# Merging Source Query Interfaces on Web Databases


| Eduard Dragut | Wensheng Wu | Prasad Sistla, Clement Yu | Weiyi Meng |
|---|---|---|---|
| CS. Dept. | CS. Dept. | CS. Dept. | CS. Dept. |
| UIC | UIUC | UIC | SUNY at Binghamton |
| edragut@cs.uic.edu | www2@uiuc.edu | {sistla,yu}@cs.uic.edu | meng@binghamton.edu |



## Abstract

*Recently, there are many e-commerce search engines that return information from Web databases. Unlike text search engines, these e-commerce search engines have more complicated user interfaces. Our aim is to construct automatically a natural query user interface that integrates a set of interfaces over a given domain of interest. For example, each airline company has a query interface for ticket reservation and our system can construct an integrated interface for all these companies. This will permit users to access information uniformly from multiple sources. Each query interface from an e-commerce search engine is designed so as to facilitate users to provide necessary information. Specifically, (1) related pieces of information such as* first name *and* last name *are grouped together and (2) certain hierarchical relationships are maintained. In this paper, we provide an algorithm to compute an integrated interface from query interfaces of the same domain. The integrated query interface can be proved to preserve the above two types of relationships. Experiments on five domains verify our theoretical study.*


## 1. INTRODUCTION

As indicated in two recent surveys [4, 6], it has been estimated that there are between 127,000 to 330,000 Web databases and over 550 billion content pages/records in these databases. A user seeking information from data sources of the same domain has to individually access each of these sources via its interface in order to find his/her desired information. For example, a user booking a flight is often interested in probing alternative sources in the same domain in order to get a good price or a suitable departure time. Therefore, one step towards the integration of these sources is the construction of an integrated query interface. Such an interface would provide uniform access to the data sources of a given domain of interest. Currently, such integrated user interfaces exist, but they are manually constructed. Though one might pursue a manual creation of a unified interface for a few query interfaces (four or five), it is hard to do so for a large number of interfaces without errors and in a timely manner. In addition, a manual approach is not resilient to even small modifications in the individual query interfaces.

Given a set of Web interfaces in the same domain, the problem is to construct automatically a unified query interface which contains all (or, alternatively, the most significant) distinct fields of all schemas and arrange them in a way that facilitates users to fill in the desired information.

The process of integrating query interfaces can be carried out in the following steps. The first step consists of extracting query interfaces from individual web pages [25] such that for each of them the attributes and their parent-child relationships are obtained. Second, the semantic relationships between the fields in different interfaces in the same domain are computed. In recent years, a considerable effort has been dedicated to finding these semantic mappings [7, 8, 9, 10, 13, 16, 24]. The accuracies of these automatic methods to identify the semantic mappings between fields can be as high as 90%. Third, the interfaces in the same domain are integrated into a unified interface [15]. In the fourth step a global query submitted through the unified interface is mapped to the queries of the individual sources [15]. Finally, the returned results are correlated and filtered according to the global query and organized into a single list for presentation to the user. In this study, we concentrate on the issue of merging interfaces, that is the third step, noting that the semantic mappings have been obtained [24]. Field value merging and format integration belong to this step, but they will not be discussed here. A comprehensive treatment on the subject can be found in [15].

The current state of the art merge algorithms [2, 18, 20, 21, 22, 23] cannot be directly employed for an automatic construction of a unified user interface over a given domain as existing works assume some human intervention.

A key problem addressed in this paper is that in a set of interfaces of a given domain that have to be merged, there are certain collective constraints induced by the interfaces

that need to be determined a priori and enforced during the merging process to form the final global interface. We observe that certain groups of attributes appear together with high frequency in a set of interfaces over a domain of discourse. For instance, subsets of fields consisting of *Departure day*, *Departure month*, *Departure year* and *Departure time* (see Figure 2) occur in different query interfaces of the airline domain. Thus, in the integrated interface these fields should be in adjacent locations. As a concrete example, we might have $D_1$ ={*Departure day*, *Departure month*, *Departure time*} in a user interface; $D_2$ = {*Departure year*, *Departure month*, *Departure day*} in another user interface. Hence, we require that the fields in $D_1$ and those in $D_2$ to be adjacent and $D_1$, $D_2$ to be grouping constraints. User interface designers have different perspectives and goals when creating an interface. Thus, it is possible for some designers to be inconsistent with most other designers of user interfaces of the same domain or to accidently make minor mistakes. Therefore, there are cases when not all the grouping constraints can be satisfied at the same time. In such situations we rely on the frequencies of occurrence of the grouping constraints to provide a solution that violates as few constraints as possible. By minimizing the number of violations, the consistencies or inaccuracies will be minimized. It is also possible to eliminate small amount of spamming as the majority view is used to construct the unified interface. Another important semantic property which is exhibited in individual user interfaces is that attributes are arranged hierarchically [24] (i.e. they have ancestor-descendant relationships). These relationships should also be reflected in the integrated user interface.

The contributions of this paper consist of:

- determining grouping constraints from individual user interfaces and constructing the groups, which satisfy the grouping constraints as much as possible.
- describing an algorithm that preserves the ancestor-descendant relationships among attributes in each of the source interfaces.
- providing an algorithm that merges two interfaces at a time until all user interfaces are merged into an integrated interface. We show that the integrated interface preserves all ancestor-descendant relationships in the individual user interfaces, while at the same time enforces the grouping constraints as much as possible.
- performing experiments on five domains (airline, automobile, book, job, real estate) to show that in each domain, the integrated interface is natural.

The organization of this paper is as follows. In Section 2, we define the problem of merging a collection of query interfaces. Our techniques are given in Sections 3 and 4. The former section presents our techniques for grouping attributes and provides an algorithm to compute such groups and the latter describes the actual merge algorithm. An experimental evaluation of our approach is shown in Section 5. Section 6 contrasts our work with related works. The paper closes with future work and conclusions. All proofs are left out due to space limitation.

## 2. PROBLEM SETTINGS

### 2.1. Representation of Query Interfaces

Data in searchable databases are accessible through form-based search interfaces (mostly HTML forms), which can be either static or dynamic. A dynamic interface could change its content upon user's input whereas static interfaces have the same set of fields displayed irrespective of user's input. For instance, in the real estate domain an all purpose interface could change the displayed fields based on the user's choices: if the user chooses to look for apartments then a dynamic interface would show only those fields relevant for querying on apartments whereas if the user selects to look for single family houses then a different set of fields is displayed. While dynamic query interfaces might occur, they are relatively rare and to the best of our knowledge none has been reported in the ongoing Deep Web projects (e.g. [14, 15, 24, 25]. Though it is possible to capture dynamic query interfaces in the model that we will introduce shortly, in what follows we will restrict our attention to static query interfaces only.

The basic building blocks for these forms are: text input boxes, selection lists, radio buttons, and check boxes. We will generically call them *fields*.

