Usage-Based Schema Matching*

Hazem Elmeleegy[1], Mourad Ouzzani[2], and Ahmed Elmagarmid[1, 2]
[1]*Department of Computer Science* [2]*Cyber Center*
Purdue University, West Lafayette, IN
{hazem,mourad,ake}@cs.purdue.edu

***Abstract*— Existing techniques for schema matching are classified as either schema-based, instance-based, or a combination of both. In this paper, we define a new class of techniques, called usage-based schema matching. The idea is to exploit information extracted from the query logs to find correspondences between attributes in the schemas to be matched. We propose methods to identify co-occurrence patterns between attributes in addition to other features such as their use in joins and with aggregate functions. Several scoring functions are considered to measure the similarity of the extracted features, and a genetic algorithm is employed to find the highest-score mappings between the two schemas. Our technique is suitable for matching schemas even when their attribute names are opaque. It can further be combined with existing techniques to obtain more accurate results. Our experimental study demonstrates the effectiveness of the proposed approach and the benefit of combining it with other existing approaches.**

I. INTRODUCTION

Schema matching has long been one of the most important, yet difficult, problems in the area of data integration. With the exploding number of information systems, the need for schema matching solutions is growing. In life sciences, information integration is becoming a bottleneck limiting what scientists can accomplish. Businesses are relying on integration more than ever before. In disaster recovery situations, several entities and authorities need to rapidly exchange information. Schema matching is a key component in all such applications. Moreover, many of today's integration tasks have to cross country boundaries, thus adding a new dimension to this vexing challenge.

The problem of schema matching is essentially to find correspondences (*matches*) between the attributes of two schemas. The set of generated matches is collectively referred to as a *mapping* between the schemas. Much attention has been paid to this problem in the literature, and many techniques have been proposed, e.g., [7,10,11,12,12,13]. These techniques are fundamentally divided into two classes based on the source of information they exploit to make their matching decisions. On one hand, *schema-based techniques* rely on the metadata available for the schemas in terms of attribute names, descriptions, data types, domains, and integrity constraints. *Instance-based techniques*, on the other hand, rely on the characteristics of the data instances such as their format, distribution, entropy, and correlation with instances of other attributes. Many systems have also been proposed to utilize a combination of those techniques [6,7,12]. The instance-based technique, proposed in [11], tackled the problem of schema matching with opaque attribute (or column) names, i.e. when attribute names are unreliable for matching purposes. This is a very realistic case, especially when matching multi-lingual schemas. However, the authors only showed how their technique can match individual tables and not complete schemas.

In this paper, we propose a new technique for schema matching, which does not fall in either of the previous two classes, but rather defines a new class of its own, which we refer to as *usage-based schema matching*. The proposed technique exploits the usage information of the attributes in the query logs to find matches, in contrast to relying on the schema information or the data instances. This may be the only option for schema matching if the information needed by the two other techniques is not available or not reliable enough to achieve good matching quality. The proposed technique first identifies co-occurrence patterns between attributes and additional features, such as their use in joins and with aggregate functions. Then, it employs a genetic algorithm to find the highest-score mappings according to the scoring function used to measure the similarity between the features of the matching attributes.

Our technique is suitable for matching schemas even when their attribute names are opaque or when they have different layouts. It is applicable to match complete schemas, rather than individual tables. In addition, our technique can be combined with other matching techniques using the combination methods proposed in the literature (e.g., [6]) to obtain higher-quality matches. In this paper, we experimentally verify the effectiveness of the usage-based technique, and show that when combined with simple matchers like a data type matcher or established matching techniques like the Similarity Flooding algorithm [13], the generated matches indeed reach high degrees of accuracy.

The paper makes the following contributions:

1. The description of a new class of techniques for matching schemas based on the usage of their attributes in the query logs. In particular, we describe details of two usage-based matchers (SLUB and ELUB).
2. A prototype implementation of the proposed techniques, which employs a genetic algorithm to find the highest score mappings.

* This work was supported by Lilly Endowment, NSF-ITR 0428168, and US DHS PURVAC.

3. An extensive experimental study showing the effectiveness of the usage-based schema matching technique and the benefit of combining it with other techniques, including the Similarity Flooding algorithm especially when attribute names are opaque.

In Section II, we discuss the related work. Section III presents the example that will be used throughout the paper. The usage based technique is described in Section IV, while the implementation issues are discussed in Section V. Experiments and their results are presented in Section VI, and finally Section VII concludes the paper and suggests directions for future work.

II. Related work

Schema matching has been extensively studied over the past two decades (See [15] for a comprehensive literature survey until 2001). Cupid [12] combines *element-level* and *structure-level* schema-based techniques to perform schema matching. Element-level techniques focus on the properties of each attribute in isolation, while structure-level techniques consider relationships between attributes. LSD [7] is an extensible framework, which employs several schema-based and instance-based matchers, and uses machine learning approaches to train and combine them. COMA [6] is another framework for combining matchers, providing several strategies for aggregating their results. Madhavan et al. [12] propose the use of a corpus of previously matched schemas to match a pair of new schemas. Using a corpus of schemas is also considered by He et al. [10]. However, their focus is on deep web applications and they follow a holistic approach to simultaneously find mappings between all the schemas of the corpus. Kang et al. [11] propose using mutual information between attributes to match tables of schemas with opaque attribute names and data values. Similarity Flooding [13] is a fixpoint computation algorithm for matching schema graphs, aided by an attribute name matcher. All of these previous works did not consider using the query logs for schema matching. Therefore, our work can be seen as either complementary to them as it can be combined with such techniques within the same matching framework or the only alternative if the information required by the previous techniques is not available or unreliable.

Much work has been focused on generating more complex types of mappings than finding simple one-to-one attribute correspondences. The Clio system [1,9] generates SQL-like mappings based on the attribute correspondences. iMap [5] focuses on finding complex relations between attributes in both schemas such as `price=rate*(1+tax)`. More recently, Bohannon et al. [2] introduced *contextual schema matching*, in which a match between a pair of attributes is valid only when certain conditions are met in the data instances. Warren et al. [18] proposed a method to find the relation between attributes in one schema and substrings of multiple attributes in the second schema. We believe that this body of work can benefit from our usage-based approach, since more evidence about these complex relations can be found in the query logs.

Finally, the idea of analyzing query logs has been used extensively in the area of self-tuning databases (e.g. [3,4]).

