Automatically Mapping and Integrating Multiple Data Entry Forms into a Database

Yuan An, Ritu Khare, Il-Yeol Song, and Xiaohua Hu

College of Information Science and Technology, Drexel University, USA
{yan,ritu,isong,thu}@ischool.drexel.edu

Abstract. Forms are a standard way of gathering data into a database. Many applications need to support multiple users with evolving data gathering requirements. It is desirable to automatically link dynamic forms to the back-end database. We have developed the FormMapper system, a fully automatic solution that accepts user-created data entry forms, and maps and integrates them into an existing database in the same domain. The solution comprises of two components: *tree extraction* and *form integration*. The tree extraction component leverages a probabilistic process, Hidden Markov Model (HMM), for automatically extracting a semantic tree structure of a form. In the form integration component, we develop a merging procedure that maps and integrates a tree into an existing database and extends the database with desired properties. We conducted experiments evaluating the performance of the system on several large databases designed from a number of complex forms. Our experimental results show that the FormMapper system is promising: It generated databases that are highly similar (87% overlapped) to those generated by the human experts, given the same set of forms.

1 Introduction

Using forms as the front-end interface mapping to a back-end database is a standard way for data collection. For building form-based applications, Do-It-Yourself (DIY) [18] and What You See Is What You Get (WYSIWYG) [28] are popular paradigms in the Cloud or in supporting non-technical users. Applications in these paradigms automatically translate forms into underlying databases and shield users from the technical details for database creation and code generation. There are a number of online systems that provide commercial services for users to create custom forms on their own. Example systems include Formassembly [1], Zoho [4], Jotform [2], and Wufoo [3]. In a previous study [16], we also developed a system that allows non-technical clinicians to create high-quality databases through forms. Despite the effort, we find that *mapping and integrating multiple forms into an existing structured database* remains largely unexplored. We call the problem ***form2db*** problem which is the focus of this paper. The following example illustrates the problem.

Example 1. Figure 1 (a) shows an existing form and an associated back-end database. The application maintains a mapping between the form and the back-end database.

Suppose that the form in Figure 1 (b) is a newly created one and will be used to collect data into the same database. It may be required to link the elements on the

M. Jeusfeld, L. Delcambre, and T.W. Ling (Eds.): ER 2011, LNCS 6998, pp. 261–274, 2011.

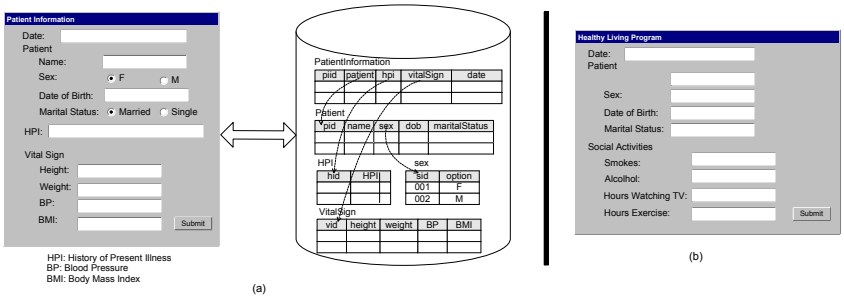

Fig. 1. Data Entry Forms and the Associated Database

form to elements in the database. For example, if the patient information is linked between the form and the database, then patient information could be filled up through auto-completion and integrated automatically. To map and integrate the new form to the existing database, a technical developer would first manually link the Name, Sex, Date of Birth, and Marital Status items on the form to the existing Patient table in the database. He/she would then extent the existing database appropriately to collect the new data items under Social Activities on the form in Figure 1 (b).

In such a manual process, the technical developer needs to directly access the database system and write the application code. However, some applications may not provide such an environment to users, for example, creating databases in the Cloud or supporting non-technical users to create forms on their own. In these applications, either **the database system is not directly exposed to the user** or **the user does not possess the necessary technical skills for generating the code**. It is desirable to develop an automatic approach that could map and integrate forms into an existing database without user intervention. Such an approach would greatly reduce human effort in system building and would benefit to many applications. ■

We have developed the FormMapper system, a fully automatic solution that accepts user-created data entry forms, and maps and integrates them into an existing database. The FormMapper system is built on our previous work [15,16] and extends it in many aspects: First, the FormMapper integrates multiple forms into a single database instead of creating individual database for each form. Second, we have attempted to implement the following requirements in the FormMapper: (i) the system can accept sophisticated forms as input, (ii) the system automatically captures the semantic relationships among form elements, (iii) the system automatically links form elements to the elements in the hidden database, (iv) the system automatically extends the hidden database for unmatched form elements, and (v) the system automatically generates mapping expressions between the form and the hidden database.

