A Mapping Repository for Meta-querier Evolution

Xiao Li and Randy Chow

CISE Technical Report 08-459
Department of CISE, University of Florida,
Gainesville, FL, 32611, USA
{xl1,chow}@cise.ufl.edu

Abstract. There exist an increasing number of online databases searchable only through query interfaces. Meta-querier with an integrated query interface is a practical solution to effective searching of information from multiple data sources. The multiplicity of data sources and their frequent changes necessitate the need for maintenance to support the evolution of meta-queriers. This paper presents a novel framework called MEQE to address this unique issue of meta-querier evolution. Central to the framework is the concept of a mapping repository, which is a domain-based information base. This information base stores all current and past mapping information, and their evolution orders, and functions as a mapping storage for query translation from a meta-querier to the underlying queriers. Moreover, it serves as a domain-based knowledge base for (semi-)automatic schema matching and merging.
The innovation in the design of the mapping repository is a conceptual graph model, which integrates both mapping and version management. In addition, the proposed framework can facilitate four advanced features: self-validation, self-configuration, self-maintenance, and virtualization. The paper describes details of the model and outlines an implementation architecture. Finally, it characterizes classes of mapping repositories and examines the potential use of the repository-based solutions for schema matching and merging.

1 Introduction

Internet has been a great source of information for our daily life. However, the explosion of information has made it difficult to efficiently retrieve useful information, especially those hidden under the *deep webs*. Deep webs refer to the on-line database searchable only through HTML query interfaces. Even in 2004, a survey estimated the scale of on-line database had already reached 450,000 sites [11]. Various researchers have attempted to develop techniques to expose these normally unreachable databases.

Studies have shown that building meta-queriers is the only practical solution to searching effectively the contents from multiple on-line databases [8, 14, 23]. A *meta-querier* refers to a system that provides a unified access (i.e., a *global*

query interface) to multiple existing online databases (i.e., *local queriers*). With a meta-querier, users only need to input a query in a global query form, which is respectively translated to multiple local queries with the same semantics (called *query translation*), and then the combined local query results are returned. In addition, meta-queriers can also be customized (called *customized meta-queriers*) [29], that is, the users are allowed to choose local queriers based on their interests and habits.

In the Web environment, online databases often change not only their data contents and formats, but also their semantics, query interfaces, and presentation styles[6]. And customized meta-queriers also can be changed because of the insertion or deletion of local queriers by users. Sometimes, a slight change in the bottom queriers could lead to a global effect and even jeopardize the whole system. In addition, changes of query forms could be frequent and unpredictable. Meta-queriers have to be incrementally evolved to adapt to these changes in order to meet the 24/7 operating requirement. Therefore, *meta-querier evolution* is more practical and considered a more critical issue than reconstruction for long-running meta-queriers.

In this paper, we propose a novel framework called MEQE to address this unique issue of meta-querier. Central to the framework is the concept of a *mapping repository*, which is the focus of this paper. The mapping repository is a domain-based information base for meta-queriers' daily operations. Its main functionalities include three points:

1) It supports mapping retrieval and management for query translation. First, the core of query translation is the mappings from query inputs in a global interface to inputs in local interfaces. Thus, mapping storage and management are fundamental issues for meta-queriers. In addition, our proposed mapping repository is shared by multiple meta-queriers to decrease storage space and prevent update anomalies, as shown in Figure 1. The intended customized meta-queriers could be numerous. Even for a specific meta-querier, the number of alternative local queriers could be large [11]. Both scalability issues indicate that the number of mappings in an integrated meta-querier system is potentially overwhelming. Thus, mapping storage and management is an important issue.

2) It provides version storage and management resulting from meta-querier evolution. Evolution implies the reuse of previous information to maintain the inoperable meta-queriers rather than rebuilding the meta-queriers from scratch. First, the unchanged self mappings can be directly reused. In addition, the development history of self and peer queriers carries important information for finding the changed or newly inserted mappings. Thus, version storage and management are added into our proposed mapping repository.

3) Since a predefined and preconfigured knowledge cannot adapt well to dynamic changing environments, the design of the proposed mapping repository addresses this shortfall by considering the following additional features: self-configuration, self-maintenance, and self-validation.

The presentation of the proposed mapping repository for meta-querier evolution is organized as follows: Section 2 provides a background of the related

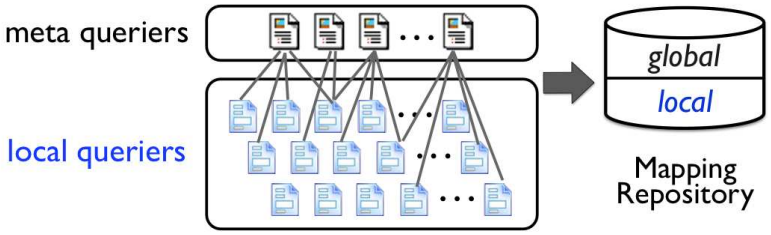

Fig. 1. A mapping repository shared by multiple meta-queriers.

work. Section 3 describes the modeling of query interfaces we developed. Section 4 gives an overview of the proposed framework MEQE. In Section 5 and 6, the prototypes and functionalities of the mapping repository are discussed in more details. Section 7 characterizes classes of mapping repositories and examines the potential applications of the mapping repository. Finally, we conclude and discuss future work in Section 8.

2 Related Work

2.1 Meta-querier

The most significant benefit of a meta-querier is its ability to utilize multiple local search engines to query multiple different but related objects at the same time. Many such commercial meta-queriers, such as Addall[1], Smartertravel[2] and Kayak[3], have implemented similar but simplified functions. Their techniques are proprietary and were often not reported.

In the research arena, multiple groups, such as *WISE*[24], *MetaQuerier*[12] and *DEQUE*[36], have attempted to analyze and attack this important problem from various perspectives. These systems can be viewed as an automatic integration of web databases to facilitate efficient web exploration[12, 24, 36]. Each complete system generally consists of the following components: source discovery and selection[22, 26, 38], interface extraction[23, 47], schema matching[19, 21, 41, 43, 44], schema merging[16, 34, 45], query translation[25, 46], results extraction[27, 48], result merging[28, 37], and system maintenance.

However, these frameworks only focus on the initial construction of meta-querier, but ignore the maintenance issue, which is considered more important than initial construction. In addition, they do not describe any mechanism for the storage of mappings, which can be used for the constructions and operations of meta-queriers, such as schema matching, schema merging, and query translation.

