Light-weight Domain-based Form Assistant: 
Querying Databases on the Web

Zhen Zhang, Bin He, Kevin Chen-Chuan Chang

University of Illinois at Urbana-Champaign
{zhang2, binhe}@uiuc.edu, kcchang@cs.uiuc.edu

Abstract
The Web has been rapidly “deepened” by myriad searchable databases online, where data are hidden behind query forms. Helping users query alternative “deep Web” sources in the same domain (e.g., Books, Airfares) is an important task with broad applications. As a core component of those applications, dynamic query translation (i.e., translating a user’s query across dynamically selected sources) has not been extensively explored. While existing works focus on isolated subproblems (e.g., schema matching, query rewriting) to study, we target at building a complete query translator and thus face new challenges: 1) To complete the translator, we need to solve the predicate mapping problem (i.e., map a source predicate to target predicates), which is largely unexplored by existing works; 2) To satisfy our application requirements, we need to design a customizable system architecture to assemble various components addressing respective subproblems (i.e., schema mapping, predicate mapping, query rewriting). Tackling these challenges, we develop a light-weight domain-based form assistant, which can generally handle alternative sources in the same domain and is easily customizable to new domains. Our experiment shows the effectiveness of our form assistant in translating queries for real Web sources.

1 Introduction
Recently, we have witnessed the rapid growth of databases on the Web, or the so-called “deep Web.” A July 2000 survey [1] estimated 96,000 “search cites” and 550 billion content pages in this deep Web. Our recent study [3] in April 2004 estimated 450,000 online databases. On the deep Web, numerous online databases provide dynamic query-based data access through their query forms, instead of static URL links. For instance, amazon.com supports a query form for searching books on author, title, subject, etc. To help users explore the deep Web, it becomes increasingly important to facilitate users’ interaction with query forms. Therefore, this paper proposes to build a form assistant to help querying databases on the Web.

In particular, with the proliferation of sources in various domains, we often need to query “alternative” sources in the same domain (e.g., Books, Airfares). We observe that such a domain-based integration scenario is useful with broad applications. For instance, we may build a Metaquerier [5] to integrate dynamically selected sources relevant to user’s queries, where on-the-fly translation of user’s queries to these sources is necessary; we may build a domain portal (e.g., pricegrabber.com) to provide a unified access to dynamic online sources in the same domain with general translation techniques; or we may build a query assistant toolkit to suggest users with potential queries they are likely to issue in query forms, e.g., if a user fills the query form in amazon.com, the toolkit can suggest potential queries in bn.com. (Section 6 will discuss the implementation of such a query assistant toolkit.)

A core component of these applications is a dynamic query translator, which translates user’s queries between dynamically selected query forms in the same domain. In particular, we define the query translation problem as translating a user’s query from a source query form to a target one, which we believe is a foundation of many translation tasks in various applications. (Section 2 will present our formal definition of the query translation problem.) Existing works mainly focus on isolated subproblems of translation (e.g., schema matching, query rewriting) to study. The goal of our work, i.e., building a complete dynamic query translator, has thus not been extensively investigated.

As the applications mandate, such a query translator should have two properties: First, source-generality: We require the built-in translation techniques can generally cope with new or “unseen” sources. Second, domain-portability: We require the translator can be easily cus-
To better understand new challenges we are facing in building a form assistant for new domains. As a realization of such a query translator, we develop a light-weight domain-based form assistant. It is domain-based because it can generally handle alternative sources in the same domain with a manageable size of domain-specific knowledge. It is light-weight because customizing to a new domain only needs a little amount of efforts to encode domain-specific knowledge. To better understand new challenges we are facing in building this form assistant, let us look at a translation example.

**Example 1:** Assume we want to build a form assistant for Books domain, which can translate queries between any two book sources. In particular, consider a translation from a source query $Q_s$ (issued on a source form $S$) to a target form $T$ in Figure 1. $Q_s$ is a conjunctive query over four predicates $s_1 :$ [author; contain; Tom Clancy], $s_2 :$ [title; contain; red storm], $s_3 :$ [age; $\geq$; 12], and $s_4 :$ [price; $\leq$; 35], i.e., $Q_s = s_1 \land s_2 \land s_3 \land s_4$. The target form $T$ supports predicate templates on author, title, subject, and ISBN one at a time with an optional predicate template on price. Figure 1 shows one of the possible translations.

To translate $Q_s$ from $S$ to $T$ in the above example, we need to reconcile three levels of query heterogeneities:

**Attribute level:** Two sources may not support querying the same concept or may query the same concept using different attribute names. For instance, $S$ supports querying the concept of reader’s age, while $T$ does not. Also, $S$ denotes book price using price range, while $T$ using price.

**Predicate level:** Two sources may use different predicates for the same concept. For instance, price predicate in $T$ has a different set of value ranges from $S$. As a result, in the predicate level, we can only translate a predicate as “close” as possible. In particular, we set our closeness goal as minimal subsumption, i.e., to subsume the source query $Q_s$ with fewest extra answers.

**Query level:** Two sources may have different capabilities on querying valid combinations of predicates. In our example, form $T$ only supports queries on one of the attributes author, title, subject and ISBN at a time with an optional attribute price. Therefore, $T$ cannot query author and title together, while $S$ can.

To realize the source query, we need to reconcile the heterogeneities at the three levels and generate a query plan expressed upon the target form $T$. Such a plan, as Figure 1 shows, in general, consists of two parts: a union query $Q^*_T$ which is a union of queries upon the target form to retrieve relevant answers from the target database, and a filter $\sigma$ which is a selection to filter out false positives. (Further optimization is possible by logic transformation of the query plan, but this transformation does not change the semantics of the plan.) To minimize the cost of post processing, i.e., filtering, we want the union query $Q^*_T$ to be as “close” to the source query $Q_s$ as possible so that it retrieves fewest extra answers. $Q^*_T$ in Figure 1 is such a query. We will discuss its construction throughout the paper.

Building such a form assistant brings us two new challenges: predicate mapping and system architecture design. First, we need to develop a general predicate mapping mechanism that can be easily adopted to new sources and domains. While extensively studies have been done to the attribute level, known as schema matching, and the query level, known as capability-based query rewriting, little has been done to reconcile the heterogeneity at predicate level, which we name as predicate mapping. Existing work on predicate mapping [2] relies on per-source based rules to encode the mapping knowledge, which cannot achieve our goal of source-generality and domain-portability. Our approach is inspired by an intriguing insight that matching predicates often form localities, which are consistent with the notion of data types. This observation enables us to encode a more general mapping knowledge based on the types and thus motivates a type-based search-driven approach for mapping predicates.

Second, guided by the application requirements, we need to carefully design the system architecture of the form assistant. In particular, to realize source-generality, each component of the system should incorporate general techniques to cope with heterogeneities for new sources. To realize domain-portability, the system should require minimal amount of human involvement to customize the form assistant for new domains.

Tackling these challenges, we develop a light-weight domain-based form assistant. We realize this form assistant in a complete application—query assistant toolkit, which starts from extracting the query capabilities of query forms and finally generates suggested queries as output. We evaluate the entire system over real Web query forms in 8 domains and the experimental results show the promise of both our system design and predicate mapping techniques. In summary, the contributions of this paper are:

- **Framework:** To realize the dynamic query translator, we develop a light-weight domain-based form assistant, which can generally handle alternative sources in the same domain and is easily customizable to new domains.
- **Techniques:** At the core of the form assistant, we propose a type-based search-driven approach for predicate mapping by leveraging the mapping localities.