III. The bookstores example

As our motivating example, we consider a fictitious company: AllBooks Inc. AllBooks mission is to provide online access to bookstores all over the world. One of their biggest challenges is how to match the schemas of the numerous bookstores to their own schema; so that the AllBooks web application can seamlessly forward queries to and retrieve answers back from each bookstore. The schemas of the bookstores have different structures, are written in different languages, and, in many cases, the table and attribute names are not easily interpretable. Moreover, the owners of the bookstores were willing to share their schemas, but not to provide labor for manual schema matching. AllBooks offered to collect the schemas of the bookstores, and provide each of them with a software tool that analyzes its query log such that the output of the analysis is sent back to AllBooks to help in the schema matching process. The rationale is to use a usage-based schema matching technique whenever the schema-based information is of low quality and cannot be relied on.

Fig. 1 shows an example of the schemas of two bookstores (X-Books and Y-Books), that AllBooks had to deal with. Although different in layout, the two schemas cover the same information about the bookstore domain. In particular, they cover information about books, book authors, customers, and ordering history. The attribute names in the figure are shown to be easily interpretable, only for illustration purposes. The two schemas were derived from the TPC-W benchmark [16], which gives the specifications for building an online bookstore. The X-Books schema is a reduced version of the TPC-W schema, while the Y-Books schema is a modified version of the X-Books schema. We will be referring to this example throughout our discussion.

IV. Usage-based schema matching

The goal of the usage-based schema matching technique is to exploit similarities in the query patterns to match attributes, which seem to play the same role in their respective databases. We are not claiming that the query patterns in the same domain will *always* be similar. However, like schema-based techniques, which will only be effective when the attribute names share some similarities, usage-based techniques will be most effective when the query patterns are close to each other; a requirement that is likely to be met in many situations. For example, as in Section III, users of many bookstores are expected to issue similar queries because the semantics of these queries is entailed by the business activities rather than by the specifics of the schema design. The same argument applies to other domains like healthcare, finance, and scientific databases.

Our proposed technique has two main phases: *feature extraction* and *matching*. The feature extraction phase collects information from the query logs characterizing the attributes' roles and their interrelationships. The matching phase examines several potential mappings, and assigns a score for

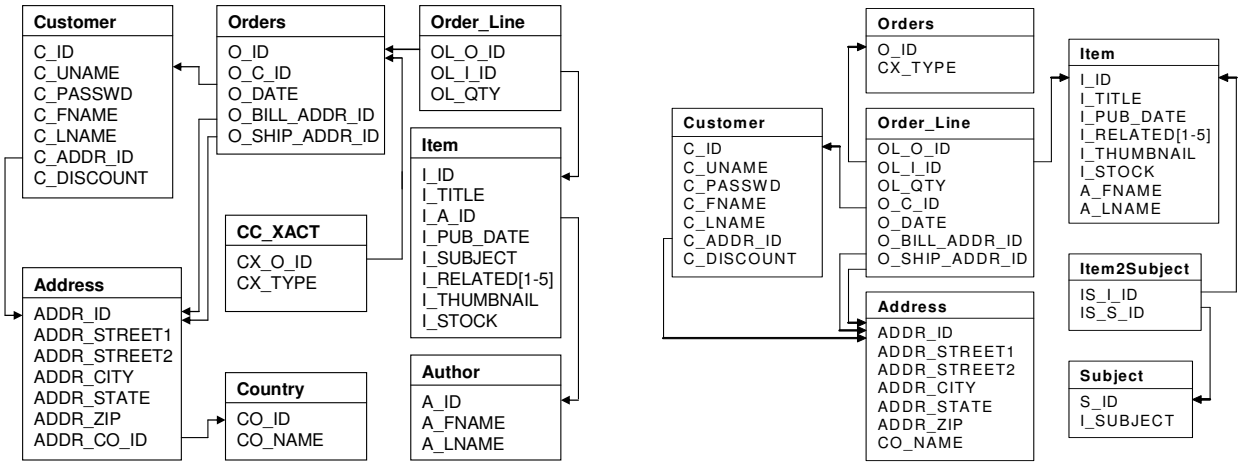

Fig. 1 Schemas of the bookstores example

each one of them. The scores are based on how well the features of the corresponding attributes match. The matching phase terminates by reporting the highest-score mapping (or mappings).

A. Feature extraction

During the feature extraction phase, the query log of each schema is scanned to collect both structure-level and element-level features. The structure-level features are used by a Structure-Level Usage-Based matcher (SLUB), whereas the element-level features are used by an Element-Level Usage-Based matcher (ELUB).

1) Structure-level features: The structure-level features capture the usage relationships between attributes of the same schema. An attribute A appearing in a query Q would normally have a role in defining Q's answer. A can be part of the answer (A occurs in the `select` clause), or it can have a filtering role (A appears in the `where` or `having` clauses), a grouping role (A occurs in the `group by` clause), or an ordering role (A occurs in the `order by` clause). Furthermore, if two attributes co-occur in the same query, they potentially have a *usage relationship* whose type is defined by the role of each of them in defining the query's answer. However, in certain complex queries, we may not consider them to be related, as will be described shortly.

Since we are considering four possible roles for the attributes, this results in 16 different types of possible relationships. The relationship types are referred to as *select-select*, *select-where*, *select-groupby*, *select-orderby*, *where-select*, etc. Note that with this naming convention, the term "where" is used to represent both the `where` and `having` clauses.

Based on the 16 types of relationships, we build 16 graphs $G_l(V,E_l)$, $l \in [1,16]$, where V represents the set of attributes and E_l represents the set of relationships of the l^{th} type between the attributes. The weights on the edges of the graph are proportional to the frequency of occurrence of their corresponding relationship in the query log. They are calculated as follows. All weights are initially zero. For each query in the query log, all the relationships between the query's attributes are identified, and the weights of their corresponding edges in the graphs are incremented by one. After scanning the whole query log, all the weights in the graphs are normalized by dividing each of them by the largest weight in its own graph. This final normalization step ensures that the graphs' weights are independent of the size of the query log.

If we denote the adjacency matrix of $G_l(V,E_l)$ by a_l, then we only need to maintain the lower triangular matrix of a_l, $l \in [1,16]$, since the upper triangular matrices can be induced. For example, the upper triangular matrix for the *select-select* graph is identical to the transpose of its own lower triangular matrix. Also, the upper triangular matrix for the *select-where* graph is identical to the transpose of the lower triangular matrix for the *where-select* graph and vice versa.

We now describe the three classes of queries we consider, and how to identify the usage relationships in each of them:

1. **SPJGO:** Single-block Select-Project-Join queries with optional grouping and/or ordering.
2. **SPJGO-UEI:** SPJGO queries with union, except and/or intersect.
3. **SPJGO-N:** SPJGO queries with nested subqueries.

