Modeling and Extracting Deep-Web Query Interfaces

Wensheng Wu, AnHai Doan, Clement Yu, and Weiyi Meng

Abstract. Interface modeling & extraction is a fundamental step in building a uniform query interface to a multitude of databases on the Web. Existing solutions are limited in that they assume interfaces are flat and thus ignore the inherent structure of interfaces, which then seriously hampers the effectiveness of interface integration. To address this limitation, in this chapter, we model an interface with a hierarchical schema (e.g., an ordered-tree of attributes). We describe ExQ, a *novel* schema extraction system with two distinct features. First, ExQ discovers the structure of an interface based on its visual representation via spatial clustering. Second, ExQ annotates the discovered schema with labels from the interface by imitating the human-annotation process. ExQ has been extensively evaluated with real-world query interfaces in five different domains and the results show that ExQ achieves above 90% accuracy rate in both structure discovery & schema annotation tasks.

1 Introduction

Besides the billions of Web pages indexed by search engines, the Web also contains a large number of databases whose contents are only accessible through query interfaces and out of reach of conventional search engines [5]. These databases form

Wensheng Wu
University of North Carolina at Charlotte, Charlotte, NC 28223
e-mail: w.wu@uncc.edu

AnHai Doan
University of Wisconsin at Madison, Madison, WI 53706
e-mail: anhai@cs.wisc.edu

Clement Yu
University of Illinois at Chicago, Chicago, IL 60607
e-mail: yu@cs.uic.edu

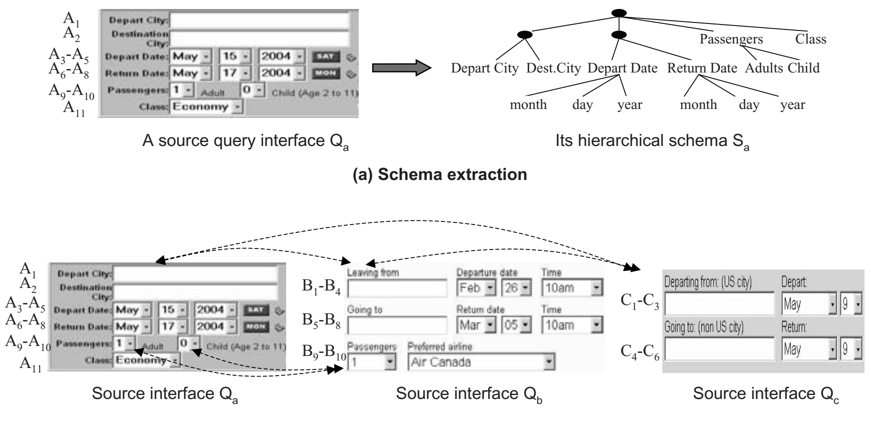
Weiyi Meng
Binghamton University, Binghamton, NY 13902
e-mail: meng@cs.binghamton.edu

the *Deep-Web*, and they are the Deep-Web data sources [4]. The Deep-Web was estimated to be at least 500 times larger than the surface Web [4], and it continues to grow at a phenomenal rate [18].

The Deep-Web covers a great variety of subject areas, ranging from business, government, education, to entertainment [18, 4]. For any domain of interest, there may be hundreds or even thousands of Web databases, e.g., book databases from Amazon, Barnes & Noble, and many other online book stores. These databases contain high-quality, structured contents, but may vary greatly in their content coverage & query capability. As a result, to find the desired information, users often need to interact with multiple sources, understand their query syntaxes, formulate separate queries, and compile query results from different sources. This can be an extremely time-consuming and labor-intensive process.

The *search* problem on the Deep-Web has received great attention from both academia and industry in the past few years. Early work includes [8, 21, 6, 15, 17, 20] in the database and AI communities. Recent efforts include [18, 9, 3, 10, 29, 24, 2, 19, 27], and recent industrial activities involve many startups, such as *Transformic*, *Glenbrook Networks*, and *Webscalers*, as well as large Internet companies, such as *Google* and *Yahoo* [18]. Given a domain of interest, an important focus of the above efforts is to build a *uniform query interface* to the data sources in the domain, thereby making access to the individual sources transparent to users.

To build such a uniform query interface, a domain developer often must solve the *interface matching* problem: given a large set of sources in a domain, find semantic correspondences, called *mappings*, between the attributes of the query interfaces of the sources [9, 29, 24]. Once the interfaces have been matched, the semantic matches are employed to construct the uniform query interface [27], to translate queries formulated over this interface to those over the interfaces of the data sources, and to translate the results obtained from the sources into a format that conform to the



(a) Schema extraction

(b) Schema matching

Fig. 1 Schema extraction & matching

uniform query interface [24]. Interface matching therefore plays an important role in the integration of Deep-Web data sources (regardless of whether the integration is virtual or materialized [26]).

Typically, interface matching involves three major tasks: *interface modeling*, *schema extraction*, and *schema matching*.

Interface Modeling: A query interface typically consists of multiple attributes. For example, there are 11 attributes on the interface Q_a shown in Figure 1.a (left). An attribute may be denoted by a label, e.g., attribute A_1 on Q has a label Depart City. An attribute may also have a set of values. For example, attribute A_{11} (Class) on Q_a has values: {Economy, First Class, Business}.

Related attributes are placed near each other on the query interface, forming a group; and closely related attribute groups may be further grouped into a super-group. For example, attributes A_9 (Adult) and A_{10} (Child) on Q_a form a group with a group label Passengers. In addition, attributes and attribute groups are intuitively ordered. For example, A_9 is placed before A_{10}. As a result, query interface may be best modeled by a *hierarchical schema* such as ordered tree. For example, Figure 1.a (right) shows such a schema S_a for the interface Q_a, where leaves and internal nodes in S_a correspond to attributes and attribute groups on Q_a respectively.

Schema Extraction: A query interface is typically rendered from a HTML form script. The script is largely concerned with the visual representation of the attributes (e.g., using a text-input field to display attribute Depart City on Q_a) and the placement of attributes and labels on the interface. It typically does not explicitly specify the attribute-label and attribute-attribute relationships on the interface. Therefore, such relationships and thus the structural aspect of the interface need to be inferred from its visual representation via schema extraction. For example, given Q_a as the input, a schema extraction algorithm might produce a schema like S_a as the output.

Schema Matching: Given a set of interface schemas extracted from source query interfaces, we need to accurately determine the mappings of attributes from different interfaces. There may be two types of mappings: simple and complex. A simple mapping is a 1:1 semantic correspondence between two attributes. For example, consider query interfaces shown in Figure 1.b. An example of 1:1 mapping is attribute A_1 (Depart city) of interface Q_a matching B_1 (Leaving from) of interface Q_b. Mappings may also be complex, e.g., 1-m mappings. A 1-m mapping is a mapping where an attribute on one interface semantically corresponds to multiple attributes on another interface. For example, attribute B_9 (Passengers) on Q_b matches both A_9 (Adult) and A_{10} (Child) on Q_a.

In this chapter, we consider the problem of *interface modeling* & *schema extraction*. Schema matching is addressed in our early work [28, 29]. While there have been many research efforts on modeling & extracting Deep-Web query interfaces, almost all existing solutions assume that query interfaces are flat, and thus largely ignore the inherent ordering and grouping relationships among the attributes on the interfaces. For example, these solutions would discover a flat schema, which contains simply a set of attributes {Depart City, Destination City, ...} for the interface

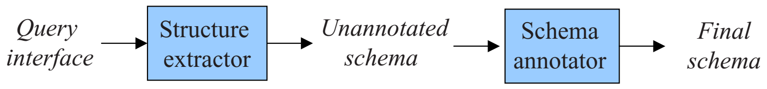


Fig. 2 The ExQ architecture

Q_a (Figure 1.a). Such a schema fails to capture the *structural* aspect of the interface, which makes it very difficult to infer the *semantic* relationships among the attributes.

To address this challenge, we propose to model an interface with a *hierarchical schema* such as an ordered tree. We show that such a hierarchical schema can capture the semantics of the interface more precisely. In [29], we showed that the captured semantics can be exploited to help find the semantic matches & resolve ambiguous match candidates among the attributes from *different* interfaces.

In this chapter, we describe ExQ, an interface modeling & extraction system. Figure 2 shows the architecture of ExQ. ExQ takes as input a query interface and outputs a schema of the interface, represented as a labeled, order tree of attributes. It consists of two major components: *structure extractor*, which takes the interface and produces an unannotated ordered-tree schema of the interface; and *schema annotator*, which then assigns labels from the interface to the nodes in the schema tree, producing the final schema.

In developing ExQ, we make the following contributions:

- A *novel* spatial clustering-based algorithm to discover the structure of the interface based on its *visual* representation.
- A *novel* label attachment algorithm to infer the labels for *both* attributes & attribute groups, based on several observations on the human-annotation process.
- *Extensive* real-world evaluation of ExQ, achieving above 90% accuracy rate in both structure discovery & schema annotation tasks.

