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 heterogeneous 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, toward on-the-fly integration of Web databases, due to the large scale and dynamic nature of the 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 approach is promising as a principle way to mediate queries for types. Our experiments over real deep Web sources show that our method can only be accessed as response to dynamically issued queries to databases (e.g., price between Slow,Shigh), and $S_3$: [reader age; in; {14:8, ...}]. 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, $T_3$, 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 $s_3 = [category, contain, "computer science"]$, as an “instantiation” of template $S_3$.

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; Slow,Shigh], and $S_4$: [reader age; in; {14:8, ...}]. 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, $T_3$, 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 $s_3 = [category, contain, "computer science"]$, as an “instantiation” of template $S_3$.

As complex queries are built upon atomic constraints, to enable query mediation, this paper addresses the constraint mapping problem—That is, how to actually translate matching constraints? As Figure 1 indicates, across a pair of “source” and “target” interfaces (e.g., $QI_1$ and $QI_2$) respectively, we must translate between the corresponding “matching” constraint templates (e.g., $S_3$ in $QI_1$ matches $T_3$ in $QI_2$). (We discuss this independent “schema matching” task below.) In particular, given a specific source constraint (e.g., constraint $s_3$ above, as an instantiation of $S_3$), with respect to a matching target template (e.g., $T_3$), what is the closest mapping? In this case, constraint mapping is to instantiate $T_3$ into $t_3=|[\text{subject all words}; \text{"computer science"}]|$, which best matches $s_3$, i.e., $s_3 \rightarrow t_3$ with respect to $T_3$. Figure 2 shows some example mappings.

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, to deal with a query interface (e.g., $QI_2$), we must first extract the supported query templates (e.g., $T_1, \cdots, T_4$). We study such query-form extractor in [8]. Second, given source and query interfaces (e.g., $QI_1$ and $QI_2$ respectively), we must find matching constraints (as Figure 1 shows). This problem is essentially schema matching, which we studied in [5] specifically for Web interfaces.

In particular, this paper aims at enabling on-the-fly constraint mapping, toward on-the-fly integration of Web databases: To begin...
with, the deep Web is of extremely large scale (at the order of 10^5 sources [3]) and of a dynamic nature (as sources are changing and new ones are emerging). Further, it is also 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. Putting together, we stress that this large-scale, dynamic, and ad-hoc nature mandates effective integration to enable translation “on the fly”– for previously unseen sources lacking specifically-configured translation rules.

While critical for integrating Web databases, such on-the-fly query translation poses a significant new challenge– As far as we know, this paper is the first to address this novel (and difficult) problem. In clear contrast, the existing works (e.g., [1, 2]) assume pre-selected sources (e.g., amazon.com and bn.com for books comparison shopping), and thus rely on a per-source, rule-driven framework. Such “statically-configured” frameworks are not suitable for our dynamic scenarios: First, they are not general because their per-source translation requires translation knowledge for each and every source. Second, they are not extensible– Since their rules (Section 2.1) encode pairwise mapping (between pairs of constraints), translating between n constraints requires O(n^2) rules– which makes extension labor extensive and hard to maintain.

In this paper, we propose a per-type search-driven translation framework to achieve both the generality and the extensibility. The idea of per-type translation is motivated by our observation that different concepts (e.g., the concept about title and the one about category) may share the same query pattern (i.e., the same operators and value format). For instance, the constraint templates of title and subject often share the same operator (i.e., “contain”) and value format (i.e., an input box). As Section 2 will discuss, the regularity in the operators and value formats indicates that there is a notion of type that decides the applicable operators and expected value formats for the attributes. Therefore, instead of presenting per-source knowledge, type-based translation “encode” more generic translation knowledge around types.

In particular, we realize the type-based translation by viewing it as a search problem. Specifically, given a source constraint, we search for the best instantiation for the target constraint template. The search space is all the possible instantiations of the target constraint template. Further, the search is guided by a closeness function, which estimates the closeness of two constraints based on a type-specific metric. Intuitively, the closeness metric is designed to evaluate the similarity of the query results between the source constraint and target constraint. For some data types such as numeric type, the closeness metric can be easily computed by directly mapping the numeric values in the constraints onto a number axis. However, for other data types such as text type, there is no straightforward way to define the closeness metric. In this paper, we propose an estimation-by-testing approach, to deal with the text type. Specifically, the estimation-by-testing approach estimates the closeness by querying the constraints against a dummy database which in principle simulates the target source. A constraint of a text type is thus mapped onto a set of query results from the dummy database, which are used for evaluating the closeness.