The rest of the paper is organized as follows: Section 2 formalizes the query translation problem. Section 3 presents the system architecture of the form assistant. Section 4 motivates the type-based search-driven predicate
mapping machinery and Section 5 discusses concrete techniques for realizing it. Section 6 presents our development of other components in building the form assistant. Section 7 reports our experimental results. Section 8 discusses related works and Section 9 concludes the paper.

2 The Query Translation Problem

In this section, we formalize the query translation problem. As Figure 1 illustrates, the input to the query translator is a source query $Q_s$ on a source form $S$ and a target form $T$. The output of translator is an expression with a filtering $\sigma$ applied upon a union query $Q'_s$. The union query is built upon a set of form queries, $f_1$, ..., $f_n$, where each form query is a valid way of filling the form $T$.

Such a translation, as Example 1 shows, needs to meet two goals: First, the union query must be a valid query. That is, it must be built upon only valid form queries $f_1$, ..., $f_n$ of the target form. Second, the union query must be “close” to the source query to minimize the selection as the post processing. As we can see, these two objectives are placed on the union query without considering the actual selection $\sigma$. The reason is that we can always apply the source query $Q_s$ as the tightest filter to remove false positives. We notice that some other works, e.g., [2], also studied the “optimality” of the filters, i.e., to choose a filter with fewest number of predicates. We do not consider this issue in our paper since it is straightforward to get an optimal filter for a union query.

We thus define the query translation problem as: Given a source query $Q_s$ and a target form $T$, among all the valid union queries of $T$, we choose the one semantically closest to $Q_s$ as the best translation. Next, we need to formally define valid union query and semantic closeness.

Query Model

The query model of a source describes templates of acceptable queries. Many specification languages, e.g., [18], have been proposed for describing general data sources, often in the form of datalog or context-free grammar. However, since developed for general purposes, those languages, although sufficiently powerful, are not very intuitive for expressing query forms. Therefore, we adopt a simple query model to describe the source capability, which we will discuss below. This query model can be automatically recognized by our previous work on form extraction [21]. Further, as Section 6 will discuss, when necessary, this model can be transformed into other specification languages and thus we can apply existing query rewriting techniques.

The query model of a form consists of vocabulary $\Sigma$ and syntax $\mathcal{F}$. Vocabulary $\Sigma$ specifies a set of usable predicate templates on the query form. A predicate template is a three-tuple [attribute; operator; value], with one or more variables as “placeholder” to be instantiated by concrete values. For instance, Figure 2(a) shows the vocabulary of the target form $T$ in Figure 1, which contains five predicate templates $\{P_1, \ldots, P_5\}$ on attributes author, title, isbn and price respectively. In particular, $P_1 = [\text{author}; \text{contain}; \text{Tom Clancy}]$.

Further, upon the vocabulary of predicate templates, the syntax $\mathcal{F}$ specifies all the valid combinations of these templates with respect to the form. Like many other specification languages, e.g., [18], our syntax focuses on conjunctive queries, because it is sufficient to capture the capabilities of most deep Web sources. We name a valid combination of predicate templates a conjunctive form.

We observe that deep Web sources often have two types of constraints on how predicate templates can be queried together. First, some predicate templates may only be queried “exclusively.” For instance, form $T$ allows only an exclusive selection among attributes author, title, subject and isbn; $P_1$, $P_2$, $P_3$ and $P_4$ can thus only appear one at a time (with an optional $P_5$). Second, a form may have “binding” constraints, which require certain predicate templates be filled as mandatory. For instance, form $T$ may require price not be queried alone and thus each form query must bind a predicate template from $P_1$, ..., $P_4$. In our modeling, we thus define $\mathcal{F}$ as all valid conjunctive forms that satisfy these two constraints. For instance, form $T$ requires one template from $P_1$, ..., $P_4$ with an optional $P_5$, and thus its syntax contains eight conjunctive forms $\mathcal{F} = \{F_1, \ldots, F_8\}$, as Figure 2(a) shows.

Based on the above modeling, we can define what a valid union query is given the target form. To begin with, we define a predicate $p_i$ as an instantiation of a predicate template $P_i$, i.e., assigning concrete values for the parameters in $P_i$. For instance, $p_1$ in Figure 2(b) is an instantiation of $P_1$. Further, a form query $f_j$ is an instantiation of a conjunctive form $F_j$, e.g., $f_1$ in Figure 2(b) is an instantiation of $F_1$. Overall, a valid union query on the target form is thus $f_1 \lor \ldots \lor f_n$, where $f_i$ is a form query. In particular, Figure 2(b) shows a valid union query $Q'_s$ on form $T$.

The reason we limit our focus on union without considering intersection is two folded: First, the union operation is “non-blocking,” which allows results to be incrementally constructed. Specifically, Web query results are often organized into multiple pages, with each page containing a set of data records. For union, we can pipeline the pro-
cessing of each page without waiting for complete input from operands. However, intersection operation is “blocking.” We are not sure whether a data record should be in the intersection result or not until we retrieve all pages from both operands. Second, an intersection usually can be more efficiently realized by a selection. Both intersection (e.g., \( A \cap B \)) and selection (e.g., \( \sigma_B A \)) achieve the same goal of removing false positives, but intersection needs to retrieve more results of \( B \). Therefore, our translation considers only union, which is consistent with many other works, e.g., [17, 18]. For simplicity, we will use query to refer to union query without ambiguity in the following discussion.

**Semantic Closeness**

To capture the closeness between a valid query and the source query, we need to define a closeness metric. In particular, we adopt the minimal subsuming metric \( C_{\text{min}} \), i.e.,

**Definition 1:** Given a source query \( Q_s \) and a target query form \( T \), a query \( Q_t^* \) is a minimal subsuming translation w.r.t. \( T \) if:

1. \( Q_t^* \) is a valid query w.r.t. \( T \);
2. \( Q_t^* \) subsumes \( Q_s \), i.e., for any database instance \( D_t \), \( Q_s(D_t) \subseteq Q_t^*(D_t) \);
3. \( Q_t^* \) is minimal, i.e., there is no query \( Q_t \) such that \( Q_t \) satisfies (1) and (2) and \( Q_t^* \) subsumes \( Q_t \).

We choose \( C_{\text{min}} \) as many other query rewriting works [15, 2] do, because it has several advantages: First, a \( C_{\text{min}} \)-translation does not miss any correct answer and contains fewest incorrect answers. Consequently, the overhead of filtering out “false positives” (in the selection operation) is minimal. Second, the \( C_{\text{min}} \) metric is database content independent, as the above definition indicates. Since human users usually are not aware of the content before querying a source, such content independent translation is consistent with users’ behavior. Also, in Section 5, we will apply this property to generate a complete database for subsumption testing. Third, it enables the separation of predicate mapping from query rewriting, as Section 3 will discuss.

The following example demonstrates a translation using the \( C_{\text{min}} \) metric.

**Example 2:** Consider the source query \( Q_s \) in Example 1 and three valid queries \( Q_{t1}, Q_{t2} \) and \( Q_{t3} \), as below, wherein predicate \( p_1, p_2, p_1^3 \) and \( p_2^3 \) were introduced in Figure 2.

\[
\begin{align*}
Q_{t1} & : (f_1 : p_1^1 \land p_1^3) \lor (f_2 : p_1 \land p_2^3) \\
Q_{t2} & : f_2 : p_1 \land p_2^3 \\
Q_{t3} & : f_1 : p_1
\end{align*}
\]

We can see that \( Q_{t1} \) and \( Q_{t2} \) subsumes \( Q_s \), while \( Q_{t3} \) does not, because it misses the price range between 0 to 25 and thus cannot be the best translation. Further, between \( Q_{t1} \) and \( Q_{t3} \), we can prune \( Q_{t3} \) because it subsumes \( Q_{t1} \) and thus cannot be the \( C_{\text{min}} \) translation. In fact, we can show that \( Q_{t1} \) is the \( C_{\text{min}} \) translation.