2.2 Mapping Repository

For storing mappings, most systems[31, 39, 49] simply employ a large table. Its columns represent different properties of mapping, such as element names, mapping expressions, and similarity values. Others use graph models in which nodes

denote schemas or fragments of schemas and edges represent mappings[9, 15]. Another approach[7, 32] is a combination of both. It models the mapping repository using graph representations with a finer granularity. That is, nodes in the graph denotes elements in the table-based implementation rather than schemas in the graph-based implementation.

Our mapping repository adopts the third approach. In the proposed graph model, nodes and edges have multiple type attributes to assist subsequent maintenance and validation. The design of this model considers both mapping storage and evolution, and thus mapping management and version management are integrated into this model.

3 Query Form Modeling

The proposed mapping repository requires a uniform representation of query forms. This section introduces an undirected graph model for representing query forms.

Current languages used to represent query forms include HTML, CSS, and some scripting languages. Structures of query forms can be classified into two groups: one-step forms and multi-step forms. Theoretically, the multi-step forms can be decomposed to multiple one-step forms based on their appearance dependence[36]. Therefore, this paper considers only the modeling of one-step forms.

For one-step form, the W3C's HTML specification[5] defines it as "a section of a document containing normal content, markup, special elements called controls (checkboxes, radio buttons, menus, etc.), and labels on those controls." The returned contents of user requests are normally based on the modification of controls (clicking radio buttons, entering text, etc.)

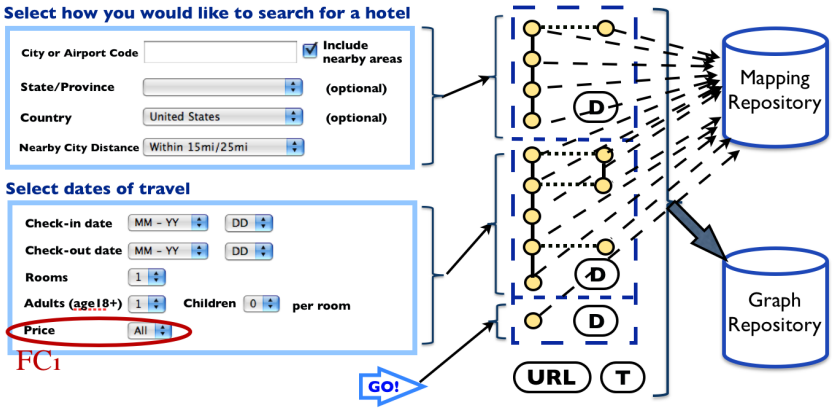

Fig. 2. A query interface and its graph models.

Typically, Interface Extractor (*IE*) can extract useful attributes such as control type, name/label, descriptive text, instances, data domain, default value, scale/unit (e.g., kg, million, dollar), and data/value types (e.g., date type, time format, char type, etc.) for each control[23, 47]. The controls and their associated attributes are referred to as *Functional Components* (**FC**) in our system. Two FCs are called **syntactically equivalent** iff all the attributes of two FCs are the same.

For instance, in the bottom left of Figure 2, FC_1 is a functional component extracted from a query interface. It has a control type "menu", a label name "price", a descriptive text "How Much?", an instance set (from "20-40" to "350-400" and "All"), data domain (from 20 to 400), default value ("All"), unit (dollar), data type (integer domain).

To capture the structural semantics among different FCs, we translate query forms from the native format into undirected graphs. A *vertex* in the graph corresponds to a FC in one form. If two FCs are vertically or horizontally adjacent to each other, an *edge* is used to connect them. Each edge is associated with a boolean property to represent its *adjacency type*, vertical or horizontal. Each *maximum connected subgraph* in the graph corresponds to a **semantic block** with a descriptive text D (if available). Each block is assigned a unique identifier called *BlockID*. In addition, each query form is identified by a *uniform resource locator* (*URL*) and a *version number* (denoted by the last valid accessed time T).

The left portion of Figure 2 shows a query interface for a hotel booking system. It requires users to input three categories of information. The middle portion of Figure 2 shows its corresponding graph model. This graph is composed of three corresponding semantic blocks. In each block, the vertical relations between FCs are denoted by solid lines, whereas the horizontal ones are denoted by dotted lines. In the proposed framework, Graph Repository stores this FC graph and Mapping Repository records a corresponding entity for each FC.

4 A framework of Meta-querier Evolution

This section gives a brief overview of the proposed maintenance framework called MEQE for **ME**ta-**Q**uerier **E**volution. MEQE is to maintain inoperable meta-queriers by mainly leveraging the previous learned information.

The maintenance of meta-queriers can be characterized in three categories: a) Deleting unneeded queriers; b) Merging newly inserted on-line databases into the current meta-querier; c) Repairing broken connections caused by the changes of existing query forms. For the first situation, it can be easily implemented by removing unneeded query forms' relations to the global query form. Thus, MEQE only focuses on the last two situations. The input to the system is a newly inserted query interface or an inoperable local query interface and the output is a fixed meta-querier.

Figure 3 is a high-level flowchart that illustrates the steps involved in maintaining a meta-querier. First, *Interface Extractor* (cf.3) extracts FCs from the

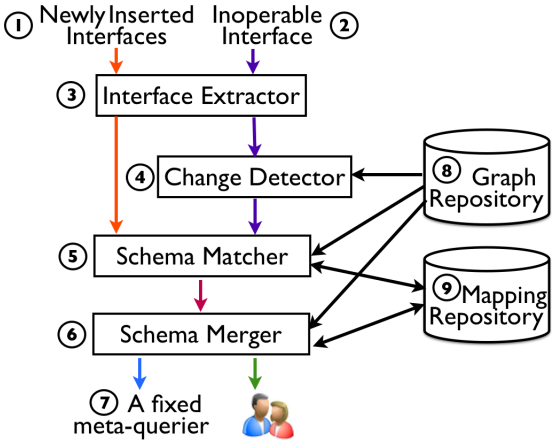

Fig. 3. A mapping repository.

newly inserted (cf.1) or inoperable (cf.2) query forms. For inoperable query forms, *Change Detector* (cf.4) [30] begins to syntactically compare the current FC graph with the last version, stored in *Graph Repository* (cf.8), and passes the difference to *Schema Matcher* (cf.5). Then, for each modified/new FC, Schema Matcher utilizes its structure information (stored in Graph Repository) and history information (stored in *Mapping Repository* cf.9) to obtain the best global-to-local mapping correspondence and expression. If one is found, Schema Matcher inserts a new mapping to Mapping Repository. Otherwise, *Schema Merger* (cf.6) attempts to merge it into the global interface and updates Mapping Repository. If the fixed meta-querier (cf.7) does not trigger any exception, the maintenance is successful. Otherwise it evokes human intervention.

