a priori defined. A group of fields (section) on a query interface can be regarded as such an object and its label as its definition. Consequently, when humans visually parse query interfaces they expect to encounter the label before the group (section). These rules act as axioms to build the data structures employed in the extraction algorithm.

The set of rules is by no means universal. It is our contention that the rules can be easily adapted to accommodate query interfaces developed for people speaking languages following other orientation patterns. We manually inspected query interfaces intended for Arabic languages. We observed that these interfaces are organized from right to left and top-down. For these interfaces, one only needs to swap "left" with "right" in the commonsense rules.

5. THE EXTRACTION ALGORITHM

5.1 Overview

First we give a high level description of the steps of the algorithm. Each but the first step is explained in detail in the following sections. These are the steps of the algorithm:

Token Extraction: An HTML Page is input into a rendering engine of a browser (e.g. IE). A list of *tokens* is extracted from it. A token is an atomic visible element on the page. The token list is cleaned and filtered. There are three types of tokens considered: *text tokens*, *field tokens* and *image tokens*. Each token is enclosed in a rectangular area that describes the layout coordinates of the token in the actual window frame. This area is called *bounding box*. For example, in Figure 4, `Departure Date` is a text token, the field showing the value `Feb` is a field token and there is no image token.

Tree of Fields: An initial tree of fields, called FT, will be generated based on the order and alignment of the fields in the rendered version of the interface. Fields and groups correspond to leaves and to internal nodes, respectively, in the tree. Additionally, a set of candidate labels is determined for each field. The tree in Figure 5 represents FT derived from the interface in Figure 1.

Tree of Text Tokens: We hypothesize that a text token in an interface has a *semantic scope*. Intuitively, this is the area of the interface which is characterized by the semantic meaning of the text token. As an example, the semantic scope of `When Do You Want to Go?` is the rectangular area that includes every text token and fields that are between the text token itself and the text token `Number of Passengers` (Figure 6). The semantic scope of `Departure Date` includes the text token and the three fields denoting month, day and time of departure. The tree of text tokens, called TT, is inferred from the inclusion relationship between the rectangular areas defining the semantic scopes of text tokens (e.g., the semantic scope of `When Do You Want to Go?` includes

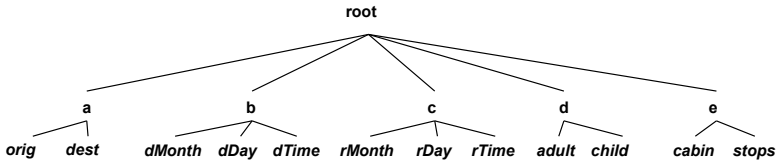

Figure 5: Tree of fields.

that of `Departure Date`, thus the latter text token is a child of the former text token). The tree in Figure 7 represents the tree of text tokens derived from the interface in Figure 1.

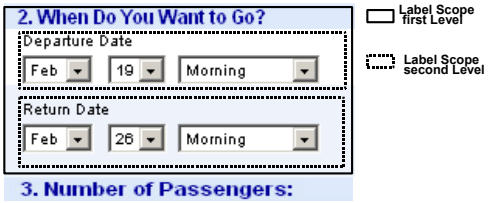

Figure 6: Semantic scopes of text tokens.

Integration: The final hierarchical representation ST (schema tree) of the interface is obtained by merging the two trees FT and TT. FT is the target tree and TT is the source tree. A directional "mapping" from TT to TF is defined. A label is mapped into a leaf (field) if it was determined to be a candidate label. The semantic scope of a label l contains a set fields (leaves). A label is mapped into an internal node if its semantic scope contains all the fields of the internal node. Multiple labels may be mapped into each node of the tree. New internal nodes may be added to the tree of fields FT. The goal of this step is to find the final schema tree and the assignment of labels to its nodes. For our running example, the final schema tree ST is depicted in Figure 1, on the right.

5.2 The Tree of Fields

This section describes the methodology for the construction of the tree of fields of a query interface. Recall that the main goal is to infer the "hidden" schema tree structure of a Web query interface. Two issues need to be addressed. First, given that the schema tree of a query interface is an ordered tree, the problem is finding the *semantic order* of the fields on a query interface. The semantic order of the fields in a query interface is the order in which a user reads and fills in the fields. On our running example (Figure 1), the semantic order of the fields is `From, To, Departure Month,..., Number of Connections`. The second problem is the grouping of related fields. The third problem is the assignment of candidate labels to the leaves of the schema tree.

Semantic Order of Fields

For each field a `tabindex` attribute can be set in HTML. The tabbing order defines the order in which elements receive focus when navigated by the user via the keyboard. Designers *may* employ this attribute to specify the fill in order of fields in a query interface. This attribute is scarcely used. Only 10 out of 100 interfaces in the ICQ dataset utilize it. The question remains, whether there is any other way to infer the fill in order of the fields when the `tabindex` is not specified for the fields. Our empirical study shows that the order in which the fields are encountered in the source code usually coincides with the fill in order. This is also the strategy employed by the rendering engines of browsers. The explanation is that, while developing Web query interfaces, designers place fields in the HTML source code in the order the user parses them. To conclude, whenever tabindex is encountered in a Web query interface we use it to determine the semantic order of fields and when it is not present we utilize the order the rendering engine provides.