Since query interfaces provide access to Web databases they have to visually convey information about the data in the databases so that the user can easily infer the necessary data he/she has to provide in order to obtain the desired information. For instance, an important aspect of user interfaces, in general, that distinguishes them from other meta-models is a sort of "spatial locality" property among the fields. That is, related fields are usually placed close to each other in the interface. For example, in the automobile domain, the *brand* and the *model* of a car are grouped together very often. Moreover, several related groups can form a super-group [24]. As an example, the field describing information about the year of a car is likely to be grouped together with the group {*brand*, *model*} to form a super-group. Thus, this bottom-up characterization gives rise to a hierarchical structure for interfaces. In addition, each field or group of fields have labels that describe to the user what the field (or group of fields) is about.

The structure of a query interface can be captured using a hierarchical schema [24, 25]. Namely, it is an ordered tree of elements so that leaves correspond to the fields in the interface, internal nodes correspond to groups or super-groups of fields in the interface, and the order among the

sibling nodes within the tree resembles the order of fields in the interface (usually this is from left-to-right).

**Example 1** *Figure 1 shows a typical example of a query interface in the airline domain and its corresponding hierarchical representation. Observe that the schema tree corresponding to the query interface has three levels, each level except for the root refines the fields in the level above. Moreover, each node of the tree has a label.*

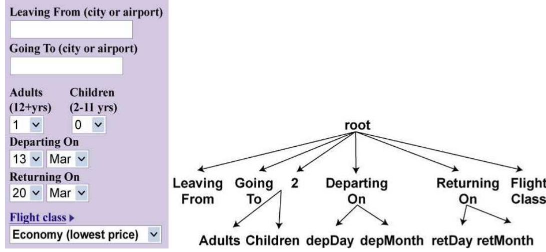


**Figure 1.** Hierarchical representation of a query interface

## 2.2. Merge Algorithm Inputs

The input to the merge algorithm consists of:

- A set of query interfaces in a given domain of interest.
- A mapping that globally characterizes the semantic correspondences between equivalent fields in the query interfaces.

The mapping to capture the relationships between all the fields over all the schemas is organized in *clusters* [24], that record 1:1 and 1:m matchings of fields. Within a given domain, *a cluster consists of all the fields of different schemas that are semantically equivalent*. As a case in point, in the example of Figure 2 *Depday*, *Departure Day* and *depDay* are semantically equivalent and they are in the same cluster, called *Dep day*, while *DepMonth*, *Departure month* and *depMonth* are in a different cluster, called *Dep month*. Additionally, a field that matches multiple fields in different clusters is placed in each of the clusters where there is a field it matches. For example, if there is another schema having the notion of departure date modelled as a single field, called *Departure Date*, then this is placed in three clusters denoting *Dep day*, *Dep month* and *Dep year*.

The elements of a cluster are pairs of the form $(SchemaName, fieldId)$, where *SchemaName* represents the name of the schema containing the field (which is unique among all the schemas in the domain) and *fieldId* uniquely identifies the field among the fields of the schema tree. Each cluster has a unique id. There may be schemas that do not have fields in all the clusters. In that case, the cluster has the entry $(SchemaName, Null)$. In order to ease our exposition we assign human readable names to the clusters. Table 1 illustrates an example of such clusters based on the mapping relationship between the three

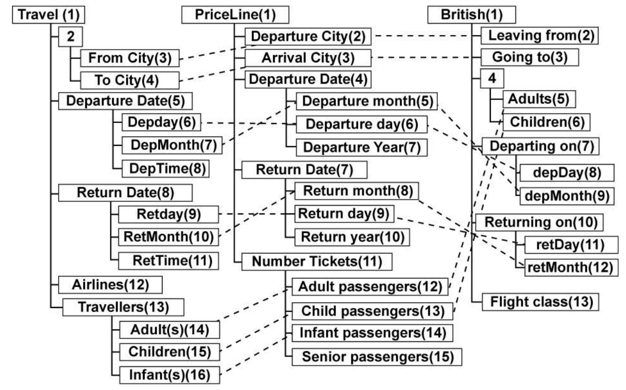


**Figure 2.** Example of three schema trees and the mapping between them

schema trees shown in Figure 2. The dashed lines represent the mapping correspondences between the schemas. The mapping between schema Travel and schema British is omitted but can be easily inferred in the figure.

In this work we assume that the semantic relationships between the attributes of the query interfaces in the same domain have been already computed. In fact, the computation of the clusters as described here is defined and analyzed in our work [24] on matching query interfaces on the Deep Web.

## 2.3. The Output of Merge Algorithm

Given a set of interfaces and a mapping definition over them, the output of the merge algorithm is a query interface that consists of all (or the most significant) elements of all interfaces. That is, it has a field for each of the clusters in the mapping definition and preserves all the constraints enforced by the interfaces being merged. The constraints to be satisfied by the global interface are the grouping constraints (to be described more precisely in the next section) and the ancestor-descendant relationships among the fields within individual interfaces.

# 3. FIELDS ORGANIZATION in an INTEGRATED QUERY INTERFACE

In this section we introduce the notion of groups of fields and develop the formalism that helps us in identifying such groups in a collection of schemas.

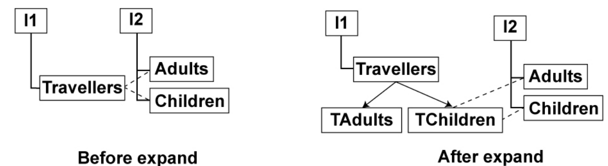


**Figure 3.** Expand procedure

| cluster name | Dep city | Dest city | Dep month | Dep day | Dep time | Dep year | Adults | Infants |
|---|---|---|---|---|---|---|---|---|
| | (Travel,3) | (Travel,4) | (Travel,7) | (Travel,6) | (Travel,8) | (Travel,null) | (Travel,14) | (Travel,16) |
| | (PriceLine,2) | (PriceLine,3) | (PriceLine,5) | (PriceLine,6) | (PriceLine,null) | (PriceLine,7) | (PriceLine,12) | (PriceLine,14) |
| | (British,2) | (British,3) | (British,9) | (British,8) | (British,null) | (British,null) | (British,5) | (British,null) |
| **cluster name** | **Ret month** | **Ret day** | **Ret year** | **Ret time** | **Airlines** | **Class** | **Children** | **Seniors** |
| | (Travel,10) | (Travel,9) | (Travel,null) | (Travel,11) | (Travel,12) | (Travel,null) | (Travel,15) | (Travel,null) |
| | (PriceLine,8) | (PriceLine,9) | (PriceLine,10) | (PriceLine,null) | (PriceLine,null) | (PriceLine,null) | (PriceLine,13) | (PriceLine,15) |
| | (British,12) | (British,11) | (British,null) | (British,null) | (British,null) | (British,13) | (British,6) | (British,null) |

**Table 1.** The cluster structure for the schema trees depicted in Figure 2

## 3.1. Modification of Query Interfaces

Granularity mismatch between two schemas occurs when there are 1:m mapping relationships between the attributes of the schemas. Such relationships are either is-a or part-of [24]. In order to have a uniform representation of the fields within all the schemas 1:m relationships have to be reduced to instances of 1:1 relationships. The resolution is done by expanding the leaf element, say $e$, on the one side of 1:m mapping into an internal element $e'$ whose children have 1:1 correspondence to the leaf elements on the many side. Consider the example of Figure 3. Field *Travellers* in interface $I_1$ participates in a 1:2 mapping relationship with *Adults* and *Children* of interface $I_2$ (see Figure 3, on the left). Thus, *Travellers* has to be expanded such that between the two interfaces there are only 1:1 correspondences. In the new organization of interface $I_1$ (see Figure 3, on the right), *Travellers* is an internal node with two children, denoted *TAdults* and *TChildren*, respectively which have 1:1 correspondences with elements *Adults* and *Children* respectively, in interface $I_2$.