### 3 System Architecture

To realize the query translation as defined in Section 2, conceptually we need to search among all valid queries for the minimal subsuming one. This is certainly inefficient. To avoid exhaustive “search,” we want to “construct” the best translation. In particular, since translation essentially is to reconcile the heterogeneities at the three levels, it is thus desirable to address them separately, and then construct the translation by putting them together. That is, we first find the matching predicates, then map each pair of matching predicates individually by \( C_{\text{min}} \) metric, and finally find the \( C_{\text{min}} \) rewriting of the mapped query on the target form.

Is such separation possible? While the separation of predicate mapping from attribute matching is obvious, how about the separation of query rewriting from predicate mapping? To guarantee the correctness of such separation, we require that the mapped query, generated by mapping predicate individually, must be rewriteable to a minimal subsuming translation. This requirement has two implications:

1. The mapped query must be rewriteable to a valid translation (if a valid translation of the source query exists at all); second, the rewriting of the mapped query must be an overall minimal subsuming translation. The following property states that under the \( C_{\text{min}} \) translation metric, the mapped query meets these two requirements and thus predicate mapping and query rewriting can be separated.

**Property 1:** Given a source query \( Q_s \) and a target query model \( T \), the mapped query \( Q_t^* \) of \( Q_s \) must be rewriteable to a valid union query w.r.t. \( T \) that subsumes \( Q_s \), if there exists a valid translation for \( Q_s \) at all.

**Proof:** Without loss of generality, let us assume the source query \( Q_s = s_1 \land s_m \) where predicate \( s_i \) maps to target predicate template \( P_i \). The mapped query by mapping each individual predicate separately is thus \( Q_t^* = t_1 \land t_m \), where \( t_i \), as we will see in Section 5, is a union of target predicates instantiated from \( P_i \).

As we can see, \( Q_t^* \) touches a set of target predicate template \( \{P_1, \ldots, P_m\} \), which we denote as \( \mathcal{P}(Q_t^*) \) or \( \mathcal{P} \) for short. To rewrite \( Q_t^* \) to a valid union query is essentially to choose a right conjunctive form from \( T \) to cover the \( \mathcal{P} \). There are three situations in choosing such a form:

1. \( \mathcal{P}(Q_t^*) \) exists in \( \mathcal{P} \), this is the form we choose. Second, if there is no form exactly covering \( \mathcal{P} \), we choose a form \( F \) that covers maximal subset of \( \mathcal{P} \). Third, if there is no form that can cover a subset of \( \mathcal{P} \), we find no form that can be used for rewriting. For situation 1 and 2, we successfully find a form \( F \) that covers a subset of \( \mathcal{P} \). Without loss of generality, let assume \( F \) covers \( P_1, \ldots, P_k \). We can then use this form to realize \( Q_t^* \) by pulling out all \( P_i \in \mathcal{P} \setminus F \), i.e., \( Q_t^* = \sigma \cdot Q_t \), where \( Q_t = P_1 \land \ldots \land P_k \). We can easily rewrite \( Q_t \) into a valid union query by transforming it to a DNF \( Q_t^* \). Therefore \( Q_t^* \) is a valid union query as the translation of \( Q_s \).

For situation 3, where for any \( F \in T \), \( F \) covers a super set of \( \mathcal{P} \). Therefore, any valid union query must have to touch additional template than \( \mathcal{P} \). However, under our 1:1 mapping assumption, any translation uses only (subset of) matching predicate templates \( \mathcal{P} \). Therefore, no translation under this assumption will have a valid rewriting.
Putting together the three situations, our statement holds.

The above property shows that the mapped query must be rewritable to a valid subsuming union query, and next we show that this mapped query must be able to be rewritten to a $C_{min}$ union query.

**Property 2:** Given a source query $Q_s$ and a target query model $T$, the mapped query $Q'_s$ of $Q_s$ can be rewritten to a $C_{min}$ union query w.r.t. $T$ if both the predicate mapping and query rewriting use $C_{min}$ metric.

**Proof:** We proof a translation $Q'_1$ obtained by $C_{min}$ rewriting of the mapped query $Q'_s$ is the $C_{min}$ translation of $Q_s$, if $Q'_s$ is obtained by predicate mapping under $C_{min}$ metric. Let us first introduce the notations we use in the proof. We denote the source query as $Q_s = s_1 \land \ldots \land s_m$, its mapped query $Q'_s = t_1 \land \ldots \land t_m$, where $t_i$ instantiates template $P_i$ and is the $C_{min}$ predicate mapping of $s_i$. Without loss of generality, let us assume the minimal subsuming rewriting of $Q'_s$ is $Q'_1$, which instantiates template $P_1, \ldots, P_k$. Let us transform $Q'_1$ back to CNF format, i.e., $Q'_1 = t_1 \land \ldots \land t_k$.

We prove our statement by contradiction. Let assume there exists a union query $Q_t = f_1 \lor \ldots \lor f_l$ which is the $C_{min}$ translation of $Q_s$, i.e., $Q'_1 \supseteq Q_t$.

We first proof that each $f_j$ instantiates a superset (or equal) of templates than $Q'_1$. Since $Q'_1$ subsumes $f_1 \lor \ldots \lor f_l$, $Q'_1$ subsumes any $f_j$. Let $f_j = p_1 \land \ldots \land p_h$, Note that all $t_i$ of $Q'_1$ are independent of each other and so does $p_j$ of $f_j$.

Therefore, in order for $t_1 \land \ldots \land t_k$ subsumes $p_1 \land \ldots \land p_h$, for any database instance, the following must be true: for any $t_i$ of $Q'_1$, there exists a $p_j$ of $Q_t$ such that $t_i \supset p_j$. Since predicates of different templates are independent, in order for $t_i \supset p_j$, they must instantiate the same template. This means that each $f_j$ instantiates a superset (or equal) of templates than $Q'_1$.

Further, we observe that $f_j$ must instantiate exactly the same set of templates as $Q'_1$ if the rewriting is minimal subsuming. The reason is that, if $f_j$ instantiates an additional template, this means that there is a conjunctive form which support a super set of templates used in $Q'_1$. Without loss of generality, let us assume this additional template is $P_{k+1}$, then the minimal subsuming rewriting of $Q'_s$ should be $Q'_1 = P_1 \land \ldots \land P_{k+1}$, which subsumes $P_1 \land \ldots \land P_k$. This is contradictory to our assumption that $Q'_1$ is minimal subsuming rewriting of $Q'_s$. Therefore, $Q'_1$ and $f_j$ instantiate exactly the same set of templates, $\{P_1, \ldots, P_k\}$.

Since every $f_j$ instantiates the same set of templates as $Q'_1$, $Q'_1$ and $Q_t$ thus instantiate the same set of templates $\{P_1, \ldots, P_k\}$. This means that $Q'_1$ and $Q_t$ both are translations of the query $s_1 \land \ldots \land s_k$ and $Q'_1$ subsumes $Q_t$. However, the work in [2] shows that under the independence assumption, the $C_{min}$ translation of $s_1 \land \ldots \land s_k$ is obtained by mapping individual predicates under $C_{min}$ metric, which makes $Q'_1 = t_1 \land \ldots \land t_k$ the $C_{min}$ translation of $s_1 \land \ldots \land s_k$. Therefore, $Q'_1$ subsumes $Q_t$. This is contradictory to our assumption that $Q_t$ subsumes $Q'_1$. With this contradiction, We therefore conclude that $Q'_1$ is the minimal subsuming translation of $Q_s$. □

![Figure 3: System architecture of the form assistant.](image)

The separation thus enables a modular design for the form assistant, which consists of three components: attribute matcher, predicate mapper and query rewritser, as Figure 3 shows. First, attribute matcher discovers semantically corresponding attributes. Then, for each pair of matching predicates, predicate mapper maps the source predicate to the target one with the $C_{min}$ metric. After matching attributes and mapping each individual predicate, we get a mapped query. Finally, query rewritser rewrites the mapped query to a valid query in terms of the capability of the target form. Example 3 illustrates the functionality of each component with a concrete example.

