On-the-Fly Constraint Mapping across Web Query Interfaces *

Zhen Zhang, Bin He, Kevin Chen-Chuan Chang
Computer Science Department
University of Illinois at Urbana-Champaign
{zhang2, binhe}@uiuc.edu, kcchang@cs.uiuc.edu

Abstract

Recently, the Web has been rapidly "deepened" with the prevalence of databases online and becomes an important frontier for data integration. On this deep Web, a significant amount of information can only be accessed as response to dynamically issued queries to the *query interface* of a back-end database, instead of by traversing static URL links. Such a query interface expresses a set of *constraint templates*, where each constraint template states how an attribute can be queried. To enable automatic query mediation among heterogenous deep Web sources, it is critical to automatically translate those constraints, which we name as *constraint mapping*. In particular, this paper aims at enabling *on-the-fly* constraint mapping, which is a critical task for integrating the large scale and dynamic deep Web. Such on-the-fly query translation poses a significant new challenge on the generality and extensibility of the translation framework. Existing works pursue a per-source rule-driven framework and thus cannot satisfy such requirements. In contrast, we propose a generic type-based search-driven translation framework by considering the constraint mapping for each data type as a search problem. In particular, in this paper, we develop search algorithms for text and numeric types. Our experiments over real deep Web sources show that our approach is promising to mediate queries for large scale integration.

1. INTRODUCTION

The Web has been rapidly "deepened" with the prevalence of databases online: A significant amount of information is now hidden on this "deep" Web, behind the *query interfaces* of searchable databases (*e.g.*, Figure 1 shows two such interfaces). Instead of direct linking through static URLs, such information is only accessible as responses to dynamic *queries* through these interfaces. With massive sources, the deep Web is clearly an important frontier for data integration. In particular, to enable query mediation for effective access of Web databases, it is critical to automatically translate queries across their query interfaces.

Such translation is, in essence, to match and express query conditions in terms of what an interface can "say": Each query interface consists of a set of *constraint templates*. A template specifies the "format" of an acceptable query condition, as a three-tuple [attribute; *operator*; value]. For example, for searching a "Books" database, query interface QI_1 (Figure 1) supports four constraint templates S_1: [title; *contain*; $v], S_2: [category; *contain*; $v], S_3: [price; *between*; $low,$high], and S_4: [reader age; *in*; {[4:8], ...}]. Note these templates use variables as "placeholders" (which we prefix by $, *e.g.*, $v) for users to fill in actual values. In contrast, QI_2 supports a different set of templates– T_1 on title, T_2 on subject, $\cdots$, and T_4 on price. Thus, querying on the deep Web is to *instantiate* these templates into actual *constraints*- by specifying concrete operators and values. For instance, we may search QI_1 by constraint s_2 = [category; *contain*; `"computer science"`], as an "instantiation" of template S_2.

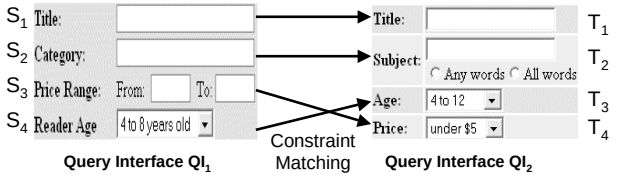

Figure 1: Two example query interfaces and their matching.

In particular, this paper addresses the problem of *constraint mapping*, not *schema matching*. Specifically, as Figure 1 indicates, between a pair of "source" and "target" interfaces (*e.g.*, QI_1 and QI_2), our focus is to translate between known "matching" constraint templates (*e.g.*, S_2 in QI_1 matches T_2 in QI_2). Formally, given a source template (*e.g.*, S_2), its matching target template (*e.g.*, T_2), and a specific source constraint (*e.g.*, constraint s_2, as an instantiation of S_2), the objective of constraint mapping is to find the target constraint with the closest semantic meaning to the source constraint. For instance, as Figure 2 shows, constraint mapping is to instantiate T_2 into t_2=[subject; *all words*; `"computer science"`], which is the best translation of the source constraint s_2, *i.e.*, $s_2 \rightarrow t_2$. As we will discuss below, we consider the discovery of semantic correspondence between S_2 and T_2, which is essentially a schema matching problem, as an independent task and beyond the scope of this paper.

While we focus on constraint mapping in this paper, for complete query translation, we note that there are other necessary tasks, which several recent works have paved the way: First, we need to extract constraint templates (*e.g.*, $T_1, \cdots, T_4$) from a query interface. We study such *query-form extractor* in [10]. Second, given source and target constraint templates (*e.g.*, QI_1 and QI_2 respectively), we need to find matching templates (as Figure 1 shows). This problem is essentially a *schema matching* problem, which has been extensively studied in the literature [6]. In particular, references [5, 8] specifically focus on matching Web query interfaces.