## 3.2. Groups

As the global interface will have a field for each cluster, the first problem to be solved is finding a partition over the set of clusters of a given domain that characterizes the way fields are grouped in the integrated schema. For instance, the groups for the example presented in Table 1 and Figure 2 are [*Dep city*, *Dest City*], [*Dep time*, *Dep day*, *Dep month*, *Dep year*], [*Adults*, *Children*, *Infants*, *Seniors*] and [*Ret time*, *Ret day*, *Ret month*, *Rep year*], with each group denoting a set of fields that form a semantic unit of information.

We say that a non-empty set of clusters have a common parent if there exists a schema tree that has a field in each of the clusters and all these fields have a common non-root parent. For instance, the set of clusters $\{Dep\ day, Dep\ month, Dep\ year\}$ described in Table 1 has a common parent, namely the node *Departure Date* of schema tree PriceLine. Observe that we do not consider the clusters that share a parent that is the root in a given schema, because an interface may have all fields as children of the root. In such a case, no reliable information can be derived.

**Definition 1** *A set of clusters $X = \{x_1, x_2, ..., x_k\}$, $k \geq 2$, is a* potential group *if (1) there is a schema in which all elements in X have a common non-root parent, $v$, (2) are adjacent, and (3) $X$ is the maximal set satisfying the previous two conditions with respect to node $v$.*

The set of clusters $\{$*Adults*, *Children*, *Infants*$\}$ represents a potential group since there is a node, *Travellers*, in schema Travel that satisfies all the conditions in the definition above. Notice that $\{$*Children*, *Infants*$\}$ is not a potential group since *Travellers* has also a child, *Adults*.

The order of the elements within a potential group depends on various factors, like the social and geographical contexts where the application is going to be used (e.g. North America vs Europe) or the application domain (e.g. academic environment vs business environment). Consider the potential group consisting of the following concepts: *First Name*, *Middle Name*, *Last Name*. This ordering of the concepts could correspond to the American way to write and address a person (e.g. John Allen Smith). However, in other cultures (e.g. Eastern Europeans) *Last Name* is normally used first, followed by *First Name* and *Middle Name* (e.g. Smith John Allen). It is well known that the European format for date is day/month/year whereas month/day/year corresponds to the American format. Given these observations we postulate that the actual order of the elements within potential groups is not important, and will not be taken into consideration for the rest of the paper.

We consider potential groups to be a natural abstraction to capture the way fields are organized within a query interface. Namely, they underline designer's perspective that these fields should be together so that users can easily understand what is required and fill in the desired information with ease. In addition, it is unlikely that fields with no apparent relationships are to be placed together. For example, one would hardly accept a query interface having fields requesting information about the type of passengers (e.g. seniors, children) intermixed with the fields about departure date (e.g. month, day, year). With the assumption that the organization of attributes within a query interface has a meaningful purpose and, therefore, enforces certain structural constraints, we conclude our discussion about potential group with the observation that an ideal situation is to have an integrated schema such that for each potential group in an input schema, all its elements are adjacent in the integrated schema. The following definition rephrases this observation in a more systematic way.

**Definition 2** *A* group*, $G$, consisting of the set of clusters*

$X = \{x_1, x_2, ..., x_k\}, k > 1$, *is a sequence of the elements of X,* $X_{seq} = [x_{i_1}, ..., x_{i_k}]$, *such that:*

1. *each* $x_i$ *belongs to at least a potential group,*
2. *if there is potential group, g, such that* $g \cap G \neq \emptyset$ *then* $g \subseteq G$,
3. *if there is a potential group, g,* $g \subseteq X$ *then all elements of g are adjacent in* $X_{seq}$
4. *for any two distinct potential groups, g and* $g'$, *in G there exists a finite sequence of distinct potential groups,* $g_1, g_2, ..., g_n, n \geq 2$ *in G such that* $g_1 = g$ *and* $g_n = g'$ *and* $g_i \cap g_{i+1} \neq \emptyset$, $1 \leq i \leq n-1$.

A *group* represents the desired organization of the fields (leaves) in an integrated schema tree. Groups subsume all the conditions enforced by each individual potential group (and implicitly by each schema tree). The first condition of the definition ensures that a group contains elements from potential groups only. The intuition behind the second condition is the requirement that if a group contains some of the elements of a potential group then it contains all of them. The rationale for the third condition is to enforce that all elements of a potential group be adjacent. The last one prevents us from having unrelated elements grouped together in the integrated schema. For instance, it is undesirable to have elements like $\{Adults, Children, Infants, Seniors\}$ put together with {*Dep city*, *Dest city*}.

**Example 2** *In the example depicted by Figure 2 and Table 1 the following set of clusters,* $X$ = {Seniors, Adults, Children, Infants}, *forms a group. The third column of Table 2 shows all potential groups pertaining to this group and it is relatively easy to check that all of them are in* $X$. *Moreover, there is a permutation of* $X$, *namely,* {Adults, Children, Infants, Seniors}, *such that all elements in each potential group having elements in* $X$ *are adjacent. Observe that for any two potential groups in the group the fourth condition of the definition is trivially satisfied by considering sequences that consists only of the potential groups themselves (i.e. sequences of length two).*

Our aim is to find an integrated schema consisting of groups of fields such that the constraints imposed by the potential groups in the individual schemas are preserved. As we will show in the subsequent sections it is not always possible to construct these groups. We will present our strategy to address this case, too.

### 3.3. Finding the Partition of Groups

The following result says that the constraints imposed by different schema trees on grouping fields is equivalent to forming disjoint groups in the integrated schema tree.

