WISE-Integrator: An Automatic Integrator of Web Search Interfaces for E-Commerce

Hai He, Weiyi Meng
Dept. of Computer Science
SUNY at Binghamton
Binghamton, NY 13902
{haihe,meng}@cs.binghamton.edu

Clement Yu
Dept. of Computer Science
Univ. of Illinois at Chicago
Chicago, IL 60607
yu@cs.uic.edu

Zonghuan Wu
Center for Adv. Compu. Studies
Univ. of Louisiana at Lafayette
Lafayette, LA 70504
zwu@cacs.louisiana.edu

Abstract

More and more databases are becoming Web accessible through form-based search interfaces, and many of these sources are E-commerce sites. Providing a unified access to multiple E-commerce search engines selling similar products is of great importance in allowing users to search and compare products from multiple sites with ease. One key task for providing such a capability is to integrate the Web interfaces of these E-commerce search engines so that user queries can be submitted against the integrated interface. Currently, integrating such search interfaces is carried out either manually or semi-automatically, which is inefficient and difficult to maintain. In this paper, we present WISE-Integrator - a tool that performs automatic integration of **W**eb **I**nterfaces of **S**earch **E**ngines. WISE-Integrator employs sophisticated techniques to identify matching attributes from different search interfaces for integration. It also resolves domain differences of matching attributes. Our experimental results based on 20 and 50 interfaces in two different domains indicate that WISE-Integrator can achieve high attribute matching accuracy and can produce high-quality integrated search interfaces without human interactions.

1. Introduction

More and more databases are becoming Web accessible through form-based search interfaces. Among these web sources, E-commerce search engines (ESEs) account for a large proportion. It is of great importance to provide a unified access to multiple ESEs selling similar products because this would allow users to search and compare products from multiple sites with ease. In this paper, we call a system that supports unified access to multiple ESEs as an E-commerce metasearch engine (EMSE for short). Currently, there are a number of EMSEs on the Internet, such as www.addall.com, www.mysimon.com, www.cnet.com, and www.dealtime.com. However, their techniques are not publicly available. To the best of our knowledge, most existing EMSEs are built manually or semi-automatically. Furthermore, as ESEs operate autonomously, changes/upgrades to them may affect the operation of the EMSE. As a result, maintaining the operation of an EMSE is a costly long-term effort.

Our E-Metabase project aims to automate the process of building large-scale EMSEs so as to significantly reduce the cost of building and maintaining EMSEs. This project consists of a number of components. First, a special crawler is used to crawl the Web and identify ESEs from the fetched Web pages. Second, the found ESEs are clustered into different groups such that ESEs in the same group sell the same type of products (i.e., in the same domain). Third, the interfaces of the ESEs in the same group are integrated into a unified interface that becomes the interface of the EMSE for this group. Fourth, a global query submitted to the EMSE is mapped to queries for the underlying ESEs. Fifth, a component that is responsible for connecting to each ESE is built so that a query can be passed to and results can be returned back from each ESE. Sixth, information of every product returned by each ESE needs to be correctly extracted from the returned result pages by an information extraction program. Finally, the extracted results from different ESEs need to be filtered according to the global query and then combined into a single list for presentation to the user based on some desired features, say price. WISE-Integrator is designed to automate the interface integration step. In this paper, we present our techniques used to build WISE-Integrator. WISE-Integrator is applied to each group of ESEs to produce an integrated

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

interface for this group of ESEs. Without loss of generality, in this paper, we assume that all ESEs under consideration are in the same product domain. (Our techniques for clustering ESEs can be found in [PMH03])

This paper has the following contributions. First, we provide a comprehensive solution to the interface integration problem. Interface integration includes schema integration, attribute value merging, format integration and layout generation of global attributes in the global (integrated) interface. In contrast, related existing works deal with only schema integration (see Section 2). Second, we propose an automated solution for interface integration using only general (i.e., domain-independent) knowledge. Most existing works employ manual or semi-automatic techniques. One of the key issues in interface integration is to identify matching attributes from different interfaces and we propose a clustering and weight-based two-step method to tackle this problem. Furthermore, this method also solves a rarely addressed issue, i.e., finding appropriate names for attributes in the global interface automatically. Our experimental results based on 20 and 50 interfaces in two different domains indicate that WISE-Integrator can achieve high attribute matching accuracy and can produce high-quality integrated search interfaces without human interactions.

The rest of this paper is organized as follows. Section 2 briefly reviews previous research works related to our work. Section 3 discusses interface representation used by WISE-Integrator. In Section 4, we present our method for matching attributes. In Section 5, we discuss merging attribute domains. In Section 6, we discuss global interface construction. Experimental results are reported in Section 7. Section 8 concludes the paper.

2. Related work

A thorough survey of approaches for automatic schema matching can be found in [RB01]. [GKD97] predefines the mapping rules for each attribute and assembles these rules into a knowledge base for interpretation when a query is handled. [LRO96] uses a *world view* to represent all sources but it does not discuss how to construct the world view automatically. [DEW96] predefines each domain description that includes information about product attributes, and then uses some heuristics and mapping functions for the fields of each search interface but it does not provide much detail about user interface. [BBB01, BCV01] use Description Logics, *Common Thesaurus* and clustering techniques for semantic schema integration. WordNet [WDNT] is also used to identify semantic relationships between schema terms. This is a semi-automatic approach as the integration process still involves human interaction. Furthermore, the approach used for matching attributes is mainly based on name affinity and structure affinity, and only a few metadata (such as key, foreign key) of schemas are used. [LC00] uses neural network techniques and focuses on utilizing both schema level and data contents level metadata to automatically identify matching attributes. Our approach has adopted some ideas from [LC00] but there are significant differences (see Section 4.3 for more comparison with [LC00]). [MBR01] investigates algorithms for generic schema matching. It combines a number of past techniques, such as linguistic-based matching and some metadata of schemas. It proposes structure-matching algorithms for hierarchy schemas (tree structures) in which a structural similarity is computed between each pair of schema elements. However, how a global schema is obtained is not discussed. [DDH01] uses and extends machine-learning techniques to semi-automatically find mappings between source schemas and the mediated schema. This approach needs human users to manually construct the semantic mappings between a small set of data sources (training) and the mediated schema. [HC03] uses a statistical approach for schema integration of query interfaces of the *deep* web. It argues that as the Web sources proliferate the aggregate schema vocabulary of sources in the same domain tends to stabilize at a relatively small size, and that underlying these sources, there exists a unified *hidden schema model*. Then it uses statistical probability and goes through three steps (hypothesis modeling, generation and selection) to obtain the hidden schema model. It uses only attribute names for statistics, and it does not apply other schema information such as domain type, default value and attribute values which, we find, based on our experiments, to be very effective in interface integration. It is not clear how semantic relationships between names (such as synonymy and hypernymy) are obtained in this work. In addition, it discusses only schema integration, but not attribute merging and global interface generation. [MGR02] uses the idea of IP packet flooding to flood the similarity of elements. It converts each schema into a directed labeled graph. On the basis of the graph model, a part of the similarity of two elements propagates to their respective neighbors. The similarity flooding algorithm terminates after a fix point is reached and some filters are used to get a subset of the result mapping. No linguistic name matching is done beyond utilizing a simple string matcher to compare common prefixes and suffixes of literals. This approach is not suitable for search interfaces because name matching plays an important role in the integration of search interfaces. [DR02] discusses combining different matching algorithms in a flexible way and supports different ways to combine match results. In [DR02], schemas are represented as rooted directed acyclic graphs. It maintains a matcher library for simple matchers such as approximate string matcher, synonym matcher, data type matcher and hybrid matchers (e.g. name matcher and structural matchers). It uses data type but not other schema and domain information to help find matches.