The large scale and dynamic features of the deep Web ask for the need of *on-the-fly* query translation. Specifically, the deep Web is of large scale (at the order of 10^5 sources [1]) and of a dynamic nature (as sources are changing and new ones are emerging). Also, it is very diverse, with various sources (*e.g.*, for finding books, airfares, patents, *etc.*)– Users will thus interact with "ad-hoc" sources to satisfy their various information need. This large-scale, dynamic, and ad-hoc nature mandates effective integration to enable "on the fly"

*This material is based upon work partially supported by NSF Grants IIS-0133199 and IIS-0313260. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

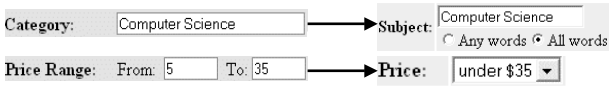

Figure 2: Constraint mapping across QI_1 and QI_2.

query translation. That is, the mapping technique should be able to translate queries for unseen sources, where no pre-configured translation knowledge can be assumed.

While critical for integrating Web databases, such on-the-fly query translation brings a new challenge. Existing work [2] on constraint mapping assumes pre-selected sources (*e.g.*, amazon.com and bn.com for book comparison shopping), and relies on a *per-source, rule-driven* framework. Such statically configured framework is not suitable for our dynamic scenario: First, It is not *general* because it manually records the translation knowledge for each source. Second, it is not *extensible*, since the translation rules encode pairwise mappings between two schemas and thus translation among n schemas requires $O(n^2)$ sets of rules, which makes extension labor extensive and hard to maintain (Section 3.1).

In this paper, we propose a *type-based*, *search-driven* translation framework to achieve both the generality and the extensibility. The idea of type-based translation is motivated by our observation that constraint templates of different concepts (e.g., the concept about title and the one about author) often share similar patterns. For instance, the constraint templates of title and author usually share the same operator (e.g., "contain") and value format (e.g., an input box). Such regularity indicates an implicit notion of *type* that decides the applicable operators and expected value formats for an attribute. Therefore, instead of per-source knowledge, we propose type-based translation, which "encodes" more generic translation knowledge for each type.

Further, we realize the type-based translation as a search problem, where the search space is defined by the template patterns of each type. Specifically, given a source query, we search for the best instantiation for the target query template that retrieves the closest set of results as the source. Such search is guided by a closeness function, which essentially evaluates the similarity of the query results between two constraints.

In summary, this paper makes the following contributions:

- Our current work plus our preceding works complete the framework to tackle the problem of on-the-fly query translation. To our knowledge, while important for large-scale integration in general (and the deep Web in particular), this problem has not been extensively studied.
- We develop a *type-based* (instead of source-specific) mechanism to generically handle on-the-fly translation– by leveraging the "regularities" across the implicit types of constraints.
- We develop a *search-driven* (instead of rule-driven) machinery to dynamically find closest mappings– by abstracting translation as a search problem.

2. RELATED WORK

Constraint mapping (which this paper focuses on) is one critical step for information integration. While on-the-fly constraint mapping in a dynamic integration setting has not be studied before, there are some works such as query mediation and schema mapping, which are closely related to our work.

First, in contrast to query mediation: Query mediation works have been mainly focusing on mediating queries across multiple sources and thus abstract the problem as a paradigm of answering query using views [4]. In particular, they assume each source has a wrapper, which encapsulates the tasks of extracting query capability, schema matching and constraint mapping for that source. The main focus of query mediation is thus on how to decompose a user query into sub-queries across multiple sources. In contrast, we are focusing on query translation between two sources other than mediating queries across multiple sources. In particular, we are dealing with the mapping of constraint heterogeneity (*i.e.*, semantically related constraints may support different operators and value formats). Further, while reference [2] specifically tackles the constraint heterogeneity, it assumes a static system setting, where per-source translation knowledge can be acquired. However, for our scenario of large scale integration, we have to on-the-fly translate queries and thus need to develop new mapping techniques. We will discuss the comparison in details in Section 3.1.

r_1	[category; *contain*; $s] → *emit:* [subject; *all*; $s]
r_2	[title; *contain*; $t] → *emit:* [title; *contain*; $t]
r_3	[price range; *between*; $s,$t] → $p=ChooseClosestNum($s), *emit:* [price; *less than*; $p]
r_4	[reader age; *between*; $s] → $r=ChooseClosestRange($s) *emit:* [age; *between*; $r]

Figure 3: Translation rules $\mathcal{T}_{12}$ between QI_1 and QI_2.