There are many challenges in developing such an automatic system. To address the challenges, we first propose a formal hierarchical model that represents the semantic relationships among elements of a form; then we develop a solution comprising of two components: a *tree extraction* component and a *form integration* component. The tree extraction component leverages a machine learning technique, Hidden Markov Model (HMM) [25], for automatically extracting a tree structure from a data entry form. In a previous study, we have applied the HMM model to the problem of segmenting search

interfaces [15]. To address the *form2db* problem, we extend the method to automatically extract a complete tree structure from a form. In the form integration component, we develop a merging procedure that maps and integrates a form tree into an existing database and extends the database as needed. Specifically, the merging procedure includes a *birthing algorithm* that creates a relational database from a form tree and a *megring algorithm* that integrates the form tree into an existing database.

In summary, we make the following contributions in this paper:

1. We identify the *form2db* problem and study related issues in developing a fully automatic approach.
2. We develop an initial solution which leverages an advanced machine learning technique for form tree extraction and takes the classic database design principles into consideration for database merging and extension.
3. We implemented the solution and conducted experimental studies. Empirically, the results generated by the system are highly similar to expert designed databases (87% overlapped). Overall, the automated system saves a great deal of human effort (seconds vs. hours) while generating results that are comparable to what human experts produced.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the formal models and the problem definition. Section 4 describes the tree extraction component. Section 5 studies the merging process. Section 6 presents the experiments. Finally, Section 7 concludes the paper.

2 Related Work

There is a broad array of related work including *form-driven database generation* [11], *DIY web services for form creation* [1] *schema mapping* [20], *view integration* [7], *schema evolution* [26], and *automatic understanding of Deep Web search interfaces* [17]. However, the problem that we address here is a new and has not been addressed before in totality. An automatic solution to the problem including its subproblems has not been developed yet.

The earlier form-driven database creation techniques were designed for IT professionals to develop databases using "form structures;" users have to specify the exact semantics of the underlying database schemas through "forms" [11,22,19]. Recent DIY web services for form creation can hide the underlying data storage details from the users. However, each form is stored individually without semantic integration [1,28].

Our problem is different from schema mapping and integration in several aspects. Schema mapping aims to discover meaningful relationships between database schemas [20,21,23,5], while view/schema integration builds a database schema by unifying a set of views/schemas [7]. Both problems primarily focus on schema elements and do not consider database extension. Also, solutions to schema mapping and view integration are semi-automatic, requiring the user to examine the final results. Our problem is a continuous and dynamic process which does not ask the user to examine the results. Schema evolution focuses on maintaining the consistency of a database when its schema is modified [26]. Our problem focuses on how to extend a database when new data needs to be collected.

There are techniques proposed for modeling and understanding search interfaces [17]. In particular, Wu et al. [27] and Dragut et al. [12] model a search interface as a hierarchical structure call *schema tree*. To extract schema trees, Dragut et al. in [12] develop a rule-based approach exploiting Web browser rendering. We develop a model-based machine learning technique for extracting a tree structure from a data entry form. Our approach does not reply on specific browser rendering and extracts information from sophisticated form structures.

3 Formal Preliminaries and Problem Definition

A form consists of a collection of form elements laid out in a particular way. Example form elements include text label, text box, radio buttons, select list, and check boxes. The underlying relationships among the elements are naturally captured by a tree structure [27,12]. Here, we formally define a *form tree* as follows.

Definition 1 (Form Tree). *A form tree is defined as a labeled, directed and ordered tree, $\mathcal{FT} = (N, E, <_{sib}, root)$, where $N = \mathcal{I} \cup \mathcal{E} \cup \mathcal{V}$ is a finite set of nodes, E is a finite set of edges, $<_{sib}$ is the next-sibling relation between children of a tree node, and $root \in N$ is the root of the tree. Moreover,*

- *$\mathcal{I}$ is a finite set of input elements (or inputs), where items of $\mathcal{I}$ are drawn from the following set of input fields: {text box, text area, radio buttons, check boxes, drop-down list, calendar} ;*
- *$\mathcal{E}$ is a finite set of logical elements. Each $e \in \mathcal{E}$ has a label $l = \lambda(e)$, a data type $t = \tau(e)$, and a constraint $k = \kappa(e)$, where the function $\lambda(e)$ returns the label l of e, the function $\tau(n)$ returns the data type t of e, and the function $\kappa(e)$ returns the constraint k of e.*
- *$\mathcal{V}$ is a finite set of values;*
- *For an edge $(n_i \longrightarrow n_j) \in E$, $n_i, n_j \in N$, n_i is called* **parent** *and n_j is called* **child**.

Figure 2 shows the form tree model of the form in Figure 1 (a). Graphically, we use different shapes to represent the different types of nodes. The data type of an element can be extracted from the source code of a form, for example, Date for calendar input. Furthermore, the source code of a form often provides constraint information about whether an input is required or optional.

Database. A (relational) database $\mathcal{D} = (I, R, \Sigma)$ is a 3-tuple, where I is a set of relational tables, R is the schemas of the tables, and Σ is a set of integrity constraints imposed on the tables.