**Proposition 1** *There exists an integrated schema such that all grouping (adjacency) constraints imposed by potential groups in the schemas are satisfied if and only if the set of attributes in the schemas that appear in the potential groups can be partitioned into disjoint groups.*

**Proposition 2** *If such a partition exists then this partition is unique up to a permutation of the elements within each group of the partition.*

Ideally, the result of the algorithm is a partition over the fields (clusters) that appear in the potential groups such that each set of the partition is a group. As we will show shortly there are instances when some of the sets of the partition do not meet the group requirements. In that case, we will minimize the number of "violations", i.e. the number of constraints (due to the potential groups in individual interfaces) that cannot be satisfied.

| Potential groups | | |
|---|---|---|
| Dep city, Dest city | Dep day, Dep month | Adults, Children |
| | Dep day, Dep month, Dep time | Adults, Children, Infants |
| | Dep month, Dep day, Dep year | Adult, Children, Infants, Seniors |

**Table 2.** Some potential groups induced by the three schemas of Figure 2.

The computation of the set of all potential groups is accomplished as follows. For each schema tree, $s$, in the set of interfaces $S$ we look for all internal nodes, except the root, that have at least two adjacent leaves. Suppose $v$ is such an internal node in schema tree $s$. We analyze its set of child leaf nodes and distinguish the following cases: (i) the set of leaves are adjacent (i.e. there is no internal node separating them), (ii) among the children of $v$ there are also internal nodes which separate its leaves. In the first case the leaf nodes (fields) induce a potential group, which consists of the clusters each field belongs to. When the leaves are separated by internal nodes we consider each maximal subset of adjacent leaf nodes to be a potential group, discarding those subsets with only one element. After we gather all potential groups induced by each individual interface we want to determine if disjoint groups can be formed. We also keep track of the number of occurrences of each distinct potential group. This will be used to minimize the number of violations, if disjoint groups cannot be formed.

Consider the example depicted in Figure 2 and assume that $S$, the set of interfaces, is composed of those three schemas. In this setting, schema Travel will induce the following set of potential groups: {{*Dep city, Dest city*}, {*Dep month, Dep day, Dep time*}, {*Adults, Children, Infants*}, {*Ret month, Ret Day, Ret time*}}. It is now easy to see that the set of all distinct potential groups induced by the three schemas contains those which are presented in Table 2.

It turns out that the potential groups of a group in the disjoint set of groups (in Proposition 1) can be easily obtained

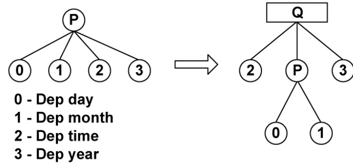


**Figure 4.** Example of applying PQ-trees in group finding

by taking the union of potential groups which have non-empty intersections. For example, the group {*Dep time, Dep day, Dep month, Dep year*} is formed from the three potential groups in Table 2.

### 3.3.1 Testing for Groups

One of the central problems this work tries to address is how to determine that a given set of potential groups amounts to a group. Formally, we reduce the problem to showing that given the set of potential groups, $G = \{g_1, ..., g_k\}, k \geq 1$, and the set $X = \{c | \exists g_i \in G \ and \ c \in g_i\}$, of all distinct clusters in $g_i$'s, we want to find a permutation $X_{seq}$ of $X$ such that all elements in each potential group, $g_i$, appear in adjacent positions in $X_{seq}$. This last formulation of our problem is identical to the set-definition of the Consecutive Ones Property (C1P). Specifically, given an universal set $U$ and a subset, $B$, of the power set of $U$ we want a permutation $\pi$ of the elements of $U$ such that all the elements in each set in $B$ appear as a consecutive sequence in $\pi$ [3, 11, 17]. In our problem, $X$ corresponds to $U$, $G$ corresponds to the power set $B$ and the permutation $X_{seq}$ corresponds to $\pi$.

The C1P exists if and only if a non-null PQ-tree can be constructed [3]. This tree can be constructed in linear time. Other solutions have been proposed for solving C1P, most of them being variations or generalization of PQ-tree, such as the PQR-tree [17]. In order to address our group finding problem we have chosen to implement a solution based on PQ-tree. Booth and Lueker [3] describe PQ-Trees as a data structure for representing the permutations of a set U in which various subsets of U are constrained to occur consecutively. They present efficient algorithms for manipulating PQ-Trees. The fundamental elements of PQ-Trees are P-Nodes and Q-Nodes. P-Nodes allow their children to be permuted in any order, while Q-Nodes allow only a reversal of the ordering of their children. These operations are referred as equivalence transformations. Two PQ-Trees are deemed to be equivalent if one can be transformed into the other by applying zero or more equivalence transformations. The initial or universal PQ-Tree is created by adding all of the elements of U as P-Node (leaf) children of a root P-Node. The example depicted in Figure 4 shows (on the left) the universal tree for the set of elements {*Dep day*, *Dep month*, *Dep time*, Dep year}. For the collection of potential groups given in the second column of Table 2, namely, {*Dep day, Dep month*}, {*Dep day, Dep month, Dep time*}, {*Dep month, Dep day, Dep year*} we will get the PQ-tree shown on the right of Figure 4.

**Algorithm Groups**$(C, S) \rightarrow G$
1. retrieve all distinct potential groups along with
   their numbers of occurrences
2. compute all disjoint sets of potential groups
3. **for** each set of potential groups /*check if it is a group*/
4.   **if** a PQ-tree exists **then** /*we have a group*/
5.     **return** the leaves of PQ-tree
6.   **else** compute the permutations with
   the smallest number of violations
7.     check for the existence of a sequence of splits
   that maintains the same amount of violations
8.     **if** such a sequence of splits exists **then**
9.       split the permutation
10.     **else** keep the original permutation.
11. **return** the partition of clusters

**Figure 5.** The group partitioning algorithm

The basic idea that we exploit in our group finding problem is that, in general, given a universal set $U$ and a subset, $B$, of the power set of $U$ there exists a permutation $\pi$ of the elements of $U$ such that the elements of each set in $B$ are consecutive in $\pi$ if and only if a PQ-tree exists. Moreover, if such a tree exists a possible permutation is obtained by taking the leaves of the tree. For the example described in Figure 4 the permutation given by the leaves is [*Dep time*, *Dep day*, *Dep month*, *Dep year*].

While nice and elegant solutions have been proposed for testing the existence of such permutations [3, 17], the problem can be shown to be NP-hard when one looks for partial (optimal) solutions in the case a solution does not exist for the original input. The common accepted strategy to deal with intractable problems is to look for an approximate solution. However, in our extensive analysis of the five domains that we have considered for testing our approach we have noticed that (1) the universal set $U$ consists of at most seven elements, which may form a group and (2) the consecutive ones property is very often satisfied (e.g. only one group out of 24 in the five domains does not obey the consecutive ones property). Therefore, we consider as reasonable a non-efficient implementation (i.e. exponential running time) that looks for the optimal solution. An optimal solution in our problem is given by a permutation that satisfies a maximum number of constraints enforced by the potential groups (or equivalently a minimum number of violations). Now it becomes clearer why we have also computed the number of occurrences of each potential group among the schemas of a given domain of discourse.

We can formally state the grouping problem as follows: given the set of potential groups, $G = \{g_1, ..., g_k\}, k \geq 1$, along with their associated frequencies $\{f_1, ..., f_k\}$ and the set $X = \{c | \exists g_i \in G \ and \ c \in g_i\}$, of all distinct clusters in $g_i$'s, we want to find a permutation $X_{seq}$ of $X$ such that all elements in each $g_i$ of $G$ appear in $X_{seq}$ in adjacent

| potential groups | occurrences |
|---|---|
| Dep day, Dep month | 7 |
| Dep day, Dep month, Dep time | 3 |
| Dep month, Dep day, Dep year | 4 |
| Dep day, Airline, Class | 2 |

**Table 3.** A collection of potential groups and their numbers of occurrences.

positions. If such permutation does not exist find a permutation $X'_{seq}$ of $X$ such that $f = \sum_{g_i \in G'} f_i$ is maximized, where $G'$ represents the set of potential groups satisfying the adjacency condition in the permutation.