**Example 3:** Consider the translation example in Example 1, where source query $Q_s = s_1 \land s_2 \land s_3 \land s_4$. First, attribute matcher will find that $s_1$, $s_2$ and $s_4$ have matching attributes in $T$, while $s_3$ does not. Then, predicate mapper will map each pair of matching predicates with the $C_{min}$ metric. In particular, for $s_1$, the mapped predicate is $p_1 = s_1$; for $s_2$, the mapped predicate is $p_2 = s_2$; for $s_4$, the mapped predicate is $p_3 = [\text{price}; \text{between}; 0-25]$ and $p_4 = [\text{price}; \text{between}; 25-45]$. After matching and mapping, the mapped query is thus $Q'_s = p_1 \land p_2 \land p_3 \land p_4$. Finally, applying the query rewriting, we can rewrite $Q'_s$ into $\sigma_{p_5}(f_1 \cup f_2)$, where $f_1 = p_1 \land p_3$ and $f_2 = p_1 \land p_4$. The union query $f_1 \cup f_2$ is thus the best translation $Q'_1$ for source query $Q_s$, and the selection $\sigma_{p_5}$ is a part of the final filter, which we omit here.

Such modularization on one hand reduces the complexity of translation– Each component only focuses on a subproblem to solve. On the other hand, it allows us to plug in, for each component, corresponding techniques suitable for specific application needs. In particular, attribute matching and query rewriting have been extensively studied. Our form assistant can thus take advantage of the existing techniques by taking them as building blocks of the system.

To prepare the input for the form assistant, we explore the form extractor we developed in [21] to automatically extract query capabilities of sources. Next, we need to design each component of the form assistant to satisfy the application requirements, i.e., source-generality and domain-portability. The key is to identify, for each component, what knowledge can be built-in as generic techniques and
what should be customized as domain-specific knowledge. Further, we want the customized knowledge to be per-domain based instead of per-source, and we want to keep such knowledge within human-manageable scale.

**Attribute Matcher:** To match attributes is essentially to identify synonym attributes. This problem, known as schema matching, has been an important problem and thus extensively studied in data integration. Depending on the application scenarios of the form assistant, we can employ either automatically discovered or manually encoded synonym thesaurus, based on which we can explore a general thesaurus-based matching approach. For instance, for the MetaQuerier system (discussed in Section 1), since sources are on-the-fly selected and thus form dynamic domains, the thesaurus has to be automatically discovered by exploring existing automatic matching techniques, e.g., [10, 11]. While for domain portals and query assistant toolkit, since sources are in pre-configured domains, the thesaurus can be manually customized with more reliable domain-specific knowledge. Our survey [3] on query forms shows that, for each domain, there only exist a small number of frequently used attributes, which makes the manual encoding of such a domain-specific thesaurus feasible in practice.

**Predicate Mapper:** To map predicates, we need to know, for each pair of matching predicates, how to choose the operators and fill in the values. To achieve the requirement of easy deployment for new domains, we thus ask whether there exists domain-generic mapping mechanism which addresses most if not all mappings in various domains. We observed that, as an enabling insight, predicates with the same data type often share similar mapping patterns. Motivated by this observation, we develop a type-based search driven mapping approach with several built-in type handlers to handle the majority cases of general “built-in” types. For those domain-specific mappings (e.g., airport code to city name in airfares domain), we can customize the required mapping knowledge by adding new type handlers if needed.

**Query Rewriter:** To tackle the query-level heterogeneity is essentially to rewrite the query to satisfy the syntax constraints of the target form. Such a task, has been studied as capability-based query rewriting [17, 19, 14], is not specific to any source or domain. Hence, as Figure 3 shows, the query rewrite component can be entirely built-in into the system. Section 6 will discuss our development of this component by exploiting existing rewriting techniques.

### 4 Predicate Mapping: The Motivation

With extensive existing study on schema matching and query rewriting, together with our modular separation of predicate mapping from query rewriting, the essential challenge of translation boils down to predicate mapping. As Section 3 motivated, the requirement of easy-customization calls for domain-generic mapping knowledge. Does there exist such generic mapping knowledge applicable to sources across different domains? How to encode such knowledge? Those are the critical questions we need to address in predicate mapping.

Existing solutions on predicate mapping usually assume a static small-scale setting. In such scenarios, it is common, e.g., as [2] studies, to use a source-based pairwise-rule driven mapping machinery to map predicates. Figure 4 gives some example rules to encode the mapping knowledge required for translation in Example 1.

However, such a mapping machinery, characterized by a “per-source” knowledge scope and “pairwise” rule encoding mechanism, is not suitable for our light-weight domain-based form assistant, since it lacks both generality and extensibility. To begin with, the source-based scope can only handle mappings between sources whose mapping knowledge has been specified. It cannot generally deal with mappings involving “unseen” sources. Also, the pairwise-rule mechanism to encode the knowledge is not extensible because writing rules for each pair of predicate templates is hard to scale when mapping are required among a large set of sources. Further, as sources are autonomously developed, their query capabilities may change, which makes maintenance of rules labor intensive.

#### 4.1 A Motivating Survey

To understand the feasibility of developing a more general and extensible mapping machinery, we thus ask: Are there generic scopes orthogonal to domains wherein the mapping happens? If such scopes exist, we thus can leverage them to encode our domain generic mapping knowledge.

To answer this question, we conduct a survey to examine the mapping correspondence between predicate templates. Such mapping correspondence hints the required translation scope: only templates with mapping correspondence needs to be translated. In particular, we examined 150 query forms in three domains, i.e., Books, Airfares and Automobiles, in the TEL-8 dataset of the UIUC Web Integration Repository [4]. This TEL-8 dataset contains around 440 manually collected deep Web sources with their query forms. We totally find 37 template patterns in 150 sources. For instance, as two popular patterns, [attr: default: $val]
uses a default operator and accepts arbitrary values from the user input (e.g., template $P_1$ in $T$ belongs to this pattern), and pattern [attr: default; $\text{val}\in\{\text{D}\}$] accepts a value from a set of given options (e.g., $P_2$ in $T$ belongs to this pattern). In the rest of the section, we will use template to generally refer to template pattern for simplicity.

We notice that two predicate templates have mapping correspondence only if there exists a concept expressed with these two templates in different sources. For example, to support querying book subject, a source may use [subject: default; $\text{val}$] and another [category: default; $\text{val}\in\{\text{D}\}$]. Since subject and category are matching attributes, templates [attr: default; $\text{val}$] and [attr: default; $\text{val}\in\{\text{D}\}$] need to be translated and thus have mapping correspondence. We record such mapping correspondence between any two predicate templates $P_i$ and $P_j$ in a correspondence matrix $CM$. In particular, $CM(i,j)$ denotes the number of concepts that are expressed using both templates $P_i$ and $P_j$. Figure 5 shows our survey result (i.e., the correspondence matrix), where the value of $CM(i,j)$ is illustrated as the degree of grayness.

As Figure 5 indicates, mappings happen mostly only within certain clusters of templates. In the figure, we order the templates in a way to clearly observe such localities: In particular, we observe three localities of templates among which mapping happens. Such mapping localities thus formulate the scope of mappings.