For SPJGO queries, the identification process is straightforward. Each pair of attributes in the query has a relationship whose type depends on the two clauses where each of them occurs. For SPJGO-UEI queries, relationships are identified separately for each subquery because the attributes in one subquery do not affect the result of another subquery. Finally, for the SPJGO-N queries, relationships are first identified separately for each block, i.e., the blocks of the outer query and each inner subquery. Then, more relationships are identified between attributes occurring in different blocks. In particular, if the inner subquery is in the `where` or `having` clauses of the outer query, then its attributes occurring in the `select`, `where` and `having` clauses have a direct filtering role

in the result of the outer query, and therefore they are considered to be related to all the attributes occurring in the outer query as if they were occurring in its own `where` clause. This is unlike attributes occurring in the `group by` and `order by` clauses of the inner subquery, which do not have any direct role in the result of the outer query. If the inner subquery is in the `from` clause of the outer query, we follow the same strategy except that the attributes occurring in the `select` clause of the inner subquery are not considered to be related to those of the outer query.

Note that equivalent queries, having different query forms, may exist in the query logs of both schemas. If the extracted features from these forms are significantly different, the matching quality would be negatively affected. However, the following example shows how the fine-grained usage relationships we identify are mostly preserved across different query forms.

Example 4.1 This example shows three equivalent queries on the X-Books schema for finding book titles whose author's last name is 'Gray'.

Q1:
```
select I_TITLE from Item, Author
where I_A_ID=A_ID and A_LNAME='Gray'
```

Q2:
```
select I_TITLE from Item
where I_A_ID in(select A_ID from Author
                where A_LNAME='Gray')
```

Q3:
```
select I_TITLE
from Item, (select A_ID from Author
            where A_LNAME='Gray')
where I_A_ID=A_ID
```

TABLE I. CONTRIBUTION OF THE QUERIES OF EXAMPLE 4.1 TO THE SELECT-WHERE RELATIONSHIP TYPE

select-where	A_ID	A_LNAME	I_A_ID	I_TITLE
A_ID	-	*Q2,Q3,Q4*	-	-
A_LNAME	-	-	-	-
I_A_ID	-	-	-	-
I_TITLE	*Q1,Q2,Q3*	*Q1,Q2,Q3*	*Q1,Q2,Q3,Q4*	-

The identified relationships (or contributions) of the above three queries to the *select-where* relationship type are shown in Table I. The three contributions are almost identical except that the relationship between `A_ID` and `A_LNAME` is identified for *Q2* and *Q3* because it occurs in their subqueries, but not for *Q1*. While this discrepancy is not considered to be large, there are cases where the discrepancy between the contributions of two equivalent queries is larger. Consider the following two queries *Q4a* and *Q4b*, which use a parameter `@list` to together perform the same task as any of *Q1*, *Q2* or *Q3*.

Q4a:
```
@list = select A_ID from Author
        where A_LNAME='Gray'
```

Q4b:
```
select I_TITLE from Item
where I_A_ID in @list
```

The contribution of *Q4a* and *Q4b* (collectively referred to as *Q4*) is shown in Table I. It has three differences from that of *Q1* and two differences from that of *Q2* and *Q3*. One approach to reduce this discrepancy is to identify relationships at the *transaction-level*. That is to find queries belonging to the same transaction and relate their attributes together. In this example, *Q4a* and *Q4b* can be considered to be in the same transaction. We will then determine that `I_TITLE` is part of the final result of the whole transaction and `A_LNAME` has a filtering role in it, so the two attributes should have the *select-where* relationship. Currently, our implementation does not take transactions into account. □

2) Element-level features: Element-level features relate to the way each attribute is used in the query log regardless of the other attributes. For instance, some real-world schemas do not specify which attributes are the primary keys and which ones are the foreign keys. In such cases, the element-level feature extractor infers that an attribute is potentially a key attribute (primary or foreign) if it occurs in a join predicate. This inference becomes unnecessary if the same information could be directly extracted from the schema. Additionally, the extractor collects information about the usage of attributes with aggregate functions. We consider five such functions and divide them into three equivalence classes: {`count`}, {`min`, `max`} and {`sum`, `avg`}. For each attribute, we record whether it was used or not with each of these three classes of aggregate functions. A useful observation is that a potentially key attribute is highly unlikely to match an attribute that is aggregated with either `sum` or `avg`. It is also possible that the extractor infers information about the data types of the attributes based on the operations, functions, and literals they are used with, or compared to, in the query logs. However, since the data type information is typically available in the schemas, it would be pointless to infer it from the query log. Therefore in our experiments, we limited the element-level usage-based features to the key and aggregate information explained above.

B. Matching and scoring functions

The matching problem can be stated as follows. Given two schemas S_1 and S_2, where S_1 has n_1 attributes $\{x_1, x_2 \ldots x_{n_1}\}$ and S_2 has n_2 attributes $\{y_1, y_2 \ldots y_{n_2}\}$, and given the features extracted for each of them; find a mapping m^* from the attributes of S_1 to those of S_2, which gives the highest score for a particular scoring function. Any mapping m should provide the following information.

1. The number of matching attributes, denoted by k_m.
2. The identity of the k_m matching attributes from each schema, denoted by $\{x_{p_1}, x_{p_2} \ldots x_{p_{k_m}}\}$ for S_1 and $\{y_{p_1}, y_{p_2} \ldots y_{p_{k_m}}\}$ for S_2.
3. The actual matches connecting each of the k_m attributes in S_1 to only one of the k_m attributes of S_2, i.e., $m(x_{p_i})=y_{q_j}$, or equivalently $m(p_i)=q_j$.

The mapping is called one-to-one mapping if $n_1=n_2=k_m$. It is called onto mapping if $n_1=k_m$ and $n_2 \geq n_1$. Finally, it is called partial mapping if $k_m < \min(n_1,n_2)$, which is the most general case. The size of the space of all possible mappings in the most general case is given by $\sum_{k_m=0}^{\min(n_1,n_2)} C_{k_m}^{n_1} C_{k_m}^{n_2} k_m!$.