5 Mapping Repository

In the core of MEQE, Mapping Repository stores all the mapping information, including the current and past mappings and their evolution order. Such information can be employed by Schema Matcher and Schema Merger to fix the broken or newly inserted local queriers. Mapping Repository is also used for mapping retrieval during query translation.

We start from the representation of mappings. Having a semantically rich representation of mappings is particularly important. In a meta-querier, a mapping can be defined as Mapping (*Expression,* $ListFC_L$*,* $ListFC_G$), where $ListFC_L$ and $ListFC_G$ are two ordered list of FCs' inputs in a local query form and a global form, respectively, and *Expression* denotes a high-level declarative expression that specifies the transformation from $ListFC_G$ to $ListFC_L$. Expressions can be list-oriented functions (e.g. equivalence, concatenation, mathematic expressions) or other more complex statements (e.g. if-else, while-loop). And mapping cardinality can be *1:1*, *1:n* and *n:m* ($n > 1$ and $m > 1$).

In addition, meta-queriers are not static. The evolution of meta-queriers complicates the mapping representation:

a) A local querier often develops multiple versions over time. Each version corresponds to a specific query form. Meta-queriers are required to update the relations between this altered local query form and the global form. Thus we need to seek and update the corresponding FC mappings. The previous self mapping information facilitates finding the proper mappings. And more recent versions carry more weight in understanding the current active version. Thus, development history needs to be stored with each mapping.

b) The collection of local queriers for a meta-querier can change, especially for a customized meta-querier. When a new querier is added, the corresponding new mappings need to be inserted into Mapping Repository. New mappings generated by matchers are individually attached with a similarity value (ranging between 0 and 1), indicating how well the match is.

Given the above discussion of mapping representation, we are ready to present a graph model for Mapping Repository and its unique managing capabilities in the following subsections.

5.1 Conceptual Graph Model

Mapping Repository is modeled using a directed graph, with three kinds of nodes (regular nodes, super nodes, and object nodes) and two kinds of edges (mapping edges and version edges), as illustrated in Figure 4.

As described in Section 3 on query form modeling, a query form is represented by an undirected FC graph, where each FC refers to a control and its associated attributes. Undirected FC graphs are stored in Graph Repository separated from Mapping Repository.

In Mapping Repository, a *regular node* (shown as a solid-line round) is intended to represent a FC. Regular nodes are the most basic building block in this model. *Super nodes* (shown as a solid-line rectangle) are the clusters of regular nodes whose FC inputs can be transformed to each other by an "equivalence" mapping. *Mapping edges* (shown as solid arrowed lines) connecting super nodes are employed to store the corresponding mapping information, such as mapping expressions, validation statuses and similarities values.

Furthermore, each querier has various different versions. Each version is a specific query form modeled in an undirected FC graph. We observe that most FCs in two subsequent versions of a querier do not change[6]. And thus, the concept of a **semantic object** is introduced to denote a set of FCs satisfying: 1) syntactically different (defined in Section 3); 2) with the same semantics; 3) from the same querier. Each FC is called a version of its semantic object. Given the above definition of a semantic object, it is easy to deduce: 1) Two syntactically equivalent FCs from different versions of a querier belong to the same version of a semantic object; 2) There exist multiple pairs of FCs which are the same version of a semantic object but respectively in two different versions of a query form.

In Mapping Repository, each *object node* (shown as a dotted line circle) refers to a semantic object, including all its versions (regular nodes). Different versions of the same semantic object are connected by *version edges* (shown as dotted arrowed lines) based on the order that they were developed.

Motivating Scenario To motivate the conceptual model, we use a scenario that is illustrated in Figure 4. Consider two customized meta-queriers, MQ_1 and

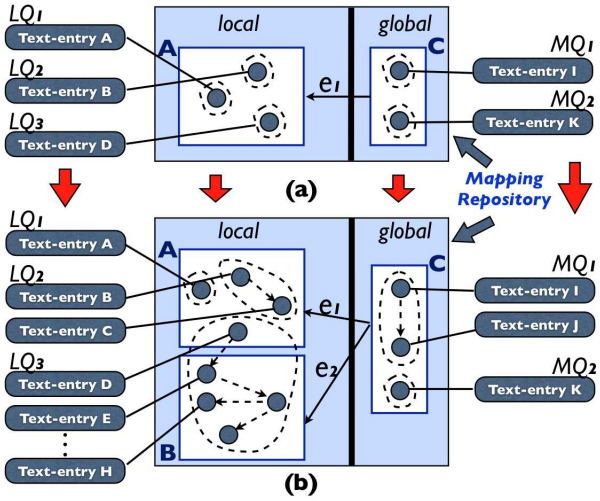

Fig. 4. A mapping repository.

MQ_2, which are respectively built on two groups of local queriers, $\{LQ_1, LQ_3\}$ and $\{LQ_1, LQ_2\}$.

Initially (illustrated in Figure 4(a)), each of these three local queriers has only one version (i.e., the currently active query form). Each version has only one FC, whose type is a text-entry box. MQ_1 and MQ_2 share the same mapping repository. In this mapping repository, three FCs in three local queriers and two FCs in two meta-queriers are represented by five regular nodes. They are divided into a global super node $\{C\}$ and a local super node $\{A\}$, respectively, based on their sources. A mapping edge e_1 records a *1:1* mapping expression from $\{C\}$ to $\{A\}$. Additionally, each querier has a object node containing its own regular node.

As time passes, LQ_2 and LQ_3 respectively develop one and four new versions, as shown in Figure 4(b). In this example, the change happened in LQ_2 does not break the current mapping, and thus, only a new regular node representing the FC ($Text\text{-}entryC$) is substituted for the FC ($Text\text{-}entryB$) to be the active regular node for LQ_2. And LQ_3's four new versions use different representation methods from the original one, such as, different control types. That is to say,

Text-entry D's inputs cannot be transformed by an "equivalence" mapping to *Text-entry E, F, G, H*'s inputs. And thus, a new super node B is created and connected by a mapping edge e_2, which stores a new mapping expression, from a global super node C.

This example assumes that all these FCs have the same semantics, and thus each querier maintains a single semantic object. In each object node, version edges are used to denote the development history.