Second, in contrast to schema mapping: Schema mapping [9] aims at translating a set of data values from one source to another one, according to given matchings. Therefore, schema mapping only concerns about the equality relation between different schema, based upon which data is converted. In particular, no constraint heterogeneity is considered in schema mapping. In contrast, constraint mapping focuses on translating specific queries other than the data values. It in general supports broader semantic translation. Specifically, since constraints templates may support different operators or value formats, equality relationship cannot always be achieved. Hence, in many situations, we have to choose the semantically closest translation other than the equal one.

3. MOTIVATION & FRAMEWORK

As discussed in Section 1 and 2, although query mediation has been extensively investigated in the literature, the existing work often assumes a static small scale system setting. In the section, we review this framework, its infeasibility for on-the-fly translation, and further propose our solution.

3.1 Preliminary

The existing work that specifically addresses the problem of constraint mapping is introduced under the scenario of a static small scale system setting, where only a small number of pre-configured sources are integrated. Based on such a setting, a per-source pairwise-rule driven translation framework is developed [2].

Example 1: To translate queries from QI_1 to QI_2 in Figure 1, the per-source rule-driven framework needs a set of manually-written rules $\mathcal{T}_{12} = \{r_1, r_2, r_3, r_4\}$ as shown in Figure 3. In particular, rule r_1 specifies that when mapping constraints from category to subject, we choose operator *all* and fill in the same value as in the source constraint.

The rules are designed only for these two sources - it specifically tells what the matched constraints are (*e.g.*, category *vs.* subject), which operator to choose (*e.g.*, $op="all") and what value to fill in (*e.g.*, $c = s$). Hence, the rules are pairwise - it handles the translation between two specific constraints, each of them is in a specific source. ■

Such a framework is designed and works well for small scale integration systems. Consequently, it lacks of generality and extensibility required for on-the-fly query translation in large scale integration scenarios, in particular the deep Web sources. First, generality: Per-source translation framework cannot generally handle the translation between arbitrary "unseen" sources because it needs translation knowledge for each and every source.

Second, extensibility: Pairwise translation cannot be easily extended. As we will see in next section, adapting the pairwise-rule-

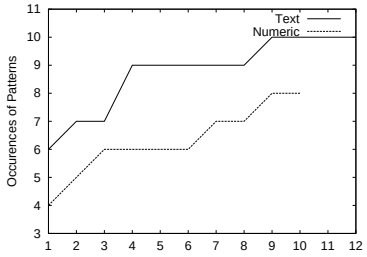

Figure 4: Growth of patterns over constraint groups.

based framework for on-the-fly translation needs rules between every pair of constraint templates, and thus adding a new constraint template needs to add multiple rules mapping back and forth to every existing one, which makes the system difficult to extend.

3.2 Motivation: Type-based, Search-Driven

To achieve the generality, we need "source-independent" translation framework that can generally handle translations without rules (*e.g.*, $\mathcal{T}_{12}$ in Example 1) that is tailored for specific sources (*e.g.*, QI_1 and QI_2). Our goal is to develop such a generic framework.

Our solution is motivated by our observation on the deep Web sources. In particular, to survey the constraints, we explore several domains in the TEL-8 dataset of the UIUC Web Integration Repository [3]. TEL-8 dataset contains about 500 deep Web sources in 8 domains e.g., Books, Automobiles.

We observe that when looking at a large collection of sources, the group of matched constraint templates, which we call *constraint group* (e.g., the group of title constraints), usually have a limited number of *template patterns* that differs in operators and value formats. For instance, exploring 65 book sources in the TEL-8 dataset, we only find six patterns for the title constraint group: [title; {*all*}; $val], [title; {*all, any*}; $val], [title; {*all, any, exact*}; $val], [title; {*all, start*}; $val], [title; {*all, exact*}; $val], and [title; {*all, exact, start*}; $val], where "$val" represents the variable accepting any string presented as an input box in query interfaces. Similarly, there are four patterns for the constraint group about subject: [subject; {*all*}; $val], [subject; {*equal*}; $val:{D}], [subject; {*subsume*}; $val:{D}], and [subject; {*all, start*}; $val], where $val:{D} represents the variable accepting values from the given domain D. For example, the second pattern allows choosing one value from a selection list D and the third choosing multiple values from a select list.