**Proposition 3** *The grouping problem is NP-hard. Moreover, the problem is NP-hard even if each potential group has 2 attributes.*

It is obvious that if all the elements of $G$ can be accommodated in the permutation then we have an optimal solution since $f$ is the maximum. If a group cannot be formed, but the maximum number of constraints is achieved we say that the grouping constraints are satisfied *as much as possible*. Based on the experience we accumulate from studying the schemas in the five domains we have established that when such permutations do not exist this is an indication that there is no apparent relationship between all elements in the set of related potential groups. Therefore, we try to determine those subsets of elements among which a structural relationship exists. We do this by determining a way to split them in disjoint subsets such that the value of $f$ is maintained. In addition we require that any new subset has at least two elements. If such splits cannot be determined the original permutation is accepted as optimal.

**Example 3** *Consider an example consisting of the potential groups as shown in the first column of Table 3. The second column of the table contains the frequency of each potential group. Using the PQ-tree algorithm [3], it can be shown that a group cannot be formed. By enumerating all permutations of the six elements* Dep time, Dep day, Dep month, Dep year, Airline, Class *the maximum number of constraints that can be satisfied is $f = 14$. And this is achieved by splitting the set into [*Dep time, Dep day, Dep month, Dep year*] and [*Airline, Class*].*

# 4. Merging the Interfaces

Our goal is to construct a global interface that satisfies the grouping constraints as much as possible and preserves the ancestor-descendant relationships in the individual schema trees. Note that while the computation of the groups has been carried out in a global manner, the actual merge process is performed two interfaces at the time. The chief reason is that we strive to preserve the ancestor-descendant relationships in all the individual schemas.

In hierarchical modeling of data the same information can be represented in various ways. For instance, in the book domain the relationship between the Authors and the Books entities can be captured as either Authors having Books, which makes Books a descendant of Authors, or Books having Authors, and in this case Books is an ancestor of Authors. This sort of heterogeneity among merged models violates the preservation of ancestor-descendant relationships. Given such a possibility, we have undergone a thorough study of more than 300 query interfaces in 8 domains used in various Deep Web projects [14, 15, 24, 25] and we have not identified any occurrence of the problem depicted above. Our possible explanation for the Authors-Books example is that in a query interface both Authors and Books are internal nodes without ancestor-descendant relationship between them. The children of Authors are First Name, Last Name while the children of Books are ISBN, Title, etc. We believe that there is a distinction between general hierarchical modeling of data (such as those in XML) and query interfaces. In the former, an element can be the ancestor of another element in one hierarchy but the reverse is true for another hierarchy. In the latter, ancestor-descendant relationships among elements are usually preserved. Thus, in this paper ancestor-descendant relationships in individual interfaces need to be preserved in the global user interface as they convey significant semantic information.

Observe that, by the end of the partitioning phase, there could be clusters that do not belong to any group. Such clusters are produced either by fields that appear as leaf children of the root or as isolated children of internal nodes, other than the root. In the merging process, these two types of clusters will be treated differently. Namely, the former will be discarded before the merge and the clusters in this category will be added as children of the root of the integrated schema and for the latter we rely on the merge algorithm to find their appropriate locations within the integrated schema so as to preserve their ancestor- descendant relationships.

## 4.1. Pairwise Merge Algorithm

In the merging process, the potential mappings between the internal elements (nodes) are identified based on the mappings of their sub-elements which can be either leaf elements (nodes) or internal elements. The merging of a set of schemas is carried out pairwise: we start by merging the first two schemas, $S_1$ and $S_2$. The resulted integrated schema, denoted by $S_{12}$ is then further merged with the third schema $S_3$, and so on.

We first define several functions on the elements of trees as well as the set of merge operations to be applied by the algorithm for the merging of schemas. The algorithm uses the following functions defined on elements of trees: (1)

for an element *e* in a tree, *rmls(e)* returns the rightmost left sibling of *e* in the tree; or null if it does not have a left sibling; (2) *lmrs(e)* is similarly defined to return the leftmost right sibling of *e*; (3) for a set of elements *W* in a schema tree, *lca(W)* returns the lowest common ancestor element of the elements in *W*. There are several types of operations in merging schema tree S into schema tree T. (1) Combine: *Cmb(S, e, T, f)* which is to combine the element *e* in the schema *S* into the element *f* in the schema *T*; (2) Insert elements: *InsRs(S, e, T, f)* to insert the element *e* in the schema *S* into the schema *T* as the immediate right sibling of the element *f*, denoted as $e'$. Note that if *f* is the root of *T*, then a new root $f'$ is created with *f* as left child and $e'$ as right child. Similarly, *InsLs(S, e, T, f)* is to insert the element e in the schema *S* into the schema *T* as the immediate left sibling of the element *f*; (3) Insert subtrees: *InsRsSub(S, e, T, f)* which is similar to *InsRs* except that now the whole subtree rooted at *e* is inserted as the immediate right sibling of *f* in the schema *T*; and *InsLsSub(S, e, T, f)* is similarly defined for the subtree insertion but as the immediate left sibling; (4) Expand: *Exp(S, X, T, f)* where *X* is a set of elements in *S* and *f* is a leaf in schema *T*. This operation expands the element *f* by transforming it into an internal element $f'$ with the set of elements in *X* as children. The purpose of the expansion has been discussed in Section 3.

In Figure 6, we give the pseudo-code for the process of merging two schemas, $S_i$ and $S_j$ where $G$ is the set of groups computed in the previous step. There are cases when interfaces consist only of fields that belong to clusters whose elements appear always as children of the root. Therefore, the removal of these fields may produce empty schema trees (i.e. they consist of root only). If both schema trees are empty after this operation we return an empty tree. If one of them is not empty and the other is the algorithm returns the nonempty tree.

The main body of the algorithm is executed when both schema trees are not empty. This part starts by choosing one of the two schemas as blueprint or target schema, denoted by $T$, the other schema denoted by $S$, also called source schema. A heuristic is employed for this choice: (1) if the schema trees for two schemas are of different depths, the schema with more levels is chosen as the target schema; (2) for two schemas of the same depth, the one with more leaf nodes is chosen as target schema. If a tie occurs, then we randomly pick one as the target schema.