The main difference between our work and existing works is that we aim to perform comprehensive interface

integration automatically while others perform only schema integration, employing mostly manual or semi-automatic techniques. There are basically no published work, to the best of our knowledge, on automatic attribute value merging, format integration and layout generation. Furthermore, compared with other approaches for Web sources integration, we utilize a richer set of schema and domain information to find matching attributes and we utilize the information differently (see Section 4).

3. Interface representation

A search interface for E-commerce is usually presented through an HTML **form** in a Web page [HTL4]. It may contain elements such as *text box*, *radio button*, *check box* and *selection list*, and each element usually has a *label* (descriptive text) associated with it. Users fill out the form and then submit the filled form as a query through the browser to the remote server. The server then returns to users the results that satisfy the query conditions. In general, much useful information is embedded in the HTML source file of each local interface and such information needs to be extracted for interface integration. In this paper, we do not discuss how to extract the needed information (some related work can be found in [RGM01, DEW96], and our work on this will be reported in another paper). Instead, we focus on what information should be used to represent each search interface for the purpose of *interface integration*.

Each local ESE interface can be conceptually viewed as a partial export relational schema of the underlying product database. In our interface representation, each **label** is considered as the **name** of an **attribute** of the underlying products. Each attribute has one or more associated **elements**. Each element has a **format** which is the input format of the element. There are generally four types of formats: *text box*, *radio button*, *check box* and *selection list*. Each element also has a **domain** that defines the set of **values** that can be used to instantiate the element when forming a query. *Text box* allows users to input whatever value they want and thus the corresponding domain can be considered to be *infinite*. A selection list provides a finite number of pre-determined values for users to select while a check box and radio button have one associated value. These three formats thus have a *finite* domain. Often multiple check boxes or multiple radio buttons are used together to accomplish the same function as a selection list. In addition, each element or a group of elements may have its or their **default value**, which is used to help forming a query when a user does not make a different selection. For each attribute, there is a **type** for its values. Six value types are considered and they are *date, time, currency, number*, *char* and *id*. The *id* type indicates that the attribute is used for identification purpose (e.g., product number, order number). The type information can be obtained through analyzing attribute name (containing date, time, price etc.) and the pattern or format of attribute values (that are viewable on the interface). For example, $300 for *currency* and 3:00PM for *time*. When the value type is difficult to determine, a default value type, i.e., *char*, is used. Whenever possible, the **scale/unit** of the attribute values is also extracted. For example, all values with US$ are considered to have the same unit but US$ and CAN$ have different units even though they are both of currency value type. Finally, each attribute has its **layout position** in the interface. The position value is determined by the layout order of attributes in an interface. More important attributes are usually arranged ahead of less important ones.

In addition to the label of an attribute, each element of the attribute may have its own label. For example, in Figure 1, attribute "publication year" has two text box elements with their own labels "after" and "before", respectively. Such label helps define the semantic meaning of the element.

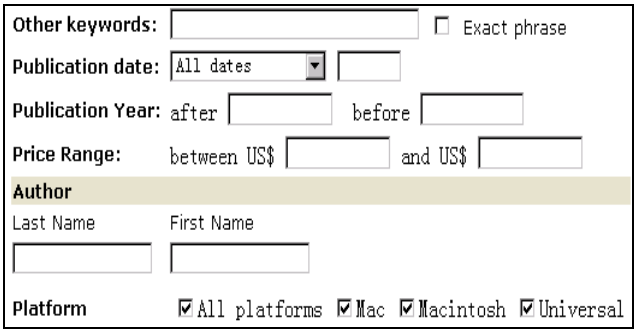

Figure 1: Examples of element relationship type

When an attribute has multiple elements, these elements are related in some way. We identify the following four relationship types among related elements based on our observations.

- **Range type:** It refers to the situation where two or more elements are used to specify the range semantics for an attribute. For example, in Figure 1, the "price range" has two related elements indicating the minimum and the maximum values allowed.
- **Part type:** It refers to the part-of relationship. For example, in Figure 1, "author" has two elements "first name" and "last name" and each of them is part of "author". Range type is a special case of part type.
- **Group type:** Multiple check boxes/radio buttons are sometimes used together to form a single semantic concept (attribute). In this case, the labels associated with the check boxes/radio buttons are values of the attribute. In Figure 1, attribute "Platform" has a group of check boxes.
- **Constraint type:** An element can be used as a constraint for another element. For example, for a text input box, a check box may be used to specify whether or not the input is case sensitive. In this case, the check box is meaningless without being related to the text input box.

To summarize, in our approach, each attribute A is represented as A = (N, P, DT, DF, VT, SU, ES, R), where N is the name (label) of A, P is the layout position of A, DT is the domain type of A, DF is the default value of A (possibly null and there is at most one default value for each group of check boxes and radio buttons),VT is the value type of A, SU is the scale/unit of A, ES is the set of elements associated with A and R is the relationship type between the elements in ES. For example, for attribute "Price Range" in Figure 1, its ES contains two text box elements labeled "between US$" and "and US$", and their relationship type is "range type". If ES contains only one element, then R is null. Each element E in ES is itself represented as a quadruplet E(L, F, V, DV), where L is its label (possibly empty), F is the format, V is the set of values (for finite domain type of elements only), and DV is the default value of the element (possibly null).

4. Matching attributes

In this section, we present our method for matching attributes from multiple local interfaces.