Definition 2 (The *form2db* Problem). *Given a set of data entry forms $\{\mathcal{F}_i, i = 1..n\}$ and a relational database $\mathcal{D} = (I, R, \Sigma)$, the* form2db *problem is to (1) discover the semantic mappings between the elements in a from $\mathcal{F}_i, i = 1...n$ and the elements in the database $\mathcal{D}$ and (2) extend the database to cover the unmapped elements in the form. The following is a set of desired properties for a solution:*

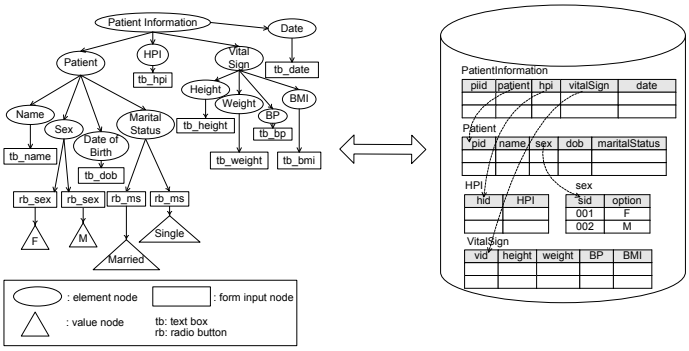

Fig. 2. The Form Tree for the Patient Information Form

- **correctness:** *a mapping linking a form element to a data element indicates that both elements represent the same concept in the application domain.*
- **completeness:** *every form element is mapped to a database element.*
- **compactness:** *every form element is mapped to at most one database element.*
- **normalization:** *the extended database should respect the normalization principles with regard to the functional dependencies that can be identified on the forms.*
- **optimization:** *the extended database should be optimized in terms of query efficiency (e.g., minimizing joins) and storage (e.g., minimizing NULL values.)*

For multiple forms $\{\mathcal{F}_i, i = 1..n\}$, we consider the problem of mapping and integrating them in a sequential order $\mathcal{F}_1$, $\mathcal{F}_2$, ..., $\mathcal{F}_n$. An individual form $\mathcal{F}_i$ is mapped and integrated to a database $\mathcal{D}$ that has already integrated the forms $\mathcal{F}_1$ to $\mathcal{F}_{i-1}$. With this consideration, in the rest of the paper, we focus on developing methods for mapping and integrating a single form to an existing database. Our solution first extracts a form tree from a form, and then maps and integrates the tree to an existing database.

4 Extracting Form Trees

Tree extraction is the process of clustering the elements on a form into semantically related groups and organizing the groups into parent-child hierarchical relationships. We have developed an effective and efficient method for segmenting form elements into semantically related groups [15]. To address the *form2db* problem, we extend the previous segmentation technique to a tree extraction method. The core of the method is a probabilistic process, Hidden Markov Model (HMM), that simulates the process of designing a data entry form. In general, an HMM [25] can model a probabilistic process that generates a sequence of observations. Particularly, an HMM consists of a set of states that satisfy the Markov assumption, i.e., the state of the process only depends on the previous states of the process. In a first order Markov model, which we assume for our problem, a current state only depends on the previous state and not on any earlier states. At each state, e.g., corresponding to an object in a domain conceptual model, the process generates an observable value, e.g., a form element, based on a fixed

Table 1. Observation Space T_HMM

Code	Description
σ_0^T	textbox/textarea
σ_1^T	select group
σ_2^T	radiobutton
σ_3^T	checkelement
σ_4^T	long text(more than 4 words)
σ_5^T	lower case non-colon-ending multi-character text
σ_6^T	colon ending text
σ_7^T	special character text
σ_8^T	uppercase text
σ_9^T	uppercase colon-ending text
σ_{10}^T	paranthesized text

probability distribution. In an HMM, an observer can observe a sequence of observable values emitted by a (first order) Markov process, but cannot see which states emit which values. An important inference problem is: *given a sequence of observed values, to find the sequence of states that is most likely to have generated those observations.*

We model the process of designing forms as a Markov process where a designer creates form elements when moving from object to object in a conceptual model following the relationship links. An object in a conceptual model could be an entity, attribute, relationship, or value. At each object, the designer decides whether to create a logical element, an input field, a nested element, etc. We simulate this design process into 2-layered HMM. The first layer T_HMM is aimed to tag the elements of the forms (observations) with their semantic roles (states), and the second layer S_HMM is used to segment the forms into groups (and sub-groups) of elements. Formally, an HMM is specified as a 3-tuple (Π, A, B) over a set of states $Q = \{q_i, i = 1..n\}$ and a set of observable values $\Sigma = \{\sigma_j, j = 1..m\}$, where $\Pi = \{\pi_i\}$, $\pi_i = Pr(q_i)$, is a vector of initial state probability, $A = \{a_{ij}\}$, $a_{ij} = Pr(q_j|q_i)$, is a matrix of transition probability among the states, and $B = \{b_{ij}\}$, $b_{ij} = Pr(\sigma_i|q_j)$, is a matrix of emission probability between states and observations. Tables 1 presents the observable values of the T_HMM, while Table 2 and Table 3 describe the states of T_HMM and S_HMM. In the 2-layered HMM, the states of T_HMM also become the observable values of the S_HMM.