Further, we observe that constraint groups of the same data type (e.g., text type or numeric type) often naturally share similar common patterns. This observation seems to imply that mapping of constraints depends on their syntactic data types instead of semantic attributes (e.g., subject and title). To further understand to what extent such commonality exists, we survey two sets of similar constraint groups: "text like" groups and "numeric like" groups. For each set, we collect up to four most popular constraint groups (if there are so many) from three domains in the TEL-8 dataset: Books, Automobiles and Airfares. For example, the four "text like" constraint groups from Books domain are title, author, keywords and subject. Also, the four "numeric like" constraint groups from Automobiles domain are price, mileage, distance and cylinder. Figure 4 shows how the patterns increase when new constraint groups are observed, where the x-axis denotes the number of observed constraint groups and y-axis the number of observed patterns accumulatively. As we can see, the emergence of the patterns generally converge. In particular, in "text like" groups, no new patterns appear after the $9th$ group.

Motivated by this observation, we achieve the generality by leveraging the regularity among constraint groups and thus propose *type-based* translation. The type of the constraint group generally determines the applicable operators and accepted value formats for constraints of this type. For instance, `text` type constraints usually support operators such as *any, all, exact, start* and accept string values, and `numeric` type constraints usually support operators such as *equal, greater than, less than, between* and accept numeric values. Therefore, different constraints of the same type share the translation knowledge, which can be exploited to direct the query translation.

However, for this type-based translation, how can we achieve the extensibility? As we can see from the Example 1, traditional rules realize a pairwise translation paradigm: each rule specifies how to map between two specific template patterns. Therefore, a type with m patterns will need $m * (m - 1)$ rules to handle the translation within the type. For instance, with 10 patterns in text type (as Figure 4 shows), we need to have 90 rules to enable the translation between any two patterns. Consequently, such a framework does not give good scalability and extensibility - adding a new pattern needs $2 * m$ rules to map back and forth to all existing patterns, which is labor intensive and hard to maintain.

To achieve the extensibility, we explore a *search-driven* approach. Given a source constraint and a target constraint template, our constraint mapping framework searches possible target instantiations for the closest one to the source constraint. The search is guided by a *closeness function*, which evaluates the proximity of a mapping based on a *closeness metric*. Such dynamic search mechanism eliminates the need for static, pairwise rules.

In summary, we develop a type-based search-driven framework for large scale constraint mapping. This framework essentially employ a search mechanism (instead of static rules) for each type (instead of each source or pattern).

3.3 System Framework

In this section, we give an overview of the constraint mapping framework (as Figure 5 shows), which starts from a source constraint s and a target constraint template T, and outputs the closest target constraint t_{opt}, that T can generate, to s. In particular, the *type recognizer* first identifies the type of the constraints, and then dispatches them accordingly to the *type handler*. The *type handler* then performs the search to find a good instantiation among possible ones described by T, which is then returned as the mapping.

The *type recognizer* takes the source constraint s and target constraint template T as input, and infers the data type by analyzing the constraints syntactically. The type of a constraint are often hinted by its syntactical features. Consider the constraints in Figure 1, to recognize the data types, we can exploit the distinctive patterns (*e.g.*, the *from-to* pattern for `numeric` type, as used in price range), the operators presented (*e.g.*, *all, any, exact* for `text` type), the values filled in the source constraint (*e.g.*, 35 in price range) and the value domain (*e.g.*, a selection list in price). Currently, we use simple rules to recognize the types based on the above features. In the future, we may explore machine learning approach to train a classifier for automatic type recognition.

As the major component of the framework, the *type handler* takes the constraints dispatched by the *type recognizer* as input and performs search among possible instantiations of the target constraint template for the best one. In next section, we will discuss how to perform the search and how to evaluate the closeness of mappings in details.

4. CONSTRAINT MAPPING

In this section, we discuss how constraint mapping is realized by a search process to find out the good translations. In particular, we study two most common types - `text` and `numeric` to illustrate the principle of our search-driven approach for constraint mapping.

4.1 The Translation Problem

As abstracted in Section 3, given a source constraint and a target constraint template, constraint mapping is essentially to find the best target constraint *w.r.t.* a closeness metric. More formally, we define the problem as follows:

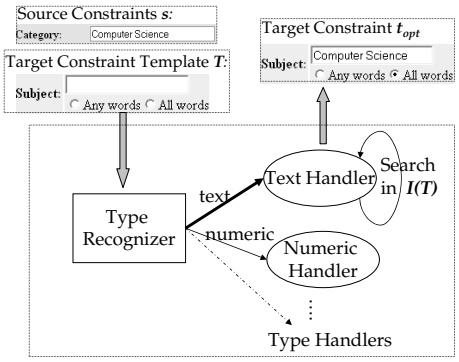

Figure 5: Framework of constraint mapping.

t_1	[subject; *any*; `"computer science"`]
t_2	[subject; *all*; `"computer science"`]
t_3	[subject; *any*; `"computer"`]
t_4	[subject; *any*; `"science"`]
...	...