4.1 Semantic relationships

Semantic relationships between concepts or objects are very important in the database schema integration and Web source integration. In our approach, we identify the following three semantic relationships between terms (attribute names or element's values): *Synonymy*, *Hypernymy/Hyponymy* and *Meronymy* [M95, WDNT, BCV01, BBB01]. Given a term, we use WordNet [M95, WDNT] to get its synonyms, hypernyms and meronyms, if applicable.

- *Synonymy*. Term T_1 is a synonym of term T_2, denoted by $S(T_1,T_2)$, if T_1 is in the synonym-set of T_2.
- *Hypernymy/Hyponymy*. Term T_1 is a *hypernymy* of term T_2, denoted by $H(T_1,T_2)$, if T_1 is more generic than T_2. For example, *H*(*tree*, *maple*) and *H*(*format*, *hardcover*).
- *Meronymy*. Term T_1 is a meronym of term T_2, denoted by $M(T_1,T_2)$, if T_1 is a part of T_2. For example, *M*(*first name*, *name*) and *M*(*last name*, *name*).

However, hypernymy and meronymy terms that can be found from WordNet are very limited. In WISE-Integrator, we also identify hypernymy and meronymy relationships of two terms using the information in the interface representations. For example, suppose we have two interfaces, one has a "hardcover" attribute and the other has a "format" attribute that contains a value "hardcover". From this, we can identify the hypernymy between the two attributes: *H(format, hardcover)*. For meronymy, we use the *part* relationship of elements. For example, if a search interface contains an "author" attribute that has two parts: "first name" and "last name", we can say *M(first name, author)* and *M(last name, author)*. Other interfaces that contain "first name" or "last name" without "author" can use the relationship to match.

4.2 Normalization

Before integration, attribute names and element values are normalized as follows to reduce mismatches.

- Convert each name or value string to lower case equivalents.
- Remove all content in parentheses, including parentheses.
- Replace all characters that are not alphanumeric with a space character.
- Tokenize each string using space, replace abbreviation and acronym (if any) [MBR01] and use WordNet to get the base form of each token.
- Remove stop words when a name or a value consists of multiple words.

4.3 Merging attributes

Merging attributes has two tasks: one is to find the matching attributes from search interfaces to be integrated, and the other is to determine what global attribute name should be used for each group of matching attributes. To the best of our knowledge, no in-depth discussion of the second task has been reported in the literature.

The SEMINT approach in [LC00] utilizes and extends the metadata characteristics in [LNE89] to determine matching attributes. SEMINT introduces three levels of metadata that can be used: attribute names (the dictionary level), field specification (the schema level, e.g., data type and primary key) and attribute values and patterns (data content level). SEMINT just focuses on using the metadata at the schema level and data content level to determine attribute correspondences. It describes 20 characteristics at the two levels, such as data length, data type, nullable, primary key, default scale, minimum, maximum, average and so on. We adopt the basic idea of the SEMINT approach for the attribute-matching task in the sense we also use metadata characteristics in multiple levels. Our approach differs from the SEMINT approach in four aspects. First, the set of characteristics used is different. For example, primary key information and maximum value are readily available in a database context but they are not available for interface integration. On the other hand, information such as element format applies to only interface integration. Second, we utilize all three levels of metadata instead of just two. Third, we classify matches based on different metadata into positive matches and predictive matches (see below). Fourth, SEMINT uses neural network techniques but we don't. Furthermore, the SEMINT approach does not address the second task of merging attributes.

As mentioned above, in our approach, we use the three levels of metadata to determine matching attributes. At the dictionary level, we explore six possible matches on attribute names: exact match, approximate string match [WM92], vector space similarity match [FB92] (see section 4.3.2), synonymy match, hypernymy match and meronymy match. At the schema level, scale, value type,

domain type, default value and *Boolean* property are used. At the data content level, we focus on comparing values in the elements.

In our approach, we classify the different matches into two types: *positive* matches and *predictive* matches. *positive* matches include exact name match, semantic (synonymy, hypernymy and meronymy) matches and value-based match. For value-based match, we employ exact match, approximate string match, synonymy match and hypernymy match to compare values. When *enough* values from the two attributes are matched (a threshold is used), value-based match is recognized as succeeded. When one of the positive matches occurs during our integration process, the corresponding attributes are recognized as matched. *Predictive* matches consist of approximate name match, vector space similarity match of names, and matches based on scale, domain type, value type, default value, *Boolean* property and value pattern. Predictive matches must be sufficiently strong (based on a weight threshold) for two attributes to be recognized as matched.

Our approach for accomplishing the two tasks of merging attributes is described in the next two subsections.

4.3.1 Clustering (positive match)

This is to group attributes into clusters based on the positive matches between attributes. This step considers all interfaces. There are three steps for the clustering:

- Group attributes into clusters based on the exact match of attribute names in all interfaces of the same domain. Thus, after this step, all attributes in the same cluster have the same attribute name. For each distinct attribute name, the number of interfaces having the attribute is counted. Values of all attributes in each cluster, if any, are unioned.
- Merge the clusters produced in the first step based on the matching of values in each cluster and the semantic (synonymy, hypernymy and meronymy) matches of attribute names. New clusters are generated in this step.
- Determine the representative attribute name of each cluster produced in the second step. This attribute name is a candidate to be the global attribute name to which other attributes in the cluster are mapped.

To determine the representative attribute name of each cluster, generally we employ the *majority rule*. In other words, the attribute name that appears in most interfaces in a cluster would be chosen as the representative attribute name of the cluster. However, we also consider the semantic relationships among attribute names in the cluster. For example, if a cluster contains four different attribute names: "format", "binding type" "hardcover" and "paperback", we do not choose "hardcover" or "paperback" as the representative name of the cluster even if they appear in more interfaces. The reason is that "hardcover" or "paperback" is *a kind* of "format" or "binding type". Therefore, during the clustering we build hypernymy hierarchy trees for attribute names in the cluster. We then choose the representative attribute name among the roots using the majority rule. For the previous example, we just need to compare the number of occurrences between "format" and "binding type".

In our approach, the clustering step does preliminary attribute matching and representative attribute identification. The step just collects the knowledge about what attributes should be matched based on the positive information. No intermediate integrated interface is yet constructed after this step. There are several reasons to perform clustering. First, count the number of interfaces an attribute name appears in; this information is important for determining the global attribute names. Second, determine the representative attribute name of one cluster in advance. Third, make sure that attributes that should be matched (based on positive matches) are matched together. This can simplify the comparisons in the weight-based match step and avoid mismatches. Our experiments indicate that this two-step approach is effective.