In the tagging phase, the T_HMM automatically tag the elements of a form with states q_0^T - q_6^T. These states indicate whether a form element is labeled as an Entity, Attribute, or Value. In addition, a form may contain other types of elements such as Value Separator, Value Data Type, Instructions, and Misc. Text. For example, a Value Separator is a piece of text with special characters. A Value Separator separates two input fields which collect data for the same attribute. For instance, blood pressure is often collected on a patient encounter form using `BP:` [ ]/[ ], where the text labels "`BP`" and "`/`" are followed by text boxes, respectively. The text "`/`" is a Value Separator rather than an attribute. In the next grouping phase, the S_HMM accepts the output of the T_HMM and assigns new states q_0^S - q_3^S. These states indicate which set of elements should be grouped into a segment.

Table 2. State Space T_HMM (Also Observation Space S_HMM)

Code	Description
q_0^T	Entity
q_1^T	Attribute
q_2^T	Value
q_3^T	Value Separator
q_4^T	Value Data Type
q_5^T	Instructions
q_6^T	Misc. Text

Table 3. State Space S_HMM

Code	Description
q_0^S	Begins a segment
q_1^S	Inside a segment
q_2^S	Begins a subsegment
q_3^S	Inside a subsegment

After a form is tagged and segmented, the tree is derived by the segmentation information. The topology of nodes within a branched segment is determined based on the semantic tags. The tree branching structure is determined in the following manner. Each segment is represented by a tree node. The root is the container segment that represents the entire form. A sub-segment within a segment becomes the child of the node represented by the segment. After the initial branching, each segment node is elaborated into a segment tree based on the semantic tags associated with the segment elements using the following rules.

- The entity becomes the root of the segment tree.
- A attribute node becomes the child of the segment root
- The value nodes associated with a given attribute node become the children of the attribute node.
- Some value nodes (radio, check, select) may need to be extended to contain the value text nodes as their children.
- A subsegment becomes the child of the root of the container segment.

The main tasks for the tree extraction include learning the parameters, Π, A, B, of the model by acquiring training examples, and applying the model to new instances. We train the HMMs by the Expectation Maximization algorithm [25] using a set of manually labeled examples. We solve the inference problem of finding the most likely explanation by the Viterbi algorithms [25]. In the Experiments section, we show that the accuracy of extraction is 96% based on a number of example forms we collected and manually labeled.

5 Mapping and Integrating Form Trees

Given a form tree $\mathcal{FT}$ and a database $\mathcal{D}$, our solution to the *form2db* problem first discovers initial element correspondences (or matchings) between the atomic elements in the tree and the database. Then it integrates the form into the database by discovering valid correspondences and extending the database. For element correspondences, we employ an element matching function $\delta(\mathcal{FT}, \mathcal{D})$ (e.g., a schema matching program [8]) which returns a set of correspondences $\mathcal{M} = \{\mathcal{FT}{:}\mathsf{P/e} \rightsquigarrow \mathcal{D}{:}\mathsf{d}\}$, which relates an

atomic element P/e reached by a simple path P in the form tree with an atomic element d in the database. The element matching function $\delta(\mathcal{FT}, \mathcal{D})$ returns a similarity (≥ 0 and ≤ 1) between every pair of elements in the tree and the database. We set up a high threshold value (e.g., 0.99) and take all pairs of elements that have similarity above the threshold as an initial set of correspondences.

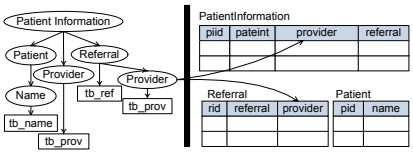

Fig. 3. Ambiguous Correspondences

Having the initial set of element correspondences, the merging process needs to address two challenges: First, there may be one-to-many or many-to-one initial correspondences. For example, Figure 3 shows a case where the form element Provider is linked to two different columns in the database. The merging process needs to resolve the ambiguities by cleaning up the set of initial correspondences. Second, in general, there could be several modeling alternatives for the same concept. For example, the relationship Patient-----Provider could be modeled as Patient-----Hospital-----Provider in a database. Consequently, an automatic method would not be able to resolve the heterogeneity without human expert's intervention.