To adapt to the local evolution, MQ_1 wants to update its own global query form by replacing its currently inaccurate FC (*Text-entry I*) with the FC (*Text-entry J*). But this change does not affect the mappings from local queriers, and thus a new regular node is added to super node C.

In the above discussion, several concepts and operations are introduced informally. To make these concepts effective and usable by developers, their semantics are specified precisely in the following subsections.

Regular nodes Regular nodes represent the most fundamental components FCs in the Mapping Repository.

Definition 1. A *regular node* represents a FC in a four-tuple $\{FCContent, URL, BlockID, LS\}$, where *FCContent* refers to this FC's associated attributes, *URL* and *BlockID* respectively identify the query form and the semantic block that this FC is located, and *LS* is the life span, i.e., the active period of this FC.

Regular nodes can be identified by a triple, {*URL, LS, BlockID*}. First, different queriers could contain syntactically equivalent FCs, but these FCs are represented by different regular nodes in Mapping Repository. In order to recognize which querier the regular node belongs to, the querier's unique identifier, URL, is necessary. Second, a querier often develop multiple versions over time. Normally, the changes of a querier only influence a small part of its FCs, thus syntactically equivalent FCs possibly appear in multiple versions of a querier. To avoid storing repetitive regular nodes, Mapping Repository employs time spans, instead of a time point, as the version identifier of a FC. Third, it is observed that syntactically equivalent FCs does not co-exist in the same semantic block[6]. And it is possible that two syntactically equivalent FCs with different purposes exist in two different semantic blocks of a query form. Thus, BlockID is also included. In sum, a specific FC can be fetched from Mapping Repository by its triple identifier, and we also can use this triple to obtain a set of its source forms (different versions of a query form) from Graph Repository.

Regular nodes can be further classified into local or global based on the source of query forms.

Super Nodes and Mapping Edges The relations among FCs can be distinguished to three categories: 1)*global-to-local*: the mappings from global FCs to local FCs; 2)*local-to-local*: the mappings among local FCs; 3)*global-to-global*: the mappings among global FCs.

All these three types of mappings are required for the proposed framework MEQE. In Mapping Repository, *super nodes* are intended for storing local-to-local or global-to-global mappings whose relations are one-to-one semantic-equivalent. *Mapping edges* represent the remaining ones.

Definition 2. A *super node* is an unordered collection of regular nodes satisfying the following two requirements:

1) All regular nodes in this collection must be either all local or all global.

2) For each pair of distinct regular nodes, their FC inputs can be transformed to each other by an "equivalence" mapping.

Depending on the constitution of super nodes, Mapping Repository distinguishes local super nodes from global super nodes. Additionally, since the inputs to regular nodes in the same super node are totally same, they always share the same mappings to other super nodes, i.e., the same correspondences and expressions.

Definition 3. A *mapping edge* is a directed edge from a source super-node list $NodeList_1$ to a destination super-node list $NodeList_2$, where $NodeList_1$ and $NodeList_2$ are two ordered lists of super-nodes. Each mapping edge has three attributes {ME, SimV, VS}. *ME* denotes a mapping expression specifying a directional function operating on the inputs of $NodeList_1$ to produce the inputs of $NodeList_2$. *SimV* and *VS* are two matrixes which respectively express the similarities degree (called *similarity matrix*) and the status (called *status matrix*) of each mapping denoted by this mapping edge.

The concept of ordered node list is proposed for supporting the storage of $n : m$ mappings. The node numbers of sources and destination lists depend on the mapping cardinality, that is, *1:1*, *1:n*, or *n:m*. ME is based on the specific sequence of source and target nodes, that is, different orders of super nodes in the source or destination list might produce different MEs.

Each mapping edge represents a set of mappings (called a *mapping set*) satisfying:

1) Its mapping expression is the same as ME;

2) Its input consists of a list of regular nodes, each of which is chosen from a unique source super node. The regular nodes in the output list are similarly chosen from the destination super nodes.

3) All its inputs must come from the same query form. It is similarly true for the outputs.

The third requirement is required; otherwise, the semantics of a global query input is separated into multiple parts of different local query forms.

The validation state and similarity value of a mapping in such mapping set are respectively stored in the corresponding element in VS and SimV matrixes. Its status can be in one of the three states: *human-validated, computer-validated* and *not-validated* (default). Its similarity value ranges from 0 (minimum) to 1.0. Its initial value is equal to the similarity degrees with all the other regular nodes in the super node it is clustered into. This value will fluctuate under the control of self-validation mechanism (discussed in Section 5.2.1).

Digging a little deeper into the definitions of super nodes and mapping edges, we can see an important consideration embedded within it: A super node bundles the regular nodes into a single logical entity. This allows for coarser granularity of presentation and explicit modeling of relationships among the super nodes. As long as a regular node is assigned to a super node, all the pre-configured mappings associated with this super node are also applied on this regular node. Thus, schema matching and merging can be transformed to reused-oriented strategies in the context of Mapping Repository.

Definition 4. A *mapping graph* is a directional graph $G =< V, E >$, where V is a set of all super nodes in Mapping Repository and E is a set of all mapping edges in Mapping Repository.

A mapping graph contains complete mapping information to support mapping management operations such as insertion and modification of mappings which might involve changes of expressions and similarity values.

For example, Figure 5 (a) illustrates a mapping graph and Figure 5 (b) depicts the composition of the super node $\{H\}$. This mapping graph covers all

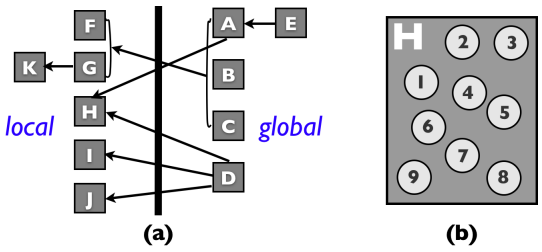

Fig. 5. A mapping graph.

three mapping types: a global-to-global one ($E \rightarrow A$), five global-to-local ones (e.g., $A \rightarrow H$), and various local-to-local one (e.g., $G \rightarrow K$ and $1 \rightarrow 2$). There exists a *3:2* complex mapping from the super-node list $\{A, B, C\}$ to the super-node list $\{F, G\}$. The global super node $\{D\}$ is mapped to three different super nodes, $\{H\}$, $\{I\}$, $\{J\}$.

Object Nodes and Version Edges Mapping Repository serves as not only the repository for mapping retrieval in query translation, but also a knowledge base for (semi-)automatic Schema Matcher and Schema Merger in the proposed MEQE framework. Besides self and peer mapping information[20, 44], their development history can also facilitates finding the proper current mapping. Development history denotes all the mapping changes and their sequences during a period. Object nodes and version edges are employed to store development history.