Next, the algorithm tries to determine for each element *e* in *S*, whether *e* matches with some element *f* in *T*. If yes, *e* will be combined with *f*. Otherwise, *e* will be inserted into *T* at a position such that if $e$ is a leaf then both the ancestor-descendant relationships and the grouping constraints associated with $e$ are preserved; if $e$ is an internal node then only its ancestor-descendant relationships are preserved. As indicated throughout this paper, the relationships among the

```
Algorithm Merge2Alg(S_i, S_j, G) → I
1. if S_i and S_j empty trees then
2.     return empty tree;
3. if S_i empty tree then
4.     return S_j;
5. if S_j empty tree then
6.     return S_i;
7. (T, S) = CHOOSEMERGEDIRECTION(S_i, S_j)
8. l = depth(S)
9. while l > 0, for each element e at level l of S
10.    repeat until no changes
11.      if e is a leaf then /* α */
12.        if e matches f in T then
13.          Cmb(S, e, T, f)
14.        else
15.          f = DETERMINEINSERTLOCATION(e, T, G)
16.          if f not null then
17.            InsLs(S, e, T, f)
18.          else if rmls(e) matches some h in T
19.            InsRs(S, e, T, h)
20.          else if lmrs(e) matches some h in T
21.            InsLs(S, e, T, h)
22.          end if
23.        end if
24.      else /* β */
25.        if all children of e are matched then
26.          W = their matching elements in T
27.          Cmb(S, e, T, lca(W))
28.        else if rmls(e) matches f in T
29.          InsRsSub(S, e, T, f)
30.        else if lmrs(e) matches f in T
31.          InsLsSub(S, e, T, f)
32.        end if
33.      end if
34.    l = l - 1
35. return T as the integrated schema I
```

**Figure 6.** The pairwise merge algorithm

elements within an interface often reflect important semantic information and thus it is important to preserve them as much as possible in the integrated schema.

The algorithm works in a bottom-up fashion, starting from the level farthest from the root of schema *S* and progressing towards the root of *S*. For each element *e* at current level, if the element is a leaf, several sub-cases are further considered. Note that this corresponds to the conditional branch marked with $\alpha$ in the figure. First, if *e* matches with some element *f* in the target schema *T*, it simply combines *e* with *f*. If *e* is an unmatched leaf element we first check to see if there is an element $f$ in $T$ such that $e$ and $f$ are in the same group. This is accomplished by a call to DETERMINEINSERTLOCATION, which takes as input the element $e$, the target schema $T$ and the set of groups. Note that if a group $g$ exists such that $e, f \in g$ this is unique. In addition, if such a group exists the insertion using *InLs* or *InRs* gives only an approximate location as the arrangement

of the elements in $g$ is given by the PQ-tree. It is possible that the target schema does not have any element in the group where $e$ belongs to. In order to insert $e$ we look for the matching behavior among its siblings. If its leftmost right sibling (or rightmost left sibling) $e'$ matches with an element $h$ in $T$, $e$ will be inserted into $T$ as the left sibling (or right sibling) of $h$. If $e'$ is a leaf instead then the potential group containing $e$ and $e'$ induced by $S$ has been split during the partitioning phase. Therefore, $e$ and $e'$ are in different groups now. Note that it is possible that $e$ and none of its siblings is matched. In this case, we proceed with the computation relying on the branch $\beta$ to insert one of the subtrees that contains $e$ into $T$. It is also possible that $e$ might have a matching sibling other than the leftmost right sibling or the rightmost left sibling. But by the repeated application of the algorithm (i.e. $\alpha$ and $\beta$ branches) for the elements in $S$ at the same level with $e$, the leftmost right sibling or the rightmost left sibling will eventually be matched to some element in $T$ and thus $e$ will be inserted into some location of *T*.

If *e* is an internal element (corresponding to the branch marked with $\beta$), then either all of its children have been matched or none of them has been matched, for if at least one child of *e* has been matched, the remaining children will be matched as described above. In such a situation, *e* will be combined with the lowest common ancestor of the matching elements of its children in $T$. This choice is critical to the preservation of ancestor-descendant relationship of elements of *S* in the integrated schema. If no children of *e* is matched, then the whole subtree rooted at *e* will be inserted into *T* if a sibling of *e* has been matched.

**Proposition 4** *The integrated schema computed by the above algorithm (Figure 6) preserves the ancestor-descendant relationships among the elements in the input schemas.*

It should be noted that parent-child relationship cannot be preserved in general. For example, if there is an interface having $A$ as parent of $B$, which is a parent of $C$, and there is another interface where $A$ is a parent of $C$, then the integrated interface would have $A$ as both grandparent and parent of $C$, hence, violating the requirement that an interface is a schema tree. If the parent-child relationship between $A$ and $C$ is removed in the integrated interface, then the ancestor-descendant relationships in the two individual interfaces are preserved in the integrated interface, but the parent-child relationship between $A$ and $C$ is lost.

## 4.2. Post Merge Processing

By applying Merge2Alg algorithm repeatedly, the resulted integrated schema complies with the ancestor-descendant relationships in each of the source schemas. As

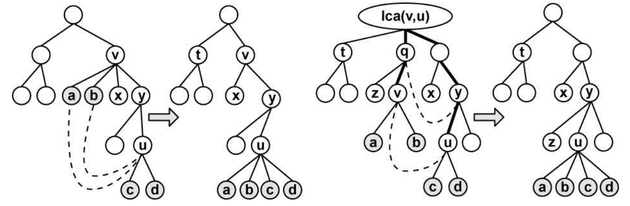


**Figure 7.** Examples of the two possible post merge scenarios.

stated earlier, we aim for an integrated schema that conforms also with the group constraints. It is possible that within the resulted integrated schema certain elements that belong to the same group do not share a common parent. Therefore, we have to perform certain transformations on this schema such that these elements are set under the same parent. We will refer to this schema tree as the *partial integrated schema*. The partial integrated schema, $I$, the set of groups, $G$, and the set of clusters whose fields always occur as children of root nodes in any schema tree, $R$, form the input of the post merge process. At the end of this process the resulted integrated schema, $J$, is guaranteed to satisfy the two main characteristics: the grouping constraints as much as possible and the ancestor-descendant relationships.

First, we identify those groups whose elements do not have the same parent node in I. In our algorithm, for a given group, $g$, let $W$ denote the set of all internal nodes having leaf nodes in $g$. If $W$ has more than one node it implies that the elements of $g$ do not have the same parent. Hence, for each such group the algorithm will place the elements of the group such that they finally have the same parent in the unified interface, while simultaneously maintains the existing ancestor-descendant relationships.

Given two internal nodes $v$ and $u$ in $W$ the algorithm pursues two distinct transformation scenarios based on the existence or not of the ancestor-descendant relationship between the two nodes. Suppose that $v$ is an ancestor of $u$. Then the algorithm simply deletes the leaf nodes of $v$ belonging to $g$ and makes them children of $u$. Suppose that $a$, $b$, $c$, $d$ are in the same group and that the partial integrated schema (the first tree in Figure 7) does not have these elements as children of a single parent. Therefore, we have to redistribute them such that they share a common parent. Since $v$ is an ancestor of $u$ then $a$, $b$ (the children of $v$ in the same group with $c$, $d$) are placed under $u$ (see the second tree in Figure 7). Observe that this transformation preserves the current ancestor-descendant relationships in the tree.