To develop a fully automatic approach, we take a *syntactic and structural* approach by assuming there is a well-defined forward engineering method for creating databases from form trees. Consequently, the mapping part in our solution is to discover "equivalent" structures between two databases generated by the forward engineering method; and the extension part is to merge two databases generated by the forward engineering method. By "equivalent", we mean the syntactically same. In the following sections, we first present a forward engineering method – the birthing algorithm. Next, we describe a merging algorithm that integrates a database generated by the birthing algorithm into an existing database. Overall, we develop our merging process into the following steps:

1. Discover a set of initial element correspondences.
2. Derive a new database from the given form tree.
3. Clean up the initial correspondences.
4. Extend the existing database for the unmapped elements.

5.1 Birthing Algorithm

The birthing algorithm takes as input a form tree and creates new database tables. Consider a form tree $\mathcal{FT} = (N, E, <_{sib}, root)$ as a conceptual model. An internal logical element $e \in \mathcal{E}$ corresponds to an entity and an edge between two logical elements $(n_i \longrightarrow n_j) \in E$, $n_i, n_j \in \mathcal{E}$ corresponds to a relationship in an ER model. The birthing algorithm creates tables corresponding to logical elements and associated edges. However, a form tree is different from a traditional ER model in many ways. A form tree contains *input fields* and *values* which can be organized in complex and messy ways. Figure 4 shows a complex data-entry form in which an input check box with a value, e.g., Obesity, is extended by another input text box. Moreover, the hierarchical relationships between form elements capture important semantics regarding

the information collected on the form. For example, the hierarchy Health Problems – Diagnosed – Depression indicates that the value Depression is a diagnosed problem, rather than a concerned symptom.

To achieve largely the desired properties specified in Definition 2, we consider the following requirements when developing the birthing algorithm:
Requirement 1: *All the user-specified elements and values should be captured in the database.*
Requirement 2: *All the hierarchical relationships among logical and value elements should be maintained in the database.*
Requirement 3: *Requirements 1 and 2 considered, tables should be merged for query efficiency.*

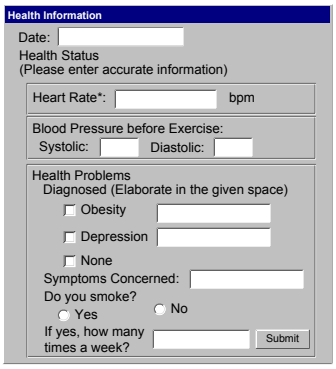

Fig. 4. A Complex Form

A form tree is pre-processed for extracting the data type $\tau(e)$ and identifying constraint $\kappa(e)$ of an element e. For example, the element Heart Rate is a required field (i.e., NOT NULL) and has a data type bpm in Figure 4. In addition, we also consider extension of an input filed, e.g. the check box *Obesity* extended by a text box. The birthing algorithm creates a relational database in a top-down fashion. In particular, the procedure starts with the root of the original form tree and implements the key patterns illustrated in Figure 5 (the full implementation catches all the rare exceptions.)
Pattern (a): The root of the form tree is mapped to a table representing an n-ary relationship.
Pattern (b): If n's children are text boxes and n has a parent, then n is mapped to a column named after n in the parent table. The text boxes are concatenated and mapped to the values of the column.
Pattern (c): An edge between two logical elements (except for Patterns (d) and (e)) is mapped to a many-to-many relationship table connecting two tables corresponding to the two elements.
Pattern (d): The values of a set of radio button are stored in the database as a lookup table. The logical element covering a set of radio buttons is linked by a functional relationship from the parent logical element.
Pattern (e): The values of a set of check boxes are stored in the database as a lookup table. The logical element covering a set of check boxes is linked by a many-to-many relationship from the parent logical element.

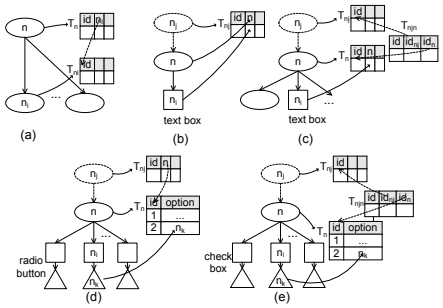

Fig. 5. Mapping Form Tree Patterns to Database

An extension field is added as an extra column of the table referencing the extended field. To meet the **Requirements 1 and 3**, the procedure works as follows: (1) It stores values in individual lookup tables; (2) It merges root's children to an n-ary relationship

table; and (3) It inlines an element with only text box children to the element's parent. To verify **Requirement 2**, we turn a relational database into a *database graph* where the nodes are tables, columns, and values; and edges are table-column, column-value, table-table (foreign key referencing) relationships. For a foreign key column, we replace the table-column relationship with a referencing relationship between two tables. Let $\mathcal{FT}' = (N', E', <'_{sib}, root)$ be the tree that is obtained from the original form tree by removing *input nodes* and reconnecting value nodes to logical nodes. Then, for an edge $(n_i \longrightarrow n_j) \in E'$ in the tree $\mathcal{FT}'$, there are corresponding two nodes, v_i and v_j, in the database graph, such that, either v_i and v_j are linked by an edge or they are linked by a path passing through an intermediate node v_k, where v_k is a many-to-many relationship table.