4.3.2 Weight-based match (predictive match)

This step is to utilize the knowledge obtained in the first step and the predictive matches to construct the integrated interface and finalize the global attribute names.

Initially, there is no intermediate integrated interface. In this case, given a local interface, our approach takes an attribute in it and looks up the representative attribute name of the cluster in which the attribute appears. Then the representative attribute name is added to the intermediate interface (it is empty initially) as the global attribute name. These two operations are repeated until all attributes in the local interface are handled. The global attribute names may be adjusted later.

Once the first intermediate interface is generated, weight-based match begins to work. First, we present the definition of weight-based match as follows:

Definition: *Given an integrated intermediate interface* $\boldsymbol{I} = \{A^I_1, A^I_2, A^I_3, ..., A^I_n\}$ *and a local interface* $\boldsymbol{L} = \{A^L_1, A^L_2, A^L_3, ..., A^L_m\}$, *where* A^I_i *is an attribute of* $\boldsymbol{I}$, A^L_j *is an attribute of* $\boldsymbol{L}$, *the mapped attribute in* $\boldsymbol{I}$ *for an attribute* A^L_j *is the one with the highest weight:* $W(A^L_j, A^I_i) > W(A^L_j, A^I_k) > w, i = 1,...,n,\ k = 1,...,n, i \neq k,$ *where* $W(A^L_j, A^I_i)$ *is the weight of attributes* A^L_j *and* A^I_i, *w is the weight threshold.*

In our approach, the weight-based match computes the matching weight between two attributes and then predicts whether the two attributes are matched based on the weight. The weight between attributes A^L_j and A^I_i can be computed based on the following metrics (predictive matches):

1. *Approximate string match*
   Compare the two attribute names to find out if the edit-distance between the two strings is within the allowed threshold. We use an approximate string-match algorithm [WM92] to find the match. If the edit-distance is within the allowed threshold *T*, assign a positive weight *Wam*; otherwise *Wam* is 0.
2. *Vector space similarity*
   The vector space similarity is the similarity between two text strings based on the Vector Space Model [FB92]. The approach is also used in [Coh98]. We tokenize each string and get the term frequency of each term in each string. The weight of this metric is the Cosine similarity of two strings.

$$Wvss\ (v, w) = \frac{\sum_{j=1}^{m} v_j \cdot w_j}{\sqrt{\sum_{j=1}^{m} (v_j)^2 \cdot \sum_{j=1}^{m} (w_j)^2}}$$

   where *m* is the number of unique terms in the two strings, w_j is the term frequency of the *j*th term in attribute string *w* and v_j is the term frequency of the *j*th term in attribute string *v*.
3. *Compatible domain*
   We consider four domain types: *finite*, *infinite*, *hybrid* and *range*. The domain type of an attribute is derived from its associated element(s). If the element(s) of the attribute has the range semantics, the domain type of the attribute is *range*. *Hybrid* is the combination of *finite* and *infinite*. If an attribute domain is hybrid, users can either select from a list of pre-compiled values or fill in a new value. In our approach, the *hybrid* type is only limited to the intermediate interface and the global interface. *Hybrid* is compatible with *finite* and *infinite*; the same types are compatible. If two attributes have compatible domain types, assign a weight *Wcd*; otherwise *Wcd* is 0. In addition, we observed that *range* type is used much less often than *finite* and *infinite* types. Thus, if two attributes have *range* domain type, we double *Wcd*.
4. *Value type match*
   As mentioned in Section 3, we consider six value types: *date, time, currency, number*, *char* and *id*. If two attributes have the same value type, assign a weight *Wvtm*; otherwise *Wvtm* is 0
5. *Scale/unit match*
   Consider two attributes that have the same value type. If they also have the same scale or unit, assign a weight *Wcs*; otherwise (i.e., if they have different value types or different scales/units), *Wcs* is 0. For example, if two attributes are both of currency type and their values are in US$, then *Wcs* is assigned to the overall match of the two attributes.
6. *Default value*
   In a search interface, some elements may have their default values. In some cases, an element may have no associated label, but it has a default value which is important for the element to find a matching attribute. In addition, if an attribute is in a cluster, then its default value is considered as one of the default values of the cluster. So when we check default values of two attributes we check default values of the two attributes themselves as well as their clusters. If two attributes have the same default value, assign a weight *Wdv*; otherwise *Wdv* is 0;
7. *Boolean property*
   If an attribute has just a single check box, this check box is usually used to mark a yes-or-no selection. Such an attribute is considered to have a B*oolean* property. If both attributes have the *Boolean* property, assign a weight *Wbp*; otherwise *Wbp* is 0.
8. *Value pattern*
   We apply value pattern only to the numeric attributes. We compute the average of all numeric values in each attribute. If the two averages are close, assign a weight *Wvp*; otherwise *Wvp* is 0.

The weight between attributes A_j^L and A_i^I is the sum of the above eight metric weights (the values of these weights are determined experimentally, see Section 7.2.2):

$$W(A_j^L, A_i^I) = Wam + Wvss + Wcd + Wvtm + Wcs + Wdv + Wbp + Wvp$$

Given the intermediate interface and an attribute A_j^L in a local interface, the approach first looks up the attribute thesaurus to see if the attribute is already mapped to a global attribute in the intermediate interface. The attribute thesaurus is established incrementally during the weight-based matching process. If it has been mapped, the attribute A_j^L would directly be mapped to the global attribute name. If it has not, the representative attribute name would be found using the name of attribute A_j^L. Then recheck the attribute thesaurus using the representative attribute name to see if the representative attribute name is mapped to a global attribute. If the mapping is found, A_j^L is mapped to the global attribute; otherwise compute the weights between A_j^L and all attributes in the intermediate interface. After these weights are computed, the attribute with the highest weight is selected. If this weight is greater than the threshold *w*, the selected attribute is considered as the matching attribute of the attribute A_j^L; otherwise, we assume that no matching attribute is found. In the former case, we have to determine the global attribute name between the two attributes. In our approach, the attribute name that appears in more local interfaces would be selected. If applicable, the corresponding entry in the attribute mapping table (which keeps mappings between

each global attribute and its corresponding local attributes) is changed and so is the thesaurus. In the latter case, the attribute A_j^L is added as a new attribute to the intermediate interface and a new entry for the attribute is added in the attribute mapping table.