The rest of the chapter is organized as follows. Section 2 discusses related work. Section 3 describes hierarchical modeling of query interfaces. Sections 4 & 5 describe ExQ's structure extractor and schema annotator in detail. Section 6 presents experimental results and Section 7 concludes the chapter.

2 Related Work

We discuss related work from several perspectives.

Structure Discovery & Label Attachment: As discussed earlier, almost all existing solutions assume that the interface is flat, i.e., containing a set of attributes. Thus, the focus is largely on attaching right labels to the attributes on the interface.

[12] proposes several algorithms for attaching labels to the attributes on the interface. The algorithms are largely based on matching labels with the names of the attributes (as specified in the HTML script). Group labeling is also considered, but limited to groups of check boxes or radio buttons. Further, their accuracy rate (80%) is much lower than ours.

[21] exploits spatial location, font size, and font style of labels for label attachment. In contrast, our approach is mainly based on annotation patterns (described in Section 5). [21] does not consider group labeling. In addition, our accuracy rate (95.5% in F-measure) on attribute labeling is higher than that in [21], and is achieved on a data set which is much more complex than that in [21].

[30] focuses on extracting query conditions from interfaces. The query conditions may indicate a restricted form of attribute groups, e.g., a text input box for author may be associated with a group of radio buttons, indicating if the required input is first name, last name, or full name. Such specific grouping of attributes may be handled by a grouping pattern by our structure extraction algorithm. Besides, our accuracy rate in attribute grouping (92.3%) is much higher than that in [30] (85%).

[11] proposes an approach to extracting attributes and their labels from the interfaces based on layout expressions. Its accuracy rate is comparable to ours. But similar to other existing solutions, it also assumes that interfaces are "flat" and therefore does not extract grouping relationships among the attributes on the interface.

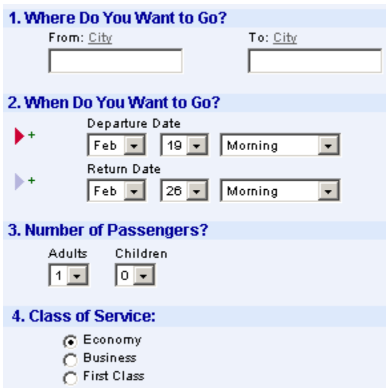
Wrapper Construction: Wrapper construction [15, 16] studies the problem of extracting structured contents from semi-structured documents (such as HTML pages). Therefore, the problem of extracting schemas from query interfaces is closely related to wrapper construction. But wrapper construction largely focuses on discovering presentation patterns (e.g., special HTML tags or tag paths) of the contents from a set of training examples, and then employs the patterns to extract additional contents from similar pages (e.g., pages generated by the same Web site). On the other hand, our work is *specifically* targeted at HTML forms. We seek an *automatic* solution to infer the relationships among interface attributes and associate labels with the attributes, by exploiting their presentation patterns and spatial layout.

Interface Integration: The accuracy of schema extraction is critical to the next two steps in interface integration, namely schema matching & schema merging. There has been a large body of work especially on schema matching (see [22] for an excellent survey). The problem of matching interface schemas is addressed in [9, 10, 29]. In particular, [29] shows that the ordering, sibling, and grouping relationships of attributes can be utilized to effectively discover both 1:1 and complex mappings among interface attributes.

[10, 27] propose solutions to merging interface schemas. In particular, [10] orders the attributes on the unified interface by observing how the attributes are ordered on the source query interfaces. [27] proposes an optimization framework to interface merging, where each source interface expresses constraints on how attributes should be ordered and grouped, and a desired (i.e., intuitive & user-friendly) unified interface is one that maximally satisfies these constraints.

3 Modeling Query Interfaces

In this section, we first describe query interfaces, and show how prior work has modeled such interface with a flat set of attributes and how we model it with a tree of attributes.



1. Where Do You Want to Go?

From: City To: City

2. When Do You Want to Go?

Departure Date: Feb 19 Morning

Return Date: Feb 26 Morning

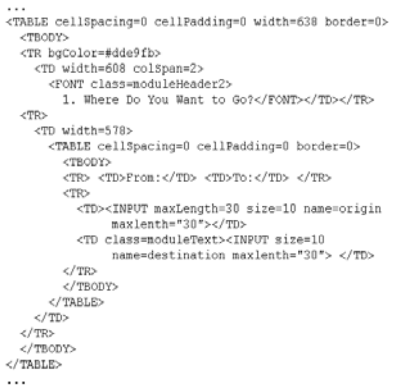
3. Number of Passengers?

Adults: 1 Children: 0

4. Class of Service:

[✓ Economy] Business First Class

(a) An airfare query interface Q



```
...
<TABLE cellSpacing=0 cellPadding=0 width=638 border=0>
  <TBODY>
  <TR bgColor=#dde9fb>
    <TD width=608 colSpan=2>
      <FONT class=moduleHeader2>
        1. Where Do You Want to Go?</FONT></TD></TR>
  <TR>
    <TD width=578>
      <TABLE cellSpacing=0 cellPadding=0 border=0>
        <TBODY>
        <TR> <TD>From:</TD> <TD>To:</TD> </TR>
        <TR>
          <TD><INPUT maxLength=30 size=10 name=origin
               maxlenth="30"></TD>
          <TD class=moduleText><INPUT size=10
               name=destination maxlenth="30"> </TD>
        </TR>
        </TBODY>
      </TABLE>
    </TD>
  </TR>
  </TBODY>
</TABLE>
...
```

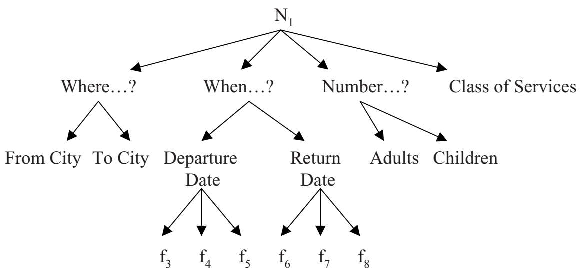
(b) The HTML script of Q

Attribute	Name	Label	Domain
f_1	origin	From: City	{s|s is any string}
f_2	destination	To: City	{s|s is any string}
f_3	departureMonth	""	{Jan, Feb, …, Dec}
f_4	departureDay	""	{1, 2, …, 31}
f_5	departureTime	""	{1am, …, 12pm}
f_6	returnMonth	""	{Jan, Feb, …, Dec}
f_7	returnDay	""	{1, 2, …, 31}
f_8	returnTime	""	{1am, …, 12pm}
f_9	numAdultPassengers	Adults	{1, 2, …, 6}
f_{10}	numChildPassengers	Children	{0, 1, …, 5}
f_{11}	cabinClass	Class of Services	{Economy, …, Business}

(c) The attributes on Q

{from city, to city, …, class of services}

(d) A flat schema of Q



(e) A hierarchical schema of Q

Fig. 3 A query interface, its HTML script, attributes, and schemas

3.1 Flat vs. Hierarchical Modeling of Query Interfaces

Query interfaces are typically written in HTML forms. For example, Figure 3(a) shows a query interface Q in the airfare domain and Figure 3(b) shows the HTML form script of Q.

A query interface can be modeled using multiple attributes. For example, Q contains 11 attributes whose details are shown in Figure 3(c). Note that the attributes are numbered in the order of their appearance (left-right, top-down) on the interface. Each attribute consists of three components: label, (internal) name, and domain.

- **Label:** The label of an attribute is a piece of text on the query interface, which denotes the meaning of the attribute to the user. For example, the first attribute on Q (i.e., f_1) has a label From: City.
- **Name:** The name of an attribute is the internal name of the attribute given in the HTML script for the identification purpose. For example, the name of attribute f_1 is origin.
- **Domain:** The domain of an attribute is a set of values the attribute may take. For example, the domain of the attribute f_9 (with label Adults) on Q is $\{1,2,...,6\}$.

This chapter focuses on exploiting presentation patterns and spatial properties of labels in schema extraction. But note that attribute names and domains may also be useful [12] and it would be interesting to consider combining all these evidences.

Note that an attribute may be represented in a variety of ways on the query interface: (1) an *input field* (e.g., attribute f_1 on Q), where the user may enter any suitable value; (2) a *selection list* (e.g., attribute f_3), where the user may only select from a list of pre-defined choices; (3) a *radio-button group* (e.g., attribute f_{11}), where each button in the group provides an exclusive choice, and the domain of the attribute is the set of all choices while the name of the attribute is taken to be the name of the radio button group; and (4) a *checkbox group*, which is similar to a group of radio buttons except that here the user may select more than one choice at a time.

Note also that label is visible to the user while name is not. As a consequence, words in the label are usually ordinary words which can be understood semantically, while words in the name are often concatenated or abbreviated. Nevertheless, we found that the name of an attribute often can be very informative, and particularly useful when the attribute does not have a label.