5.2 Merging Algorithm

The merging algorithm takes as input a new database $\mathcal{D}'$, an existing database $\mathcal{D}$, and a set $\mathcal{M}$ of correspondences between $\mathcal{D}'$ and $\mathcal{D}$. It aims to merge the two databases. Traditional schema integration [9,24] finds a unified representation from a set of source schemas. However, existing approaches to schema integration require substantial amount of human feedback during the integration process. It has been shown that there can be multiple possible schemas that integrate data in different ways and each may be valuable in a given scenario [10]. In our case, we focus on developing a fully automatic solution that only uses the structural information for merging. We assume that the existing database was also generated by the birthing algorithm. The method first cleans up the correspondences by checking whether the linked two elements are compatible in terms of types and constraints (e.g., NOT NULL). It then extends the existing database by comparing two partially matched databases and adding the missing elements in the existing one. *The current method considers different structures as having different semantics even though the structures may model the same concepts.* (We will develop an ontology-based approach for fully automatically resolving semantic heterogeneity in the future work.)

Cleaning up Correspondences. Besides ambiguous correspondences, an invalid correspondence may link two elements that are incompatible in terms of types or constraints. We develop a procedure cleanup($\mathcal{D}'$, $\mathcal{D}$, $\mathcal{M}$) that removes invalid correspondences. For each correspondence, the procedure first ensures that the pairs of elements linked by the correspondence are compatible in terms of their types, i.e., tables are linked to tables, columns are linked to columns, and values are linked to values. Second, for a correspondence linking two columns, the procedure checks their data types if available. Third, for two foreign keys linked by a correspondence, the procedure ensures the referenced tables are linked by a correspondence. Finally, the procedure removes correspondences that cannot be used to insert data due to relevant NOT NULL constraints. For example, if a correspondence $\mathcal{FT}$:P/e $\rightsquigarrow$ $\mathcal{D}$:T.c_i links the element P/e (P is a simple path in the tree) to the column c_i of the table T(c_1,..., c_i, ...c_j,..., c_m). If the column c_j has a constraint NOT NULL and there are no other correspondence from the form tree to the column c_j, then the correspondence $\mathcal{FT}$:P/e $\rightsquigarrow$ $\mathcal{D}$:T.c_i is prevented by the NOT NULL constraint of c_j, because a tuple that only contains the data about P/e cannot be inserted in the table T.

Extending Database. Let $\mathcal{M}'$ be the set of correspondences returned by the procedure cleanup($\mathcal{D}'$, $\mathcal{D}$, $\mathcal{M}$). If there is an element $d \in \mathcal{D}'$ that is not covered by any correspondences in $\mathcal{M}'$, we need to extend $\mathcal{D}$ to accommodate d.

We develop a procedure extendDB($\mathcal{D}'$, $\mathcal{D}$, $\mathcal{M}'$) which returns an extended database and new mappings. The issues we consider in extending database are optimizations for data storage and query processing (as proposed in Definition 2.) It is desirable to minimize the number of columns that have NULL values. However, adding a column to an existing table may bring in a lot of NULL values to the table. Consider two tables $T'(a_1, a_2, ..., a_n) \in \mathcal{D}'$ and $T(b_1, b_2, ..., b_m) \in \mathcal{D}$ such that $T' \rightsquigarrow T$ and there are h correspondences between the columns of T' and T, that is, $a_{i_1} \rightsquigarrow b_{j_1}$, $a_{i_2} \rightsquigarrow b_{j_2}$, ..., and $a_{i_h} \rightsquigarrow b_{j_h}$. If $m > h$ or/and $n > h$, then there are dangling columns in the tables that are not matched. If we add a new column to T, then all the existing tuples as well as all the new tuples inserted by other forms will have NULL values under the column. However, if we create a new table for new columns in order to reduce the number of NULL values, the number of tables is increased. To balance the trade-off between reducing the number of tables (joins) and reducing NULL values, two of the fundamental concerns in database design, we employ a user-defined *quality tuning factor*(qf) $(0 \leq qf \leq 1)$. qf = 0 indicates a high preference to reducing NULL values, while qf = 1 indicates a high preference to reducing the number of tables. When merging two tables, we compare the tuning factor with a numeric metric *null value ratio* (nvr) to make decision. The *null value ratio* ($nvr = \frac{(m-h)+(n-h)}{h}$) reflects the possibility of having NULL value columns if the two tables are merged together. If the nvr is lower than qf, we merge the two tables, otherwise, we create a new table.

6 Experiments

We conducted experiments on evaluating the performance of the FormMapper system. The goal of the experiments is two-fold: (1) to evaluate the effectiveness of the tree extraction component and (2) to evaluate whether the merging process can generate "good" results while dynamically and continuously mapping and integrating user-created forms.

