Tuning Schema Matching Software using Synthetic Scenarios

Mayssam Sayyadian, Yoonkyong Lee, AnHai Doan
University of Illinois
{sayyadia, ylee11, anhai}@cs.uiuc.edu

Arnon S. Rosenthal
The MITRE Corporation
arnie@mitre.org

Abstract

Most recent schema matching systems assemble *multiple components*, each employing a particular matching technique. The domain user must then *tune* the system: select the right component to be executed and correctly adjust their numerous "knobs" (e.g., thresholds, formula coefficients). Tuning is skill- and time-intensive, but (as we show) without it the matching accuracy is significantly inferior.

We describe eTuner, an approach to *automatically* tune schema matching systems. Given a schema S, we match S against synthetic schemas, for which the ground truth mapping is known, and find a tuning that demonstrably improves the performance of matching S against real schemas. To efficiently search the huge space of tuning configurations, eTuner works sequentially, starting with tuning the lowest level components. To increase the applicability of eTuner, we develop methods to tune a broad range of matching components. While the tuning process is completely automatic, eTuner can also exploit user assistance (whenever available) to further improve the tuning quality. We employed eTuner to tune four recently developed matching systems on several real-world domains. eTuner produced tuned matching systems that achieve higher accuracy than using the systems with currently possible tuning methods, at virtually no cost to the domain user.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

1 Introduction

Schema matching finds semantic correspondences called *matches* between the schemas of disparate data sources. Example matches include "location = address" and "name = concat(first-name,last-name)". Application that manipulates data across different schemas often must establish such semantic matches, to ensure interoperability. Prime examples of such applications arise in numerous contexts, including data warehousing, scientific collaboration, e-commerce, bioinformatics, and data integration on the World-Wide Web [37].

Manually finding the matches is labor intensive, thus numerous automatic matching techniques have been developed (see [37, 35, 5, 21] for recent surveys). Each individual matching technique has its own strength and weakness [37]. Hence, increasingly, matching tools are being assembled from *multiple components*, each employing a particular matching technique [37, 21].

The multi-component nature is powerful in that it makes matching systems highly extensible and (with sufficient skills) customizable to a particular application domain [8, 38]. However, it places a serious burden on the domain user: *given a particular matching situation, how to select the right matching components to execute, and how to adjust the multiple "knobs" (e.g., threshold, coefficients, weights, etc.) of the components*? Without tuning, matching systems often fail to exploit domain characteristics, and produces inferior accuracy. Indeed, in Section 6 we show that the untuned versions of several off-the-shelf matching systems achieve only 14-62% accuracy (in F-1 score) on four real-world domains.

High matching accuracy is crucial in many applications, so tuning will be quite valuable. To see this, consider two scenarios. First, consider data exchange between automated applications, e.g., in a supply chain. People do check correctness of each data value transmitted, so erroneous matches will cause serious real world mistakes. Thus, when building such applications, people check and edit output matches of the automated system, or use a system such as Clio [41]

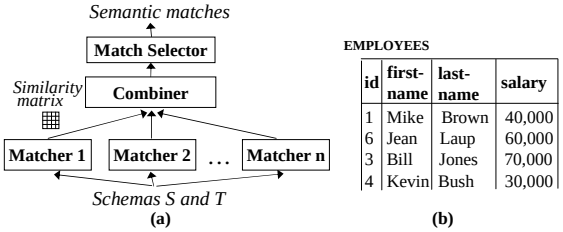


Figure 1: An example of multi-component matching systems.

to elaborate matches into semantic mappings (e.g., in form of SQL queries [41] which specify exact relationships between elements of different schemas) [39]. Here, improving the accuracy of the automated match phase can significantly reduce peoples' workload, and also the likelihood that they overlook or introduce mistakes.

Second, large-scale data integration, peer-to-peer, and distributed IR systems (e.g., on the Web [1]) often involve tens or hundreds of sources, thus thousands or tens of thousands of semantic matches across the sources or metadata tags. At this scale, humans cannot review all semantic matches associated with all sources. Instead, the systems are likely to employ the automated match results, and return the apparent best answers for human review. Here, each improvement in matching accuracy directly improves the result the user receives.

While valuable, tuning is also very difficult, due to the large number of knobs involved, the wide variety of matching techniques employed (e.g., database, machine learning, IR, information theory, etc.), and the complex interaction among the components. Writing a "user manual" for tuning seems nearly impossible. For example, tuning a matching component that employs learning techniques often involves selecting the right set of features [16] (Section 6.2), a task that is difficult even for learning experts [16]. Further, since we rarely know the ground truth for matches, it is not clear how to compare the quality of knob configurations.

For all above reasons, matching systems are still tuned manually, largely by trial and error – a time consuming, frustrating, and error prone process. Consequently, developing efficient techniques for tuning seems an excellent way to improve matching systems to a point where they are attractive in practice.

In this paper we describe eTuner, an approach to automatically tune schema matching systems. In developing eTuner, we address the following challenges:

Define the Tuning Problem: Our first challenge is to develop an appropriate model for matching systems, over which we can define a tuning problem. To this end, we view a matching system as a combination of matching components. Figure 1.a shows a matching system which has $(n + 2)$ components: n matchers, one combiner, and one selector (Section 3 describes these components in detail).