Current works represent a query interface with a flat set of attributes, as defined above. For example, Figure 3(d) shows such flat schema of Q. But in fact, as Q shows, closely related attributes (e.g., f_1 and f_2, both on the location of the flight) may be grouped together. Furthermore, attributes and attribute groups may be intuitively ordered (e.g., f_1, for origin, is placed before f_2, for destination). As a result, the query interface has a much richer structure. Such a structure conveys domain knowledge and may be exploited for the effective integration of interfaces.

3.2 Hierarchical Modeling of Query Interfaces

To capture both the grouping and ordering relationships of attributes on a query interface, we model query interface with a hierarchical schema. Figure 3(e) shows an example of such hierarchical modeling, which is technically an ordered tree. A leaf node in the tree corresponds to an attribute on the interface. An internal node corresponds to a group or a super-group of attributes on the interface. Nodes with the same parent are sibling nodes. Sibling nodes are ordered by the sequence of their corresponding attributes or attribute groups (if they are internal nodes) appearing on the interface.

Note that nodes are annotated with the labels of their corresponding attributes or attribute groups. If a node does not have a label, its ID is shown instead, where N_i's represent internal nodes and f_j's leaf nodes. In the rest of the chapter, we may also call the nodes of the tree as the elements of the schema.

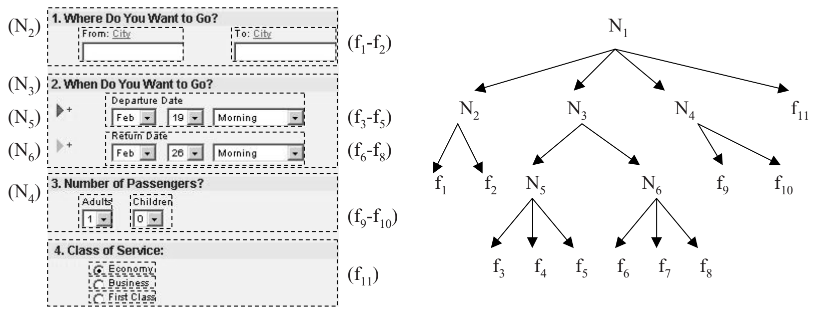
In [29], we show that the ordering relationships of the attributes may be exploited to resolve the ambiguous 1-1 matches among attributes and the grouping relationships of the attributes may be exploited to effectively identify the attributes involved in complex matches.

From now on, when we refer to such modeling, we use the phrase "query interface". Extracting such query interfaces is difficult for the following reasons. First, we must group the attributes appropriately. Next, we must extract the labels and assign them to the right places. We describe how to extract such interfaces next. Note that the names of attributes can be easily obtained from the HTML script of the query interface. If an attribute is represented as a selection list, then its values can also be easily obtained from the option sub-elements of the list. We will describe in Section 5 how to obtain the values of an attribute represented as a radio-button or checkbox group.

4 Extracting the Tree Structure of an Interface

In this section, we describe ExQ's structure extraction algorithm, which is based on spatial clustering. The algorithm takes as the input a query interface (e.g., Q in Figure 4.a) and produces an unannotated ordered-tree schema of the interface (e.g., S_Q in Figure 4.b). In the next section, we will describe ExQ's label attachment algorithm which then assigns the labels from the interface to the nodes in the schema to produce the final schema (e.g., Figure 3(e)).

The main idea of the algorithm is to exploit the spatial relationships (e.g., proximity, alignment, and direction) of attributes on the query interface to effectively discover both the grouping and ordering relationships among the attributes. In the following, we start by describing a basic version of the algorithm which produces a schema tree where each node can have at most two children. We then describe how to remove this limitation via n-way clustering. Next, we discuss how to exploit other information such as lines separators to help determine the grouping relationships of attributes. Finally, we present the complete extraction algorithm.



(a) Query interface Q (b) S_Q, the schema of Q (*before* label attachment)

Fig. 4 Example of extracting tree structure of an interface

4.1 Structure Extraction via Spatial Clustering

The basic version of the extraction algorithm can be regarded as a conventional hierarchical agglomerative clustering algorithm [13] where the objects to be clustered are *attribute blocks*. Attribute block is the spatial representation of an attribute, which can be obtained as follows.

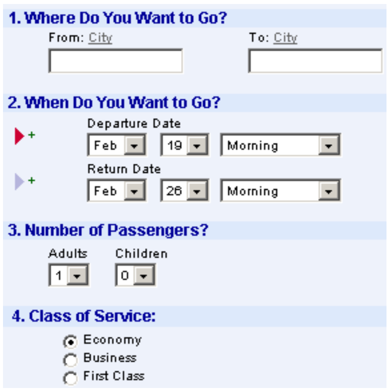
If an attribute f is rendered as an input field (e.g., f_1 and f_2 on interface Q in Figure 5(a)) or a selection list (e.g., $f_3 - f_{10}$ on Q), then f's attribute block is taken to be the smallest rectangular region enclosing the input field or the selection list (see Figure 5(b)). On the other hand, if f is represented as a group of radio buttons (e.g., f_{11}) or checkboxes, then f's attribute block is taken to be the smallest rectangular region enclosing all the radio buttons or checkboxes in the group (see Figure 5(b)).

In the following, we may also denote an attribute block B as $[(x,y),(m_x,m_y)]$, where (x,y) is the top-left corner of B (with x as the x-coordinate and y as the y-coordinate), and (m_x,m_y) is the bottom-right corner of B. Note that the top-left corner of the webpage is considered to be the origin of the coordinate system. The x-coordinate and y-coordinate of an object are its horizontal and vertical offsets to the origin, respectively.

We consider three types of spatial relations between the blocks: *topological relations* (contain, overlap, and disjoint), *direction relations* (above, below, left, and right), and *alignment relations* (top/bottom-aligned and left/right-aligned).

Definition 1 (Topological Relations). *A block U is* contained *in a block V if $\forall p \in U$ (i.e., p is a point in U), we have $p \in V$. A block U* overlaps *with a block V if $\exists p \in U$ such that $p \in V$ and $\exists q \in U$ such that $q \notin V$. A block U is* disjoint *with a block V if $\forall p \in U$, we have $p \notin V$.* □

Definition 2 (Direction Relations). *A block U is* above *(*below*) a block V if $\forall p \in U$ and $\forall q \in V$, we have $p_y > q_y$ ($p_y < q_y$), where p_y denotes p's y-coordinate. A block U is to the* left *(*right*) of a block V if $\forall p \in U$ and $\forall q \in V$, we have $p_x < q_x$ ($p_x > q_x$), where p_x denotes p's x-coordinate.* □


1. Where Do You Want to Go?

From: City

To: City

2. When Do You Want to Go?

Departure Date

Feb 19 Morning

Return Date

Feb 26 Morning

3. Number of Passengers?

Adults 1

Children 0

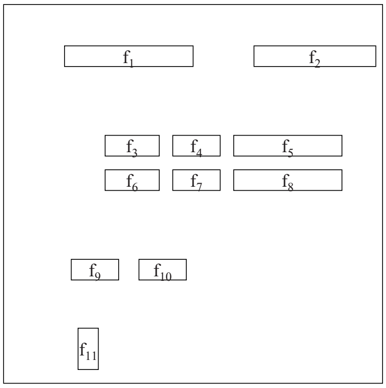
4. Class of Service:

[✓ Economy]

Business

First Class

(a) A query interface Q



(b) Q's attribute blocks

Fig. 5 A query interface and its attribute blocks

Definition 3 (Alignment Relations). *Consider two blocks* $U = [(x,y),(m_x,m_y)]$ *and* $V = [(s,t),(m_s,m_t)]$. *U is* left-aligned *(*right-aligned*) with V if* $x = s$ $(m_x = m_s)$*; and U is* top-aligned *(*bottom-aligned*) with V if* $y = t$ $(m_y = m_t)$*.* □

Distance Function: Intuitively, if two blocks are close to each other and aligned, it is likely that they belong to the same group. Accordingly, we define a distance function between two blocks U and V, denoted as dist(U, V), as follows:

$$\text{dist}(U,V) = \frac{\text{point-dist}(U,V)}{\text{align}(U,V)}. \tag{1}$$

point-dist(U,V) is the minimum Euclidean distance between any two points in U and V. align(U, V) is given by left-align(U,V) + right-align(U,V) + top-align(U,V) + $2*$bottom-align(U,V), where left-align(U, V) takes the value of one if U is left-aligned with V, and zero otherwise. Other alignment functions are similarly defined. If U and V are not aligned, align(U, V) is set to one, i.e., no adjustment will be made to the point-dist. (Alternatively, the denominator in Formula 1 may be changed to align$(U,V)+1$, and then align(U, V) may be set to zero when U and V are not aligned. In our experiments, the original Formula 1 was used.) Note that the weight coefficient for bottom-align is set to two since intuitively two adjacent blocks on the same line are more likely to be closely related.