6.1 Testing the Tree Extraction Component

Testing tree extraction consists of two parts: (1) training the HMM models and (2) applying the models to test data. We collected 52 data-entry forms in the healthcare domain and manually labeled them with semantic tags as training examples. Because the limited number of examples, we conducted a k-fold cross-validation for training and testing. That is, we divided the entire set into k sets (folds). For each fold, we trained the model on the other folds and test on the fold. The smallest form has 50 elements and the largest form has 183 elements. The average size of form happens to be 100. For measuring the effectiveness, we define the *extraction accuracy* as the percentage of the correctly extracted patient-child relationships. Our experiments on the 52 examples showed the average accuracy was 96%. The average time for generating a tree structure was within one second.

6.2 Evaluating the Merging Process

We conducted comparative experiments on evaluating the performance of the merging process. We used sets of complex and overlapping forms to incrementally build databases in the FormMapper system. We manually examined the resultant databases and compared them with "gold standard" databases created by human experts. The experimentation was carried out as follows. For a set of forms, we put the forms in a random order and applied the system to them one by one. We started with an empty database. The system generated a new database from the first form and sequentially integrated other forms. We were concerned with the following two questions: (i) did the resultant database have the desired properties? and (ii) was the resultant database comparable to the "gold standard" databases?

Datasets and Benchmarks. We selected 3 sets with total 16 forms (among the 52 forms) belonging to 3 different healthcare institutions. The average form size is 61. We asked two human experts to manually generate databases from the 3 sets of forms. These manually generated databases serve as "gold standard" for comparison. Both experts have more than 10 years of experience in designing databases for various enterprises. We were informed that the experts required *several hours* to prepare the "gold standard".

Implementation and Experiment Setting. The system is implemented in a IBM x3400 server with 8 GB memory. In all the experiments, the *quality tuning factor* qf (defined in Section 5.2) which is used in the database extension process is set to 0.5. For discovering element correspondences, we developed our own solution by exploiting linguistic and structural information.

Experimental Results. First of all, the system took on average 3 seconds to create a new database or integrate a form to an existing database. On an average, the databases contain 56 tables, 172 columns, 98 values, and 61 foreign key references. Next, we manually examined the resultant databases and compared them with the "gold standard" databases for evaluating desired properties and quality. Our examination showed that all the resultant databases are *complete*. In addition, the system built *normalized* databases based on the functional dependences deduced from the many-to-one relationships. Figure 6 shows a summary of the comparison between the system generated databases and the "gold standards" in terms of the total number of database elements.

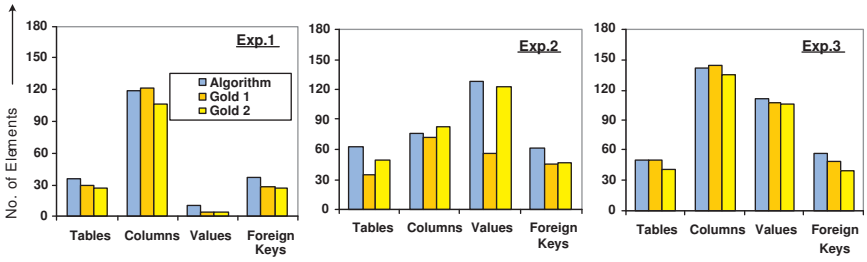

Fig. 6. Total Elements - Algorithm Vs Gold

On an average, 87% of system generated databases are considered to be "matched" to the "gold standard" databases.

We are more interested in the discrepancies. A small portion of the discrepancies is due to missing correspondences. For discrepancies under the same set of correspondences, we analyzed the results and identified 7 form patterns where the result differed between the algorithm and the gold standards. We found that the algorithm resulted in a superior result in 3 out of the 7 cases, and also both the gold standards resulted in superior results in 3 cases each. For the superior cases in the "gold standards", we found that they were mainly due to human judgement based on a personal understanding of the domain semantics. For example, a human expert extracted many-to-one relationships from the category-subcategory relationships on forms based on personal domain knowledge. Considering the algorithms generate databases with "good" properties by only taking as input the forms, we would conclude that the algorithms are promising and have the potential to replace human developers.

7 Conclusions

Jagadish et al. recently illustrated 5 painful issues in database usability [13]. Among them, the birthing pain is related to the difficulties of creating a database and putting information into a database. We are motivated to study an easy and flexible way for users to use a database for storing information. Forms are a user-friendly way for interacting with databases. We develop a solution for automatically generating databases from data-entry forms. In terms of form creation, our problem is an inverse process of that studied in [14], where query forms are automatically generated from databases. For our problem, we show that with the fully automatic solution, users do not need a clear knowledge of the final structure of a database. As users create more forms for evolving needs, the structure of the database grows automatically, however, in a principled way with predictive characteristics. The limitations of our approach include the missing correspondences and syntactic nature of the merging process. In future, we intend to exploit more sophisticated schema matching techniques [6] and apply an ontology to resolving semantic heterogeneity.