4.3.3 Maintenance of integrated interface

After a global interface is generated, it is likely that new local interfaces need to be added to or some existing local interfaces need to be removed from the global interface from time to time. This requires maintaining the global interface. For adding new local interfaces, the first step of clustering needs to be performed on the new local interfaces, followed by the second step of clustering to cluster the output of the first step to the existing clusters. The representative attribute name may need to be updated based on the current and previous statistical and semantic knowledge. Then, the weight-based match is performed. For removing some local interfaces from the global interface, we remove the attribute names and their corresponding values from the clusters and the related mapping information from the attribute mapping table. In both cases, the count indicating the number of interfaces containing an attribute needs to be updated accordingly, and if applicable, the global attribute name of the global interface may also need to be changed.

5. Merging attribute domains

When a local attribute is mapped to a global attribute in the intermediate interface, we must determine the global attribute domain after the mapping. This includes the following two aspects:

1) *The global domain type*. As mentioned previously, four domain types are supported in our approach and they are *finite, infinite, hybrid* and *range*. A compatible domain type between the two attributes should be used.
2) *The attribute values.* Need to merge the values that represent the same concept and provide a set of values for the global attribute.

To deal with these two issues, differences between the two domains should be identified and resolved, including format difference, semantic conflict, scale difference, range difference and constraint difference. Here we need to take a closer look at the range difference. In Figure 3, we can see that there are various range formats. Two aspects need to be considered in resolving range conflict, one is about *range modifiers* such as "from", "to", "less than", "under" and so on, and the other is about *range width*. Figure 3 shows that different range domains may have different range modifiers and different range width. The resolution of range conflicts is to generate a global range domain that is compatible with the range domains of the matching attributes (see details in section 5.3).

5.1 Determine global domain type

For a given local attribute A_j^L and a matching global attribute A_i^I, we use the following rules to determine the new domain type for A_i^I:

1) finite + finite → finite
2) infinite + infinite → infinite
3) range + any type → range
4) (finite + infinite) or (hybrid + finite) or (hybrid + infinite) → hybrid

The first rule can be explained as follows: if the local attribute A_j^L is *finite* and the global attribute A_i^I is also *finite*, then the new global domain type of A_i^I is *finite*. Other rules can be explained similarly.

5.2 Merging alphabetic domains

If a local attribute and its matching global attribute are *finite* or *hybrid* and have alphabetic values, we should consider how to merge their values and form a new value set for the global attribute. In WISE-Integrator, this is carried out in two phases. The first phase is in the clustering step discussed in Section 4.3.1. In this phase, attributes that have some values in common are grouped into the same cluster. Furthermore, due to the matching techniques employed (exact match, approximate string match, synonymy match and hypernymy match), semantic relationships between values are identified. In the second phase, we use the knowledge of the relationships between values to merge values and generate a global value set.

Phase 2 consists of the following steps. First, we cluster all values into categories based on approximate string match, vector space similarity match, synonymy match and hypernymy match. Thus, all values that are similar, synonymy or hypernymy are clustered into the same category. Next, we solve the following two problems: (1) Which value should be chosen as the global value if multiple similar and synonym values are in the same category? (2) How to provide values to users if the values in the same category have hypernymy relationships? For the first problem, we can keep a counter for each value and use the *majority rule* to choose the most popular value. As to the second problem, we need to make a tradeoff between choosing ***generic*** concepts and choosing ***specific*** concepts as the choice would have different effects on query cost and interface friendliness. The cost of evaluating a global query includes the cost of invoking local ESEs to submit sub-queries, the cost of processing sub-queries at local ESEs, result transmission cost and post-processing cost (e.g., result extraction and merging). If we choose only generic concepts as global values and do not use specific concepts, a query against the global interface may need to be mapped to multiple values (corresponding to specific concepts) in some local

interfaces, leading to multiple invocations to the local search engines. On the other hand, if we keep only specific concepts and ignore generic concepts, users who want to query generic concepts (i.e., have broader coverage) may have to submit multiple queries using specific concepts, resulting in less user-friendly interface. Our approach is to provide a concept hierarchy of values to users, including generic and specific concepts. This remedies the problems of the previous two options and gives the users more flexibility to form their queries. Value clustering may produce multiple categories and a value hierarchy is created for each category. Each hierarchy is limited to at most three levels to make it easier to use.

After these two phases, the mappings between global values and local values are established.

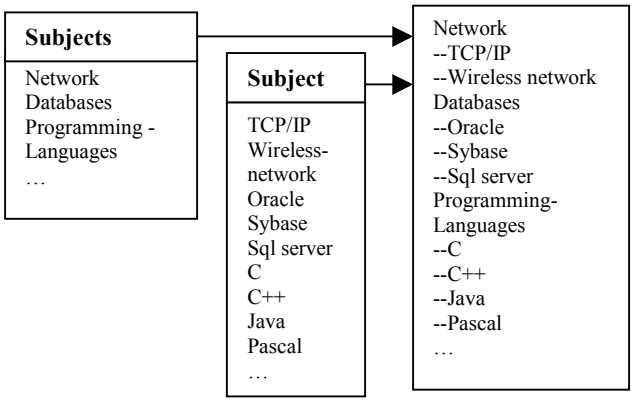

Figure 2: Example of merging domain values

Example 1: Consider two Web bookstore interfaces, one has an attribute "subjects" with values "Network", "Databases", "Programming Languages" and so on and the other has a corresponding attribute "subject" with values "TCP/IP", "Wireless network", "Oracle", "Sybase", "Sql server", "C", "C++", "Java", "Pascal" and so on. After clustering the values, some semantic hierarchies between the values from the two interfaces can be identified. There are three possible ways to generate the global domain values for "subject". One is to use only ***generic*** concept values, i.e., values from the *first interface,* namely "Network", "Databases", "Programming Languages" etc. In this case, suppose a user wants to find information about Oracle. Since "Oracle" is not available, the user has to select "Databases" on the global interface and submit the query. This global query would have to be mapped to three sub-queries for the *second interface*, namely "Oracle", "Sybase", and "Sql server". Obviously, searching based on "Sybase" and "Sql server" wastes the resources at the second site and returns more useless results to the user. The second option is to use only the values with more ***specific*** concepts, i.e., the values from the second interface. In this case, a user who wants to find information about database (not of any specific type) needs to submit three queries respectively using "Oracle", "Sybase" and "Sql server". This is inconvenient to the user. Our approach will organize related values into a ***hierarchy*** (see the box on the right side in Figure 2). In this case, if the user selects "Databases", then the meta-search engine will generate three sub-queries for the second site on behalf of the user. On the other hand, if any of the three sub-concepts of "Databases" is selected, only that concept will be used for the second site but "Databases" will be used for the first site. This solution remedies the problems of the first two solutions.