We further ask: What templates can share a locality? Are there something common among templates in the same locality? Not surprising, we found there is indeed an underlying explanation for the phenomenon. While the predicate templates in a locality are used by various concepts in different domains, those concepts often share the same “data type.” In particular, the first locality in Figure 5 corresponds to templates usually used by concepts of datetime type, the second one numeric type and the third one text type. The only outlier is the template [attr: default; $\text{val}$], as shown in Figure 5. Concepts with different data types can all use this template to express their query predicates and this template thus has mapping correspondence to most other templates.

4.2 Type-based Search-driven Predicate Mapping

Our observations above clearly reveal that mapping between predicate templates are not arbitrary– Their associations not only suggest “locality,” but also are clearly “aligned” with their underlying data types. Such observation motivates a type-based search-driven predicate mapping machinery.

First, our observation shows the promise of pursuing a type-based knowledge scope. Since mappings are only worthy inside localities of types, we can express the mapping knowledge in the scope of types. Such a type-based mapping scope is more general than the source-based or domain-based scope, since data types are widely reused across different sources and domains, and we are able to translate queries to unseen sources or adapt the knowledge to new domains as long as they reuse those data types.

Second, we find that data type gives us a “platform” to compare semantics (i.e., the subsuming relationship) of different mappings and thus enables an extensible search-driven mechanism. Instead of writing rules to hard code the mapping for every pair of predicate templates, we can encode our mapping knowledge as an evaluator for each template. This evaluator “materializes” the semantics of a query against a type specific platform, as Section 5.1 will discuss. By doing so, semantic comparison can thus be performed on the materialized results. For instance, to realize rule $r_3$ in Figure 4, we can project the source predicate as well as all target predicates onto an axis of real numbers, and thus compare their semantics based on their coverage. Finding the closest mapping thus naturally becomes a search problem - to search for the ranges expressible in the target form that minimally cover the source. Such a search-driven approach achieves extensibility by exploring evaluators rather than static pairwise rules. With a data type of $n$ templates, we only need $n$ evaluators instead of $n^2$ rules; when adding a new pattern, only one evaluator needs to be added instead of $n$ rules.

5 Predicate Mapping: The Solution

In this section, we discuss our development of predicate mapper, which realizes the type-based search-driven mapping machinery. According to our system design (Section 3), predicate mapper takes a source predicate $s$ and a matched target predicate template $P$ as input, and outputs the closest target translation $t^*$ for $s$. In particular, predicate mapper consists of two components: type recognizer and type handler, as Figure 6 shows. Type recognizer recognizes the data type of the predicates and then dispatches to a corresponding type handler. A type handler maps predicates of a specific type with a search approach.

Since our focus in this section is type handler, we only briefly discuss the development of type recognizer. In particular, since the data type of a predicate can often be hinted by its syntactic features, we explore those syntactic clues to implement the type recognizer. For instance, we can exploit distinctive operators (e.g., all, any for text type), the value filled in the source predicate (e.g., $s_3$ in Example 1 contains value 35) and the value domain in the target template (e.g., a selection list of $P_5$ in Example 1) to infer type information. We have developed a type recognizer in our previous work for schema matching [11], wherein the type information is used to help matching. Due to space limitation, please refer to [11] for more details.

5.1 Overview of Type Handler

At the core of the predicate mapping, each type handler realizes the search-driven mechanism for its responsible type. Like any other search-based algorithms, a type handler needs to have three key components: search space, closeness estimation, and search strategy.

Example 4: Consider a predicate mapping problem. The source predicate is $s = [\text{category}; \text{contain}; \text{"computer science"}]$ and the target predicate template is $P =$
Defining Search Space

Defining a reasonable search space is essential to any search process, because it impacts the complexity of search significantly. Given a predicate template $P$, we define the instance space of $P$, denoted by $I(P)$, as all possible instantiations (i.e., predicate) of $P$. Due to the predicate-level heterogeneity, we often need to map a source predicate into multiple target predicates (e.g., the price predicate in Example 1). We thus define the search space of $P$, denoted by $\Omega(P)$, as any disjunction of predicates in $I(P)$. Each item in $\Omega(P)$ thus corresponds to a possible mapping.

As $I(P)$ often can be infinite, we take a “close-world” assumption to shrink the search space. Specifically, while operators are clearly limited by $P$ (e.g., two operators any, all in Example 4), the values can often be infinite--for a predicate template without pre-defined domain of values (e.g., an attribute name with a text input box), any value can be filled in and thus constitutes an infinite instance space. To define a reasonable space of $I(P)$, we make a “closed-world” assumption: We denote the values filled in the source predicate $s$ as $W_s$. For a target template without pre-defined domain of values, we assume the values of the target predicate can only choose from $W_s$. Therefore, we define the instance space $I(P)$ as all possible instantiations of $P$ using values either in a pre-defined domain if it is available or composed from $W_s$ otherwise. For instance, in Example 4, the target template $P$ is thus restricted to the words used in the source predicate, i.e., $W_s = \{ \text{"computer"}, \text{"science"} \}$. Figure 7 shows the instance space and part of the search space of $P$. In particular, with the close-world assumption, the instance space of $P$ contains 6 possible instantiations.

Such a close-world assumption is reasonable, because without domain-specific knowledge, it is difficult to invent new values. In fact, the search space can be enriched (e.g., by expanding queries words with their synonyms) if more domain knowledge is available (e.g., by providing synonym thesaurus), as we will discuss in Section 5.3.

Closeness Estimation

Given the search space $\Omega(P)$ covering all possible mappings, finding a $C_{min}$ mapping boils down to inferring subsumption relationship between a mapping and the source predicate, and between two mappings. This inference is specific to data types—For some types, it is straightforward, while others, it is not. In particular, for numeric type, since a predicate with numeric type can be projected to an axis of real numbers, evaluating subsumption relationship upon the numeric axis becomes an easy task. Datetime type can be processed in similar way as numeric type and thus we discuss their handlers together in Section 5.2. However, for text type, the inference of subsumption relationship is not trivial since it essentially needs logical reasoning. To avoid convoluted logical inference, we develop an equivalent “evaluation-by-materialization” approach. Section 5.2 will discuss this approach in details.

Search Algorithm

With the subsuming testing, we can find the $C_{min}$ mapping using the following simple algorithm, where $s, P$ is the source predicate and target template, $H$ records all subsuming mappings and $R$ records $C_{min}$ mappings and $x$ is the returned mapping:

1. $H = \emptyset, R = \emptyset$
2. for $\forall t \in \Omega(P)$:
   3. if $\text{subsume}(t, s)$: add $t$ to $H$
4. for $\forall t \in H$:
   5. if $\nexists t' \in H$ and $t' \neq t$ such that $\text{subsume}(t, t')$
   6. add $t$ to $R$
7. choose an $x \in R$ with minimal number of predicates

This algorithm basically exhausts the entire search space $\Omega(P)$ to find all $C_{min}$ mappings, and then chooses the one with minimal number of predicates to minimize the number of form queries issued on the target form. To improve its efficiency, we may refer to an approximate algorithm. In particular, we can take a greedy approach in practice: We find the mapping iteratively and in each iteration, we look for a (not-selected) instantiation from $I(P)$ that maximally covers the uncovered part of the source predicate $s$ until we can entirely cover $s$. (In case there is no subsuming mapping for $s$, this approach will generate a maximal subsumption mapping, which is very desirable.)

5.2 Built-in Handlers

Text Type Handler: The $\text{Text}$ is the most commonly used type in query forms for querying string-based fields (e.g., subject in Example 4). Operators of $\text{Text}$ type predicates are typically string match operations including all, any, exact, etc., and values are strings.