In summary, this paper makes the following contributions:

- We identify the problem and propose an overall framework for on-the-fly constraint matching. To our knowledge, while important for large-scale integration in general (and the deep Web in particular), this problem has not been 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) machine

---

**2. MOTIVATION & FRAMEWORK**

As discussed in Section 1, existing work on query translation realizes a per-source rule-driven translation framework, which is designed for the static setting of small scale information integration system. In the section, we review this framework, its infeasibility for on-the-fly translation, and further propose our solution.

### 2.1 Preliminary & Related Work

Automatically querying the deep Web has not been extensively studied in the literature. In particular, reference [6] develops a task-specific human-assisted crawler to query a specific deep Web source and does not consider the problem of query translation. To the best of our knowledge, this paper is the first work on translating queries among Web query interfaces. On the other hand, query translation (for mapping constraints across sources) has been extensively investigated under the scenario of translating queries among a static small scale system setting, where only a small number of pre-configured sources are mediated. Based on such a setting, a per-source rule-driven translation framework is developed [1, 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 T\_1 = \{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., $\text{Sp}=\text{ChooseClosestRange}($title$)$) and what value to fill in (e.g., $\text{Sp} = s$). 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: Rule-based pairwise translation cannot be easily extended. As we will see in next section, adapting the rule-based framework for on-the-fly translation needs rules between every pair of constraints, and thus adding a new constraint needs to add multiple rules mapping back and forth to every existing one, which makes the system difficult to extend.

### 2.2 Motivation: Type-based, Search-Driven

To achieve the generality, we need "source-independent" translation framework that can generally handle translations without source
specific rules (e.g., $T_{12}$ 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 [4]. 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 constraints, which we call constraint group (e.g., the group of title constraints), usually have a limited number constraint patterns that differs in operators and value formats. For instance, exploring 65 book sources in the TEL-d dataset, we only find six patterns for the title constraint group: [title: {all} ; Sval], [title: {all, any} ; Sval], [title: {all, any, exact} ; Sval], [title: {all, start} ; Sval], [title: {all, exact} ; Sval], and [title: {all, exact, start} ; Sval], where “Sval” 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} ; Sval], [subject: {equal} ; Sval: {D}], [subject: {subsume} ; Sval: {D}], and [subject: {all, start} ; Sval], where Sval: {D} represents the variable accepting values from the given domain D. 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 exist, 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(b) 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, rules realizes how to map between two specific constraint patterns. Therefore, a type with m patterns will need $m \times (m - 1)$ 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 2m 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).

2.3 System Framework

In this section, we give an overview of the constraint mapping framework (as Figure 5 shows), which starts from a source constraint 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 for the best one. In next section, we will discuss how to perform the search and how to evaluate the closeness of constraints in details.
3. CONSTRAINT MAPPING: A SEARCH-DRIVEN APPROACH

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.

3.1 The Translation Problem

As abstracted in Section 2, 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:

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.,

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

(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 Q1 and subject in Q2. The source constraint $s = \{\text{category: contain; "computer science"}\}$ is instantiated from template $S = \{\text{category: contain; Sval}\}$ by populating Sval="computer science." The target constraint template $T = \{\text{subject: SOP; Sval}\}$ accepts operators SOP from \{'any words\", \"all words\"\} (simply written as \"any\", \"all\" in the following paper), and value Sval 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.

$$P(s, t) = \frac{|s \land t|}{|t|}, \quad R(s, t) = \frac{|s \land t|}{|s|}$$

(2)

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

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

(3)

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

---

Figure 6: Search space of constraint mapping.

Figure 7: Venn Diagram of Precision and Recall.

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 type text and numeric to illustrate the principles in our search-driven approach.

3.2 Text Handler

Text is the most commonly used type in query interfaces 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 constraints, 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 $s$.

In Example 2, $W_s$ is thus restricted to \{\"computer\", \"science\"\}, and accordingly the values for Sval 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 queries words with its 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 $t_i$ 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 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

---

1 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"]
expression, reasoning on their containment relations is the regular
language containment problem, which has proven to be \textsc{psp}_{ace}-
complete [7].