The challenge of formulating an effective scoring function to compare between different mappings has been addressed in [11]. The key idea is to match two tables by building a complete graph for the attributes of each table, where the weight on each edge equals the mutual information between its end nodes. The table matching problem then reduces to graph matching. The authors proposed two possible scoring functions to measure the distance between a pair of graphs given a certain mapping. In particular, they classified the scoring functions into *monotonic* and *non-monotonic* in k_m, and they proposed an example for each class. Monotonic functions are not suitable for automatically estimating the correct number of mappings k_{m^*}. If k_{m^*} is not known and a monotonic function is used, the matching algorithm will conclude that either no attributes match (k_{m^*}=0), or that the maximum number of attributes match (k_{m^*}=min(n_1,n_2)), depending on the direction of monotonicity. Clearly, this is not the desired behavior for the matching algorithm. Thus, if the matching algorithm uses a monotonic scoring function, it has to be provided with a user estimate $\hat{k}_{m^*}$ for k_{m^*}. This way, it will only consider mappings with $k_m = \hat{k}_{m^*}$. However, if it uses a non-monotonic function, it might be able to automatically estimate the correct value of k_{m^*}, as we will explain shortly.

We adapt the scoring functions proposed in [11] to fit our context. Firstly, for the structure-level features, we have 16 pairs of graphs rather than one pair, so we assign a weight w_l for the graphs representing the usage relationship of the l^{th} type, where $\sum_{l=1}^{16} w_l = 1$. Secondly, the scoring functions are adjusted to fall in the range [0,1] to make sure they are comparable to each other when more than one matcher is combined. Thirdly, for the same reason, the goal should be to *maximize* all the scoring functions, rather than a mixture of maximize and minimize.

We will now present the scoring functions used by the SLUB and ELUB matchers. We denote the adjacency matrices of the structure-level feature graphs of S_1 and S_2 as a_l and b_l respectively, $l \in [1,16]$.

Structure-level monotonic scoring function

$$f_{su}^{m}(m) = 1 - \sum_{l=1}^{16} \frac{w_l}{r_l} \sqrt{\sum_{i=1}^{k_m}\sum_{j=1}^{k_m} (a_l[p_i][p_j] - b_l[m(p_i)][m(p_j)])^2} \quad (1)$$

This function uses the Euclidean distance to measure the dissimilarity between the 16 feature graphs of S_1 and their corresponding graphs of S_2 given a certain mapping m. Obviously, as k_m increases, the dissimilarity can potentially increase, and consequently the score decreases. Therefore, when the estimate $\hat{k}_{m^*}$ is given to the matcher, all occurrences of k_m in (1) should be replaced by $\hat{k}_{m^*}$. The variable r_l is an upper bound to the value of the square root, which guarantees that the score does not exceed 1. Since the elements of a_l and b_l are normalized, i.e., they are less than or equal to 1, the upper limit for the summation under the square root is k_m^2. Since a_l and b_l represent sparse graphs, a tighter upper bound for that summation can be $(t_{l_1}+t_{l_2})^2$, where t_{l_1} and t_{l_2} are the sums of nonzero elements in a_l and b_l respectively. Therefore r_l is given by $\min(k_m, t_{l_1} + t_{l_2})$.

Structure-level non-monotonic scoring function

$$f_{su}^{n}(m) = \sum_{l=1}^{16} \frac{w_l}{r_l} \sum_{i=1}^{k_m}\sum_{j=1}^{k_m} \Psi_{lij}(m) \quad (2),$$

where

$$\Psi_{lij}(m) = \begin{cases} 0, & a_l[p_i][p_j] = 0 \text{ and } b_l[m(p_i)][m(p_j)] = 0 \\ \left(1 - \alpha_{lij} \left| a_l[p_i][p_j] - b_l[m(p_i)][m(p_j)] \right|\right), & \text{otherwise} \end{cases} \quad (3)$$

This function uses the absolute difference between the corresponding elements in a_l and b_l given a certain mapping m to measure the dissimilarity between the graphs. When both elements are zero (no edge in both graphs), nothing is contributed to the score. However, if at least one of them is non-zero, then the mapping is either rewarded or penalized according to the control variable α_{lij}. To explain how α_{lij} is calculated, let β_{lij} be the value multiplied by α_{lij} in (3), which represents the dissimilarity between an element in a_l and its corresponding element in b_l. If we consider the distribution of β_{lij} across all possible mappings, we may expect that the value of β_{lij} for m^* (call it β^*_{lij}) is among the smallest values in this distribution (since m^* is expected to minimize the dissimilarity between a_l and b_l). Therefore if β^g_{lij} is the value of the g-quantile of β_{lij}, for some small value g (e.g., $g \in [0.1,0.3]$), then β^*_{lij} should still be smaller than β^g_{lij}. Consequently, if we set α_{lij} to $1/\beta^g_{lij}$, the value of $\psi_{lij}(m^*)$ will be positive for most combinations of l, i and j, and therefore m^* is expected to be rewarded the most among other mappings. Additionally, if any mapping m has a value for k_m greater than the correct value, it will have to make wrong matches between elements of a_l and b_l whose β_{lij} is expected to be greater than β^g_{lij} and therefore the value of ψ_{lij} will be negative and m will be penalized. Similarly, if a mapping m has a value for k_m less than the correct value, it will not be as rewarded as m^*. This way, a matcher using a non-monotonic scoring function can estimate the correct k_{m^*}.

The drawback of this technique is that when the graphs being matched are not very close to each other, the value of β^*_{lij} can be greater than β^g_{lij} in many cases, which results in many false positives and false negatives in the produced matches. In [11], the authors assumed that the value corresponding to β_{lij} in their problem setting, which they called the *normal distance*, is uniformly distributed. Therefore, they used the average of β_{lij} to calculate their control variable α. However, using the average can result in poor results if the distribution of β_{lij} is skewed. To address this issue, we use quantiles instead of average. Furthermore, they used a single value α irrespective of which element in the graph's adjacency matrix is being considered, while we calculate a different value α_{lij} for each graph type and combination of i and j, which gives a higher accuracy.

The role of r_l in (2) is similar to its role in (1). It guarantees that the score does not exceed 1, being an upper limit for the innermost two summations in (2). Observing from (3) that the maximum value for $\psi_{lij}(m)$ is 1, then based on an analysis

similar to that used for (1), r_l is calculated as $\min([\min(n_1,n_2)]^2, c_{l_1}+c_{l_2})$, where c_{l_1} and c_{l_2} are the number of nonzero elements in a_l and b_l respectively. Note that $\min(n_1,n_2)$ is used instead of k_m so that r_l does not change as the mapping changes.

Before considering the scoring functions for the ELUB matcher, we first show how we calculate a score $Score_{eu}$ for a matching pair of attributes from S_1 and S_2 respectively, based on their element-level features. Table II shows a simple feature compatibility matrix, which contrasts the features of the first attribute (rows) to those of the second one (columns). $Score_{eu}$ is initially zero. If both attributes have the same feature, it is incremented by 1/4. If they have contradicting features, it is decremented by 1. $Score_{eu}$ never exceeds 1.