As declared in the previous section, a mapping is defined as a triple (*Expression, $ListFC_L$, $ListFC_G$*). In Mapping Repository, a mapping is expressed by a

mapping edge and two regular node lists. We assume that changes on *Expression* is for correcting the previous one, so its evolution history is ignored. The changes on a mapping can be represented by the evolutions of its two regular node lists.

Definition 5. A *version edge* is a directed edge from a regular node list $NodeList_1$ to another regular node list $NodeList_2$, where $NodeList_2$ is the next version of $NodeList_1$ for the same semantic object. A *version graph* is a directional graph $G =< V, E >$, where V is a set of all regular nodes in Mapping Repository and E is a set of all version edges in Mapping Repository. An *object node* is a maximum connected subgraph in a version graph.

A *version edge* reflects the order of two subsequent versions. The different versions of a semantic object indicate the same semantics in different formats. In a global or local object node, two regular nodes (versions) may differ in their representation formats, (i.e., FC's attributes) such as control type, descriptive texts and default values. For instances, there are two regular nodes, $N_1\{FCContent_1, URL_1, BlockID_1, LS_1\}$ and $N_2\{FCContent_2, URL_2, BlockID_2, LS_2\}$ If they satisfy: $FCContent_1 \neq FCContent_2$, $URL_1 = URL_2$, $LS_1 \neq LS_2$ and $BlockID_1 = BlockID_2$, and both N_1 and N_2's super nodes correspond to the same global nodes, we regard them as the different versions of one object.

Each semantic object corresponds to a maximum connected subgraph (denoted as an object node) in a version graph. Thus, an object node stores all the versions of a semantic object and its development orders. Since version orders are based on the development history of FCs, each object node should be a tree-type regular node graph (e.g., a version graph illustrated in Figure 6).

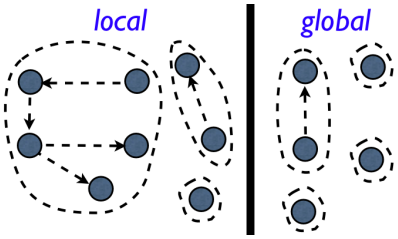

Fig. 6. A version graph.

Given the concepts described above, the critical issue is to automatically identify which regular nodes belong to the same semantic object and which nodes are newly inserted.

For global queriers, it is easy to capture the contents and orders of changes since MEQE tracks the whole evolution process. But for local queriers, all we have are two subsequent versions of a querier (i.e., two query forms) without any knowledge of the steps that led from one form to another. Thus we need to find the correspondent FCs between two versions automatically.

We observe that the semantics of a FC remain unchanged if it stays in the same semantic block in different versions [6]. Furthermore, since synonym FCs rarely co-exist in the same semantic block [21, 40], each global FC corresponds to no more than one local FC in a semantic block.

Based on these two observations, our algorithm can be built upon the following three assumptions:

1) The relations between semantic objects can be expressed by the orders and locations of FCs in the FC graph.

2) If a FC changes its position but remains in the same semantic block, the semantic of this FC will not be changed.

3) In the same semantic block of each form, two FCs do not correspond to the same global FCs.

Therefore, a local object consists of a set of local regular nodes satisfying three requirements:

1) Corresponding to the same global super node list;

2) In the same query form;

3) In the same semantic block.

Based on the above criteria, local objects can be built and maintained automatically by Algorithm 1 sketched below. The algorithm is for inserting a regular node LR to a proper object node ON in Mapping Repository R. Notice that the algorithm is only for $1 : n$ mappings, but it is easy to extent it to $n : m$.

Input: Local regular node LR, Mapping Repository R
Output: Object node ON
BF = NodesInSameBlockForm(LR, R);
GN = GlobalDestination(LR);
foreach *Local regular node* i *in* BF **do**
  N_i = GlobalDestination(i);
  if $N_i \cap GN \neq \emptyset$ **then**
    ON = Object node of i;
    ON.Insert(LR);
    return ON;
  end
end
ON = new ObjectNode();
ON.Insert(LR);
return ON;

Algorithm 1: Insert a regular node to a proper object node in Mapping Repository.

Algorithm 1 is composed of three steps:

1) Searching the whole Mapping Repository (R) to obtain all the regular nodes (BF) in the same block and query form with LR.

2) Finding out all the possible global super nodes lists (GN) which can reach LR through different directed mapping edges.

3) Comparing GN with all possible global super nodes lists (N_i) of each regular node i in BF. If there exists the same global super node list, then the object node of regular node i is the target ON. If there is no such regular node i, a new local object node will be built to contain LR.

Mapping Repository integrates the version graph with the mapping graph. Its basic building blocks are regular nodes, mapping edges and version edges. Object nodes and super nodes are employed for coarser granularity based on different purposes, version and mapping management.

5.2 Advanced Features

In order to accommodate the dynamic of web environment and the complexity of query forms, Mapping Repository needs to provide advanced features such as self-validation, self-configuration, self-maintenance, and virtualization.

Self-validation In meta-queriers, the mappings from global forms to local forms are precisely engineered, and thus have one special property: high similarity with a value close to 1.0. If human experts insert or modify a regular node, the repository automatically assumes that experts have validated the mappings related to this regular node, i.e., their validation statuses will be set to "human-validated" with the highest similarity values (1.0).

For the mappings derived by Schema Matcher (computers), their initial similarity values depend on the algorithm implemented in the matcher. But these values are not statistic. As time passes, the meta-queriers continuously examine the usage of mappings, and adjust the similarity values based on the validation of mappings. We propose the following similarity function to obtain the similarity value of a mapping at the time point t2:

$$SimVt2 = \begin{cases} 1 & \text{if } E(n,m) + SimVt1 > 1; \\ 0 & \text{if } E(n,m) + SimVt1 < 0; \\ \text{E(n,m) + SimVt1} & otherwise. \end{cases}$$

where n and m are the numbers of correct execution and improper execution between the time points $t1$ and $t2$, respectively, $SimVt1$ is the similarity value at the time point $t1$, and $E(n,m)$ is equal to the degrees of changes between $t1$ and $t2$. In the initial implementation, we adapt the following definition: $E(n,m) = a \times n + b \times m (b > a)$, where a and b are constants. The penalties of improper executions are larger than the benefits from possibly proper executions, and thus the value of b is greater than a.