The case when $v$ and $u$ are not in an ancestor-descendant relationship is handled as follows. First, the algorithm retrieves the lowest common ancestor of the two nodes, $w$. Then it computes the paths of $v$ and $u$, respectively, to $w$, denoted in the algorithm by $pv$ and $pu$, respectively. Note that in order to move a certain subset of children of one of them to become children of the other and also to preserve the current ancestor-descendant relationships the key obser-

vation is that all the ancestor nodes of the moved leaf nodes along the path to $w$ have to be placed somewhere on the other path. Therefore, the solution we approach is to merge the nodes of the two paths starting with $v$ and $u$, and stop just before $w$ is first reached. Specifically, between the two paths we pick the longer as the target (or blueprint) and the shorter as the source. Suppose the path of $v$ to $w$, $pv$, is the shorter. Then in a bottom-up manner, starting from $v$ and moving up towards $w$, we merge the nodes on the path $pv$ into the nodes of path $pu$ as follows: first we merge $v$ into $u$, then the parent of $v$ into the parent of $u$ and so forth just before $w$ is reached.

We illustrate this case with the third tree in Figure 7. Consider $a$, $b$, $c$, $d$ to be a set of nodes that have to have a common parent in the integrated schema. Observe that initially the schema tree does not comply with this constraint. Hence, those leaf nodes have to be reorganized such that they will have a common parent. The parent of $a$ and $b$ (i.e. $v$), and that of $c$, $d$ (i.e. $u$) do not have an ancestor-descendant relationship. Therefore, the algorithm will merge the nodes of the paths from $v$ and $u$ to their lowest common ancestor (drawn with a thicker line in Figure 7). First, it merges $v$ into $u$, followed by $q$ into $y$ then it stops since the parent of $q$ is the lowest common ancestor of $v$ and $u$. The fourth tree in Figure 7 shows the new integrated schema tree. It should be stressed that this transformation preserves the ancestor-descendant relationships in the tree, too.

At this point the new integrated schema tree satisfies the ancestor-descendant relationships of the initial schemas and the elements in the same group have a common parent. However, there is one last detail regarding the groups that has to be addressed. Namely, due to the reorganization steps described above we might end up having multiple groups sharing the same parent node. We handle this problem by employing the following strategy: (i) for each internal node, $v$, whose set of leaf nodes span more than one group, if the number of its child non-leaf nodes is at least as large as the number of groups minus one, then separate the elements in distinct groups using these internal nodes else (ii) for each group, $g$, introduce a new internal node, $v_g$, as child of $v$ and make all leaf nodes of $v$ in $g$ children of $v_g$. In both cases the algorithm has to arrange the leaf nodes such that they reflect the permutation of elements within the group.

Finally, for each cluster in $R$ (the set of clusters whose fields always occur as children of root nodes in any schema tree), if any, a leaf node is added to the root of the partial integrated schema tree. Each of these leaf nodes is a representative of the fields of the source schemas that belong to a cluster in $R$.

**Proposition 5** *The output of the postMerge algorithm is an integrated interface that satisfies the ancestor-descendant relationships and the grouping constraints with respect to the potential groups in the individual schemas as much as possible.*

| Domain | Leaf Nodes Avg | Internal Nodes Avg | Depth Avg |
|---|---|---|---|
| **Airline** | 10.7 | 5.1 | 3.6 |
| **Automobile** | 5.1 | 1.7 | 2.4 |
| **Book** | 5.4 | 1.3 | 2.3 |
| **Job** | 4.6 | 1.1 | 2.1 |
| **Real Estate** | 6.7 | 2.4 | 2.7 |

**Table 4.** Characteristics of interfaces per domain.

| Domain | Potential groups | Groups | Violations | Integ. interface | |
|---|---|---|---|---|---|
| | | | | Leaves | Depth |
| **Airline** | 46 | 8 | 2 | 24 | 6 |
| **Automobile** | 22 | 4 | 0 | 18 | 3 |
| **Book** | 34 | 4 | 0 | 19 | 3 |
| **Job** | 12 | 1 | 0 | 19 | 2 |
| **Real Estate** | 47 | 7 | 0 | 28 | 4 |

**Table 5.** The characteristics of the integrated interfaces.

A problem specific to merging query interfaces is that the unified interface may have too many fields to be user-friendly. To remedy this problem fields that are less important (e.g. they can be detected by frequencies of occurrences in the various schemas) can be trimmed from the integrated interface. A detailed solution to the problem has been discussed in [15] and is readily applicable to our framework.

# 5. EXPERIMENTS

In order to evaluate our theoretical results we have performed experiments using the data sets collected for five domains on the Web. The experimental evaluation has two main goals: (1) the extent to which real user interfaces satisfy the grouping constraints and (2) natural and easily understood integrated query interfaces can be constructed from individual interfaces automatically in different domains of interest.

## 5.1. Experiment Setup

We have considered five real world domains on the "deep" Web: airline, automobile, book, job and real estate. A comprehensive characterization of each of these domains is given in [24]. Here, we only mention some of their main characteristics. Each domain consists of 20 query interfaces. These interfaces were collected searching (1) listed sources in profusion.com, which maintains a collection of "invisible" data sources along with their query interfaces and (2) web directory maintained by yahoo.com. After query interfaces were collected they were manually transformed into schema trees. Such trees can also be obtained automatically [25]. Table 4 shows the main characteristics of the query interfaces in each domain.

The input of our algorithm consists of the set of query interfaces and the mapping definition between them, i.e. the set of clusters, which can be obtained with high accuracy

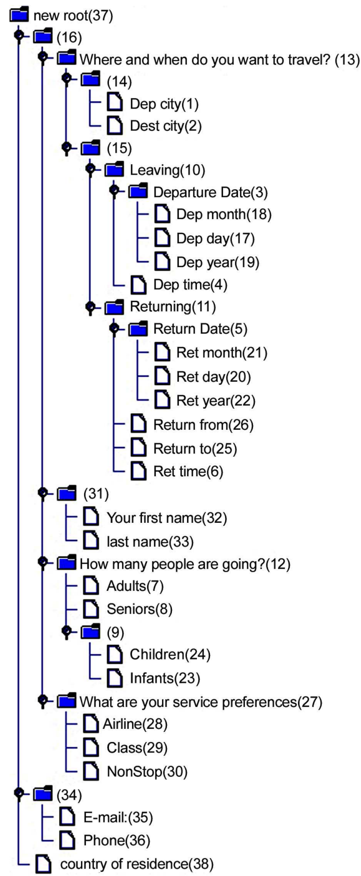


**Figure 8.** Integrated interface for airline domain

[24]. Each query interface is specified in an ASCII file that encodes the hierarchical organization of the interface and various properties of the fields (e.g. name). We use a tree structure to visually display each individual schema. User can browse through all search interfaces that are considered for an experiment. The algorithm displays, for a particular set of user interfaces, the integrated interface as a tree view in a separate panel as shown in Figure 8. The characteristics of the resulted global interfaces are illustrated in Table 5. All constraints given by the potential groups in the five domains are satisfied with the exception of two potential groups in the airline domain. Each of these two potential groups occurs exactly once. The permutation that minimizes the number of violations and the resulting split yields the following two groups [*Seniors*, *Adults*, *Children*, *Infants*] and [*Airline*, *Class*, *NonStop*].