TABLE II. ELEMENT-LEVEL USAGE-BASED FEATURE COMPATIBILITY MATRIX

	key	`count`	`min, max`	`sum, avg`
key	1/4	0	0	-1
`count`	0	1/4	0	0
`min, max`	0	0	1/4	0
`sum, avg`	-1	0	0	1/4

We now consider the monotonic and non-monotonic scoring functions used by the ELUB matcher.

Element-level monotonic scoring function

$$f_{eu}^{m}(m) = \max\left(0, 1 - \frac{1}{k_m}\sum_{i=1}^{k_m}\left(1 - Score_{eu}(p_i, m(p_i))\right)\right) \qquad (4)$$

This function calculates the dissimilarity between pairs of matching attributes in m using $Score_{eu}$. It ensures that the score is non-negative, and does not exceed 1.

Element-level non-monotonic scoring function

$$f_{eu}^{n}(m) = \max\left(0, \frac{1}{k_m}\sum_{i=1}^{k_m}(1 - \alpha_i(1 - Score_{eu}(p_i, m(p_i))))\right) \qquad (5)$$

This function uses the control variable α_i in the same way as in (3), where β_i, here $(1\text{-}Score_{eu}(p_i,m(p_i)))$, represents the dissimilarity between a pair of attributes. It also ensures that the score is non-negative and does not exceed 1.

To be resilient to the differences in schema layouts, the matching phase, which uses the scoring function we have discussed so far, is performed in two steps. In the first step, only *non-foreign-key* attributes are matched, while foreign key attributes are totally ignored. In the second step, foreign keys are matched, while ensuring that the matches obtained in the first step are left unchanged. The details of how the two steps are implemented will be explained in Section V. To see the intuition for such two steps, consider a group of non-foreign-key attributes that are frequently queried together. Moreover, they exist in the same table in S_1, but in different tables in S_2. The two-step method, we described, will ensure that the matching of these attributes will not rely on their relationship with any foreign keys, which may or may not exist in the queries depending on the schema layout. For example, attributes `I_TITLE` and `I_SUBJECT` exist in `Item` table in the X-Books schema, but in `Item` and `Subject` tables respectively in the Y-Books schema. In our implementation, we use a conservative approach, where the attributes ignored in the first step are those which *only* appear in join predicates, rather than any general foreign key.

V. IMPLEMENTATION

For feature extraction, we used a free SQL parser [19]. Any other parser can be used as well. For the matching phase, since the space of all possible mappings is very large and an exhaustive search is too expensive, we implemented a genetic algorithm to find the optimal mapping, as an example of heuristic optimization methods. Other methods are also applicable. This section gives the details of the search algorithm and how several matchers can be combined together including SLUB, ELUB, and non-usage-based matchers.

A. Genetic search algorithm

Genetic algorithms represent an approximate method for solving optimization problems [14]. In our search algorithm, each chromosome (or candidate solution) represents a possible mapping. Hence, all mappings should have a uniform representation regardless of the number of matches or the identity of matched attributes. For this purpose, we introduce the notion of *dummy attributes* that are added to each of S_1 and S_2, such that they both end up with the same number of attributes n'. Thus, each mapping will contain precisely n' matches connecting all attributes in both schemas. However, a match is considered a *real match* only if it does not involve dummy attributes. A mapping can now be uniformly represented as an ordering of the n' attributes of S_2 in the form of $(y_{p_1}, y_{p_2}\ldots y_{p_{n'}})$, where the real matches are of the form $m(i)=p_i$, $i\in[1,n']$ and both x_i and y_{p_i} are non-dummy attributes.

The number of dummy attributes to be added depends on the scoring function. For a monotonic function, where the estimate $\hat{k}_{m^*}$ is given to the algorithm, then exactly $n_2\text{-}\hat{k}_{m^*}$ and $n_1\text{-}\hat{k}_{m^*}$ dummy attributes are added to S_1 and S_2 respectively, leading to $n'=n_1+n_2\text{-}\hat{k}_{m^*}$. This guarantees that the generated mappings will contain at least $\hat{k}_{m^*}$ real matches (in case all dummy attributes match to non-dummy attributes). However, since the monotonic functions presented in Section IV-B are all decreasing in k_m, m^* will have exactly $\hat{k}_{m^*}$ real matches. For a non-monotonic scoring function, then $\hat{k}_{m^*}$ is considered to be zero, i.e., n_2 and n_1 dummy attributes are added to S_1 and S_2 respectively. Thus, the generated mappings may contain any number of real matches in $[0, \min(n_1,n_2)]$, depending on how many dummy attributes match to non-dummy attributes. Since the scoring function is non-monotonic in k_m, m^* may also contain any number of real matches within the same range.

In the scoring function, used as the fitness function of the genetic algorithm, only the real matches in the mapping are

taken into account. Constraints are specified such that only mappings containing some given matches are considered in the search space. We use these constraints in the second step of our matching phase, where the matches obtained for the non-foreign-key attributes in the first step have to remain fixed. The constraints are also used when the user has prior information about some correct matches. We specify constraints by dividing S_1 attributes into *constrained* and *non-constrained* attributes. Each constrained S_1 attribute can only match a certain S_2 attribute, while unconstrained S_1 attributes are permitted to match any of the remaining S_2 attributes. The stopping criterion we set is that the highest score encountered remains unchanged for some fixed number of iterations, *N_Iterations*.

The backbone of our search algorithm is similar to any genetic algorithm [14]. Its details are not shown here for the lack of space. However, the specific algorithms used to generate new mappings, make crossovers and make mutations, all while ensuring that the generated mappings satisfy the constraints, need more elaboration. To form the initial population and introduce new immigrants, new mappings are generated. The higher the score of these mappings, the faster the search algorithm can converge. Algorithm 1 is designed to fulfill this purpose. It is an iterative algorithm which starts by matching a random S_1 attribute to a random S_2 attribute, to which it can match. Then, at each iteration, it picks an unmatched S_1 attribute that is *related* to those attributes previously matched in S_1 and matches it to another unmatched S_2 attribute that is also *related* to the previously-matched attributes in S_2. The pair of attributes are selected such that (a) they are permitted to match and (b) their two *relations* with their predecessors (previously-matched attributes) are *closest* to each other compared to the corresponding relations of any other pair of unmatched attributes from S_1 and S_2 respectively. This criterion ensures the relatively high score desired for the generated mapping. If no such pair is found, then any two random unmatched attributes are selected, one from each schema, and matched together, if they are permitted to match. This process continues until all the n' attributes have been matched. Note that if the structure-level features are not involved in the matching phase (i.e., no graphs are involved), Algorithm 1 assumes that all the attributes are *unrelated*.