Figure 6: Search space of constraint mapping.

Problem Statement: Let S and T denote the source and target constraint template respectively, and $I(S)$ and $I(T)$ denote a set of constraints that S and T can instantiate respectively. $C(s,t)$ denotes the closeness metric that assesses the closeness of the constraint s and t. Constraint mapping is that, given $s \in I(S)$ and T, find $t_{opt} \in I(T)$ such that $C(s, t_{opt})$ is maximized, *i.e.*,

$$t_{opt} = \arg\max_{t_i \in I(T)} C(s, t_i) \quad (1)$$

Let us use an example to illustrate the components of our search problem for constraint mapping.

Example 2: Consider the example shown in Figure 5 to map the constraints between category in QI_1 and subject in QI_2. The source constraint s = [category; *contain*; `"computer science"`] is instantiated from template S = [category; *contain*; $val] [1] by populating $val=`"computer science."` The target constraint template T = [subject; *$op*; $val] accepts operators $op from { `"any words"`, `"all words"`} (simply written as `"any, all"` in the following paper), and value $val from any string. Therefore, the search space $I(T)$ contains possible instantiations of T as Figure 6 enumerates some of them. Among the candidate target constraints $t_1, t_2, \ldots$, from $I(T)$, the constraint mapping thus searches for the $t_i \in I(T)$ that is closest to s, *i.e.*, $C(s, t_i)$ is maximized. In the example the best mapping $t_{opt} = t_2$. ■

To quantify the closeness of the mapping, the closeness metric C is defined. Ideally, the mapped constraints t should retrieve exactly the same results as the original one s. However, since such an exact mapping may not exist (as Figure 2 shows an example on mapping price), the *approximate mapping* may introduce *false positives* or *false negatives* as opposed to the original constraint. Figure 7 illustrates those errors using a Venn diagram for original constraint s and its translation t. To quantify those errors, two metrics are introduced as *precision* to capture the false positives and *recall* to capture false negatives.

$$\mathcal{P}(s,t) = \frac{|s \wedge t|}{|t|}, \mathcal{R}(s,t) = \frac{|s \wedge t|}{|s|} \quad (2)$$

With the two metrics to capture the mapping errors, the closeness metric is thus a formula defined on the two, *i.e.*, $C(s,t) = \mathcal{F}(\mathcal{P}(s,t), \mathcal{R}(s,t))$. For example, if we measure the closeness as $|s \wedge t|/|s \vee t|$, then C is defined as:

[1]We assume operators of a constraint are known. For constraints without explicit operators (*e.g.*, S), we assign a default one (*e.g.*, *contains*) based on the most commonly used operator observed in constraints of same type.

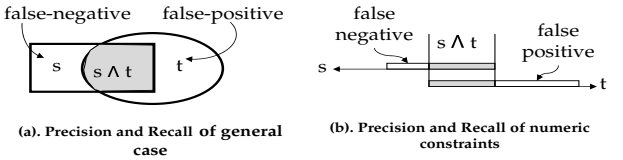

Figure 7: Venn Diagram of Precision and Recall.

$$C(s,t) = \frac{1}{\frac{1}{\mathcal{P}(s,t)} + \frac{1}{\mathcal{R}(s,t)} - 1} \quad (3)$$

As abstracted in Equation 1, to realize a search paradigm, the type handler needs to do the following with the inherent challenges:

1. *define a search space* $I(T)$ to traverse: How to define a reasonable space that on one hand does not lose good translations, on the other hand is of manageable size to be traversed.
2. *enumerate the candidate mappings* (s, t_i) to be evaluated;
3. *assess the closeness metric* $C(s, t_i)$ to measure the closeness of the mapping (s, t_i): How to implement a closeness function to evaluate $C(s, t_i)$? Do we need semantic reasoning to infer their closeness?

In the following section, we study the two most common used types `text` and `numeric` to illustrate the principles in our search-driven approach.

4.2 Text Handler

`Text` is the most commonly used type for querying string based fields in the database (*e.g.*, subject in Example 2). The operators of `text` type are typically string match operations including *all*, *any*, *exact* and *start*, and the values are strings. In the section, we will use the mapping between category and subject in Example 2 to illustrate how the search proceeds towards finding the best mapping.