Grouping of Related Fields

Although the HTML language and its add-ons have some constructs (e.g. `<fieldset>`) that could be used to emphasize certain grouping of fields in a HTML page, which in turn could be employed to infer the hierarchical structure, our experience with real world Web query interfaces shows that these constructs are sparsely found within the source code of Web pages. (Designers are mostly interested in the layout of their products rather than in their semantic annotations). But, are there any other layout hints that could be employed to infer groupings of fields?

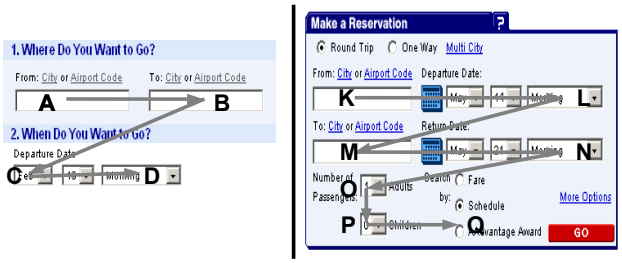

Figure 8: Semantic order and inflection points.

We noticed the following geometric pattern for Web query interfaces. If straight line segments are drawn between any two consecutive fields in the semantic order, then all these line segments form one connected curve. The curve consists of horizontal, vertical and diagonal line segments. A horizontal (vertical) line segment corresponds to a set of fields laid out row-wise (column-wise) on the visual rendering of the query interface. Figure 8 (left half) shows the curve for a fragment of the interface in Figure 1. The curve may also have a number of *inflection points*. An inflection point is a point on a curve where the curve changes from being concave upwards (positive curvature) to concave downwards (negative curvature), or vice versa. Thus, in our case, an inflection point is a point where two non-parallel line segments meet. An inflection point marks either the end or the beginning of a semantic group of fields in the interface. For instance, in Figure 8 (left half) inflection point B marks the end of the group of fields `From` and `To` and point C marks the beginning of the semantic group `Departure Date`. An inflection point is the geometric "evidence" that the designer finishes/begins the organization of a subset of fields into a group of semantically related fields. Hence, a horizontal/vertical line segment emphasizes the presence of a group of fields. For example,

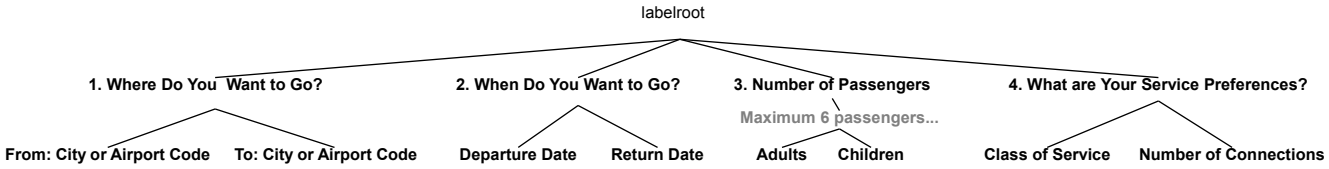

Figure 7: Tree of text tokens.

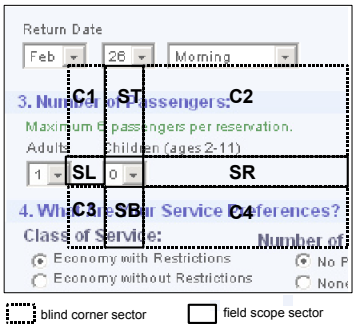

Figure 9: Exemplified field scopes.

the line segment $[C, D]$ denotes the group `Departure Date` while the vertical line segment $[O, P]$ in the right half of Figure 8 represents the group `Number of Passengers`.

Once the groups of fields have been determined, the tree of fields is constructed bottom up as follows. We start with a flat tree—all fields are children of the root and ordered from left to right according to their semantic order. Then, an internal node is added to the tree for each determined group of fields. Figure 5 shows the derived tree of fields for the running example (e.g., an internal node was added for the groups of fields `From` and `To`).

There is one more issue to be addressed: Can a field belong to multiple groups of fields? In our geometric interpretation this corresponds to an inflection point that joins a horizontal with a vertical line segment. On such an occurrence the field is assigned to the group corresponding to the horizontal line segment since the fields on query interfaces are mostly row-wise organized.

Candidate Labels for Fields

Another important problem in the construction of the schema tree is the semantic tagging of its leaves (fields) and internal nodes (group of fields). There is no consistent pattern across query interfaces as to where a label is positioned with respect to the field it defines. A label may be to the left, to the right, above or below of the field. This enumeration is the order of places a label is most likely to appear for a field. One may choose to assign labels according to this prioritization. This is the strategy employed by [24, 13], but it is not generic and leads to errors. Our strategy instead is to collect first all possible candidate labels for each field and then, in a later step (integration, see Section 5.4), to decide which are the appropriate ones. More in detail, first, for each field, a set of candidate labels is found from all text tokens, using Rule 6. This reduces the set of text tokens which are possible labels for the field. Then, the groups of fields are computed. Afterward, a text token is chosen from the set of candidate labels as the final label, if it satisfies Rules 5 and 7. These steps are applied iteratively, as whenever a final label is determined for a field, the set of candidate labels for every other field or group of fields is updated (Rule 2).