Because of the way the number of dummy attributes is selected, all the mappings generated by Algorithm 1 will have $k_m \geq \hat{k}_{m^*}$. When monotonic scoring functions are used, although the best mapping should have $k_m = \hat{k}_{m^*}$, mappings of intermediate generations having $k_m > \hat{k}_{m^*}$ are kept in the population if they have relatively high scores because they can later result in finding better mappings having $k_m = \hat{k}_{m^*}$ (i.e., through mutations and crossovers).

Algorithm 2 is used to generate two child mappings, c_1 and c_2, from two parent mappings, m_1 and m_2, after crossing them over, while algorithm 3 generates a child mapping, c, from a parent mapping, m, after mutating it.

Algorithm 1: Generate_New_Mapping (m)

M_i: set of matched S_i attributes, $i \in [1,2]$
L_i: set of S_2 attributes permitted to match x_i, $i \in [1,n']$
$R_{1,i}$: sum of edge weights from x_i to all $x_j \in M_1$ averaged over the 16 feature graphs of S_1, $i,j \in [1,n']$, $i \neq j$
$R_{2,i}$: sum of edge weights from y_i to all $y_j \in M_2$ averaged over the 16 feature graphs of S_2, $i,j \in [1,n']$, $i \neq j$

1- $M_1=\{\}$; $M_2=\{\}$;
2- **for each** iteration t
3- **if** $|M_1|=n'$
4- **return** m;
5- Find an unmatched S_1 attribute x_{i_t} and an unmatched S_2 attribute y_{j_t} such that $y_{j_t} \in L_{i_t}$, $R_{1,i_t}>0$, $R_{2,j_t}>0$, $|R_{1,i_t}-R_{2,j_t}| \leq |R_{1,u}-R_{2,v}|$, $u \neq i_t$, $v \neq j_t$, $x_u \notin M_1$, $y_v \notin M_2$;
6- **if** such pair (x_{i_t}, y_{j_t}) does not exist
7- Let x_{i_t} be any random unmatched S_1 attribute, y_{j_t} be any random unmatched S_2 attribute, $y_{j_t} \in L_{i_t}$;
8- Let $m(i_t)=j_t$;
9- Add x_{i_t} to M_1;
10- Add y_{j_t} to M_2;
11- Remove y_{j_t} from L_u, $u \neq i_t$;

Algorithm 2: Make_Crossover (m_1, m_2)

c_i : the i^{th} child mapping to be generated, $i \in [1,2]$

1- Copy m_1 into c_1;
2- Randomly divide c_1 into two parts;
3- Keep the first part of c_1 unchanged;
4- For the second part, keep the matches for the constrained S_1 attributes unchanged;
5- Reorder the matching S_2 attributes for the unconstrained S_1 attributes in the second part of c_1 to follow the ordering of m_2;
6- Generate c_2 in the same way as c_1 after switching the roles of m_1 and m_2;
7- **return** $\{c_1, c_2\}$;

Algorithm 3: Make_Mutation (m)

c : the child mapping to be generated

1- Copy m into c;
2- Pick two random unconstrained S_1 attributes x_i and x_j;
3- Swap $c(x_i)$ and $c(x_j)$;
4- **return** c;

5.2 Integration with other matching techniques

We allow the combination of any number of matching techniques, whenever applicable, to improve the quality of the generated mappings. In particular, we follow an aggregation approach similar to the COMA framework [6], where an overall score is used to capture the scores of each individual matcher. In this scenario, each candidate mapping, generated by the genetic algorithm, is passed to the individual matchers, which return their individual scores. The genetic algorithm then calculates a weighted average of the individual scores and uses it as the fitness value. The overall scoring function can either be monotonic or non-monotonic, depending on whether the aggregated individual scoring functions are themselves monotonic or non-monotonic respectively.

As an example of a non-usage-based matcher, we implemented a data type matcher to assess the value of combining it with usage-based matchers. The data type

information is typically available with each schema or can be inferred from the query log to a certain degree of accuracy (See Section IV-A-2). Moreover, the data types used by different DBMSs are usually very similar. Therefore, in practice, we would always be able to combine this matcher with our usage-based matchers. The data type matcher (DT) considers only three classes of data types: numeric, string and datetime. For numeric data types it considers the scale and precision properties, and for the string data types it considers the length property. It uses a data type compatibility matrix to calculate a score $Score_{dt}$ for each match between two attributes from S_1 and S_2 respectively.

The monotonic and non-monotonic scoring functions used for the DT matcher are exactly similar to (4) & (5) respectively in Section IV-B, except that $Score_{dt}$ is used instead of $Score_{eu}$. The monotonic scoring function is:

$$f_{dt}^{m}(m) = \max\left(0, 1 - \frac{1}{k_m}\sum_{i=1}^{k_m}\left(1 - Score_{dt}(p_i, m(p_i))\right)\right) \quad (6),$$

while the non-monotonic scoring function is given by

$$f_{dt}^{n}(m) = \max\left(0, \frac{1}{k_m}\sum_{i=1}^{k_m}\left(1 - \alpha_i\left(1 - Score_{dt}(p_i, m(p_i))\right)\right)\right) \quad (7)$$

VI. Experiments and results

A. Experimental setup

We build upon the bookstores example described in Section III. We used the Wisconsin implementation of the TPC-W benchmark [17] to which we added a query logger component. After each TPC-W run, two query logs are generated for the X-Books and the Y-Books schemas respectively. The attributes which do not appear in the query logs were not included in these two schemas. Normally, all schema attributes would be queried, but this is not the case for TPC-W since it only focuses on certain aspects of the bookstore business. For example since the integration part with the banking system is not considered, some attributes of the credit card transactions do not appear in the query log.

The TPC-W benchmark specifies three types of workload: browsing mix (B), shopping mix (S), and ordering mix (O). Read-only web interactions constitute 95%, 80%, and 50% in browsing, shopping, and ordering mixes respectively, while the remaining percentage is for read-write web interactions.

We generated query logs for both schemas corresponding to the three workload mixes. In each run, 30 emulated browsers submit requests to the bookstore application simultaneously for about 3 hours. The size of the generated query logs range from 10,120 to 17,819 queries. These sizes are large enough to guarantee that the logs are representative of the workload. The logged queries are mostly SPJGO queries with a few SPJGO-N queries. In the TPC-W implementation, two additional tables were used to capture the shopping cart information. We added them to the schemas of X-Books and Y-Books, such that the total number of attributes in each schema became 44 and 46 respectively. The correct number of matching attributes is 41.