To address the features of self-configuration and self- maintenance, we need to introduce the notion of mapping transitivity as illustrated in Figure 7.

Definition 6. *Transitive*: for any three super node lists A, B and C, if there exist two mapping edges m_1 (from B to A) and m_2 (from C to B), another new mapping edge m_3 (from C to A) can be derived by combining m_1 and m_2 in order, denoted by $m_1 \bullet m_2$.

Mapping composition is a challenging problem. Research results [10, 18, 33] state that not all the mappings can be composed, that is, the intermediate states cannot be circumvented. For instance, in Figure 7 (a), C has to be (partly) converted to the B's format and then to the A's.

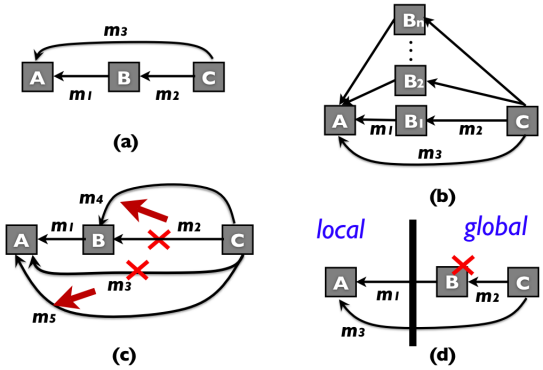

Fig. 7. Mapping transitivity.

In the current Mapping Repository, we do not compose the mappings but combine them in order to predict the new one, i.e., going through every intermediate state in the mapping path. Its similarity value can be computed by multiplying the similarity values of all the mappings involved in the mapping path.

Self-configuration *Self-configuration* is an automatic mechanism to generate the unknown global-to-local mappings for each local super node. By exploiting mapping transitivity, it calculates the similarity values for all the mapping paths. We can obtain a new mapping by combining the mappings through the mapping path with the highest similarity value.

For instance in the Figure 7 (b), there exist n mapping paths from C to A. The path $C \rightarrow B_1 \rightarrow A$ consists of the mapping m_2 from C to B_1 and m_1 from B_1 to A. Its similarity value is equal to $Sim[C, B_1] \times Sim[B_1, A]$. Suppose this path has the highest similarity value among all the paths, the new mapping m_3 can be derived by the combination of m_1 and m_2, i.e., $m_1 \bullet m_2$.

Self-maintenance *Self-maintenance* is to facilitate efficient restructuring of Mapping Repository upon changes, such as deletion, modification and insertion

of edges and nodes. We demonstrate three cases common in the evolution of meta-queriers:

1) Removing a mapping edge: Improper mapping edges can be removed automatically or manually. As illustrated in the Figure 7 (b), if the mapping edge m_3 is removed, Mapping Repository will attempt to find a new mapping edge from the mapping paths from C to A, just like self-configuration.

2) Updating derived mappings: Changes of any mapping edge will trigger the update of its derived mappings. If this edge is deleted and cannot be replaced by the combination of other edges, the derived mapping edges are also removed; otherwise, they should be updated to the new ones. For instance in the Figure 7 (c), the mapping m_3 derived by m_1 and m_2 is updated to m_5 due to the fact that m_2 is changed to m_4, where $m_5 = m_1 \bullet m_4$.

3) Replacing a global super node: In meta-queriers, human experts sometimes need to re-organize the global query form. Some global super nodes will be replaced. It is a laborious, time-consuming and error-prone task to update numerous mappings between the replaced global super nodes and local super nodes. As long as experts specify the mapping from the new global super nodes to the replaced, Mapping Repository can update all the related global-to-local mappings automatically. For instance in the Figure 7 (d), given the mapping m_2 from the new super node C to the replaced node B, a new mapping m_3 from C to A can be derived by combining m_2 and m_1 in order.

Virtualization Mapping Repository can be shared by many customized meta-queriers, although they are built on different collections of local query engines. A *view* is a virtual or logical mapping repository. Each view is a sub-graph of the whole mapping and version graph. Each meta-querier or local querier owns its own view, which includes all the related nodes and edges.

Views hide the complexity of graphs and make it easier to maintain the mapping repository for human editors. Views can be categorized to updatable or read-only views to increase the security of Mapping Repository.

6 System structure

Mapping Repository is implemented above the open-source system AlignmentAPI[4]. Our proposed query form model is transformed to ontology format in OWL. Each query form can be regarded as an instance of this form ontology.

Our system is organized in two layers, as shown in Figure 8. Layer 1 (*storage layer*) is responsible for physical storage of mapping and version graphs. Layer 2 (*logic layer*) implements all the logics of mapping repository discussed in the previous sections.

Storage layer stores mapping and version graphs in tables. It also provides the $storageAPI$ to translate the graph operations to the operations on the relational schema, such as, seeking all the reachable super nodes from a super node N. Roughly, logic layer can be divided into three levels. Each level offers a functional interface the operations of which may be used by higher levels. Level 1 (*node*

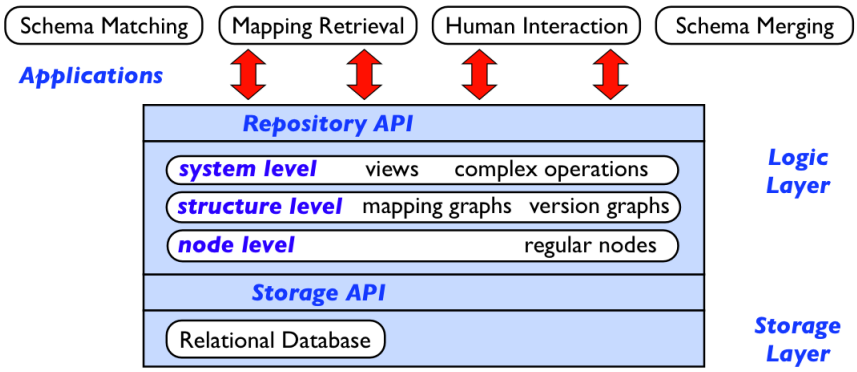

Fig. 8. The system structure.

level) implements the concept of regular nodes. Level 2 (*structure level*) and 3 (*system level*) implement the mechanisms for mapping management and version management.

Node level offers the basic operations on regular nodes. Regular nodes can be viewed, inserted, edited or deleted. At this level, no relation between regular nodes is considered. All the regular nodes are operated separately.