Based on the above block distance function, the distance between two clusters of attribute blocks can be defined as follows. Consider a cluster C which contains a set of attribute blocks $S = \{B_1, B_2, ..., B_k\}$. We define a block for the cluster C, denoted as B_C, as the *smallest* rectangular region enclosing all the attribute blocks in S. Then, the distance between two clusters C and C' is measured by dist(B_C, $B_{C'}$). We are now ready to describe the clustering algorithm.

Clustering: The algorithm accepts as input a set of attributes on a query interface, where each attribute is represented by its corresponding attribute block as described above; and outputs a hierarchical clustering over the attributes. It starts by putting each attribute in a cluster by itself, and then repeatedly merges two clusters with the minimum distance, until all the attributes are put into a single cluster.

Note that the algorithm produces only *binary* clusterings, i.e., a cluster can only have two sub-clusters. This does not correspond well to the grouping relationships of attributes, since an attribute group may contain more than two sub-groups of attributes. For example, attribute group $\{f_3, f_4, f_5\}$ on the interface Q (Figure 3(a)) contains three attributes. To cope with this, we extend the algorithm to handle n-way clustering.

4.2 N-Way Clustering

The extended algorithm works similarly as the basic one: initially we have a set of clusters, each containing a single attribute, and we repeatedly merge the clusters until we have a single cluster with all the attributes. The key difference is in the merge operation: rather than immediately merging two clusters with the minimum distance, it first expands them into a proximity set of clusters, and then merges all the clusters in the proximity set in a single step.

Specifically, consider two clusters C_1 and C_2, where dist(B_{C_1}, B_{C_2}) = d. A proximity set with respect to C_1 and C_2, denoted as S, can be obtained as follows. To start with, we set $S = \{C_1, C_2\}$. We then use d as the reference distance, and keep growing S by adding a new cluster C_x such that $\exists C_i \in S$, $|\text{dist}(B_{C_x}, B_{C_i}) - d| < \delta * d$, where δ is a small constant (e.g., $\delta = .1$ in our experiment). This growing process stops when no such cluster can be found.

Example 1. Suppose clusters $C_1 = \{f_3\}$, $C_2 = \{f_4\}$, and $C_3 = \{f_5\}$. Then $\{C_1, C_2, C_3\}$ may be a proximity set with respect to C_1 and C_2, since the distance between C_3 and C_2 is very close to the distance between C_1 and C_2. □

4.3 Exploiting Non-distance Information as Constraints

Besides the distance among the attributes, query interfaces may also contain other information such as section titles or horizontal lines, which can be exploited to help determine the grouping relationships of the attributes. For example, the attributes on the interface Q (Figure 3(a)) can also be divided into four sections by the section titles: “1. Where Do You Want to Go?”, “2. When Do You Want to Go?”, so on.

In this section, we describe how to search for these additional information on the query interface and how to exploit them to obtain a partial clustering over the attributes. We will describe in Section 4.4 how to incorporate the obtained partial clustering to constrain the merging process in the spatial clustering algorithm.

Grouping Patterns: To systematically search for these information, we employ a set of grouping patterns. Each grouping pattern specifies a way of grouping some

Table 1 Grouping patterns

Pattern Type	***Examples***
Separator-based	Attributes separated by a set of section labels which are left-aligned and have the same large font. Or attributes separated by a set of left-aligned horizontal lines.
Alignment-based	Multiple rows of attributes which are top and bottom-aligned along the row, and left and right-aligned across the rows.
Indentation-based	A group of attributes which are all indented relative to a label which is located right above and has a large font.

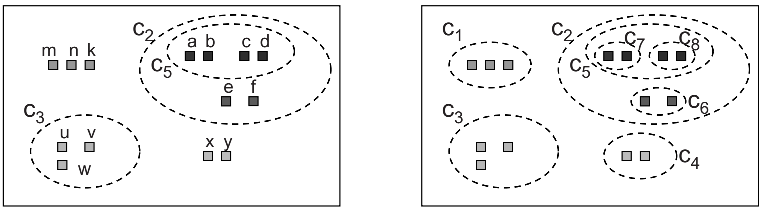
attributes on the interface. These grouping patterns fall into three categories (see Table 1 for examples on each category).

- *Separator-based* patterns, which utilize separators such as section titles and horizontal lines to divide the attributes into groups. Note that the labels which have a larger font (compared to the most common font among the labels) and are located at the left-most of the interface are regarded as section titles.
- *Alignment-based* patterns, which identify groups of attributes which are highly aligned to one another. The discovery is done in a top-down fashion by examining the HTML script to find a set of attributes which might be aligned into rows or columns using a HTML table element. The goal is to overcome the limitation of the spatial clustering algorithm, which proceeds in a bottom-up fashion and might lack a gloal picture on the objects. The discovered patterns are then employed to constrain the clustering process to ensure that the discovered attribute groups are respected and retained in the final results.
- *Indentation-based* patterns, which identify groups of attributes based on their indentation relative to labels. The discovery may also be based on the fonts & colors of the labels. See Table 1 for an example.

Partial Clustering: The above patterns may then be employed to obtain a partial clustering over the attributes on the query interface. Note that such a partial clustering may not be a complete clustering, rather it gives a rough idea of how the final complete clustering should look like. For example, the partial clustering might not indicate the grouping relationships of the attributes within each section on the interface Q in Figure 3(a). Partial clusterings can be formally defined as follows.

Definition 4 (Partial Clustering). *Consider a set of attributes* $S = \{f_1, f_2, ..., f_n\}$. *A* flat *partial clustering* $\mathscr{P}$ *over the attributes in S is a set of subsets of attributes, i.e.,* $\mathscr{P} = \{S_1, ..., S_k\}$, *such that* $S_i \subset S$ *and* $S_i \cap S_j = \emptyset$ *for* $i \neq j$. *Note that* $\mathscr{P}$ *might not have the property that* $\cup_{1 \leq i \leq k} S_i = S$. *Otherwise,* $\mathscr{P}$ *is a* complete *clustering over the attributes in S.*

Such a partial clustering may be further formed over some *of the subsets in* $\mathscr{P}$. *Proceed recursively, the resulted nested clustering is called a* hierarchical *partial clustering over the attributes in S.* □



(a) A partial clustering (b) A complete clustering

Fig. 6 Partial vs. complete clusterings

Since we are only concerned with hierarchical clusterings over the attributes, we will simply use partial clusterings to refer to hierarchical partial clusterings.

Example 2. Figure 6 shows a partial clustering vs. a complete clustering over the same set of attributes, where clusters are represented by dotted ovals. We observe that at the first level, the complete clustering forms four clusters over the attributes, but only two of them (C_2 and C_3) are given in the partial clustering. □

Obtain Partial Clustering: Based on the above discussions, we are now ready to describe PRECLUSTER, a procedure which takes as input a set S of attributes on a query interface Q, and outputs a partial clustering P over the attributes in S. PRECLUSTER proceeds in a top-down fashion. It first finds attributes groups among the attributes in S by applying a set G of grouping patterns. These attribute groups form the top level clusters of the partial clustering. It then recursively finds subgroups among the attributes within each group.

Specifically, PRECLUSTER consists of the following steps. (a) *Pattern matching:* apply the patterns in G on S. Each pattern returns a set of subsets of attributes in S, denoted as $\{S_1,...,S_k\}$, where $S_i \subset S$ and $S_i \cap S_j = \emptyset$ for $i \neq j$. Let G_S be a set of all such subsets given by the patterns in G. If $G_S = \emptyset$, then stop. (b) *Maximization:* from the subsets in G_S, select a set of maximum subsets, denoted as G'_S. A subset $S_x \in G_S$ is a maximum subset if there does not exists $S_y \in G_S$ such that $S_y \subset S_x$. (c) *Recursion:* if there is at least one subset in G'_S which has more than two attributes, recursively apply steps a–b on each such subset in G'_S.

The maximum subsets obtained over the iterations of the above recursive procedure form a top-down partial clustering over the attributes on the interface Q.

So given a partial clustering (e.g., Figure 6.a) over the attributes on a query interface, the goal of the spatial clustering algorithm is in a sense to obtain a complete clustering (e.g., Figure 6.b) which respects the partial clustering. As we will show next, one way of doing this is to use the partial clustering to constrain the merging process of the algorithm.