Defining Search Space

Defining a reasonable search space $I(T)$ is an essential problem in any search process (*e.g.*, the left-deep trees in query optimization for optimizing join orders), because a huge space or even infinite space is impossible to exhaust. While *operators* are clearly limited by T (*e.g.*, two operators *any,all* in Example 2), how about values? Theoretically any texts may fill in the values of the target constraint template, which constitute an infinite space. What is the right scope for the search to focus? To define a reasonable search space $I(T)$ to traverse, in the current implementation, we make a "closed-world" assumption: the values of any target constraint t_i use only constants W_s mentioned in the source constraint s. [2] In Example 2, W_s is thus restricted to {`"computer"`, `"science"`}, and accordingly the values for $val is all possible combinations (with different ordering) of the words from W_s. By doing so, we define the search space $I(T)$ as Figure 6 shows part of it. Such a close world assumption is reasonable, because without domain specific knowledge, it is very hard to create new words out of blue. Further, the search space can be enriched (*e.g.*, by expanding the query words with their synonyms) if more domain knowledge is available (*e.g.*, by providing synonym lookup).

Estimating the Closeness

Given the source and target constraint s, t_i, closeness estimation $C(s, t_i)$ essentially needs to evaluate how close the result retrieved by t against the target database is to that retrieved by s. However, the lack of target database content makes such estimation difficult. Ideally, if it is possible to reason about their closeness without looking at the database, we can always find the best mapping t_{opt}. However, in general, such reasoning is very hard because it needs not

[2]Or, in general, any constant that can be functionally determined by W_s, *e.g.*, synonyms or hyponyms from the dictionary. This generalization is useful *e.g.*, in mapping constraints with different vocabulary such as [subject; *contain*; `"computer science"`] *vs.* [subject; *contain*; `"Internet"`].

Original Database	Isomorphic Database
Computer Science	w_1, w_2
Science Computer	w_2, w_1
Computer Science And Internet	w_1, w_2, w_3, w_4
Computer Game	w_1, w_5
Biology Science	w_6, w_2
Art And Architecture	w_7, w_3, w_8
Computer Software	w_1, w_9
Computer Hardware	w_1, w_{10}

Figure 8: Dummy database and isomorphic database.

only the knowledge for reasoning (*e.g.*, logic rules) but also algorithms to apply the knowledge to realize the reasoning. For example, when expressing the constraints in regular expression, reasoning on their containment relations becomes the regular language containment problem, which has proven to be $\mathcal{PSPACE}$-complete [7].

To address the difficulty of closeness estimation, we employ an "estimation-by-testing" approach. We query the two constraints s and t_i against a *dummy database*, and then by comparing their results, we calculate the precision and recall, hence the closeness metric. Such a dummy database in principle simulates the target database so that the relative goodness of the mappings can be evaluated. It can be achieved, for example, by random sampling the objects in the target database. Currently, our database is generated "uniformly", as we explain later.

Example 3: In Example 2, to estimate the closeness of mappings between `text` constraints (s, t_i), we build a dummy database D that contains a set of tuples with a single textual attribute. The left column of Figure 8 shows an example of such a dummy database. To estimate the closeness of s =[category; *contain*; `"computer science"`] and t_1 =[subject; *any*; `"computer science"`]), we query s and t_1 against D, and retrieves 3 and 7 tuples respectively. Among those tuples, three is in the intersection of s and t_i. According to Equation 2, the precision and recall is $\frac{3}{7}$ and 1 respectively. Using the closeness metric in Equation 1, their closeness is thus $\frac{3}{7}$. Similarly, estimating (s, t_2) against D gets closeness 1. Therefore, t_2 is a better mapping than t_1. ∎

To fully reflect the relations between s and t_i, the dummy database should capture the "interesting" values with various compositions of the queried terms W_s, *e.g.*, tuples with both `"computer"` and `"science"` by different orderings, with only one of them, without any of them *etc.*. To make this happen, we customize the alphabet of the constructed database to subsume W_s. Therefore, in Example 3, the alphabet of database contains {computer, science} plus some other random words. In the implementation, such customization is realized by using an *isomorphic* alphabet of words, *e.g.*, $\{w_1, w_2, \ldots, w_n\}$, and mapping the constants in W_s into words from the alphabet, *e.g.*, `"computer"` $\rightarrow w_1$, `"science"` $\rightarrow w_2$. Figure 8 shows the database on the isomorphic alphabet. By doing so, we do not need to construct a dummy database for every time to translate a query. To make sure that the database captures interesting values, we keep the alphabet small and the database size relatively large so that every value pattern including the interesting ones has a better chance to appear in the database. The values in the database are *uniformed-generated*: the length of the value follows a uniform distribution, and the words in the database are randomly picked from the isomorphic alphabet.