In the experiments, we use the average F-measure (*f*) metric to measure the average quality for all the mappings generated with the same highest score under varying conditions. We study the effects of combining several matchers, changing the quality of attribute names (when schema-based techniques are combined with usage-based techniques), changing the types of usage relationships used, changing the parameter $\hat{k}_{m^*}$ (when monotonic scoring functions are used), changing the parameter *g* (when non-monotonic scoring functions are used), and using query logs of the same workload type versus the two most different types (BB and BO). We compare our technique to a hypothetical optimal matcher assumed to always return the correct mapping as one of its highest-score mappings. Note that because some attributes may be indistinguishable to the matcher (e.g. they have the same data type when only a data type matcher is used), some incorrect mappings may have the same score as the correct mapping. Thus, *f* values for that optimal matcher are not necessarily 1.

For the comparison with established techniques, we use the Similarity Flooding (SF) algorithm [13], which is available as open source. We could not compare our technique to that of [11] (being also independent from the attribute name information) because it only considered matching individual tables rather than complete schemas as in our case.

In the graphs, we use "m" and "n" to denote monotonic and non-monotonic scoring functions respectively. The weights for relationship types, w_l, were set to 1/16, $l \in [1,16]$. The number of iterations, *N_Iterations*, for which the highest-score mapping should remain unchanged before the genetic algorithm stops (stopping criterion), was set to 500. The initial population size was set to 50. Throughout the experiments, the average total number of iterations required was 2305.

B. Effect of combining different matchers

In this experiment, we show the impact of combining three basic matchers, namely SLUB, DT and ELUB, according to eight different combinations of weights, including the cases where they are used individually, in pairs, or all three together. For monotonic scoring functions, a correct estimate $\hat{k}_{m^*}$ for k_{m^*} is used, and for non-monotonic scoring functions, the parameter *g* is set to 0.2 (Recall from Section IV-B that *g* controls how mappings are rewarded/penalized).

Figures 2 and 3 show the *f* values when the eight combinations of matchers are used. The best results are obtained when SLUB is combined with DT, where the *f* value reaches up to 0.8. This figure is very close to what the optimal matcher could achieve: 0.83. When used individually, SLUB achieves higher accuracy than DT and ELUB, as its *f* value reaches 0.7 when similar query logs are used (B and B, or simply BB) and 0.5 when the most different query logs are used (B and O, or simply BO). We also observe that using a non-monotonic scoring function makes the matcher more sensitive to the discrepancies between the two query logs compared to the case when a monotonic scoring function is used. This was expected because of the drawback of non-monotonic scoring functions, discussed in Section IV-B.

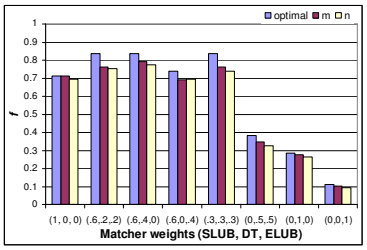

Fig. 2 Effect of combining matchers (BB query logs)

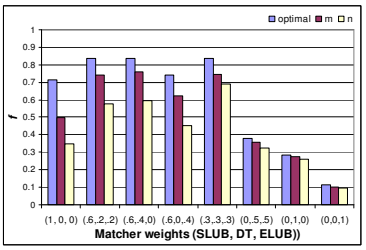

Fig. 3 Effect of combining matchers (BO query logs)

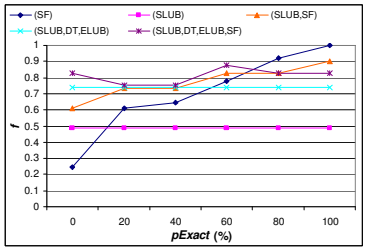

Fig. 4 Effect of varying *pExact*

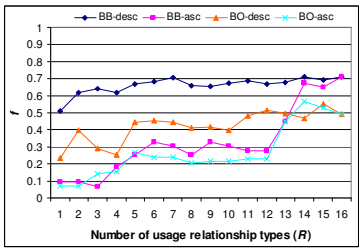

Fig. 5 Effect of usage relationship types

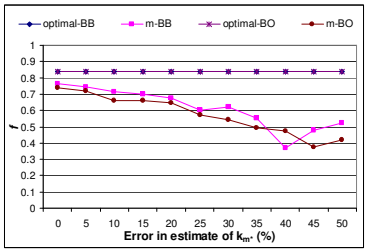

Fig. 6 Effect of varying $\hat{k}_{m^*}$

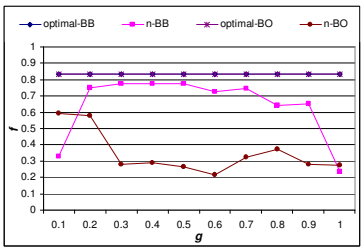

Fig. 7 Effect of varying *g*

ELUB proved to be more useful with the BO query logs as it improved the accuracy when combined with each of DT and SLUB separately (e.g. (1,0,0) vs (.6,0,.4)). However, when combined with them together, the accuracy was not significantly improved (e.g. (.6,.4.0) vs (.3,.3,.3)). In general, the value ELUB can add in a combination of matchers depends on the discriminative power of the other matchers and the richness of the query logs in terms of element-level features. When used separately, ELUB gave almost the same accuracy both with BB and BO query logs. The reason is that the same element-level features are preserved in both types of query logs.

C. Effect of the quality of attribute names

In this experiment, we compare the matching quality of the usage-based approach to that of SF. We study the impact of the attribute naming quality when the following combinations of matchers are used: SF, SLUB, (SLUB, SF), (SLUB, DT, ELUB), and (SLUB, DT, ELUB, SF). When used in combination, matchers are assigned equal weights. To vary the quality of attribute names, we considered that a percentage of the attribute names of the target schema are random strings; i.e., they cannot be matched to the attributes of the source schema, while the remaining percentage (*pExact*) represent exact matches to those of the source schema.