References

[1] Form Assembly, `http://www.formassembly.com`

[2] Jotform, `http://www.jotform.com/`

[3] Wufoo, `http://wufoo.com/`

[4] Zoho Creator, `http://creator.zoho.com`

[5] An, Y., Borgida, A., Miller, R.J., Mylopoulos, J.: A Semantic Approach to Discovering Schema Mapping Expressions. In: ICDE 2007, pp. 206–215 (2007)

[6] Aumueller, D., Do, H.-H., Massmann, S., Rahm, E.: Schema and ontology matching with coma++. In: SIGMOD (2005)

[7] Batini, C., Lenzerini, M., Navathe, S.B.: A comparative analysis of Methodologies for database schema integration. ACM Computing Surveys 18(4), 323–364 (1986)

[8] Bellahsene, Z., Bonifati, A., Rahm, E. (eds.): Schema Matching and Mapping (Data-Centric Systems and Applications). Springer, Heidelberg (2011)

[9] Buneman, P., Davidson, S.B., Kosky, A.: Theoretical aspects of schema merging. In: Pirotte, A., Delobel, C., Gottlob, G. (eds.) EDBT 1992. LNCS, vol. 580, pp. 152–167. Springer, Heidelberg (1992)

[10] Chiticariu, L., Hernández, M.A., Kolaitis, P.G., Popa, L.: Semi-automatic schema integration in clio. In: VLDB, pp. 1326–1329 (2007)

[11] Choobineh, J., Mannino, M.V., Tseng, V.P.: A form-based approach for database analysis and design. Commun. ACM 35(2), 108–120 (1992)

[12] Dragut, E.C., Kabisch, T., Yu, C.T., Leser, U.: A hierarchical approach to model web query interfaces for web source integration. PVLDB 2(1), 325–336 (2009)

[13] Jagadish, H.V., Chapman, A., Elkiss, A., Jayapandian, M., Li, Y., Nandi, A., Yu, C.: Making database systems usable. In: SIGMOD 2007, pp. 13–24. ACM, New York (2007)

[14] Jayapandian, M., Jagadish, H.V.: Automated creation of a forms-based database query interface. Proc. VLDB Endow. 1(1), 695–709 (2008)

[15] Khare, R., An, Y.: An empirical study on using hidden markov model for search interface segmentation. In: Proceedings of 18th ACM Conference on Information and Knowledge Management (CIKM), pp. 17–26 (2009)

[16] Khare, R., An, Y., Hu, X., Song, I.-Y.: Can clinician create high-quality databases? a study on a flexible electronic health record (fehr) system. In: The Proceedings of the 1st ACM Health Informatics Symposium (IHI 2010), Washington, DC, USA (2010)

[17] Khare, R., An, Y., Song, I.-Y.: Understanding search interfaces: A survey. SIGMOD Record 39(1), 33–40 (2010)

[18] Kowalczykowski, K., Ong, K.W., Zhao, K.K., Deutsch, A., Papakonstantinou, Y., Petropoulos, M.: Do-it-yourself custom forms-driven workflow applications. In: CIDR 2009 (2009)

[19] Luković, I., Mogin, P., Pavićević, J., Ristić, S.: An approach to developing complex database schemas using form types. Softw. Pract. Exper. 37(15), 1621–1656 (2007)

[20] Madhavan, J., Bernstein, P.A., Rahm, E.: Generic schema matching with cupid. In: VLDB 2001, pp. 49–58 (2001)

[21] Miller, R.J., Haas, L.M., Hernandez, M.A.: Schema Mapping as Query Discovery. In: VLDB, pp. 77–88 (2000)

[22] Pavicevic, J., Lukovic, I., Mogin, P., Govedarica, M.: Information system design and prototyping using form types. In: ICSOFT (2), pp. 157–160 (2006)

[23] Popa, L., Velegrakis, Y., Miller, R.J., Hernández, M.A., Fagin, R.: Translating web data. In: VLDB, pp. 598–609 (2002)

[24] Pottinger, R., Bernstein, P.A.: Merging models based on given correspondences. In: VLDB, pp. 826–873 (2003)

[25] Rabiner, L.R.: A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE, 257–286 (1989)

[26] Rahm, E., Bernstein, P.: An on-line bibliography on schema evolution. SIGMOD Record 35(4), 30–31 (2006)

[27] 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, pp. 95–106. ACM, New York (2004)

[28] Yang, F., Gupta, N., Botev, C., Churchill, E.F., Levchenko, G., Shanmugasundaram, J.: Wysiwyg development of data driven web applications. Proc. VLDB Endow. 1(1), 163–175 (2008)