To address the difficulty of closeness estimation, we employ an
“estimation-by-testing” approach. We query the two constraints \( s \) and \( t \) against a \textit{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 eval-
uated. 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 \textit{text} constraints \( \{s, t\} \), 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 \( s = [\text{category: contain; "computer science"}] \)
and \( t_1 = [\text{subject: any; "computer science"}] \), 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 \( t_1 \). According
to Equation 2, the precision and recall is \( \frac{3}{7} \) and \( \frac{1}{1} \) respectively.

Using the closeness metric in Equation 1, their closeness is thus \( \frac{3}{7} \).
Similarly, estimating \( \{s, t\} \) against \( D \) gets closeness 1. Therefore,
\( t_2 \) is a better mapping than \( t_1 \).

To fully reflect the relations between \( s \) and \( t_1 \), the dummy database
should captures 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 other random words. In the implementation, such customiza-
tion is realized by using an \textit{isomorphic} alphabet, e.g., \{w₁, w₂, \ldots, wₙ\},
and mapping the constant in \( W_s \) into the alphabet, e.g.,
"computer" \rightarrow w₁, "science" \rightarrow w₂. Figure 8 shows the
database on the isomorphic alphabet. By doing so, we do not need
to construct dummy database for every time to translate a query. To
make sure that the database capture 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 \textit{uniformed-
generated}: the length of the value follows a uniform distribution,
and the words in the database is randomly picked from the isomor-
phic alphabet.

Although evaluating against the dummy database may not al-
ways give the best mapping, it does eliminate the bad candidates
and give good ones. For example, the dummy database may assess
\( t_1 \) (with closeness \( \frac{3}{7} \) better than \( t_3 \) (with closeness \( \frac{1}{7} \), however,
the real distribution may suggest otherwise. Although the subtle
difference between the goodness of \( t_2 \) and \( t_1 \) cannot be captured by
our dummy database, it generates mapping of comparable quality
to the ones chosen by the user if he has no background knowledge
on the sources. In the future work, we will consider how to make
our database more “intelligent,” which captures the characteristics
of target database and also adequately test the constraints at hand.

3.3 Numeric Handler

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

Defining Search Space

Similar as \textit{text} handler, the challenge for \textit{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 em-
ploy 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 \textit{hill-climbing}. Starting from some initial solution, the hill-
climbing always go for a better solution (known as \textit{uphill move}).
Suppose we have only one variable \textit{Val} in target constraint tem-
plate \( T \) to be instantiated and we adapt a walk of fixed-length \( k \). At
any assignment of values to \textit{val}, we have 2 possible move-
ment as \( \textit{val} = \textit{val} + k \) or \( \textit{val} = \textit{val} - k \). The search performs a
series of uphill moves until it reaches a \textit{local optimal}. We can fur-
ther 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
data is applicable for numeric data to estimate the closeness met-
ric \( C \), we find that systematic reasoning on the numeric constraints
is possible due to the continuous nature of the numeric data. To
estimate the query results of numeric constraints, we map the con-
straint into ranges on the numeric line (e.g., \( 5 \ll 35 \) in Figure 2),
and therefore the false positives and false negatives can be evalu-
ated based on the coverage and overlapping of the two constraints,
as Figure 7(b) illustrates.

4. CASE STUDIES

In this section, we report our preliminary study on the perfor-
mance of our framework. The study simulates a \textit{query assistant}
system as the application scenario where the system automatically
fill 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 map-
ning can help users in interacting with those sources.

In the experiment, the patterns captured are the common ones
(with appearance more than 5) we collected during the survey. There
are 8 such patterns for \textit{text} type and 6 for \textit{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-
mance metric: \textit{number of correct mappings}, because it reflects the
amount of efforts the system saves for the user. The correct map-
pings are manually generated by looking at the source and target
constraints ourselves. Counting only the “absolutely” correct con-
straints is actually very stringent because it does not capture “how
wrong” an incorrect mapping is. An incorrect mapping may re-
trieve similar results as the correct one (i.e., still have good preci-
sion and recall although not the best). However, in our measure-
work 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 3.2), to define more comprehensive search space (Section 3.2); 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.

Last, Supporting successive queries: Currently the constraint mapping shoots for a best single constraint w.r.t. the source constraints. However, a best mapping may involve a series of constraints issued to the database. For instance, when translating constraints such as [title: any; database system] to [title: contain; $v$], the mapping generated is $v$="database". However, if we can issue a series of successive queries and combine their results, then the best mapping should be [title: contain; database] U [title: contain; system]. In the future work, we plan to support such successive queries to achieve better accuracy in translation.

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.

6. REFERENCES