We now consider the possible locations where the label of a field should be placed with respect to the rectangular box enclosing the field. This is called the *scope of the field*. The space around a field is partitioned into 8 sectors by the vertical and horizontal lines going through the top-left and bottom-right corner points of the bounding box of the field. Figure 9 shows the sectors around the field `Children` (the combo box having the value 0 selected). The corner sectors, marked with C_1 to C_4 in the figure, are called *blind spots*, because the label of a field cannot reside *entirely* in those areas. If the label were entirely in one of these sectors, then between the label and the field would be a diagonal relationship. This would be a violation of the common sense Rule 0, that information is organized top-down and left-to-right. The four sectors, denoted ST (top sector), SL (left sector), SR (right sector) and SB (bottom sector) constitute the scope of the field. A candidate label must lie in one of these four areas, although it might extend into a blind spot. For example, the scope of the field `Children` is the shaded area in Figure 9. Each sector is a bounded rectangular area. Each sector starts at the bounding box of the field and stretches in one of the four directions (e.g., top sector stretches upward) until the bounding box of another field or the boundary of the interface is met. A candidate label is a text token whose bounding box intersects the scope of the field. For each sector an ordered list of candidate labels is computed. The order is given by the distance between the field bounding box and the candidate label bounding box. In our example, the left and right lists of candidate labels of the field `Children` are empty. The top list consists of `Children, Maximum 6 passengers per reservation` and `Number of Passengers`. The bottom list consist of `What are Your Service Preferences?` and `Class of Service`.

5.3 The Tree of Text Tokens

The second piece of information employed for determining the hierarchical structure of a query interface is the layout relationship between text tokens, which is covered in this section. We distinguish between *text tokens* and *labels*. The former refers to any text appearing in a query interface, e.g., comments. The latter is a text token that was identified to be the semantic tagger of a field or a group of fields.

The hierarchical structure of headings is easy to extract for documents because headings are explicitly tagged. The labels in Web query interfaces lack such tagging, thus the problem of their hierarchical subordination is harder. Nevertheless, using the analogy between headings and the labels assigned to a group of fields, a technique can be designed. There are two main observations. First, headings at the same depth in the heading hierarchy of a document have the same text-style. Consequently, we expect that the labels assigned to the groups at the same depth in the schema tree of query interface to have the same text-style (Rule 5). Thus, the first task is to cluster/classify the text tokens appearing on a query interface based on their text-style properties.

Second, for any two headings H and h, with h subheading of H, the content area covered by h is a subset of that of H. So, if we knew the (semantic) area each label covered on the interface then the hierarchical subordination relationship between the labels in a query interface could be determined by using the inclusion relationship. We show how the semantic areas of text tokens are estimated and used to infer the hierarchical relationship between them in this section.

Clustering of Text Tokens

Each text token has a complex style attribute, which describes its layout properties: font-color, background-color, font-size, font-style, font-weight and font-family. The properties are retrieved from the rendering engine of a browser. The text tokens with the same values for their style properties are clustered together. For our running example (Figure 1), there are three clusters of text tokens:
$C_1 =$ { `Where do you want to go?`, `When...`, `...`};
$C_2 =$ { `From, To, Departure Date, ...`};
$C_3 =$ { `Maximum 6 passengers per reservation, ...`}.

Semantic Scopes of Text Tokens

Since the relationships between a label l of a group of fields and the objects it characterizes are not explicitly given in the HTML source code, we hypothesize that the semantics of l summarizes a (rectangular) area on the visual rendering of the interface. Thus, any object in this area is semantically described by the label.

The *semantic scope* of a text token t, denoted $scope(t)$, is the maximal rectangle with the following properties:

1. Its left-upper corner coordinates are the coordinates of the left-upper corner of the the bounding box of text token t.

2. It extends downward and to the right until the semantic scope of another text token p in the same style cluster or the boundary of the interface is met.

3. Let q be a text token from a different style cluster than that of t. If the bounding box of t is included in the semantic scope of q then the semantic scope of t is included in the semantic scope of q.

The semantic scope of the text token `Departure Date` (Figure 6) starts at the left-upper corner of the bounding box of the text token and continues downwards until text token `Return Date` is met and to the right until the boundary of the interface. Its semantic scope is included by the semantic scope of the text token `When Do You Want to Go?`, because its bounding box is inside the semantic scope of `When Do You Want to Go?`.