We should point out that not every category of values can form a hierarchy. In that case, we just provide a list of values.

5.3 Merging numeric domains

To merge numeric domains, we need to perform the following tasks:

1) Resolve scale difference. We assume the identification of the scale/unit of a numeric attribute has already been done by the interface extractor. In our approach, we build a scale relationship dictionary in advance for some popular scales. The system can look up the dictionary to find out how to map one scale to another scale. The numeric values in those attributes are transformed to the same global scale during value merging.
2) Understand the semantic differences involved.
3) Generate a global domain with query cost taken into consideration.

We identify two types of numeric domains: *range numeric domain* and *non-range numeric domain*. *Non-range numeric domain* attributes may come from the numeric attributes that are either *finite* or *infinite*. If the domains of the matching local attributes are *non-range numeric*, we just union all values of these attributes for the global attribute.

For the rest of this subsection, we focus on *range numeric domain*. For the *range numeric domain*, three types of formats can be identified as shown in Figure 3.

1) *One selection list*. The range type consists of only one selection list, for example, the first four selection lists in Figure 3.
2) *One selection list* and *one text box*. The range domain is like the "publication date" in Figure 3, which has two elements, one is a *selection list* for range modifier and the other is a *text box* for numeric value.
3) *Two textboxes* or *two selection lists*. The type consists of two elements and each of them may be a *textbox* or a *selection list*. The examples are "price range", "publication year" in Figure 3.

From Figure 3, we can see that numeric values are mostly combined with other semantic words. To help the system understand such formats, we need to let the system know the meaning of the range modifiers such as "less

than", "from", "to" and "over". For this purpose, we build a semantic dictionary that keeps all possible range modifiers for numeric domains. In addition to these range modifiers, we also save the meaning of other terms related to numeric values. For example, in Figure 3, we can see that "baby" and "teen" are in "reader age". We have to specify the real meanings of these words to help the system know what they are. We can say that "baby" represents "under 3 years", "teen" is "13-18 years" and "adult" is "over 18 years". Then, we design a special extractor that can extract the range modifiers and the numeric values, and use the semantic meanings in that dictionary to build a semantic range table that can be understood by the system. The semantic range table keeps multiple ranges corresponding to the original ranges in the element(s). This table can be used in query mapping and submission.

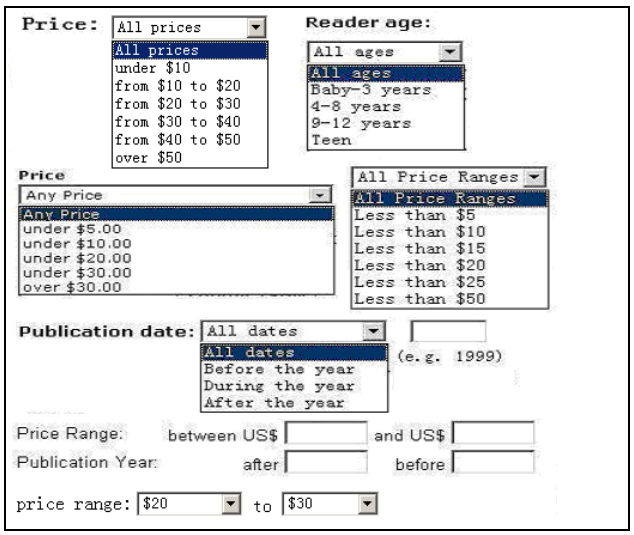

Figure 3: Examples of different range formats

Suppose we handle the element that has "less than" range modifiers in Figure 3. From this element, we can obtain numeric values: *5, 10, 15, 20, 25* and *50* by extraction. We can also get the semantic words: *"all price ranges"* and *"less than"*. With the information, we can build a range semantic table as shown in Table 2. The internal values are the values in the HTML text that correspond to the values of the element.

So far we have solved the first two problems of merging range numeric domains. The last thing we need to do is to generate a global range format that is compatible with the local domain formats of the matching attributes. And the global range format should consider query efficiency as much as possible. Intuitively, a larger range condition in the global interface would lead to more invocations to some local sites, causing more local server processing effort, more data transmission and more post-processing effort. Therefore, we aim to reduce the range width of each range condition in the global interface. To this end, we keep a list for the matching numeric attributes. Every time when a numeric attribute is mapped or added to the intermediate interface, the numeric values that are previously extracted from the numeric attribute are added to the list (scale conflicts are resolved before this step). When all attributes are matched, the list is sorted in ascending order of the values. The ranges are generated using every two consecutive numeric values in the list. For the minimum and the maximum values, "under" and "over" range modifiers are used, respectively.

Range modifiers	Meaning
Less than	<
Over	>
Under	<
Greater than	>
From*	>=
To*	<=
Between*	>=
And*	<=
After	>
Before	<
…	…
All	All range
Any	All range

Table 1: Range modifiers dictionary

* modifiers to be used in pairs

Lo	Hi	Internal value
0	5	'lessthan5'
0	10	'lessthan10'
0	15	'lessthan15'
0	20	'lessthan20'
0	25	'lessthan25'
0	50	'lessthan50'
0	∞	'allrange'

Table 2: A range element semantic table

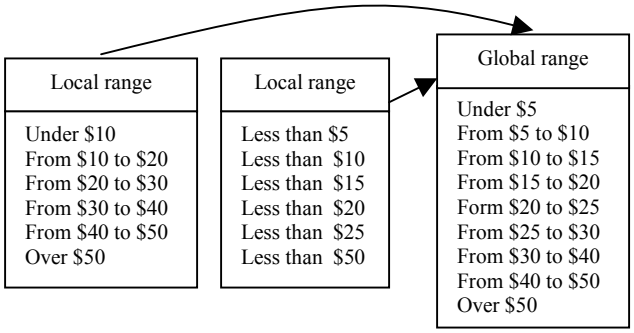

Figure 4: Example of a global range domain

Example 2: Suppose in Figure 3 two attributes with "from" and "less than" range modifiers are matched, then the list of values kept for the two matching attributes is: 5, 10, 15, 20, 25, 30, 40 and 50. The global range format is shown in Figure 4. From Figure 4, we see that one global range condition is translated to only one appropriate local range condition. For example, "from $10 to $15" in the global range format is respectively mapped to "from $10 to $20" and "less than $15" in the local range formats. Thus multiple query invocations to local interfaces are avoided and other costs including post-processing time are also reduced.