4.4 The Structure Extraction Algorithm

Figure 7 shows the complete structure extraction algorithm **EXTR**. **EXTR** accepts as input S, a set of attributes on an interface, and outputs $\mathscr{T}$, an unannotated

```
EXTR($S$) → $\mathscr{T}$:
Input: $S$, a set of attributes on an interface
Output: $\mathscr{T}$, an unannotated ordered tree schema
1. Utilize grouping patterns to obtain partial clustering:
    $\mathscr{P} \leftarrow$ PRECLUSTER($S$)
2. Form initial clustering:
    /* $\mathscr{C}$ contains a singleton cluster for each attribute $f \in S$ */
    $\mathscr{C} \leftarrow \{\{f\} \mid f \in S\}$
3. Repeat the following steps until all attributes are in one cluster:
    /* each iteration performs a n-way constrained merging operation */
    a. Obtain clusters to be considered in the current iteration:
        $\mathscr{C}_{\mathscr{P}} \leftarrow$ CONSTRAIN($\mathscr{C}$, $\mathscr{P}$)
    b. Find two clusters $C_1, C_2 \in \mathscr{C}_{\mathscr{P}}$ with the minimum distance
    c. Expand them into a proximity set:
        $X \leftarrow$ OBTAINPROXIMITYSET($C_1, C_2, \mathscr{C}_{\mathscr{P}}$)
    d. Merge clusters in $X$ into a new cluster $C_X$
    e. Evaluate distances of $C_X$ with remaining clusters via Formula 1
4. $\mathscr{H} \leftarrow$ the hierarchical clustering output by step 3
5. Order attributes and attribute groups in $\mathscr{H}$:
    $\mathscr{T} \leftarrow$ ORDER($\mathscr{H}$)
6. Return $\mathscr{T}$
```

Fig. 7 The structure extraction algorithm

ordered-tree schema of the interface. At the high level, **EXTR** is a hierarchical agglomerative n-way clustering algorithm where the merging process is constrained so that it does not violate the partial clustering obtained by PRECLUSTER.

It proceeds as follows. First, it applies PRECLUSTER to obtain a partial clustering $\mathscr{P}$ over the attributes in S. $\mathscr{P}$ is then used to constrain the merging process via the CONSTRAIN function at step 3(a). Given the current clusters in C and the partial clustering $\mathscr{P}$, CONSTRAIN finds a minimum cluster $C_m \in \mathscr{P}$ such that C_m contains a set of clusters in C, denoted as $C_{\mathscr{P}}$. Note that a cluster $C_m \in \mathscr{P}$ is minimum if $\nexists C'_m \in \mathscr{P}$, such that C'_m also contains all clusters in $C_{\mathscr{P}}$ and $C'_m \subset C_m$. If such a minimum cluster $C_m \in \mathscr{P}$ is found, CONSTRAIN returns the corresponding $C_{\mathscr{P}}$ as the output; otherwise, it returns C as $C_{\mathscr{P}}$.

Example 3. Suppose the partial clustering $\mathscr{P}$ is as given in Figure 6.b. Then in the first iteration of the step 4, $C_m = C_5$ and $C_{\mathscr{P}}$ is a set of singleton clusters with the attributes in C_m. □

Then, in the remaining of step 3, only the clusters in $C_{\mathscr{P}}$ are considered. First, two clusters $C_1, C_2 \in C_{\mathscr{P}}$ with the minimum distance are chosen. C_1 and C_2 are then expanded into a proximity set X as described in Section 4.2. Note that X only contains the clusters in $C_{\mathscr{P}}$. Next, the clusters in X are merged into a new cluster C_X. Finally, the distances of C_X with the remaining clusters are evaluated, before the next iteration.

The result of step 3 is a hierarchical clustering $\mathscr{H}$ over the attributes on the interface. $\mathscr{H}$ corresponds to an *unordered* schema tree of the interface. Finally, step 5 orders the nodes in $\mathscr{H}$ to produce an ordered schema tree $\mathscr{T}$ via the ORDER function.

ORDER considers the internal nodes of $\mathscr{H}$ in turn, and for each internal node I, it arranges I's child nodes by the spatial location of their corresponding attributes or attribute groups on the interface. Specifically, suppose I has k children I_1, I_2, ..., I_k. Denote the smallest rectangular box which encloses all the attributes (i.e., leaf nodes) of the subtree rooted at I_i as B_{I_i}. Then, I_i precedes I_j in the ordering, if one of the following two conditions holds: (1) B_{I_i} and B_{I_j} overlaps in the y-direction, and B_{I_i} is to the *left* of B_{I_j}; or (2) B_{I_i} and B_{I_j} does not overlap in the y-direction, and B_{I_i} is above B_{I_j}. Such an ordering corresponds to the intuitive left-right top-down viewing sequence of the attributes on the interface by the users.

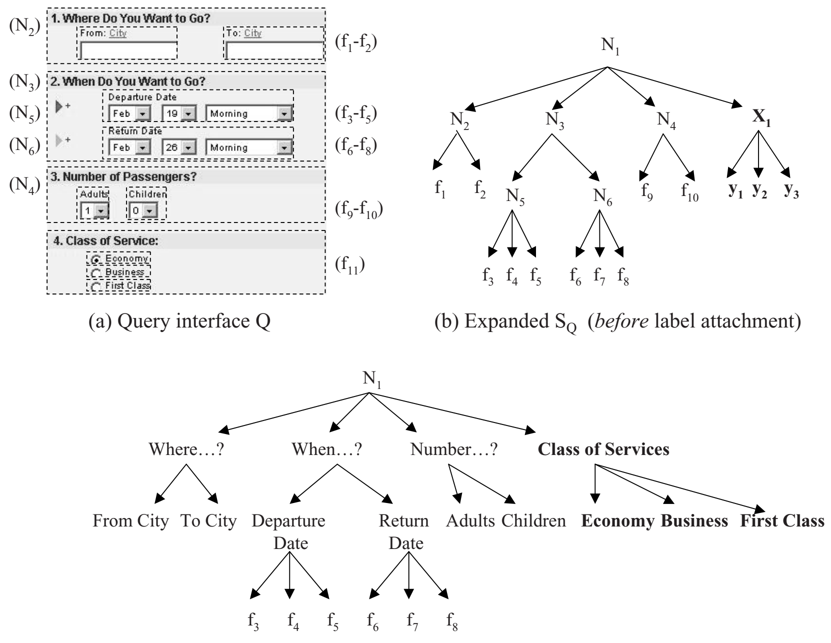
5 Extracting and Attaching the Labels

In Section 4.4, we described a structure extraction algorithm which takes as the input a query interface (e.g., Q in Figure 8.a) and produces an unannotated schema of the interface (e.g., S_Q in Figure 8.b). In this section, we describe ExQ's label attachment algorithm which finds the labels from the interface for the nodes in the schema.

As described earlier, if an attribute is represented as a group of radio buttons or checkboxes, then its values are the labels of the individual radio buttons or checkboxes. In order to also extract these labels, we expand the schema of the interface before label attachment so that every such attribute (e.g., f_{11} on Q) is transformed into an attribute group (e.g., X_1 in Figure 8.b) which contains as many (pseudo) attributes as the number of radio buttons or checkboxes for the original attribute (e.g., y_1, y_2, and y_3). Then, after the label attachment is finished, the pseudo attributes will be removed from the expanded schema (e.g., Figure 8.c) to produce the final schema (e.g., Figure 3(e)). Note that the labels for the pseudo attributes (e.g. Economy, Business, First Class) will become the values of the original attribute and the label of the attribute group (e.g., Class of Service) will become the label of the original attribute.

While there have been some works on label attachment [21], they either assume that query interfaces are flat and thus do not consider the attachment of group labels, or only handle groups of radio buttons and checkboxes (see related work section for more details). Furthermore, the current solutions commonly employ distance-based heuristics where labels are attached to the attributes with the smallest distances. Such heuristics may not work well, especially for group labels. For example, consider the interface snippet in Figure 9, which contains a group of two attributes (one for each selection list). We observe that the group label Passengers is closer to the first attribute than its actual label Adult.

To address these challenges, we take a closer look at the process of annotating attributes and attribute groups on a query interface with labels. For each annotation, we define an *annotation block* as the smallest rectangular region enclosing the



(a) Query interface Q

(b) Expanded S_Q (*before* label attachment)

(c) Expanded S_Q (*after* label attachment)

Fig. 8 Example of label attachment

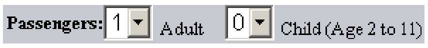

Passengers: 1 Adult 0 Child (Age 2 to 11)

Fig. 9 Examples of label attachment where distance-based methods fail

annotating label and the attribute or attribute group the label annotates. For example, Figure 8.a shows the annotation blocks (represented by dashed rectangular boxes) for the attributes and attribute groups on the interface Q. The following observations can be made. (In addition, a useful observation used in [11] is that labels followed by ":" are more likely to be group labels.)

First, *non-overlapping annotation blocks:* It is unusual that annotation blocks would overlap with each other. In other words, for any two annotation blocks, there may only be two possibilities: either they are disjoint or one is contained within another. This observation can be illustrated using Figure 8.a. For example, consider the annotation block B_{N_1} for the group N_1, which encloses the group label (i.e., 1. Where Do You Want to Go?) and the attributes in the group (i.e., f_1 and f_2). We can observe that B_{N_1} contains the annotation blocks for attributes f_1 and f_2 and does not overlap with any other annotation blocks.