In order to estimate the semantic scope of a text token t on the visual rendering of a query interface we need to know the text tokens of the same text-style "closest" to it in either rightward or downward directions. We call them *rightward neighbor*, t_r, and *downward neighbor*, t_d, respectively. The boundary of the semantic scope of t is computed with respect to its neighbor text tokens or the boundary of the interface. Denote by $\mathcal{T}$ the set of text tokens of the same text style as that of t in a given query interface. The expressions below define t_r and t_d, respectively. The left-upper and right-bottom coordinates of the bounding box of a text token $p \in \mathcal{T}$ are $(p.X_1, p.Y_1)$ and $(p.X_2, p.Y_2)$, respectively.

$$t_r = \min_{p.X_1}\{p | p \in \mathcal{T} \wedge p.X_1 > t.X_2 \ \wedge \ p.Y_1 > t.Y_1\} \quad (1)$$

$$t_d = \min_{p.Y_1}\{p | p \in \mathcal{T} \wedge p.Y_1 > t.Y_2 \ \wedge \ p.X_2 > t.X_1\} \quad (2)$$

Equation 1 says that from the set of all text tokens in the same style cluster as t that reside to the right of t and are not above of it, the text token with the smallest X_1-coordinate is the rightward neighbor. In a similar way, Equation 2 defines the downward neighbor. It is possible that the boundary of the interface is met, a small adjustment is made to each equation. In our running example, the downward neighbor of the text token `When Do You Want to Go?` is the text token `Number of Passengers` (Figure 1) and the rightward neighbor is the right boundary of the interface.

Algorithm 1 ComputeTokenTree($WF, root$)

Input: current window frame WF
Output: the root of the tree of tokens

```
topT = ComputeTopLevelTokenSet(WF, root);
for all t ∈ topT do
    create node node_t for token t;
    root.addChild(node_t);
    scope_t = getSemanticScope(t, WF);
    ComputeTokenTree(scope_t, node_t);
end for
```

Having these concepts defined we can provide the algorithm for the computation of semantic scopes and of the tree of labels. The algorithm for computing the hierarchical relationship between text tokens is depicted in Algorithm 1. The algorithm is a recursive algorithm. The input of the algorithm consists of the current rectangular window frame WF on the visual rendering of a query interface. The output is the root (artificially created) of the tree of text tokens. The algorithm is initially called with the window frame of the entire query interface. In the current window frame, the algorithm first retrieves the set of text tokens $topT$ in the same style cluster satisfying the properties: (1) the union of their semantic scopes covers the entire set of fields in the current window frame and (2) it is the smallest set out of all such sets of text tokens. This set of tokens is called *top level tokens*. The procedure `ComputeTopLevelTokenSet` finds this kind of tokens and is omitted here for brevity.

Then, for each token t in the set $topT$ the following steps are performed. First, token t is appended as a child to the current root. Second, the semantic scope of token t is computed using the neighbor text tokens of t, Equations 1 and 2 (procedure `getSemanticScope`). Third, the set of text tokens that are inside of the window frame defined by the semantic scope of t is retrieved. Finally, the algorithm is recursively called with the window frame defined by the semantic scope of t. The recursive call terminates when there are no text tokens $topT$ in the current window frame.

We show how the tree of text tokens (Figure 7) for the running example is obtained by the algorithm `ComputeTokenTree`. The initial window frame is the entire query interface. The text tokens of each of the first two clusters C_1, C_2 cover all the fields of the interface. The text tokens of cluster C_3 do not cover fields such as the one with label `From` and are therefore discarded. Since the cluster C_1 has a smaller number of text tokens, its tokens become the first level of the

tree. Then, the semantic scope of each token in C_1 is computed. For example, the semantic scope of `When Do You Want to Go?`, called SC, stretches all the way to the right boundary of the interface and down to the token `Number of Passengers`. SC includes the text tokens `Departure Date, Return Date`. The algorithm recursively computes the semantic scope of each of these two text tokens. The union of the semantic scopes of `Departure Date` and `Return Date` includes all the fields within SC. Therefore these text tokens become the children of the text token `When Do You Want to Go?`. Furthermore, the algorithm recursively goes into the semantic scopes of these two text tokens. Since there are no other text tokens in their scopes, this branch of the recursive calls terminates. All other subtrees are obtained similarly. The resulting tree is shown in Figure 7.

Adjustments

As shown in the histogram in Figure 3, Rule 3 is not always satisfied. The main issue is that designers choose to assign labels with the same text-style to both fields and groups of fields. For instance, the labels for fields `From` and `To` have the same style as the label `Year` in Figure 2 on the right. The semantic scope of `Year` is "empty", because its rightward neighbor is `From` and its downward neighbor is the boundary of the interface. Therefore, its scope contains no field. To overcome this issue, we expand the semantic scope of such a text token to the right and downward until one of the following conditions (1) the next token with an empty semantic scope, (2) a text token with at least two fields in its scope or (3) the interface boundaries is met. The expanded semantic scope of `Year` contains the fields with labels `From` and `To` due to the satisfaction of (3).

5.4 Integration

Given the tree of fields FT (e.g., that in Figure 5), the tree of text tokens TT (e.g. that in Figure 7) and the lists of candidate labels for the fields of a Web query interface we develop an algorithm to derive an integrated tree structure that represents the schema tree of the interface ST (e.g., that in Figure 1). The body of the algorithm is given in Algorithm 2. The tree of text tokens is used to determine labels for the internal nodes (group of fields) of the schema tree and to prune the initial candidate labels of fields as determined in Section 5.2.