While the quality of mapping closely depends on the dummy database, we find that in most cases dummy database can give reasonably good answers. The reason is that the containing relationship between operators (e.g., *any* contains *all*) can be reflected from any dummy database, independent of the real data distribution. That is, no matter how we construct the dummy database, the result of applying operator *any* always contains that of applying *all*. Our observations show that, across the deep Web, most sources only use a small set of four operators: *any*, *all*, *exact* and *start*. Since these four dominating operators have a chaining containing relationship, i.e., *any* contains *all* contains *exact* contains *start*, we can achieve reasonably good translation with a uniformly distributed dummy database. For other relatively rare operators that may not guarantee the containing relationship, the dummy database can be exploited to help find the statistically optimal translation. Certainly, how to make our database more "intelligent," to capture the characteristics of the target database and also adequately test the constraints at hand, is an important part of our future work, and we are currently working on the related techniques (*e.g.*, understanding query results) to make this happen.

4.3 Numeric Handler

As another most frequently used type, `numeric` constraints query the database fields of numeric values. The typical operators are numeric comparisons including *less than*, *greater than*, *between* and *equal*. Due to space limitation, we only discuss the numeric handler very briefly.

Defining Search Space

Similar as `text` handler, the challenge for `numeric` handler is how to define the right search space so that search can be performed in a reasonable small but still good scope. To address the problem, two approaches are possible. First, we may again employ the closed-world assumption to refine the space $I(T)$ based on the source constraint s. This is currently the approach used in our implementation. Second, we take the infinite numeric space, and perform a systematic search using existing search algorithms such as *hill-climbing*. Starting from a initial solution, the hill-climbing always goes for a better solution (known as *uphill move*). Suppose we have only one variable $val in the target constraint template T to be instantiated and we adapt a walk of fixed-length k. At any state (assignment of values to val), we have 2 possible movements as $val = val + k$ or $val = val - k$. The search performs a series of uphill moves until it reaches a *local optimal*. We can further refine the hill climbing strategy by choosing the starting point based on the source constraints s instead of starting from a random solution. Due to space limitation, we will not discuss in details.

Estimating the Closeness

While similar idea of constructing the dummy database for text type is applicable for numeric type to estimate the closeness metric C, we find that systematic reasoning on the numeric constraints is possible due to the continuous nature of the numeric data. To estimate the query result of a numeric constraint, we map the constraint into ranges on the numeric line (*e.g.*, $5 \sim 35$ in Figure 2), and therefore the false positives and false negatives can be evaluated based on the coverage and overlapping of the two constraints, as Figure 7(b) illustrates.

5. CASE STUDIES

In this section, we report our preliminary study on the performance of our framework. The study simulates a *query assistant* system as the application scenario where the system automatically fills out the query forms (which may be dynamically collected by a search engine) based on the user's original query. The goal of the experiment is to evaluate how well the automatic constraint mapping can help users in interacting with those sources.

In the experiment, the constraint template patterns captured are the common ones (with appearance more than 5) we collected during the survey. There are 8 such patterns for `text` type and 6 for `numeric` type. For each pattern, we build an interpreter, which knows how to estimate the query results. In particular, the text pattern interpreter needs to know how to query the constraint against the dummy database, and the numeric pattern interpreter will map the constraints into ranges on the numeric line.

Based on our application scenario, we adapt a simple perfor-

Source Constraint	Target Constraint Template	Target Constraint
[title; *contain*; `"database system"`]	[title; *$op:{any, all}*; $val]	[title; *all*; `"database system"`]
[title; *start*; `"computer theory"`]	[title; *$op:{any, all, exact}*; $val]	[title; *all*; `"computer theory"`]
[subject; *contain*; `"Programming Language"`]	[subject; *equal*; $val:{art,. . . , computer,. . . }]	[subject; *equal*; `"all subject"`]
[price; *less than*; 5000]	[price range; *between*; $low,$high]	[price range; *between*; 0,5000]
[price; *between*; 1000,5000]	[price; *less than*; $val]	[price; *less than*; 5000]

Figure 9: Examples of generated mappings.

Constraint	#Total	#Correct	%Correct
title	19	17	0.89
subject	10	7	0.7
keywords	10	9	0.9
Text	39	33	0.85
mileage	8	7	0.88
price	16	15	0.94
Numeric	24	22	0.92
Overall	63	55	0.87

Figure 10: Experiment data set and results.

mance metric: number of *correct* mappings, because it reflects the amount of efforts the system saves for the user. The correct (or closest) mappings are manually generated by checking the semantics of the source and target constraints ourselves (without really querying the actual database). Counting only the "absolutely" correct constraints is actually very stringent because it does not capture "how wrong" an incorrect mapping is. An incorrect mapping may retrieve similar results as the correct one (*i.e.*, still have good precision and recall although not the best). However, in our measurement no credit is given to such mappings. In the future work, we will quantify the quality of the mappings in a finer granularity instead of simple binary *correct vs. incorrect*.