6. Generating global interface

WISE-Integrator uses the results of both the attribute matching and the attribute domain merging to generate the global interface and show the interface in HTML format. It also has to decide which attribute should appear in the global interface and the layout of all the attributes.

6.1 Attribute position

Each attribute has its layout position in a given local interface. These layout positions reflect the importance of the attributes as perceived by local interface designers and their users, and they may influence users' behaviors of selecting attributes to use. To be user-friendly, we aggregate local importance of each attribute and arrange more important attributes ahead of less important ones. In WISE-Integrator, the global layout position of a global attribute is computed as follows.

$$P(A_i) = \sum_{j=1}^{m} P(A_i^j)$$

where $P(A_i)$ denotes the position value of the i-th global attribute A_i, m is the number of local interfaces to be integrated, $P(A_i^j)$ is the layout position of the local attribute in the j-th local interface that is mapped to A_i; $P(A_i^j)$ is assigned the total number of global attributes when no matching local attribute exists in the j-th local interface. All global attributes are laid out in increasing order of their position values. Clearly, using this method, attributes that appear in high positions (the first position is the highest) in many local interfaces are likely to appear in high positions in the global interface.

6.2 Attribute selection

When a large number of local interfaces are integrated, the global interface may have too many attributes to be user-friendly. While some key attributes about the underlying products appear in most or all local interfaces, some less important attributes appear in only a small number of local interfaces. One way to remedy this problem is to trim some less important attributes from the global interface. We use the global position of each global attribute to trim off less important attributes, i.e., those that have large global position values. A user-adjustable threshold can be used to control this.

7. Implementation and experimental results

7.1 Implementation

WISE-Integrator is developed using JDK1.4 and is operational. WordNet1.6 is embedded into the system through APIs based on C. The GUI of the system is shown in Figure 5. The system can read the interface description of each Web site and then display the interface description visually in a tree structure. From the tree view, users can see all information on each search interface. The global interface and the attribute matching information are shown after the integration is finished. Through the GUI, users can remove or add any interface at any time on the fly. And the new global interface is generated without starting from scratch. In addition, a user can choose any parameter value to trim attributes from the global interface.

To see a demo of WISE-Integrator, go to the Web site: http://www.cs.binghamton.edu/~haihe/projects/wise.html.

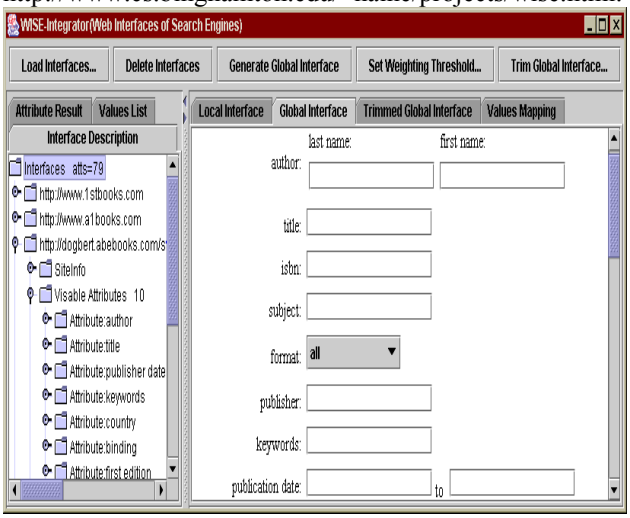

Figure 5: WISE-Integrator interface

7.2 Experiments

7.2.1 Evaluation criteria

Three qualitative criteria for measuring the quality of a global conceptual schema in the context of database schema integration are proposed in [BLN86] and they are *Completeness and Correctness*, *Minimality* and *Understandability*. We rephrase these criteria and propose the following principles to guide the evaluation of search interface integration.

Correctness. Attributes that should be matched are correctly matched; attribute domains for the matching attributes are correctly merged and constructed.

Completeness. If a result can be retrieved directly through a local interface, then the result can also be retrieved through the global interface.

Efficiency. Global interface construction should consider query cost. While query cost is usually considered at the query evaluation time, a bad global interface may cause a high query cost despite of good query evaluation algorithms. For example, supporting only very wide range conditions in the global interface may cause too many local queries to be submitted to a local engine and too useless results to be transmitted to the metasearch engine.

Friendliness. A global interface should be simple and easy to understand and use by users. As an example, it is better to provide users a list of values for an attribute

Domain	The number of Interfaces	Total Attributes	Case 1	Case 2	Case 3	ama(%)	amc(%)
Book	10 (1st round)	79	76	0	3	96.20	100
	20 (2nd round)	159	150	2	7	94.34	98.74
	30 (3rd round)	210	201	2	7	95.71	99.05
	40 (4th round)	259	250	1	8	96.53	99.61
	50 (5th round)	313	302	3	8	96.49	99.04
Electronics	10 (1st round)	68	63	5	0	92.65	92.65
	20 (2nd round)	135	128	5	2	94.81	96.30
Average						95.25	97.91

Table 3: Attribute matching correctness and completeness

when these values are available for the attribute rather than let users fill out without any knowledge. As another example, frequently used attributes should be arranged ahead of less frequently used ones.

Efficiency and friendliness of the global schema are taken into consideration by WISE-Integrator (see Sections 5 and 6). In the next subsection, we report our experimental results for completeness and correctness for matching attributes.

7.2.2 Experimental results

To perform the experiments, we collected the search interfaces of 50 book Web stores and 20 electronics Web sites, and then constructed the interface representation for each search interface by hand (Tools for automatic construction of ESE interface representation is under development and will be reported in another paper).

Correctness requires that attributes that should be matched across all search interfaces be matched and that attributes that should not be matched not be matched. It also requires that the attributes in the global interface be semantically unique. To help measure the correctness of attribute matching, the global attribute name and semantics are used as a reference to measure how well local attributes are matched to the global attribute. If there exist multiple global attributes that are semantically the same in the global interface, the global attribute with more local attributes matched is considered as the only real global attribute while others should be matched to it. There exist three cases for attribute matching:

1) Attributes are correctly matched to a unique global attribute.
2) Attributes are incorrectly matched to a global attribute.
3) Attributes are correctly matched to a global attribute, but they should belong to another matched group that has more matching attributes.

Our evaluation metric is called Attribute Matching Accuracy (*ama*), which defines what percentage of all attributes is correctly matched.

$$ama = \frac{\sum_{i=1}^{n} m_i}{\sum_{i=1}^{n} a_i}$$

where n is the number of all local interfaces used for integration, m_i is the number of correctly matched attributes in the i-th interface (*case 1*), a_i is the number of all attributes in the i-th interface.