Using the semantic scopes of text tokens, the set of candidate labels of a field can be further pruned. If a field f is within the semantic scope of some text token t, then any candidate label for f is t, one of its descendent text tokens or null. For example, in Figure 6 the field dTime (the first field having default value `Morning`) cannot have the text token `When Do You Want to Go?` as a candidate label from its upward direction, because the field is within the semantic scope of `Departure Date` and `When Do You Want to Go?` is not a descendant of `Departure Date`. Similarly, candidate labels of a field from the other directions can be pruned.

The labels for the fields are determined from the remaining candidate labels. The algorithm first assigns those text tokens as labels to fields that are explicitly specified by designers in the HTML code (tag `<label>`), if such specifications exist. The algorithm also discards the text tokens from the tree of tokens, whose semantic scope does not contain any field. Then, the algorithm iteratively performs two tasks. It assigns labels to leaves (fields) according to Rules 5 and 7. That is, the labels of the fields in a group have to be on the same side of the fields and to have the same text-style. The algorithm may discover new groups of fields in addition to those already found in the tree of fields and assigns, whenever possible, labels to internal nodes. This step is accomplished by procedure `doMerge` (Algorithm 3). These steps are performed as long as there are *changes* to the schema tree. There are two types of changes: new internal nodes are added to the schema tree ST (i.e., new grouping of fields) or labels are assigned to the nodes in ST.

Algorithm 2 TreeIntegrator(FT, TT)

Input: Tree of fields FT, tree of labels TT
Output: Schema tree ST

```
ST = FT
prune candidate label sets of leaves in FT using TT;
discard text tokens from TT whose semantic scopes contain no fields;
assign text tokens to fields explicitly specified in HTML;
while no more changes to ST do
  for all sets of sibling nodes in ST do
    assign labels to leaves according to Rules 5 and 7;
  end for
  ST = doMerge(TT.root,ST);
end while
ST = postProcess(ST, TT);
```

The schema tree may require additional adjustments. In a final step, `postProcess`, the algorithm may further adjust the structure of the tree and may assign additional text tokens to nodes. This step handles those fields which should not form a group but are incorrectly placed within a group in the tree of fields (described in Section 5.2). On some interface, this is caused by fields which are grouped horizontally by its designer for the sole purpose of optimizing the space occupied by the interface on the Web page. We observe that such a group on this kind of interfaces has the following characteristics: it contains only two fields, does not have labels and the interface is entirely constructed out of such groups. The parent of these fields is removed from the schema tree ST and the grandparent becomes their parent.

Procedure `doMerge` recursively integrates the tree of text tokens into the tree of fields. The target tree is the tree of fields and the source tree is the tree of labels. The initial schema tree is the tree of fields. The output is the tree resulted from the integration of the two trees. In a preorder traversal of the tree of text tokens TT the algorithm performs the following operations. A node v with text token F in the tree of text tokens is mapped into a node lca of the schema tree. The node lca represents the lowest common ancestor of the set of fields contained in the semantic scope of the text token F (procedures `mapFieldList` and `getLCA`). Three cases are handled based on the relationship between the set of descendent leaves D_{lca} of the node lca and the list of fields, S_F in the semantic scope of the token F represented by node v. (i) If $D_{lca} = S_F$, the text token F is assigned as a candidate label to lca. (ii) If $S_F \subset D_{lca}$ and the fields in S_F are children of lca, then a new node is inserted in the schema tree as a child of lca. The children of lca in S_F become children of the new node. (iii) If $S_F \subset D_{lca}$ and and the fields in S_F are *not* all children of lca, then descendent leaves of lca in S_F are reorganized. A new internal node is created and the set of fields in S_F become its

Algorithm 3 doMerge(node,FT)

Input: the root of the tree of text tokens, schema tree ST
Output: integrated schema tree ST

```
V = node.getChildren;
for all v ∈ V do
  F = mapFieldList(v);
  lca = getLCA(F, ST);
  D_lca = getDescendentLeafSet(lca);
  if D_lca = F then
    v candidate label for lca;
  else
    if all f ∈ F children of lca then
      lca.appendChild(F);
    else
      w = lca.createChildInOrder(F);
      w.addChildren(F);
    end if
  end if
  doMerge(v,FT);
end for
```

children. The new node is inserted among the children of lca such that the semantic order of the first field in S_F is preserved. Note that this reorganization of the leaves may be inconsistent with the initial semantic order of the fields. This occurs when the groups constructed from the semantic order of fields are in contradiction with the groups suggested by the semantic scopes of text tokens. As a rule of thumb, whenever such a contradiction occurs, the semantic scope of the label provides the intended way the fields should be grouped.

The result of integrating the two trees in our running example is shown in Figure 1, on the right.

Note that the schema tree produced by the algorithm `TreeIntegrator` has the following property. If label l_v is assigned to node v and label l_w is assigned to node w with v an ancestor of w then label l_v is an ancestor of l_w in the tree of text tokens TT. This result guarantees that Rules 4 and 8 are satisfied by the labels assigned to the groups and subgroups of fields in the final schema tree of a query interface. A formal proof is omitted due to space constraints.