As the experiment data, we collected 40 query interfaces from the TEL-8 UIUC Web Integration Repository [3]. The query interfaces are from two domains Books and Automobiles, with 20 query interfaces from each. For each domain, we collected the popularly queried constraint groups as our experiment data set. Figure 10 reports the constraint groups in the data set and the number of constraints in each group. Among the constraints groups, title, subject, keywords are from Books domain to test `text` type mappings, and price, mileage from Automobiles domain to test the `numeric` type mappings.

The experiment essentially evaluates the mapping accuracy between two randomly picked constraints from the same constraint group. In particular, suppose the constraint group (*e.g.*, title) has n constraint. We first number all the constraints in the group, *i.e.*, $[C_1, C_2, \ldots, C_n]$. We then random shuffle those constraints to get a random permutation $[C_{i_1}, C_{i_2}, \ldots, C_{i_n}]$. Last we map the constraints between (C_1, C_{i_2}), (C_2, C_{i_2}), $\ldots$, (C_n, C_{i_n}). We use Equation 3 as the closeness metric for closeness estimation.

Figure 9 shows several examples of constraint mappings in the experiment. As we can see, the system can generally work well for most interesting cases, but further improvement is possible and important. For example, we did not get good result for the second mapping due to the lack of domain knowledge of hyponyms. Figure 10 generally reports the experiment results as the percentage of the correct mappings for each constraint group, the average for `text` type mappings, `numeric` mappings, and the overall for both types. As we can see, our framework generally achieves good accuracy: among the 63 mappings tested, 55 are correct, which amounts to 87% accuracy.

6. CONCLUDING DISCUSSION

In this preliminary work of the type-based search-driven translation framework for on-the-fly constraint mapping, while the initial result is promising, we also observed several further opportunities and open issues that warrant more investigation.

First, *handling complex mappings*: Currently, we study only the simple $1:1$ constraint mapping, which is the basic and also most common mapping in query translation. In general, the mapping can be $m:n$, *e.g.*, last name + first name is a synonym of author in Books domain. We are interested in modelling the compound mappings in a general way, similar to what we did for simple constraints using types. Also, we plan to investigate how to extend the search-driven approach to support such a model.

Second, *incorporating domain knowledge*: Our current framework does not assume any domain specific knowledge. However, as mentioned in the paper quite a few times, whenever available, such knowledge may help improve the mapping quality from various aspects, *e.g.*, to construct representative dummy database (Section 4.2), to define more comprehensive search space (Section 4.2); More importantly, some mappings are domain specific, which have to refer to domain knowledge (e.g., the mapping from city name to airport code). Therefore, we need to extend our framework to incorporate the domain knowledge whenever they are available.

Third, *constructing representative dummy database*: We currently generate a "uniformly-distributed" dummy database by treating all the keywords equally. However, the importance of words can be various. For instance, when translating constraints such as [title; *contain*; database system] to [title; *any*; $v], the mapping generated is $v=`"system"`. However, if we know that `"database"` is a more distinguishing word than `"system"`, we may construct our database to reflect the distribution of word frequency and thus generate better mappings.

In summary, this paper aims at developing a framework to help automatically mapping constraints among deep Web sources. In particular, we propose a generic type-based search-driven translation framework, which is well suited for the requirements of the on-the-fly constraint mapping among large scale data sources. Our preliminary case studies validate the effectiveness of our approach and open several future research issues.

7. REFERENCES

[1] M. K. Bergman. The deep web: Surfacing hidden value. Technical report, BrightPlanet LLC, Dec. 2000.

[2] K. C.-C. Chang and H. Garcia-Molina. Mind your vocabulary: Query mapping across heterogeneous information sources. In *SIGMOD Conference*, 1999.

[3] K. C.-C. Chang, B. He, C. Li, and Z. Zhang. The UIUC web integration repository. Computer Science Department, University of Illinois at Urbana-Champaign. http://metaquerier.cs.uiuc.edu/repository, 2003.

[4] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[5] B. He and K. C.-C. Chang. Statistical schema matching across web query interfaces. In *SIGMOD Conference*, 2003.

[6] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[7] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *5th annual ACM symposium on Theory of computing*, 1973.

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

[9] L. L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD Conference*, 2001.

[10] Z. Zhang, B. He, and K. C.-C. Chang. Understanding web query interfaces: Best-effort parsing with hidden syntax. In *SIGMOD Conference*, 2004.