Second, *label positioning:* A group label is usually located either *above* or to the *left* of the group, while an attribute label may also be located to the right of the attribute, but seldom located *below* the attribute. For example, all the group labels are located above the groups on the interface Q in Figure 8.a, and none of the

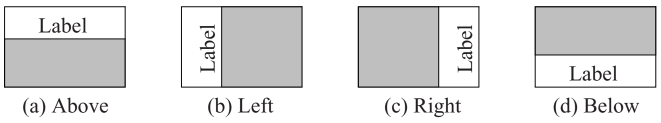


Fig. 10 Positions of the annotating label in an annotation block

attribute labels is located below the attributes. Figure 10 shows possible layouts of an annotation block and the respective positions of the annotating label.

The Label Attachment Algorithm. Motivated by the above observations, we propose a label attachment algorithm ATTACH. ATTACH accepts as input an unannotated schema tree $\mathscr{T}$ for an interface, and a set $\mathscr{L}$ of all labels on the interface. It annotates the nodes in $\mathscr{T}$ with the labels in $\mathscr{L}$, and returns an annotated schema tree $\mathscr{T}^a$. The main ideas of ATTACH are as follows.

- *Bottom-up:* One way of annotating the nodes in a schema tree is to proceed in a bottom-up fashion: we start with the leaf nodes and annotate a node only when all of its child nodes have been annotated. For example, consider interface Q in Figure 8.a. We first find labels for attributes f_1 and f_2 before finding label for group N_1.
- *Group-based:* Rather than annotating nodes in isolation, we may consider the annotation of a node and its sibling nodes (i.e., nodes within the same group) together. Intuitively, knowing that a label is unlikely to be assigned to neighbor nodes helps determine the node which the label should be attached to.

Based on the above ideas, ATTACH considers the groups (i.e., internal nodes) in the schema tree in the *post-order*. For example, the groups in the schema S_Q (Figure 8.b) are considered in this order: N_2, N_5, N_6, N_3, N_4, X_1, N_1. For each group N, it annotates the child nodes of N via ATTACHONE described below. For example, when N = N_2, ATTACHONE annotates attributes f_1 and f_2, and when N = N_1, ATTACHONE annotates N_2, N_3, N_4, and X_1. We now first define several necessary concepts.

Definition 5 (Attribute Set and Block of a Node). *For each node x in a schema tree, we define an attribute set, denoted as $\mathscr{A}_x$, as a set of attributes (i.e., leaf nodes) in the sub-tree rooted at x; and a block, denoted as B_x, as the smallest rectangular region enclosing all the attributes in $\mathscr{A}_x$.* □

ATTACHONE: ATTACHONE accepts as the input a group N and a set L_a of available labels. It assigns some labels from L_a to the child nodes of N and returns the unassigned labels. It proceeds in three major steps: candidate generation, candidate pruning, and match selection. We now describe them in detail.

(1) Candidate generation: For each child node x of N, ATTACHONE determines which labels in L_a may be assigned to x, according to the *non-overlapping annotation areas* observation. Specifically, a label l is regarded as a candidate label for x if the annotation block enclosing the label l and the attributes in $\mathscr{A}_x$, the attribute set of x, does not overlap with any attributes not in $\mathscr{A}_x$ and any other labels in L_a.

For example, label l = From: City (Figure 8.a) is a candidate label for f_1 since the annotation block enclosing l and f_1 does not overlap with any other attributes or labels. On the other hand, l may not be assigned to f_2, since the annotation block enclosing l and f_2 overlaps with attribute f_1 (and also another label To: City).

This step results in an attachment matrix M, whose rows correspond to the child nodes of N, and columns correspond to the labels in L_a. The entry $M[i, j]$ is one if the j-th label is a candidate label for the i-th child node of N, and zero otherwise.

(2) Candidate pruning: This step prunes the candidates in M according to the *label positioning* observation as well as the distances between labels and blocks. The pruned matrix is denoted as M'.

It proceeds as follows. First, all candidate labels for a node x are pruned if the distance between the labels and the node block B_x is larger than a threshold d. Next, if x is an attribute and has a candidate label which is *not* located *below* B_x, then all the labels below B_x are pruned. Finally, if x is an attribute group, then all its candidate labels which are located below or to the right of B_x are pruned.

For example, since attribute f_1 has a candidate label From: City located above it, another candidate label 2. When Do You Want to Go?, located below it, is pruned.

```
ATTACH($\mathscr{T}$, $\mathscr{L}$) → $\mathscr{T}^a$:
Input: $\mathscr{T}$, an unannotated schema tree; $\mathscr{L}$, all labels on the interface
Output: $\mathscr{T}^a$, an annotated schema tree
1. Obtain a list of internal nodes of $\mathscr{T}$ in post-order:
      $<N_1, N_2, ..., N_k>$ ← POSTORDER($\mathscr{T}$)
2. Initialize $L_a$ to contain all labels on the interface:
      a. Let $L_a$ be a set of available labels
      b. $L_a \leftarrow \mathscr{L}$
3. For each $N_i$, annotate its child nodes via ATTACHONE:
      a. Let $L_r$ be a set of unassigned labels
      b. for $i = 1$ to $k$ do
            $L_r$ ← ATTACHONE($N_i$, $L_a$)
            $L_a \leftarrow L_r$
4. Return $\mathscr{T}^a$ = the annotated $\mathscr{T}$

ATTACHONE($N$, $L_a$) → $L_r$:
Input: $N$, an internal node; $L_a$, available labels
Output: $L_r$, remaining labels
1. Generate label candidates:
      a. Let $\mathscr{N}$ be a set of child nodes of $N$
      b. Let $M$ be a $|\mathscr{N}| \times |L_a|$ matrix
      c. $M$ ← OBTAINATTACHMENTMATRIX($\mathscr{N}$, $L_a$)
2. Prune candidates based on annotation rules:
      a. Let $M'$ be the updated attachment matrix
      b. $M'$ ← PRUNECANDS($M$)
3. Assign labels to nodes based on $M'$ via best-first strategy
4. Return $L_r$ = unassigned labels in $L_a$
```

Fig. 11 The label attachment algorithm

(3) Match selection: Based on the pruned attachment matrix M' from step 2, this step assigns labels to blocks via a *best-first* strategy, starting with the most confident assignments. Specifically, the following cases are considered in turn: (a) a label l can only be assigned to a node x and x does not have any candidates other than l; (b) a label l can only be assigned to a node x and l is inside B_N (i.e., the node block of N); (c) a label l can only be assigned to a node x and l is to the right of B_N; and (d) a label l can only be assigned to a node x, but not in case (b) or case (c). Note that cases (b) and (c) are considered before case (d), since the labels which are inside or to the right of B_N are unlikely to be a label for the group N.

For each case, all the entries in M' are checked. If an entry $M'[i, j]$ falls into the case, then the j-th label will be assigned to the i-th block and all entries at the i-th row and the j-column of M' will be set to zero. The above process is then repeated until none of the entries in M' falls into any of the cases.

For example, it can be shown that, after the candidate pruning step, the label From: City will be the only candidate label for the attribute f_1 and f_1 does not have any other candidate labels. Thus From: City will be assigned to f_1 according to case (a). For another example, consider assigning labels to a group of two attributes (g_1 and g_2, each represented by a selection list) on the interface snippet shown in Figure 9. First, Child (Age 2 to 11) will be assigned to g_2 according to case (c), since Child (Age 2 to 11) can only be assigned to g_2 and is located to the right of the group. Next, Adult will be assigned to the g_1 according to case (b). Note that Passengers will not be assigned to g_1 since g_1 has already been assigned a label.

Figure 11 gives the pseudo code of the label attachment algorithm, where AttachOne is shown at the bottom.

6 Empirical Evaluation

We have evaluated ExQ with query interfaces of Deep-Web sources over varied domains. In this section, we present experimental results.

Data Set: All experiments were performed on a real-world data set available from the UIUC Web integration repository [1]. The data set contains query interfaces to Deep Web sources in five domains: airfare, automobile, book, job, and real estate, with 20 query interfaces for each domain. Before the experiments, we manually transformed the query interfaces in the data set into ordered-tree schemas, and used them as the gold standard to gauge the performance of the algorithms.

Table 2 shows the details of the data set. For each domain, columns 2–7 show the minimum, maximum, and average numbers of leaf nodes and internal nodes in the schema trees of the interfaces in that domain. Columns 8–10 show the similar statistics on the depth of the schema trees.

For each domain, we first evaluated the performance of the *structure extractor* on capturing the grouping and ordering relationships of the attributes on the interfaces;

[1] http://metaquerier.cs.uiuc.edu/repository/

Table 2 Domains and characteristics of the data set

Domain	**Leaf Nodes**			**Internal Nodes**			**Depth**		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
Airfare	5	15	10.7	1	7	5.1	2	5	3.6
Auto	2	10	5.1	1	4	1.7	2	3	2.4
Book	2	10	5.4	1	2	1.3	2	3	2.3
Job	3	7	4.6	1	2	1.1	2	3	2.1
Real Estate	3	14	6.7	1	6	2.4	2	4	2.7

we then evaluated the effectiveness of the *schema annotator* in finding the right labels for both the attributes and attribute groups.