6. EXPERIMENTAL EVALUATION

We have implemented an operational prototype extraction system. We have conducted extensive experiments over several domains of Web sources to evaluate our approach. Our study intends to evaluate whether our solution can be used for arbitrary application domains, ranging from simple query interfaces to nested, multi-field forms, and whether it substantially improves on previous work.

6.1 Datasets

Three datasets were considered:

ICQ dataset consists of query interfaces in five domains: airline, automobiles, books, jobs, real estate. Each domain has 20 query interfaces, so in total there are 100 interfaces.

Tel8 is the dataset employed in [24]. It consists of 487 query interfaces, of which we could use only 50% because the others refer to no longer existing Web servers. For these interfaces the browser either crashes or displays no page. Interfaces are from eigth domains: airlines, auto, books, car rentals, hotels, jobs, movies and music records.

	Leaves			Internal Nodes			Depth		
Domain	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**
Airline	10.95	1	18	5.3	1	7	2.5	1	4
Automobiles	4.95	2	10	1.85	1	4	1.4	1	2
Books	5.45	1	12	1.85	1	5	1.45	1	2
Jobs	4.55	1	7	1.45	1	5	1.25	1	2
Real Estate	6.95	1	18	3.05	1	10	1.8	1	4
Overall	**6.57**	**1**	**18**	**3.75**	**1**	**10**	**1.68**	**1**	**4**

Figure 10: Characteristics of ICQ dataset.

	Leaves			Internal Nodes			Depth		
Domain	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**
Books	4.73	1	22	1.73	1	8	1.41	1	3
Electronics	5.71	1	14	1.79	1	4	1.36	1	2
Games	5.56	2	18	1.78	1	7	1.22	1	2
Movies	5.08	1	24	1.4	1	7	1.16	1	3
Music	4.12	1	11	1.2	1	3	1.16	1	2
Toys	5.13	3	6	1.25	1	3	1.13	1	2
Watches	6.85	1	25	1.77	1	5	1.38	1	2
Overall	**5.08**	**1**	**11**	**1.57**	**1**	**8**	**1.29**	**1**	**3**

Figure 11: Characteristics of WISE dataset.

The ICQ and Tel8 datasets are publicly available[1].

WISE is the dataset used in [13]. It consists of 147 interfaces, out of these we were able to work with 134 due to the same reason as mentioned for the Tel8 dataset. Interfaces come from seven domains: books, electronics, games. movies, music, toys and watches. Overall, we evaluated our approach on more than 500 web interfaces from 15 distinct domains. Interfaces differ largely in terms of number of fields, depth of nesting, layout etc. Figure 10 and Figure 11 give some figures on ICQ and WISE datasets.

6.2 Performance Metrics

We evaluated our algorithm according to different metrics, concentrating on the difficult tasks in interface extraction. In each metric, we compared a particular type of information obtained automatically from our method with the "true" information as defined in a gold standard. How these gold standards were obtained is described below. The metrics are:

Leaf Labeling: The problem of finding the right label for leaves (fields) is difficult because labels are not explicitly assigned to fields (as highlighted previously). We compute the ratio of the number of correctly labeled fields to the total number of fields (accuracy).

Schema Tree Structure: This measure aims to quantify the distance between the structure of the extracted schema tree and that of the ideal schema tree while ignoring the labels assigned to the nodes. It shows how well the groups of fields are identified and how accurate the semantic order of fields is obtained. We use as measure the *structural tree edit distance* which is the minimum number of operations (insert, delete) to convert one tree into another. The precision per interface is $P_s = (N_e - D_s)/N_e$, where N_e is the number of nodes in the extracted tree and D_s is the structural tree edit distance. The recall per interface is $R_s = (N_e - D_s)/N_i$, where N_i refers to the number of nodes in the gold standard tree. Finally, we compute F-score $F_s = 2P_sR_s/(P_s + R_s)$.

Overall Metric: This measure aims to quantify how well the trees along with their labels are extracted. Therefore, we use the *tree edit distance*, i.e., the minimum number of operations (insert, delete and relabeling) to convert one

[1] see the University of Illinois at Urbana Champaign Web Repository `http://metaquerier.cs.uiuc.edu/repository/`

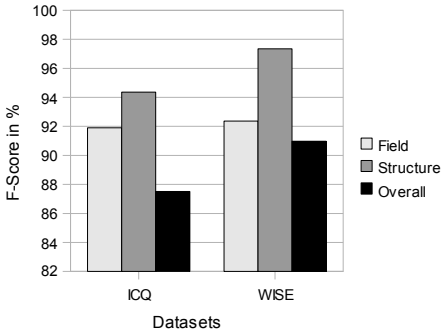

Figure 12: Experimental results.