Structure level manages mapping graphs and version graphs. Super and object nodes, mapping and version edges are defined and implemented in this level. There are three kinds of operations: 1) Creation and deletion of super nodes and object nodes; 2) Manipulation of mapping graphs and version graphs (Creation and deletion of edges, modification of mapping edge's ME, mapping edge shift between two super nodes, and the version edge shift between two regular nodes); 3) Search and comparison of graph elements including regular nodes, mapping edges, version edges, super nodes, and object nodes.

In system level, the support of four advanced features (described in Section 5.2) is implemented, e.g., providing a mechanism to define customized views to adapt customized meta-queriers. This level also provides some common complex operations, a sequence of basic operations, such as, combining the mapping expressions through a mapping path.

Mapping Repository can be employed by different applications such as mapping retrieval, schema matching and schema merging. In the next section, two repository-based applications, schema matching and merging, are discussed in details.

7 Discussion

7.1 Design Issues

Mapping repositories function as knowledge bases for storing mapping information. There are many dimensions by which mapping repositories can be classified. Mapping repositories have the following four properties:

Single-domain v.s. *cross-domain*: whether mapping repositories can be from the queriers in the same domain or multiple domains. The representation methods in the same domain are limited. For example, the vocabulary used in query forms tends to converge at a relatively small size[11]. Thus, the previous mappings in the same domain are very useful for (semi-)automatic construction and maintenance of meta-queriers.

The current version of Mapping Repository mainly focuses on the storage of mappings in a single domain. In the future, Mapping Repository needs to consider more issues for supporting the meta-queriers that cross multiple domains, such as, dealing with the contradicted and common mappings from different domains.

Dynamic v.s. *static*: whether mapping evolution can be supported or not. Mapping is inherently dynamic and abstract perception of relation between local and global queriers. The proposed mapping repository can modify or add the corresponding contents based on the changes in the local and global queriers.

Single-version v.s. *multi-version*: whether the previous versions are stored or not. Mapping information is active and valid at different time intervals. Although the previous versions become invalid, they are very precious resources for understanding the current valid version of local queriers. The proposed Mapping Repository stores all the versions of mappings.

Single-meta-querier v.s. *multi-meta-querier*: whether mapping repositories only serve a specific meta-querier or multiple ones. A mapping repository ought to be designed for the sharing by multiple meta-queriers if they have many common mapping information. Compared with the storage space used by multiple individual mapping repositories, the space for the shared repositories is much smaller. And the shared repository also prevents update anomalies. Multiple customized meta-queriers can be simultaneously maintained by modifying the common part in the mapping repository. Lastly, the more mapping information a mapping repository owns, the better performance (semi-)automatic construction and maintenance of meta-queriers have. Thus, this property is also supported by the proposed Mapping Repository.

7.2 Applications

The designers of meta-queriers can implement their own operations on Mapping Repository based on the specific application (domain). For example, 1) a *repository-based mapping retrieval* to utilize the storage of mappings to fetch the required mappings for query translation; 2) supporting human interaction for *repository-based mapping validation* and *modification*. In the following, two application examples that use Mapping Repository are presented in details.

Repository-based schema matching is a reuse-oriented matching strategy. It aims to find a global FC correspondence (FC_G) and a mapping expression for each pending local FC (FC_L). It leverages the evolution history of self and peer mappings to learn new mappings. This process is composed of three steps:

1) Scanning Mapping Repository to find the most similar (higher than a threshold) local super node (SN_L) for each pending FC_L. Each super node is indeed a cluster of highly similar regular nodes, and thus this search problem is equivalent to the *incremental clustering* algorithms which is widely studied[13, 42]. And the similarity degrees between regular nodes are determined by various associated information, such as their FC attributes, development history, and their locations and adjacencies in FC graphs. These values can be calculated by most existing schema or ontology matching strategies[17, 35], since the relations between FC_L and the FCs stored in SN_L are totally equivalent (i.e., "=") rather than other mapping expressions.

2) Finding the most suitable mapping path (MP) which is from global super nodes to the SN_L (found in step 1). The possible paths from global super nodes can be obtained by enumerating all the paths to SN_L. Then the similarity value of each path is calculated by exploiting mapping transitivity, just like the example in Section 5.2.2. The path with the highest value is finally selected as MP.

3) Choosing a FC_G from the super node SN_G which is the source of MP (found in step 2). In most cases, the selection should best-effort maintain unchanged with the subsequent version of the global interface. The other principles of selection depend on the specific application. In addition, the corresponding expression can be achieved by combining all the mappings involved in the MP. As shown in Figure 7 (b), consider A and C are respectively a local super node and a global super one. If A is the SN_L and the path $C \rightarrow B_1 \rightarrow A$ is MP, C is the SN_G with a mapping expression m_3.

Repository-based schema matching focuses on reusing previous matching results of self and peers. For providing better match candidates, this approach should be combined with other schema-based or instance-based schema matching techniques[17, 35].

Repository-based schema merging is a reused-oriented merging strategy. It aims to integrate multiple local query forms into a unified global query form by using the previous validated self and peer mappings. Local and global query forms are made up of FCs, which are individually represented by a regular node. The primary objective of this approach is to find a set of global regular nodes (RN_G) that can be mapped to all (or the most significant) local regular nodes (RN_L) of local query forms.

Since each RN_L corresponds to a local super node (SN_L), it is not hard to obtain a set of global super nodes ($SNSet_G$) which can reach each SN_L through directed mapping edges. The principles of choosing a set of RN_G from all the $SNSet_G$ are listed in the order of priority:

1) RN_G should not conflict with each other, such as naming and structural conflicts.

2) RN_G should maximally satisfy the constraints of all the RN_L.

3) The number of chosen RN_G should be minimized.

In essence, it is a NP-hard optimization problem. Many approximation or heuristic algorithms have been proposed in the previous research results[16, 34, 45].

8 Conclusion and future work

In this paper, we propose a framework named MEQE to tackle a unique and important topic, meta-querier evolution. In the center of MEQE, Mapping Repository functions as a mapping storage for query translation and serves as a domain-based knowledge base for Schema Matcher and Merger. Mapping Repository employs an innovative conceptual graph model which integrates both mapping and version management. In addition, it supports four advanced features: self-validation, self-configuration, self-maintenance, and virtualization. Although Mapping Repository is primarily designed for the meta-querier evolution, it should be useful to other applications in the field of information integration.

Currently, we have implemented the basic Mapping Repository. Future work in this direction includes:

1) Modification and improvement of Mapping Repository to support cross-domain meta-queriers.

2) Implementation of other components of MEQE, such as Schema Matcher and Merger.

3) Collection and analysis of the evolutions of real query forms.