Completeness requires that all contents and capabilities of each local interface be preserved in the global interface. As we mentioned above, three cases exist for attribute matching. Among these three cases, *case 2* would reduce the completeness because some attributes are mismatched to a global attribute; using such global attributes may lead to incorrect results from some local search engines. For *case 3*, although the uniqueness requirement is not satisfied, using these global attributes can still retrieve results from the matching local interfaces. Therefore, *case 3* matches do not affect completeness.

We define the Attribute Matching Completeness (*amc*) measure as follows:

$$amc = \frac{\sum_{i=1}^{n} (a_i - r_i)}{\sum_{i=1}^{n} a_i}$$

where r_i is the number of mismatched attributes in the i-th interface (*case 2*).

We performed 5 rounds of experiments on book interfaces. In the first round, 10 interfaces were randomly selected and a global interface was generated for them. In each subsequent round, 10 additional interfaces were randomly selected and added to previously selected interfaces. Then a global interface was generated from all selected interfaces from scratch. Then we manually checked how well the attributes are matched. We also performed 2 rounds of experiments using interfaces of electronics sites. The experimental results are shown in Table 3. We can see that, on the average, the overall correctness and completeness of our approach for the two domains are **95.25**% and **97.91**%, respectively. In addition, the results are remarkable stable (with all correctness and completeness values within a narrow range) despite the differences in the number of interfaces used and the product types.

In all experiments, the weights for the seven metrics in section 4.3.2 (the other metric, similarity match, has no fixed weight) are: *Wam=0.5, Wcs=0.2, Wcd=0.1, Wvtm=0.4, Wdv=0.6, Wbp=0.1* and *Wvp=0* (value pattern match is not used in our experiments) and the weight

threshold *w* is 0.63. These values are obtained from the experiments using the book interfaces and they are applied to the electronics interfaces without changes. As the interfaces for books are very different from those for electronics, the experimental results indicate that the above parameter/threshold values are robust.

8. Conclusions

In this paper, we provided a comprehensive solution to the problem of automatically integrating the interfaces of E-commerce search engines. The problem is significantly different from schema integration for traditional database applications. Here we need to deal with not only schema integration, but also attribute value integration, format integration and layout integration. In this paper, we described our techniques used to build WISE-Integrator. With appropriate interface representation of local interfaces, WISE-Integrator automatically integrates them into a global interface using only domain (application) independent knowledge. Our two-step approach based on positive matches and predictive matches for merging attributes was shown to be very effective by our experiments. We believe that the proposed approach can also be applied to other domains of E-commerce or ones beyond E-commerce such as digital library and some professional databases on the Internet.

While good results were obtained using our method, there is room for improvement. One possibility is to use the Open Directory Hierarchy to find more hypernymy relationships. One possible way to reduce the case 3 problem in Section 7.2.2 is to allow an attribute in a local interface to match more than one attribute in the intermediate interface in Section 4.3.2. We will investigate these possibilities in the near future.

Acknowledgements

This work is supported in part by the following grants from National Science Foundation: IIS-0208574, IIS-0208434, EIA-9911099 and ARO-2-5-30267. We thank Mr. Leonid Boitsov for providing us his agrep algorithm implementation (itman.narod.ru/english/aboutnotser.html).

References

[BLN86] C. Batini, M. Lenzerini, S. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. ACM Computing Surveys, 18(4):323-364, December 1986.

[BBB01] I. Benetti, D. Beneventano, S. Bergamaschi, F. Guerra and M. Vincini. *SI-Designer:* An Integration Framework for E-Commerce. 17th IJCAI-01, Seattle.

[BCV01] S. Bergamaschi, S. Castano, M. Vincini, D. Beneventano. Semantic Integration of Heterogeneous Information Sources. Journal of Data and Knowledge Engineering, 36(3):215-249, 2001.

[Coh98] W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. ACM SIGMOD Conference, Seattle, WA, 1998.

[DR02] H. Do, E. Rahm. COMA- A System for Flexible Combination of Schema Matching Approaches. The 28th VLDB conference, Hong Kong, 2002.

[DDH01] A. Doan, P. Domingos, A. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine-learning Approach. ACM SIGMOD Conference, May 2001.

[DEW96] R. B.Doorenbos, O. Etzioni, and D. S.Weld. A Scalable Comparision-Shopping Agent for the World Wide Web. Techinical Report UW-CSE-96-01-03, University of Washington, 1996.

[FB92] W. Frakes and R. Baeza-Yates. Information Retrieval: Data Structures & Algorithms. Prentice Hall, Englewood Cliffs, N.J. 1992.

[GKD97] M. Genesereth, A. Keller, O. Duschka. Infomaster: An Information Integration System. ACM SIGMOD Conference, May 1997

[HC03] B. He, K. Chang. Statistical Schema Integration Across the Deep Web. ACM SIGMOD Conference, 2003.

[HTL4] HTML4: http://www.w3.org/TR/html4/.

[LNE89] J. Larson, S. Navathe, R. Elmasri. A Theory of Attribute Equivalence in Databases with Application to Schema Integration. IEEE Transactions on Software Engineering, Vol.15, No.4, April 1989.

[LRO96] A. Levy, A. Rajaraman, J. J.Ordille. Querying Heterogeneous Information Sources Using Source Description. The 22nd VLDB Conference, India, 1996

[LC00] W. Li, and C. Clifton. SEMINT: A Tool for Identifying Attribute Correspondences in Heterogeneous Databases Using Neural Networks. Data & Knowledge Engineering, 33: 49-84, 2000.

[MBR01] J. Madhavan, P. Bernstein, E. Rahm. Generic Schema Matching with Cupid. VLDB Conference, 2001.

[MGR02] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. IEEE Conference on Data Engineering, San Jose, 2002.

[M95] A. Miller. WordNet: A Lexical Database for English. Communications of the ACM, 38(11): 39-41,1995.

[PMH03] Q. Peng, W. Meng, H. He, and C. Yu. Clustering of E-Commerce Search Engines. Submitted for publication, 2003.

[RGM01] S. Raghavan, H. Garcia-Molina. Crawling the Hidden Web. The 27th VLDB Conference, 2001.

[RB01] E. Rahm, P. Bernstein. A Survey of Approaches to Automatic Schema Matching. VLDB Journal, 10:334-350, 2001.

[WDNT] WordNet: http://www.cogsci.princeton.edu

[WM92] S. Wu and U. Manber. Fast Text Searching Allowing Errors. Communications of the ACM, 35(10):83-91, October 1992.