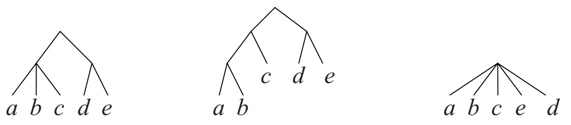
Performance Metrics for Structure Extraction: For each interface, the schema tree produced by the structure extraction algorithm was compared with the schema tree in the gold standard with respect to their structures, i.e. the grouping and ordering of the attributes. A possible metric for comparing two trees is the *tree edit distance* [25], where the distance between two trees is taken to be the number of insertion, deletion, and relabeling operations necessary for transforming one tree into the other. But this metric does not sufficiently capture the *semantic* aspects of two trees, that is, the semantic closeness of two attributes in terms of their grouping relationships, and the relative semantics of two attributes in terms of their ordering relationships.

To address this challenge, we observe that the semantics of a schema tree can actually be encoded with the constraints which the schema enforces on its elements. In particular, we observe that the grouping and ordering relationships of attributes in the schema may be captured with *least-common-ancestor (LCA) constraints* and *precedence constraints*, to be formally defined below. Then, the *semantic* differences of two schema trees may be measured by the extent which the constraints from one schema tree are satisfied by the other schema tree.

Definition 6 ***(LCA Constraint).*** *Consider a schema tree S and denote the lowest common ancestor of two attributes (i.e., leaf elements) x and y in S as LCA(x, y). Consider three attributes x, y and z in S. We say that there exists a* LCA constraint *in the form of* $(x,y)z$ *in S, if* $\mathrm{LCA}(x,y) < \mathrm{LCA}(x,z)$ *and* $\mathrm{LCA}(x,y) < \mathrm{LCA}(y,z)$*, where* $n_1 < n_2$ *denotes that element* n_1 *is a proper descendant of element* n_2*.* □

Intuitively, the LCA constraint $(x,y)z$ indicates that two attributes x and y are semantically closer to each other than either to the attribute z. LCA constraints thus capture the semantic closeness of attributes expressed by the schema. It is interesting to note that given all the LCA constraints from an unordered schema tree S, S can be *fully* reconstructed in polynomial time [1].

Example 4. The LCA constraints in the schema S_1 shown in Figure 12.a are: $(a,b)d$, $(a,b)e$, $(a,c)d$, $(a,c)e$, $(b,c)d$, $(b,c)e$, $(d,e)a$, $(d,e)b$, and $(d,e)c$. □



(a) Schema S_1 (b) Schema S_2 (c) Schema S_3

Fig. 12 Examples on constraints of schemas

Definition 7 ***(Precedence Constraint).*** *Consider a schema S and a sequence of attributes, denoted as q_s, obtained from a* pre-order *traversal of S. We say that there exists a precedence constraint between two attributes x and y, denoted as $x \prec y$, in the schema S, if x appears* before *y in q_s.* □

The precedence constraints thus capture the relative ordering of the attributes, both within the same group and across different groups.

Example 5. q_{S_1} is $<a,b,c,d,e>$. As such, some examples of the precedence constraints in S_1 are: $a \prec b$, $a \prec c$, $a \prec d$, and $d \prec e$. □

Based on these constraints, we evaluated the performance of schema extraction via *grouping metrics* and *ordering metrics* given as follows.

Grouping metrics: We measured the grouping performance of the structure extraction algorithm with three metrics: (LCA) precision, (LCA) recall, and (LCA) F-measure [23]. Denote the schema tree for an interface obtained by the structure extraction algorithm as S', and the schema tree given in the gold standard for the interface as S. The precision is then taken to be the percentage of the LCA constraints which are *correctly* identified by the algorithm (i.e., they are in both S' and S) over all the LCA constraints identified by the algorithm (i.e., they are in S'). And the recall is the percentage of the LCA constraints which are correctly identified over all the LCA constraints in S. F-measure incorporates both precision and recall. We use the F-measure where precision P and recall R are equally weighted, i.e., $F = 2PR/(R+P)$.

Example 6. Suppose that S_1 in Figure 12.a is the schema tree given by the gold standard for an interface. Further suppose that S_2 in Figure 12.b is the schema tree given by the structure extraction algorithm for the same interface.

It can be verified that S_2 have all the nine LCA constraints in S_1 (see Example 4) plus an additional constraint $(a,b)c$. As such, the LCA precision of S_2 is 9/10 = .9, while the LCA recall of S_2 is 9/9 = 1. □

Ordering metrics: We measured the ordering performance of the structure extraction algorithm with two metrics: (precedence) precision and (precedence) recall. Since the number of precedence constraints in a schema with n attributes is always $n(n-1)/2$, precedence precision is always the same as precedence recall. In other words, precedence precision and precedence recall are both given by the ratio of the number of the precedence constraints correctly identified by the algorithm over $n(n-1)/2$.

Example 7. Suppose that S_1 in Figure 12.a is the schema tree given by the gold standard for an interface. Further suppose that S_3 in Figure 12.c is the schema tree given by the structure extraction algorithm for the same interface.

It can be verified that the only different precedence constraints in S_1 and S_3 are $d \prec e$ in S_1 vs. $e \prec d$ in S_3. As such, both the precedence precision and the precedence recall are 9/10 = .9 (note that since $n = 5$, $n(n-1)/2 = 10$). □

Performance Metrics for Label Attachment: We measured the performance of the label attachment algorithm on finding both attribute labels and group labels.

Attribute labeling metrics: The performance on finding attribute labels was measured with two metrics: (attribute labeling) precision and (attribute labeling) recall. The precision is the percentage of the correctly identified labels (i.e., labels attached to correct attributes) over all the labels identified by the algorithm. And the recall is the percentage of the correctly identified labels over all the attribute labels given in the gold standard.

Group labeling metrics: Similarly, the performance on finding group labels was measured with two metrics: (group labeling) precision and (group labeling) recall. But since the groups identified by the structure extraction might not be always correct, a more accurate way of evaluating the group labeling is to base it on the attributes. In particular, for each attribute on the interface, we associate with the attribute the labels of all groups which contain the attribute.

Example 8. The group labels associated with the attribute f_3 in Figure 3.c are Departure Date and When Do You Want to Go?. Intuitively, the group labels associated with an attribute, together with the label of the attribute, denote to the users what the attribute means. □

Based on the above discussions, the group labeling precision is taken to be the percentage of the group labels correctly associated with the attributes by the algorithm over all the group labels associated with the attributes by the algorithm (note that typically a group label may be associated with more than one attributes). And the recall is taken to be the percentage of the group labels correctly associated with the attributes by the algorithm over all the group labels associated with the attributes in the gold standard.

6.1 Evaluating the Structure Extractor

Columns 2–4 of Table 3 show the performance of the structure extraction algorithm.

Grouping: Columns 2–3 show the performance of the structure extraction algorithm on discovering the grouping relationships of the attributes over the five domains. We observe that the precisions range from 82.4% in the job domain to 94.0% in the book domain, with an average of 92.1%. Note that the job domain is the only domain whose precision is lower than 90%. Detailed analysis indicated that some interfaces in this domain use shaded area to indicate attribute groups. And since some of the

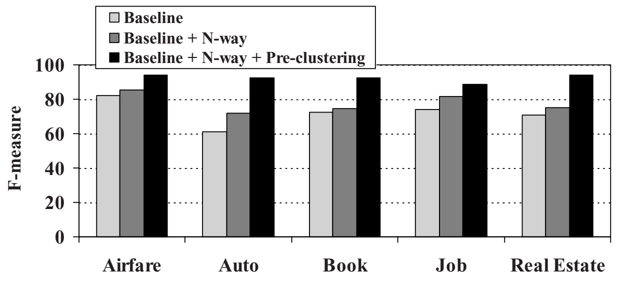


Fig. 13 Effects of the n-way clustering and pre-clustering

attributes in these groups are actually farther away from each other than from the attributes not from the same group, as a result, several attribute groups found by the algorithm were incorrect. A possible remedy is to introduce a new grouping pattern to recognize the attribute groups which are delimited with shaded areas.

We further observe that the recalls range from 90.8% in the book domain to 95.6% in the real estate domain, with an average of 93.9% over the five domains. These indicate that the algorithm is highly effective in identifying the grouping relationships of the attributes.

We also examined the effects of the n-way clustering and pre-clustering on the grouping performance. Figure 13 shows the results. For each domain, the three bars (from left to right) represent the performance produced respectively by the algorithms without the n-way clustering and pre-clustering, with only the n-way clustering, and with both the n-way clustering and pre-clustering being incorporated. All the performances are measured by F-measure. Note that the last bars correspond to the figures shown in Table 3.