References

1. http://www.addall.com/.
2. http://www.smartertravel.com/compare-prices.
3. KAYAK. http://www.kayak.com/.
4. Alignment API and Alignment Server. http://alignapi.gforge.inria.fr/
5. W3C's HTML specification. http://www.w3.org/TR/html4/interact/forms.html.
6. The UIUC web integration repository (TEL-8 query interfaces). Computer Science Department, University of Illinois at Urbana-Champaign. http://metaquerier.cs.uiuc.edu/repository, 2003.
7. S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. SIGMOD Rec., 28(1):5459, 1999.
8. M. K. Bergman. The Deep Web: Surfacing hidden value. Journal of Electronic Publishing, 7(1), August 2001.
9. P. A. Bernstein. Applying model management to classical meta data problems. In CIDR, 2003.
10. P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash. Implementing mapping composition. The VLDB Journal, 17(2):333353, 2008.
11. K. C.-C. Chang, B. He, C. Li, M. Patel, and Z. Zhang. Structured databases on the web: Observations and implications. SIGMOD Rec., 33(3):6170, 2004.
12. K. C.-C. Chang, B. He, and Z. Zhang. Toward large scale integration: Building a metaquerier over databases on the web. In CIDR, 2005.
13. M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. SIAM J. Comput., 33(6):14171440, 2004.

14. S.-L. Chuang, K. C.-C. Chang, and C. Zhai. Context-aware wrapping: Synchronized data extraction. In VLDB, 2007.
15. H.-H. Do. Schema Matching and Mapping-based Data Integration. Ph. D. dissertation, Department of Computer Science, University of Leipzig, Germany, 2006.
16. E. Dragut, W. Wu, P. Sistla, C. Yu, and W. Meng. Merging source query interfaces on web databases. In ICDE, 2006.
17. J. Euzenat and P. Shvaiko. Ontology Matching. Springer-Verlag New York, Inc., 2007.
18. R. Fagin, P. G. Kolaitis, L. Popa, and W.-C. Tan. Composing schema mappings: Second-order dependencies to the rescue. ACM Trans. Database Syst., 30(4):9941055, 2005.
19. B. He and K. C.-C. Chang. Statistical schema matching across web query interfaces. In SIGMOD, 2003.
20. B. He and K. C.-C. Chang. A holistic paradigm for large scale schema matching. SIGMOD Rec., 33(4):2025, Dec. 2004.
21. B. He, K. C.-C. Chang, and J. Han. Discovering complex matchings across web query interfaces: A correlation mining approach. In KDD, 2004.
22. B. He, T. Tao, and K. C.-C. Chang. Organizing structured web sources by query schemas: a clustering approach. In CIKM, 2004.
23. H. He, W. Meng, Y. Lu, C. Yu, and Z. Wu. Towards deeper understanding of the search interfaces of the deep web. World Wide Web, 10(2):133155, Jun. 2007.
24. H. He, W. Meng, C. Yu, and Z. Wu. Wise-integrator: An automatic integrator of web search interfaces for e-commerce. In VLDB, 2003.
25. G. Kabra, C. Li, and K. C.-C. Chang. Query routing: Finding ways in the maze of the deep web. In ICDE-WIRI, 2005.
26. Y. Lu, H. He, Q. Peng, W. Meng, and C. Yu. Clustering e-commerce search engines based on their search interface pages using wise-cluster. Data Knowl. Eng., 59(2):231246, Nov. 2006.
27. Y. Lu, H. He, H. Zhao, W. Meng, and C. Yu. Annotating structured data of the deep web. In ICDE, 2007.
28. Y. Lu, W. Meng, C. Y. L. Shu, and K. L. Liu. Evaluation of result merging strategies for metasearch engines. In WISE, 2005.
29. Y. Lu, Z. Wu, H. Zhao, W. Meng, K. Liu, V. Raghavan, and C. T. Yu. Mysearchview: a customized metasearch engine generator. In SIGMOD, 2007.
30. R. McCann, B. AlShebli, Q. Le, H. Nguyen, L. Vu, and A. Doan. Mapping maintenance for data integration systems. In VLDB, 2005.
31. S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: a programming platform for generic model management. In SIGMOD, 2003.
32. P. Mitra, G. Wiederhold, and M. L. Kersten. A graph-oriented model for articulation of ontology interdependencies. In EDBT, 2000.
33. A. Nash, P. A. Bernstein, and S. Melnik. Composition of mappings given by embedded dependencies. ACM Trans. Database Syst., 32(1):4, 2007.
34. R. Pottinger and P. A. Bernstein. Schema merging and mapping creation for relational sources. In EDBT, 2008.
35. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. The VLDB Journal, 10(4):334350, Dec. 2001.
36. D. Shestakov, S. S. Bhowmick, and E.-P. Lim. Deque: querying the deep web. Data Knowl. Eng., 52(3):273311, 2005.
37. W. Su, J. Wang, Q. Huang, and F. Lochovsky. Query result ranking over e-commerce web databases. In CIKM, 2006.

38. W. Su, J. Wang, and F. Lochovsky. Automatic hierarchical classification of structured deep web databases. In WISE, 2006.
39. A. Thor and E. Rahm. Moma - a mapping-based object matching system. In CIDR, 2007.
40. F. L. W. Su, J. Wang. Holistic schema matching for web query interface. In EDBT, 2006.
41. J. Wang, F. J. Wen, Lochovsky, and W.-Y. Ma. Instance-based schema matching for web databases by domain-specific query probing. In VLDB, 2004.
42. D. H. Widyantoro, T. R. Ioerger, and J. Yen. An incremental approach to building a cluster hierarchy. In ICDM, 2002.
43. W. Wu, A. Doan, and C. Yu. Webiq: Learning from the web to match deep-web query interfaces. In ICDE, 2006.
44. W. Wu, C. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In SIGMOD, 2004.
45. W. Wu, A. Doan, and C. Yu. Merging interface schemas on the deep web via clustering aggregation. In ICDM, 2005.
46. Z. Zhang, B. He, and K. C.-C. Chang. On-the-fly constraint mapping across web query interfaces. In VLDB-IIWeb, 2004.
47. Z. Zhang, B. He, and K. C.-C. Chang. Understanding web query interfaces: Best-effort parsing with hidden syntax. In SIGMOD, 2004.
48. H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. Yu. Fully automatic wrapper generation for search engines. In WWW, 2005.
49. A. Zhdanova and P. Shvaiko. Community-driven ontology matching. In ESWC, 2006.