We will use the mapping task in Example 4 to illustrate
how the search proceeds towards finding the best mapping of the target template $P$. In Figure 7, we have illustrated the instance space and search space of $P$. Next, we need to develop an approach to evaluate the subsumption relationship of text type.

While it may be possible to logically reason the subsumption relationship, such an approach can be very convoluted. On one hand, it is usually difficult to encode the logic relationship, e.g., $\text{contain}(v_1, v_2) = \text{any}(v_1) \land \text{any}(v_2)$ and $\text{contain}(v_1) = \text{any}(v_1)$. It is also non-trivial to apply those rules to reason between complex predicates. On the other hand, it is not extensible, since adding a new operator (e.g., $\text{start with}$) requires encoding its relationship with every existing operators (e.g., $\text{any}$, $\text{all}$, $\text{exact}$).

To circumvent such reasoning, we employ an “evaluation-by-materialization” approach. The idea is that, instead of semantically reasoning the subsumption relationship between two queries $Q_1$ and $Q_2$, we can “materialize” them against a database instance $D$ and compare the query results $Q_1(D)$ and $Q_2(D)$. The question is which database instance can be used to reliably test the subsumption relationship? It is obvious that we cannot arbitrarily choose a database instance $D_i$. We observe that a “complete” database $D$, which conceptually is the union of all possible database instances $D_i$, satisfies our requirement.

**Property 3:** Given two queries $Q_1$ and $Q_2$, and a complete database $D = \bigcup D_i$, $Q_1$ semantically subsumes $Q_2$ if and only if $Q_1(D) \subseteq Q_2(D)$.

The “only if” direction obviously holds, and we only show the “if” direction informally. Let us suppose that $Q_1(D) \subseteq Q_2(D)$ but $Q_1$ does not semantically subsume $Q_2$. Then there exists a database instance $D_i$ such that $Q_1(D_i) \nsubseteq Q_2(D_i)$, i.e., $\exists x \in Q_1(D_i) \land x \notin Q_2(D_i)$. This means that $Q_2(x)$ is false and thus for any database instance $D_j$, $Q_2(D_j)$ cannot contain $x$. Therefore, $Q_2(D)$ cannot contain $x$, and $Q_1(D) \nsubseteq Q_2(D)$. This is contradictory to our assumption and thus the property holds.

Next, the question becomes how to construct such a complete database? Conceptually, a complete database is a complete enumeration of all possible phrases composed from English words (assuming we only handle English). This is simply too huge to construct and we thus need to look for a practical solution.

We observe that the “completeness” of a database depends on the “logical properties” to be tested by operators. For instance, operators contain and any concern only membership of words in a string field without caring their ordering, while operators start and exact concern both membership and sequence. In practice, we observe that membership and sequence can almost cover all the operators for text type. Therefore, we construct the database using words from $W_s$ plus some additional random words. The database is composed of all possible combinations of the words (for testing the membership) with all possible orders (for testing the sequence). Since the exact set of additional words used in the test database does not matter, we thus can use a small number of random words to generally represent all words outside $W_s$. In implementation, to avoid constructing database for every mapping, we use a set of five fixed words to construct a static database. At runtime, we dynamically designate each word to represent a particular constant in $W_s$. If the size of $W_s$ is greater than 5, we will dynamically construct a test database.

To test subsumption between a mapping and the source predicate, we build a database $D$ with alphabet $D = \{\text{computer, science, dummy}\}$, as Figure 8 shows the content. For simplicity, we use only initials for corresponding words. Figure 8 shows, for source predicate $c$ and each mapping $t_i$ in the search space, the result set against the complete database. As we can see all the listed mappings are subsuming mapping. Among them, $t_5$ is $C_{\text{min}}$ mappings as it subsumes no other mappings. Alternatively, the greedy approach searches over all predicates in $I(P)$ can also find the right mapping $t_5$, but it is much more efficient.

**Numeric & DateTime Handler:** Numeric and datetime have very similar nature in that they all form a linear space, and have similar operators, such as $\text{before vs.} \leq, \text{during vs.} \text{between}$ and so on. In fact, datetime is internally presented as an integer in many programming languages. Therefore, the mapping techniques for the two types are generally the same. In this section, we will thus focus on numeric type and the general discussion applies to datetimelike type too.

We use an example to run through the mapping process. Consider mapping between the $\text{price}$ predicates in Example 1. Since the target predicate has a pre-defined domain of values, each representing a range, our search space is restricted to disjunctions of those ranges. To estimate the closeness, we project the source predicate $c : [\text{price}; \leq: 35]$ and the target predicate, e.g., $p_1 : [\text{price}; \text{between}; 0, 25]$ into ranges in numeric line. By projecting $c$ and $p_1$ to the numeric line, their false positive range (empty) and false negatives (25,35) can be straightforwardly evaluated as their coverage on the line. Using the greedy search approach, we will choose the one that maximally covers the uncovered range. Therefore, we choose range $(0, 25)$ and $(25, 45)$ in turn, which forms our mapping $[\text{price}; \text{between}; 0, 25] \lor [\text{price}; \text{between}; 25, 45]$.

<table>
<thead>
<tr>
<th>Complete Database D: C, S, D, CS, CD, SC, SD, DC, DS, CSD, CDS, DSC, DCS, SCD, SDC</th>
<th>Materialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e : [\text{category}; \text{contain}; \text{CS}]$</td>
<td>CS, SC, CSS, CDS, DSC, DCS, SCD, SDC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Materialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1 : [\text{subject}; \text{any}; \text{C}]$</td>
<td>C, CS, CD, SC, DC, CSD, DCS, DSC, SCD, SDC</td>
</tr>
<tr>
<td>$t_2 : [\text{subject}; \text{all}; \text{C}]$</td>
<td>same as above</td>
</tr>
<tr>
<td>$t_3 : [\text{subject}; \text{any}; \text{S}]$</td>
<td>S, CS, SC, SD, DS, CSD, DSC, DCS, SCD, SDC</td>
</tr>
<tr>
<td>$t_4 : [\text{subject}; \text{all}; \text{S}]$</td>
<td>same as above</td>
</tr>
<tr>
<td>$t_5 : [\text{subject}; \text{all}; \text{CS}]$</td>
<td>CS, SC, CSS, CDS, DSC, DCS, SCD, SDC</td>
</tr>
</tbody>
</table>

Figure 8: Example of database and evaluation results.
5.3 Domain Specific Type Handler

While the type handlers discussed above can handle generic types independent of domains, there are situations where domain specific knowledge is required. First, while our built-in type handler only syntactically maps values between source and target predicate values, some mapping needs domain-specific thesaurus to find out the semantic correspondence. For instance, to map from [subject; equal; computer science] to [subject; equal; computer architecture], we need a subject ontology to know computer science is a hyponym of computer architecture. Second, while the built-in type handlers only deal with 1:1 mappings, we need domain-specific procedures to realize \( m : n \) mappings, e.g., to map from lastname, firstname to author.

There are two ways to implement domain specific type handlers. The first implementation is to plug in a full-fledged type handler, which takes the matching predicates as input and outputs the mapping result. This handler is thus a black box to the system, which performs whatever computation to generate the mapping. The second implementation is to plug in a domain thesaurus, which on one hand is used for generating terms for defining the search space, e.g., to map from city name to airport code, and on the other hand is for subsumption testing, e.g., to tell that “computer science” subsumes “computer network.” Therefore, this implementation requires two interfaces from the thesaurus− interface lookup takes a value as input and outputs the set of values which is semantically related to the input; interface subsume takes two values as input and output whether one is a hyponym of another. With such a thesaurus, the same search algorithm can be used for finding the best mapping.