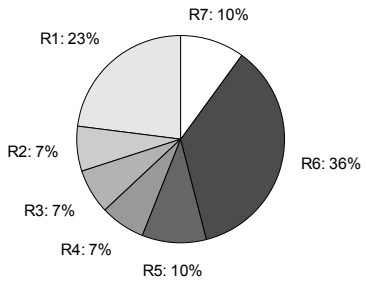

Figure 13: Relevance of commonsense rules.

tree into another. Precision, recall and F-measure are defined as above.

For the latter two metrics, we shall report the overall precision, recall and F-score obtained by averaging over all interfaces in a dataset. Note that neither metric counts the retrieval of domains, data types and default values, because these are easily retrieved from the HTML code.

Gold standard. Our extracted results are compared against gold standard interface representations. The ICQ dataset already provides the gold standard. For the WISE dataset, we manually constructed the gold standard. For the Tel8 dataset, we only compare the extracted labels for fields and groups (called *conditions* in Tel8) as specified in its gold standard. Note that our algorithm actually identifies many more important groupings, but these cannot be evaluated directly on the Tel8 gold standard.

6.3 Evaluation of the Algorithm

Figure 12 summarizes the results of our experimental study on the ICQ and WISE dataset. From left to right, the bars represent the accuracy of retrieving the right labels for fields, the F-score for the evaluation of the structure of the schema trees and the overall F-score. The identification of labels for leaves reaches 92% accuracy in both datasets. We also achieve very high accuracy for the structural extraction (>94%). Results for the overall measure are slightly worse (87.5% for ICQ and 91% for WISE), which indicates that sometimes labels for internal nodes are missed. The frequency of this problem correlates with the accuracy of the structural extraction. For instance, if the descendent nodes of an internal node in the schema tree are not properly identified, the label for that internal node may be missed. Thus, the differences in the performance between WISE and ICQ can be boiled down to the differences in the structural accuracy which again depends on the complexity of the schemas (see Figure 10 and 11).

We also ran the experiment for the Tel8 dataset. The overall measure is 87.5%. Note however, that this number gives only a partial impression of the performance of our algorithm on this dataset because the gold standard offers less information than extracted by our algorithm.

Evaluation of the Commonsense Rules. The pie chart in Figure 13 shows the influence of the different rules when being run on the ICQ set. Each slice represents the ratio of the total number of times the rule was used to produce labels over the total number of usages of any rule. Obviously, all rules were necessary to achieve a proper result. Except for Rules 1 and 6, the influence of the rules is roughly the same, which suggests that this dataset contains a balanced number of flat and hierarchical interfaces.

Efficiency of the Algorithm. We also evaluated the efficiency of the system. The entire system has two parts, rendering/loading the Web page and the extraction of the query interface. While the rendering takes on average 4 seconds per Web page, the extraction itself is efficient; it takes on average 1 second per user interface.

6.4 Comparison with Other Systems

We compared the performance of our method with that of the WISE Extractor[13] and the system from [24] (abb. as BEP: best effort parser)). Since the structures extracted by the three algorithms can not be compared directly, we resorted to a simpler evaluation by computing only the accuracy for field and internal node extractions. This measure was also used in [13, 24]. Note that it disregards the ability of our tool to extract deeply nested structures.

WISE extractor. We thank the authors of [13] for sharing code and dataset. We run WISE Extractor and our tool on all three datasets. Figure 14 shows average results for all datasets. Our tool on average is slightly better on the WISE dataset, but much better on the other two sets: 90% vs. 80% for ICQ, 88% vs. 82% for Tel8. Over all three datasets, the difference in accuracy between our tool and WISE extractor is 6.5% on average. Our explanation for the poorer performance of WISE on Tel8 and ICQ is that these datasets have more complicated interfaces (see Figure 10 and 11) than those in WISE dataset. For instance, the airline domain has on average 5.3 internal nodes whereas none of the domains in the WISE dataset exceeds on average 2 internal nodes. Furthermore, ICQ and Tel8 were collected over several years. During this period, the HTML language itself has evolved, which poses problems to parsing-based techniques as the WISE Extractor is a parsing based technique. The tool is also unable to distinguish between visible and invisible fields on a query interface. In comparison, our visual technique is more resilient to language evolution and does properly handle invisible elements.

BEP. We were not able to obtain the system described in [24]. Consequently, we can only compare to the results published in this paper, and only on the Tel8 dataset. We obtain an overall accuracy of 88% whereas [24] reports an accuracy of 85% [24]. Thus, although BEP uses a set of rules that were directly derived from the dataset it was evaluated on, our system - using a small and generic set of extraction rules - outperforms this method on its own dataset.

7. CONCLUSION

We presented a technique for extracting hierarchical schema trees from Deep Web interfaces. This representation is richer and thus easier to be used for Deep Web integration than

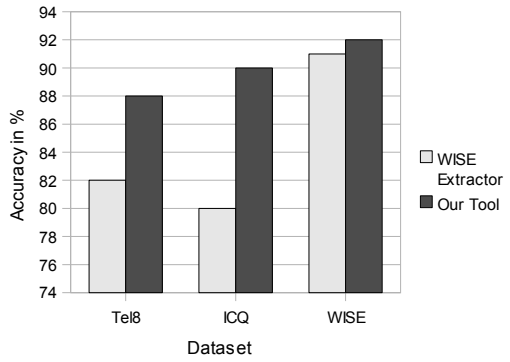

Figure 14: Experimental comparison.

previous, flat models. Our extraction technique is based on a small set of general design rules which, together with a proper exploitation of visual layout of HTML pages, allow to extract schema trees with high accuracy. We showed experimentally that our method outperforms previous approaches even if its capabilities for extracting structure are disregarded.