It can be observed that with the n-way clustering, the performance improved consistently over the domains, with the largest increase (10.4 percentage points) in the auto domain. Furthermore, the pre-clustering significantly improved the performance in all five domains, ranging from 6.7 percentage points in the job domain to as high as 20.8 percentage points in the auto domain. These indicate the effectiveness of both the n-way clustering and pre-clustering.

Ordering: Column 4 shows the performance of the structure extraction algorithm in identifying the ordering of attributes over the five domains. It can be observed that the accuracy ranges from 96% in the real estate domain to as high as 99.7% in both the auto and book domains. This indicates that the algorithm is highly effective in determining the ordering of the attributes.

6.2 Evaluating the Schema Annotator

The last four columns of Table 3 show the performance of the label attachment algorithm.

Table 3 The performance of the schema extractor

Domains	Grouping		Ordering	Attribute Labels		Group Labels	
	Prec.	Rec.	Prec.(Rec.)	Prec.	Rec.	Prec.	Rec.
Airfare	93.3	95.1	99.5	95.0	93.7	97.7	92.3
Auto	92.0	92.9	99.7	99.2	97.6	100	89.5
Book	94.0	90.8	99.7	91.6	91.6	94.4	82.7
Job	82.4	95.2	97.7	95.1	94.9	100	89.7
Real Est.	92.1	95.6	96.0	99.1	97.3	100	90.5
Average	90.8	93.9	98.9	96.0	95.0	98.4	88.9

Attribute labeling: Columns 5–6 show the performance on attribute labeling. We observe that the precisions range from 91.6% in the book domain to as high as 99.2% in the auto domain, with an average of 96% over the five domains, and that the recalls range from 91.6% to 97.6%, with an average of 95%. These indicate that the label attachment algorithm is highly accurate in determining the labels of the attributes.

Group labeling: The last two columns show the performance on group labeling. It can be observed that high precisions are achieved over the five domains, with 94.4% in the book domain, 97.7% in the airfare domain, and perfect precisions in the other three domains.

It can be further observed that the recalls range from 82.7% in the book domain to 92.3% in the airfare domain. We examined the book domain which had the relatively low recall. Detailed results indicated that there are several interfaces where some of the groups identified by the structure extraction algorithms are only partial, that is, they do not contain all the attributes in the group. As a result, the label attachment assigns the group label to the partial group, resulting in the low recall. This indicates that the label attachment algorithm could be very sensitive to the performance of the structure extraction algorithm, which is not surprising.

Overall, the average precision 98.4% and the average recall 88.9% were achieved on the five domains. These indicate that the label attachment is very effective in identifying group labels.

7 Conclusions

We have presented the ExQ system for extracting & annotating schemas of Deep-Web query interfaces. The key novelties of ExQ include: (1) a hierarchical modeling approach to capture the inherent structure of interfaces & address limitations of existing solutions; (2) a spatial-clustering based algorithm to discover attribute relationships based on their visual representation; and (3) a schema annotation algorithm motivated by human annotation process. ExQ has shown to be very effective in the experiments with query interfaces in a variety of domains. Nevertheless, ExQ is not perfect (like other automatic systems for information extraction). Besides further experiments with additional data sets, we are investigating the direction to turn ExQ into an interactive system. In other words, ExQ may ask domain experts questions to help resolve its uncertainties on attribute relationship and label attachment. We note that similar methods have been successfully employed in schema matching (e.g., [7, 29]) and other information extraction tasks (e.g., [14]).

References

1. Aho, A., Sagiv, Y., Szymanski, T., Ullman, J.: Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. SIAM Journal on Computing 10(3), 405–421 (1981)
2. Arasu, A., Garcia-Molina, H.: Extracting structured data from Web pages. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003), pp. 337–348 (2003)
3. Barbosa, L., Freire, J.: Searching for hidden-Web databases. In: Proceedings of the 8th ACM SIGMOD International Workshop on Web and Databases (WebDB 2005), pp. 1–6 (2005)
4. Bergman, M.: The Deep Web: Surfacing the hidden value. BrightPlanet.com (2000), `http://www.brightplanet.com/technology/deepweb.asp`
5. Chang, K., He, B., Li, C., Patel, M., Zhang, Z.: Structured databases on the Web: Observations and implications. ACM SIGMOD Record 33(3), 61–70 (2004)
6. Crescenzi, V., Mecca, G., Merialdo, P.: RoadRunner: Towards automatic data extraction from large Web sites. In: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001), pp. 109–118 (2001)
7. Doan, A., et al.: Reconciling schemas of disparate data sources: A machine-learning approach. In: SIGMOD 2001 (2001)
8. Doorenbos, R., Etzioni, O., Weld, D.: A scalable comparison-shopping agent for the World-Wide Web. In: Proceedings of the First International Conference on Autonomous Agents (AGENTS 1997), pp. 39–48 (1997)
9. He, B., Chang, K.: Statistical schema matching across Web query interfaces. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003), pp. 217–228 (2003)
10. He, H., Meng, W., Yu, C., Wu, Z.: WISE-Integrator: An automatic integrator of Web search interfaces for e-commerce. In: Aberer, K., Koubarakis, M., Kalogeraki, V. (eds.) VLDB 2003. LNCS, vol. 2944, pp. 357–368. Springer, Heidelberg (2004)
11. He, H., Meng, W., Yu, C., Wu, Z.: Constructing interface schemas for search interfaces of Web databases. In: Ngu, A.H.H., Kitsuregawa, M., Neuhold, E.J., Chung, J.-Y., Sheng, Q.Z. (eds.) WISE 2005. LNCS, vol. 3806, pp. 29–42. Springer, Heidelberg (2005)
12. Kaljuvee, O., Buyukkokten, O., Garcia-Molina, H., Paepcke, A.: Efficient Web form entry on PDAs. In: Proceedings of the 10th International Conference on World Wide Web (WWW 2001), pp. 663–672 (2001), `citeseer.nj.nec.com/kaljuvee01efficient.html`
13. Kaufman, L., Rousseeuw, P.: Finding Groups in Data: An Introduction to Cluster Analysis. John Wiley & Sons, Chichester (1990)
14. Kristjansson, T.T., Culotta, A., Viola, P.A., McCallum, A.: Interactive information extraction with constrained conditional random fields. In: AAAI (2004)

15. Kushmerick, N., Weld, D., Doorenbos, R.: Wrapper induction for information extraction. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997), pp. 729–737 (1997)
16. Laender, A.H.F., Ribeiro-Neto, B.A., da Silva, A.S., Teixeira, J.S.: A brief survey of web data extraction tools. SIGMOD Rec. 31(2) (2002)
17. Lerman, K., Minton, S., Knoblock, C.: Wrapper maintenance: A machine learning approach. Journal of Artificial Intelligence Research (JAIR) 18, 149–181 (2003)
18. Madhavan, J., Cohen, S., Dong, X.L., Halevy, A.Y., Jeffery, S.R., Ko, D., Yu, C.: Web-scale data integration: You can afford to pay as you go. In: CIDR (2007)
19. McCann, R., AlShelbi, B., Le, Q., Nguyen, H., Vu, L., Doan, A.: Maveric: Mapping maintenance for data integration systems. In: Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005), pp. 1018–1030 (2005)
20. Perkowitz, M., Doorenbos, R., Etzioni, O., Weld, D.: Learning to understand information on the internet: An example-based approach. Journal of Intelligent Information Systems 8(2), 133–153 (1997)
21. Raghavan, S., Garcia-Molina, H.: Crawling the hidden Web. In: Proceedings of 27th International Conference on Very Large Data Bases (VLDB 2001), pp. 129–138 (2001), `citeseer.nj.nec.com/raghavan01crawling.html`
22. Rahm, E., Bernstein, P.: A survey of approaches to automatic schema matching. VLDB Journal 10(4) (2001)
23. van Rijsbergen, C.: Information Retrieval. Butterworths, London (1979)
24. Wang, J., Wen, J., Lochovsky, F., Ma, W.: Instance-based schema matching for Web databases by domain-specific query probing. In: Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004), pp. 408–419 (2004)
25. Wang, J., Zhang, K.: Finding similar consensus between trees: An algorithm and a distance hierarchy. Pattern Recognition 34, 127–137 (2001)
26. Widom, J.: Integrating heterogeneous databases: lazy or eager? ACM Computuing Surveys (CSUR) 28(4) (1996)
27. Wu, W., Doan, A., Yu, C.: Merging interface schemas on the Deep Web via clustering aggregation. In: Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005), pp. 801–804 (2005)
28. Wu, W., Doan, A., Yu, C.: WebIQ: Learning from the Web to match Deep-Web query interfaces. In: Proceedings of the 22nd IEEE International Conference on Data Engineering (ICDE 2006), p. 44 (2006)
29. Wu, W., Yu, C., Doan, A., Meng, W.: An interactive clustering-based approach to integrating source query interfaces on the Deep Web. In: SIGMOD 2004 (2004)
30. Zhang, Z., He, B., Chang, K.: Understanding Web query interfaces: Best-effort parsing with hidden syntax. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD 2004), pp. 107–118 (2004)