Let us use an example to illustrate the second implementation of domain-specific type handler. Consider a mapping from [airport code; equal; LAX] to [city name; equal; SFO], where a set of searchable city name is enumerated in a selection list. To realize a specific type handler for this mapping, we provide a thesaurus with a subsume interface which takes an airport code and a city name as input and output whether one subsumes the other. (Note that since the target template has a pre-defined domain of values, we do not need lookup interface to define the search space.) With this interface, we can perform the same search algorithm, which tries different city names one by one to find the minimal subsuming mapping.

6 Implementation: Form Assistant Toolkit

With form assistant as a core component to many applications, in this section, we study an example application—building a query assistant toolkit, to concretely evaluate the effectiveness of our approach. Query assistant toolkit is a browser based toolbar, which gives users suggested translations across sources in the same domain. In particular, users can register a query of her interest, e.g., a query on amazon.com, and when she browses other sources in the same domain, e.g., barnesandnoble.com, she can ask for suggested translation. This toolbar is pre-configured with a set of supported domains, and users need to specify the domain of interaction when registering a query.

When activated, the assistant will automatically translate the source query to the target form. To prepare the input to translation, the assistant has a form extractor module to automatically extract the query capability of the target form. With predicate mapping addresses the predicate heterogeneity, in this section, we briefly discuss the remaining three components - form extractor, attribute matcher and query rewriter to complete the framework. Please refer to our extended report [22] for more details.

Form Extractor: Form extractor automatically constructs the query capability of a form to prepare the input for form assistant. We have studied the problem of form extraction in [21], which extracts the set of predicate templates as vocabulary from a query form. To further construct the syntax of query capability, we identify the exclusively queried attribute and the required fields in query forms. The exclusive predicates are usually presented in a pattern, where a set of attributes are enumerated in radio-button or selection list, with a textbox accepting input. Based on this pattern, we thus refer to a set of commonly queried attributes in a domain to further identify those exclusive predicates. The set of common attributes is preconfigured as domain knowledge, which is used in attribute matching as well. Identifying required fields is a challenging task because there usually is no standard way of indicating those fields. However, most existing rewriting techniques can accommodate this problem by pushing as many predicates as possible to each form query. We will discuss this issue in query rewriter module.

Attribute Matcher: Attribute matcher identifies the semantical corresponding attributes. Our attribute matcher is customized with a domain thesaurus, which stores synonyms for commonly used concepts. At runtime, attribute matcher fuzzily matches the input attributes with those synonyms to check whether they express the same concept or not. In implementation, we use a hash table to map each synonym (after normalization), which we call indexed attribute, to the corresponding concept. In particular, the attribute matcher consists of two steps:

Preprocess: Given two predicates, the preprocess step performs some standard normalization, including, stemming, normalizing irregular nouns and verbs (e.g., “children” to “child,” “colour” to “color”) and removing stop words.

Synonym check: After preprocessing, synonym check checks whether the two predicates match. We view input predicates as matched if their types match and their attributes are synonyms. The reason for checking on types is that some attributes, e.g., departure, arrival in airfares, must be accompanied by their type to reflect its semantics (e.g., departure city or departure date). To do so, we recognize the types of the predicate using the same type recognizer used in predicate mapping of Section 5. If their type matches, we further check whether they are synonyms. In particular, for each input attribute, we first search for
Algorithm: Rewriter($Q'_s$, $T$)
1. Input: Mapped query $Q'_s = \land (t_1, \ldots, t_k)$
2. Target form $T = \{F_1, \ldots, F_m\}$
3. Output: $\text{minsub} \leftarrow \text{C}_{\text{min}}$ translation of $Q'_s$
4. Begin:
5. /*Finding maximal coverage of $P(Q'_s)$ */
6. $F_{\text{max}} = \arg\max_F \{ F \cap P(Q'_s) \}$
7. covered = $\{ t_i | P(t_i) \in F_{\text{max}} \}$
8. /*Transform to a union query */
9. $Q \leftarrow t_i \land \text{covered}$
10. $\text{minsub} \leftarrow \lor (Q)$
11. End

Figure 9: Minimal subsuming query rewriting.

its “representative,” which is an indexed attribute with the highest similarity score above a predefined threshold. If the system successfully finds the representatives for both attributes and both representatives correspond to the same concept, we return them as matching attributes.

Query Rewriter: Query rewriter takes a mapped query and constructs a minimal subsuming query as the best translation. The core of this translation is a well-studied problem known as capability-based rewriting. Many existing solutions, e.g., [18, 15, 14], can apply here by transforming our query model into their specification languages. In particular, our rewriter builds upon the techniques developed in [17], which generates a minimal subsuming rewriting of a query. Essentially, this approach pushes as many predicates as possible to each form query so that it maximizes the use of source query capability.

We observe that we can optimize the algorithm in [17] by taking advantage of the restricted format of the mapped query for more efficient rewriting. In particular, the mapped query $Q'_s$ generated from predicate mapping is in a CNF format $t_1 \land \ldots \land t_k$, where $t_i = p_i \lor \ldots \lor p_{i_m}$. Each $t_i$ is a union of predicates instantiated from the same template, and we denote this template as $P(t_i)$. Therefore, the mapped query overall instantiates a set of predicate templates $\{P(t_1), \ldots, P(t_k)\}$ in $T$, which we denote as $P(Q'_s)$.

The proof of Property 2 in Section 3 already showed our process of constructing a $\text{C}_{\text{min}}$ rewriting of a mapped query. In this section, we present the algorithm more formally.

To rewrite $Q'_s$ to a minimal subsuming union query, our algorithm proceeds in two steps—finding maximal coverage of $P(Q'_s)$, transforming the query to a valid union query. Figure 9 outlines the algorithm. Given a mapped query $Q'_s$ and a target form $T$, the first step finds, among all conjunctive forms $F \in T$, the one which covers the maximal number of templates in $P(Q'_s)$. This step generates a CNF query $Q_t$ which minimally subsumes the mapped query w.r.t. $T$.

The second step further transforms $Q_t$ into an union query. This can be done by rewriting the query from CNF format to DNF format, which is straightforward: Since each $t_i = \land (p_{i_1}, \ldots, p_{i_m})$ is a conjunction of a set of predicates from the same template, we can take a cartesian product of $t_i$ in $Q_t$. Each item in the product is thus a form query, and the union of those items is the $\text{C}_{\text{min}}$ translation of the source query $Q_s$.

In our current development, given a minimal subsuming translation as a union of form queries, we present the form queries, e.g., $f_1, f_2$, in different HTML frames and user can decide which one to submit or submit all. We will investigate more GUI issues in our future study.

Finally, we discuss the issue of incomplete modeling of required fields in form extraction. Consider the target query form $T$, if we fail to identify that at least one of the four predicates on author, title, subject, ISBN is required, we may enlarge the space of valid translations by including “false” queries, e.g., [price; between; 0, 25]. However, the query rewriter will not choose this query because it is not minimal subsuming—more predicates, e.g., $p_1$, can be pushed to the form query to make it more specific. Therefore, as long as the source query specifies the corresponding predicates, which can be mapped to the required fields, our translation will not abuse those “false” queries even if we do not explicitly capture those required fields. (If the source query does not specify those required field, we cannot have any valid translation at all.)

7 Experiment

Experimental Data: To evaluate the complexity of deploying the form assistant and the accuracy of the translation, we collected 120 query forms from 8 domains of the TEL-8 dataset [4]. In particular, we separate the forms into two datasets - Basic dataset and New dataset. The Basic dataset contains 60 query forms with 20 from each of the three domains we surveyed (Section 4.1). The purpose is to evaluate the complexity of domain knowledge needed for translation and the performance over sources from which the types and their patterns are derived. The New dataset contains 60 query forms, which we sampled from 5 domains (i.e., CarRentals, Jobs, Hotels, Movies and MusicRecords) in the TEL-8 dataset. The purpose is to test how well the type handlers can apply to new domains.