Fig. 4 shows the *f* values when *pExact* varies from 0% to 100%. The monotonic scoring function is used and the query logs of both schemas are considered to be different (BO). The *f* value increases as *pExact* increases only when SF is used. Otherwise, it remains constant, since SF is the only matcher that depends on the attribute names. The *f* value for SF is lower than those of SLUB and (SLUB, DT, ELUB) when *pExact* is less than 20% and 60% respectively. This reflects the superiority of the usage-based technique when the attribute names are unreliable: the case where the usage-based technique is needed the most. We also note that if similar query logs were used (BB), the previous two figures would have been 60% and 60% respectively, since SLUB performs better with BB query logs (See Fig. 2). Furthermore, as long as *pExact* is less than 80%, the combination of all matchers provides the highest accuracy compared to any single matcher or subset of matchers. This shows how schema-based and usage-based matchers can reinforce each other when used in combination. When *pExact* is 80% or greater (almost the ideal case for SF), SF performs the best because the attribute name information becomes highly reliable, making the combination with other matchers counter-productive. We finally note that SF manages to find some correct matches even when all attribute names do not match (*pExact*=0%) because SF does not solely depend on attribute names, but it also exploits the structural similarities in the two schemas.

D. Effect of usage relationship types

In this experiment, we study the impact of using a subset of the usage relationship types as opposed to using all of them. We ranked the relationship types based on the connectivity of their source and target graphs by counting the number of edges in both graphs for each relationship type. This ranking (Table III) gives an indication of the discriminative power of the different types of relationships. Fig. 5 shows the *f* values when SLUB is used with only the top *R* relationship types, where *R* is varied from 1 to 16. The relationship types are always given equal weights.

We use the monotonic scoring function and consider both BB and BO query logs. The ranking of relationship types is considered both when it is descending in the total number of edges (as in Table III) and when it is ascending in that number. As expected, the figure shows that, in general, the more relationship types are used, the more effective the matcher is. This is because each type of relationships may

provide additional evidences as to which attributes match. Also, when the descending ranking is used, *f* converges to its highest values much faster than the ascending case, as the most discriminative types of relationships are considered first in the case of the descending ranking.

TABLE III. RANKING OF USAGE RELATIONSHIPS

Usage relationship type	Total no. of edges	Rank
select-select	339	1
where-select	91	2
select-where	91	3
where-where	62	4
orderby-select	18	5
select-orderby	18	6
groupby-groupby	17	7
groupby-select	17	8
select-groupby	17	9
orderby-where	10	10
groupby-where	10	11
where-orderby	10	12
where-groupby	10	13
orderby-orderby	9	14
orderby-groupby	5	15
groupby-orderby	5	16

E. Effect of the parameter $\hat{k}_{m^*}$

Fig. 6 shows the *f* values when the matcher uses a monotonic scoring function and the error in the estimate $\hat{k}_{m^*}$ varies from 0% to 50%. The matcher is a combination of SLUB, DT and ELUB with weights 0.6, 0.2 and 0.2 respectively. Interestingly, the accuracy does not sharply deteriorate as the error in $\hat{k}_{m^*}$ increases. For instance, the *f* value for a 25% error is almost 0.6, which is very reasonable, considering that 25% error means that the matcher is explicitly instructed to return 25% fewer matches than the correct number of matches. Fig. 5 also confirms that a matcher using a monotonic function is not very sensitive to discrepancies in the query logs, since the BB and BO curves are not far from each other. For the optimal matcher, the BB and BO curves coincide because, in our experiment setting, the B and O query logs, are similar in terms of which attributes are indistinguishable to the matcher and which attributes are not.

F. Effect of the parameter g

Fig. 7 shows the effect of varying the parameter *g* on the *f* values when the matcher uses a non-monotonic function. The specific matcher used is also a combination of SLUB, DT and ELUB with weights 0.6, 0.2 and 0.2 respectively. As expected, the matcher performs better with BB query logs compared to BO query logs. In the former case, the *f* values remain at their peak for a big range of *g* (0.2-0.7), while in the later case they peak when *g* is less than 0.3. Generally, a small *g* can result in penalizing correct mappings, while a large *g* can result in rewarding erroneous mappings, which leads in both cases to a lower accuracy.

VII. CONCLUSIONS

We introduced a new class of techniques, usage-based schema matching, where the usage information in the query logs is used to find correspondences between the attributes of two schemas. Our experimental study demonstrated the effectiveness of the proposed technique and the value of combining it with other matchers, including the Similarity Flooding algorithm. The results showed that when the attribute name information is of low quality, usage-based techniques outperform schema-based techniques. However, when combined together, the matching quality improves on average compared to using either technique in isolation.

While this paper was focusing on relational schemas and SQL, a natural next step would be to investigate the applicability of our approach in an XML context with a query language like XQuery. Furthermore, building a repository of query logs obtained from real world systems can be very useful in studying the effectiveness of any technique that relies on query log analysis, such as ours. Finally, it would be interesting to study the possibility of using the query logs to discover Global-As-View (GAV) or Local-As-View (LAV) mappings, where a table in one schema is expressed as a view over the other schema.

REFERENCES

[1] Y. An et al. A semantic approach to discovering schema mapping expressions. In *ICDE*, 2007.
[2] P. Bohannon, E. Elnahrawy, W. Fan, and M. Flaster. Putting context into schema matching. In *VLDB*, 2006.
[3] N. Bruno and S. Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *SIGMOD* 2005.
[4] B. Dageville et al. Automatic SQL tuning in Oracle 10g. In *VLDB*, 2004.
[5] R. Dhamankar et al. iMAP: discovering complex semantic matches between database schemas. In *SIGMOD*, 2004.
[6] H. Do and E. Rahm. COMA - A system for flexible combination of schema matching approaches. In *VLDB*, 2002.
[7] A. Doan et al. Reconciling schemas of disparate data sources: A machine learning approach. In *SIGMOD*, 2001.
[8] L. Haas et al. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, 2005.
[9] B. He and K. Chang. Statistical schema matching across web query interfaces. In *SIGMOD*, 2003.
[10] J. Kang and J. Naughton. On schema matching with opaque column names and data values. In *SIGMOD*, 2003.
[11] J. Madhavan, P. Bernstein, A. Doan, and A. Halevy. Corpus-based schema matching. In *ICDE*, 2005.
[12] J. Madhavan, P. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *VLDB*, 2001.
[13] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: a versatile graph matching algorithm. In *ICDE*, 2002.
[14] M. Mitchell, *Introduction to Genetic Algorithms*, MIT Press, Cambridge, MA, 1996.
[15] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4), 2001.
[16] The TPC-W benchmark. http://www.tpc.org/tpcw
[17] http://www.ece.wisc.edu/~pharm/tpcw.shtml
[18] R. Warren and F. Tompa. Multi-column substring matching for database schema translation. In *VLDB*, 2006.
[19] http://www.experlog.com/gibello/zql/