Due to the space limitation we provide here the output of the algorithm for the airline domain only. Figure 8 illustrates the resulted integrated interface before the post merge phase is applied. An analysis of this intermediary global interface reveals some of the problems that we have discovered analytically. That is, observe that this interface does not satisfy the grouping constraints. Specifically, the following groups are violated: [*Dep time*, *Dep day*, *Dep month*, *Dep year*], [*Ret time*, *Ret day*, *Ret month*, *Ret year*] and [*Seniors*, *Adults*, *Children*, *Infants*]. Thus, in the post merge phase, the schema tree has to be reorganized using only the transformations of the type depicted in Figure 7. The results of these transformations are: (1) the field *Dep time* is relocated to become a child of *Departure Date*, forming the group [*Dep time*, *Dep day*, *Dep month*, *Dep year*], (2) the field *Ret time* is relocated to become a child of *Return Date*, forming the group [*Ret time*, *Ret day*, *Ret month*, *Ret year*], and (3) the fields *Adults* and *Seniors* are moved to form the group consisting of [*Seniors*, *Adults*, *Children*, *Infants*]. The final integrated query interface is natural in the sense that the user will have no difficulty in understanding it and provide the necessary information. The integrated interfaces for the other four domains are just as natural. Thus, our algorithm constructs natural useful user interfaces as if they were created manually.

# 6. RELATED WORK

We mention here only those papers which are most closely related to our work in this paper. The quality of the integrated user interface produced by the algorithm described in this paper relies on the effectiveness of works in query interface matching [12, 13, 15, 24].

A grouping pattern among the attributes of query interfaces in a given domain of interest has been studied in [13] and [25]. The former paper focuses on the identification of complex matchings of fields across schemas, while the latter paper is concerned with the extraction of schema trees. The grouping properties in these two works are different from the one introduced in this paper. Specifically, a group in our sense may not correspond to any group as defined in [13, 25]. Furthermore, the grouping concept in [13, 25] is not used for schema merging.

Certain problems of merging query interfaces are also studied in [15]. However, [15] models interfaces as flat schemas and does not utilize the grouping constraints, the sibling, and the hierarchical structure of the interfaces, which turn out to be valuable resources in accurately computing an integrated query interface.

Theoretical studies of merge activity as a generic process over any class of data models are given in [5, 21, 19]. Model merging, in general, has received a lot of attention in various applications and contexts. Most of the recent work on model merging has been done in the context of ontology merging and alignment [1, 20, 21]. These algorithms have been tested against large pairs of ontologies, but none of them has reported their scalability to merging multiple ontologies. Rondo [18] is a model management system prototype that includes a merge algorithm. This algorithm is restricted to equality mappings, therefore it does not handle 1:m relationships as we do through the expand operation. In

addition, the algorithm is not guaranteed to preserve any of the properties discussed in this paper. Other relevant works on the problem of schema merging are studied it in the context of ER data model [22] and semi-structured data [2].

The concept of field grouping, which is central in this paper, is absent from all the works mentioned above.

## 7. SUMMARY & FUTURE WORK

Our contribution is an automatic approach to the problem of merging large-scale collections of query interfaces of the same domain. Toward this goal we have introduced a novel abstraction (i.e. *potential groups*) to capture the tendency of semantic grouping of the fields in query interfaces. The constraints imposed by the partial groups are used to form groups in the integrated user interface. We provide an algorithm which transforms a set of interfaces in the same domain of interest into a global interface such that all ancestor-descendant relationships are preserved and the grouping constraints are satisfied as much as possible. To the best of our knowledge, this is the first piece of work which makes such a guarantee for interface merging. Experiments are performed in five domains to demonstrate that the integrated interface produced by our algorithm in each of these domains is natural. There is substantial work to be done. First, the names to be assigned to the nodes in the integrated schema tree have to be done automatically. Second, our work needs to be generalized in order to be applicable to other data models, such as XML and ontology.

## References


[1] D. Beneventano, S. Bergamaschi, F. Guerra, and M. Vincini. The momis approach to information integration. In *IEEE and AAAI Conference on Enterprise Information Systems*, 2001.

[2] S. Bergamaschi, S. Castano, and M. Vincini. Semantic Integration of Semistructured and Structured Data Sources. *SIGMOD Record*, 1999.

[3] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *SIAM Journal of Computing*, (12):335–379, 1976.

[4] BrightPlanet.com. The deep web: Surfacing hidden value. In *http://brightplanet.com*, 2000.

[5] P. Buneman, S. Davidson, and A. Kosky. Theoretical aspects of schema merging. In *EDBT*, 1992.

[6] K. Chang, B. He, C. Li, M. Patel, and Z. Zhang. Structured databases on the web: Observations and implications. In *SIGMOD Record*, 2004.

[7] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering Complex Mappings between Database Schemas. In *SIGMOD*, 2004.

[8] Hong Hai Do and E. Rahm. COMA - A System for Flexible Combination of Schema Matching Approaches. In *VLDB*, pages 610–621, 2002.

[9] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach . In *SIGMOD*, 2001.

[10] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to Map between Ontologies on the Semantic Web. In *WWW*, 2002.

[11] D. Fulkerson and O. Gross. Incidence matrices and interval graphs. *Pacific J. Math*, 1965.

[12] B. He, K. Chang, and J. Han. Statistical schema matching across web query interfaces. In *SIGMOD*, pages 217–228, 2003.

[13] B. He, K. Chang, and J. Han. Discovering complex matchings across web query interfaces: A correlation mining approach. In *SIGKDD*, 2004.

[14] B. He, T. Tao, and K. Chang. Clustering structured web sources: A schema-based, model-differentiation approach. In *EDBT04 ClustWeb Workshop*, 2004.

[15] H. He, W. Meng, C. Yu, and Z. Wu. WISE-integrator: An automatic integrator of Web search interfaces for e-commerce. In *VLDB*, 2003.

[16] J. Madhavan, P. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *VLDB*, 2001.

[17] J. Meidanis, O. Porto, and G. P. Telles. On the consecutive ones property. In *Discrete Applied Mathematics*, number 88, pages 325–354, 1998.

[18] S. Melnik, E. Rahm, and P. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *SIGMOD*, pages 193–204, 2003.

[19] R. Miller, Y. Ioannidis, and R. Ramakrishnan. Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice. *Information Systems*, 19(1):3–31, January 1994.

[20] N. Noy and M. Musen. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In *AAAI/IAAI*, pages 450–455, 2000.

[21] R. Pottinger and P. Bernstein. Merging Models Based on Given Correspondences. In *VLDB*, 2003.

[22] S. Spaccapietra and C. Parent. View integration: A step forward in solving structural conflicts. *TKDE*, 6(2), 1994.

[23] J. Ullman. Information Integration Using Logical Views. In *ICDT*, 1997.

[24] W. Wu, C. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *SIGMOD*, 2004.

[25] Z. Zhang, B. He, and K. Chang. Understanding web query interfaces: best-effort parsing with hidden syntax. In *SIGMOD*, pages 107–118, 2004.