Experiment Setting: The experiment evaluates the translation accuracy between two randomly picked forms in the same domain. In particular, suppose a domain has $n$ forms as $[Q_1, \ldots, Q_n]$. We random shuffle those forms to get a permutation $[Q_{i_1}, \ldots, Q_{i_n}]$, and translate queries between $(Q_1, Q_{i_1}), \ldots, (Q_n, Q_{i_n})$. With 120 forms in total, we thus evaluate 120 translations.

We manually prepare the commonly used concepts and synonyms, as the domain-specific thesaurus, for each domain. In particular, for each of the Airfares, Books and Automobiles domains, those concepts and synonyms are prepared based on 20 forms randomly sampled from the forms used in the survey of Section 4.1. We store only concepts appearing in more than one source. We apply those knowledge to new forms to understand the feasibility of manually preparing matching knowledge and the impact of mismatching to translation accuracy.

To avoid biased queries against different sources, we
randomly generate queries. Given a source form \( Q_s \), we instantiate its query templates by randomly choosing a value for each variable. In particular, if the domain of the variable is finite with a set of pre-defined options, we randomly choose one from these options. Otherwise, we randomly generate a value with the corresponding type. The query generated in the experiment is thus “maximal,” i.e., it instantiates all predicate templates in the source form. As we will see in the following discussion, as one of our performance metric measures the percentage of queries that are perfectly translated, using a maximal query thus measures the worst-case performance.

**Performance Measurement:** In the experiment, we measure two aspects of the system performance: the translation accuracy and the complexity of customization.

**Translation Accuracy:** For the translation accuracy, we measure how many mappings in the suggested translation are correct, which indicates the amount of efforts the form assistant saves for users. In particular, we adopt a form-based metric: For a source query \( Q_s \) and a target form \( T \), let \( P(Q_s,T) \) denote the number of predicates in the target query, and \( R(Q_s,T) \) denote the number of predicate templates correctly filled by the form assistant. The form-based metric is defined as the percentage of correctly filled predicate templates and is thus \( R(Q_s,T) / P(Q_s,T) \).

To validate the impact of 1:1 mapping assumption, for each set of experiments, we measure the performance with and without this assumption. Measurement with 1:1 mapping counts, among all 1-1 mappings, how many predicates are correctly filled. For complex mapping (e.g., lastname+firstname→author), as the form assistant does not handle such mappings, those predicates - to be filled by complex mapping - are taken as a negative count in measurement with complex mapping.

To understand the impact of other components to predicate mapping, we further separately report the errors caused by form extraction and attribute matching from those caused by predicate mapping.

**Complexity of customization:** For each domain, we only customize the domain-specific thesaurus for attribute matcher. To evaluate how well the domain generic translation can achieve, we do not add any domain specific type handler. To measure the complexity of such customization, we report the number of commonly queried concepts and the corresponding synonyms for each concept.

**Experimental Result**

**Form-based Measurement:** Figure 10 (a) and (b) report the performance without counting the errors caused by form extraction and attribute matching. In particular, they show the frequency distribution over form-based metric for the two datasets respectively. As we can see, for 1:1 mappings, the system achieves very good results for both Basic and New datasets - 87% of the forms achieve perfect accuracy for the Basic dataset and 85% for New dataset. Further, as Figure 10(a) and (b) show, the system perform reasonably well even counting complex mappings– 76% of the forms achieves accuracy above 80% for the Basic dataset, and 70% for New dataset.

Figure 10(c) shows the performance impact of form extraction to the system (under 1:1 assumption). In particular, it shows average accuracy of translation with perfect input from form extraction and without. To generate perfect input, we manually corrected the errors caused by form extraction. As we can see, in average, we achieve 90.4% for Basic dataset with raw input from form extraction, and by correcting the errors of raw input, we achieve 96.1% accuracy. For the New dataset, we achieve 81.1% with raw input and 86.7% with perfect input.

Figure 10(d) further reports, among all the mappings, the number of errors caused by each component. As we can see, the impact of attribute matching is smallest– it causes only 6 errors out of 33 in total; form extractor causes 13 errors and predicate mapping causes 14 errors. The majority errors in predicate mapping are caused by the lack of domain specific mapping knowledge. For instance, to map between subject “computer science” to subject “programming language,” we need a domain ontology; to map between zipcode to city, we need a domain dictionary. Therefore, adding domain specific type handlers will be helpful to improve the mapping performance.

**Complexity of domain knowledge:** Figure 11 shows the number of concepts and synonyms we prepared for each domain. As we can see, the number of concepts and synonyms needed for translation is reasonably small. More importantly, those knowledge are collected on only 20 sources for each domain, and the accuracy of matching based on the prepared knowledge is very high. As Figure 10(d) shows, only 6 errors are caused by the mismatching of attributes,
which is very acceptable.

8 Related Work

Query translation has been actively studied in information integration systems in the literature. In particular, we observe that the existing techniques can be classified into four categories, to address data source heterogeneity at four levels. This paper, while is closely related to those works, clearly has its own focus: First, we focus on a specific sub-problem of dynamic predicate mapping, which is largely unexplored in the literature. Second, we focus on the design and development of a complete query translator, which satisfies our requirements for a light-weight domain-based form assistant. In this section, we connect and compare our work with those existing solutions.

Attribute Heterogeneity: Schema matching [7, 10, 16, 13, 11] focuses on mediating the heterogeneity at attribute level. Those works are concrete building blocks for query assistant. Different approaches of schema matching suit for different application settings. Some approaches, e.g., [10, 11, 20] require a collection of sources to mine the matchings, which are suitable for applications such as MetaQuerier. Others, e.g., [7, 13], perform matching across pairwise sources, which are suitable for applications such as a domain portal.

Data Heterogeneity: Schema mapping [12, 6] focuses on addressing the heterogeneity of data format across different sources. Its main objective is to convert a set of data records from a source schema to a target one and thus it does not need to cope with constraints on predicates (i.e., available operators and values). Unlike schema mapping, which focuses on equivalence conversion, predicate mapping must deal with more complicated semantics (e.g., any, exact in text type and less than in numeric type).

Predicate Heterogeneity: Predicate mapping focuses on addressing the heterogeneity of the predicates across different source capabilities. Existing solutions, e.g., [2] usually assume a static system setting, where sources are pre-configured with source-specific translation rules. In contrast, our approach dynamically maps predicates across unseen sources without prepared source knowledge. Further, the mapping in [2] depends on static rules to describe how to choose operators and fill in the values, while we propose a general search-driven approach to dynamically search for the best mapping.

Query Heterogeneity: Capability-based query rewriting focuses on mediating the heterogeneity of query syntax across different sources. Current query mediation works [9, 8, 19, 17, 14, 18, 15] studied the problem of how to mediate a “global” query (from the mediator) into “local” subqueries (for individual sources) based on capabilities described in source descriptions. Such rewriting is essentially transformation of Boolean expressions upon predicates, and does not consider the heterogeneity inside predicates. Therefore, predicate mapping and query rewriting are two complementary aspects of query translation. In particular, reference [17] focuses on $C_{\text{min}}$, query rewriting at query level, and is thus a concrete building block of our form assistant to achieve the overall $C_{\text{min}}$ translation.

9 Conclusion

In this work, we developed a light-weight domain-based form assistant, which is a core component in many integration applications. For building such a system, we proposed an architecture that satisfies the requirements of source-generality and domain-portability. In particular, we studied the dynamic predicate mapping problem, which, together with existing solutions for other components, complete the form assistant.

References