Overall, we reach very high accuracy values over a wide range of interfaces and domains. However, one can still strive for improvements. We manually investigated those interfaces where we performed poorly and found a number of problematic situations. First, there are interfaces whose labels are encoded in images. These interfaces account for about 2% of the investigated query interfaces. Although there are tools to extract text from an image (e.g. OCR) our current implementation does not include them. About 1% of the interfaces have labels that are aligned to the center and not to the left. The semantic scopes of this kind of labels are wrongly computed. A number of query interfaces (less than 1%) have a different semantic order than the fill-in order which our algorithm expects. Many of the remaining errors in the extraction process root in the preprocessing step of the algorithm. For instance, during tokenization a single text token obtained from the browser may contain the labels for multiple fields.

Acknowledgements: This work is supported in part by the United States National Science Foundation (NSF), grant IIS-0414939, and by the German Academic Exchange Service (DAAD), grant D0841421.

The authors would also like to express their gratitude to the annonymous reviewers for providing helpful sugestions.

8. REFERENCES

[1] Luciano Barbosa and Juliana Freire. An adaptive crawler for locating hidden web entry points. In *WWW*, 2007.

[2] Luciano Barbosa and Juliana Freire. Combining classifiers to identify online databases. In *WWW*, 2007.

[3] Andre Bergholz and Boris Chidlovskii. Crawling for domain-specific hidden web resources. In *WISE*, 2003.

[4] Alexander Bilke and Felix Naumann. Schema matching using duplicates. In *ICDE*, 2005.

[5] Michael J. Cafarella, Edward Chang, Andrew Fikes, Alon Y. Halevy, Wilson C. Hsieh, Alberto Lerner, Jayant Madhavan, and S. Muthukrishnan. Data management projects at google. *SIGMOD Record*, 37(1), 2008.

[6] Kevin Chen-Chuan Chang, Bin He, Chengkai Li, Mitesh Patel, and Zhen Zhang. Structured databases on the web: observations and implications. *SIGMOD Rec.*, 33(3), 2004.

[7] Hong Hai Do and Erhard Rahm. Coma - a system for flexible combination of schema matching approaches. In *VLDB*, 2002.

[8] Eduard Dragut, Wensheng Wu, A. Prasad Sistla, Clement T. Yu, and Weiyi Meng. Merging source query interfaces on web databases. In *ICDE*, 2006.

[9] Eduard C. Dragut, Clement Yu, and Weiyi Meng. Meaningful labeling of integrated query interfaces. In *VLDB*, 2006.

[10] Bin He, Kevin Chen-Chuan Chang, and Jiawei Han. Discovering complex matchings across web query interfaces: a correlation mining approach. In *KDD*, 2004.

[11] Bin He, Zhen Zhang, and Kevin Chen-Chuan Chang. Metaquerier: querying structured web sources on-the-fly. In *SIGMOD*, 2005.

[12] Hai He, Weiyi Meng, Clement Yu, and Zonghuan Wu. Automatic integration of web search interfaces with wise-integrator. *The VLDB Journal*, 13(3), 2004.

[13] Hai He, Weiyi Meng, Clement T. Yu, and Zonghuan Wu. Constructing interface schemas for search interfaces of web databases. In *WISE*, 2005.

[14] Oliver Kaljuvee, Orkut Buyukkokten, Hector Garcia-Molina, and Andreas Paepcke. Efficient web form entry on pdas. In *WWW*, 2001.

[15] Nicholas Kushmerick. Learning to invoke web forms. In *CoopIS, DOA, and ODBASE 2003*, 2003.

[16] Jayant Madhavan, Shawn R. Jeffery, Shirley Cohen, Xin Dong, David Co, Cong Yu, and Alon Halevy. Web-scale data integration: You can only afford to pay as you go. In *CIDR*, 2007.

[17] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*, 2002.

[18] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997.

[19] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. In *VLDB*, 2001.

[20] Jiying Wang and Fred H. Lochovsky. Data extraction and label assignment for web databases. In *WWW*, 2003.

[21] Jiying Wang, Ji-Rong Wen, Fred Lochovsky, and Wei-Ying Ma. Instance-based schema matching for web databases by domain-specific query probing. In *VLDB*, 2004.

[22] Ping Wu, Ji-Rong Wen, Huan Liu, and Wei-Ying Ma. Query selection techniques for efficient crawling of structured web sources. In *ICDE*, 2006.

[23] Wensheng Wu, Clement Yu, AnHai Doan, and Weiyi Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *SIGMOD*, 2004.

[24] Zhen Zhang, Bin He, and Kevin Chen-Chuan Chang. Understanding web query interfaces: best-effort parsing with hidden syntax. In *SIGMOD*, 2004.