To the user (and eTuner) the components are *black-boxes*, with "exposed knobs" whose values can be adjusted. For example, a knob allows the user to set a threshold α such that two schema attributes are declared matched if their similarity score exceeds α. Other knobs allow the user to assign reliability weights to the component matching techniques. Yet another knob controls how many times a component should run. In addition, given a library of components, the user also has the freedom to select which components to be used, and where in the matching system.

Given the above knobs, many possible tuning problems can be defined. As a first step, in this paper we consider the following: given a schema S, how to tune a matching system M so that it achieves high accuracy when we subsequently apply it to match S with other schemas. This is a very common problem that arises in many settings, including data warehousing and integration [37, 19].

Synthesizing Workload with Known Ground Truth: Tuning system M amounts to searching for the "best" knob configuration for matches to S. The *quality* of a knob configuration of M is defined as an aggregate accuracy of the matching system, when applied with that configuration. Accuracy metrics exist (e.g., precision, recall, and combinations thereof [17]). How can they be evaluated? How can we find a corpus of match problems where ground truth (i.e., "true" matches) are known? This is clearly a major challenge for any effort on tuning matching systems.

To address this challenge, our key idea is to employ a set of *synthetic* matching scenarios involving S, *for which we already know the correct matches*, to evaluate knob configurations. Specifically, we apply a set of common transformation rules to the schema and data of S, in essence randomly "perturbing" it to generate a collection of synthetic schemas $S_1, S_2, \ldots, S_n$. For example, we can apply the rule "abbreviating a name to the first three letters" to change the name EMPLOYEES of the table in Figure 1.b) to EMP, and the rule "replacing ,000 with K" to the column salary of this table. We note that these rules are created only once, independent of any schema S.

Since we generated schemas $S_1, S_2, \ldots, S_n$ from S, clearly we can infer the correct semantic matches between these schemas and S. Hence, the collection of schema pairs $\{(S, S_1), (S, S_2), \ldots, (S, S_n)\}$, together with the correct matches, form a *synthetic matching workload*, over which the average accuracy of any knob configuration can be computed. We then use this accuracy as the estimated accuracy of the configuration over matching scenarios involving S.

While the above step of generating the synthetic workload (and indeed the entire tuning process) is completely automatic, eTuner can also exploit user assistance, whenever available. Specifically, it can ask the user to do some simple preprocessing of schema S, then exploit the preprocessing to generate an even better synthetic workload.

Search: The space of knob configurations is often

huge, making exhaustive search impractical. Hence we implement a sequential, greedy approach, denoted *staged tuning*. Consider the matching system M in Figure 1.a. Here, we first tune each of the matchers $1 \ldots n$ in isolation, then tune the combination of the combiner and the matchers, assuming the knobs of the matchers have been set. Finally, we tune the entire matching system, assuming that the knobs of the combiner and matchers have been set. We describe in detail how to tune different types of knobs in Section 5.

In summary, we make the following contributions:

- Establish that it is feasible to tune a matching system, automatically.
- To enable estimating the quality of a matching system's result (with a given knob configuration), we synthesize matching problems for which ground truth is known. For potential applications beyond the tuning context, see Section 7.
- Establish that staged tuning is a workable optimization technique. The solution can also leverage human assistance to further increase tuning quality.
- Extensive experiments over four real-world domains with four matching systems. The results show that eTuner achieves higher accuracy than the alternative (manual and semi-automatic) methods, at virtually no cost to the domain user.

The paper is organized as follows. The next section discusses related work. Section 3 defines the problem of tuning matching systems. Sections 4-5 describe the eTuner approach in detail. Section 6 presents experimental results, and Section 7 concludes.

2 Related Work

Schema matching has received increasing attention over the past two decades (e.g, [37, 35, 5, 4, 21]). A wealth of matching techniques has been developed, employing hand-crafted rules and heuristics (e.g., [31, 36, 11, 6, 32, 28, 30]), machine learning [26, 7, 19, 15, 22, 33, 27], IR [14], information theory [25], clustering [40, 28, 27], and statistics [24, 27].

Many of the developed techniques are synergistic [37, 21]. As a result, the focus is shifting away from monolithic (stovepipe) matching systems, toward creating robust and widely useful matching components, and a plug-and-play framework for them. Many recent works [8, 18, 19, 22, 38, 15, 27, 20] have used a multi-component matching architecture, where each component employs a particular matching technique and the final predictions combine the predictions of the components. A recent work using this approach [8] aims at an industrial-strength schema matching system, while [38] examines its scalability to very large XML schemas.

A next logical direction is to make the frameworks easy to customize for a particular set of matching tasks. Our work aims at automating the customization.

Several recent works exploit previously matched schema pairs to improve matching accuracy (e.g., [18, 19, 27, 7]). Such prior match results, whenever available, can play the role of the "ground-truth" workload and thus can be used for tuning as well. However, tuning data obtained this way is often costly, ad hoc, and limited. In contrast, synthetic matching scenarios can be obtained freely, is often more comprehensive, and can be tailored to a particular matching situation. In Section 6.5 we show that tuning on synthetic scenarios outperforms tuning on previous matching results, but can exploit such results whenever available to further improve tuning quality.

Finally, our work can be seen as part of the trend toward self-tuning databases, to reduce the high total cost of ownership [2, 13, 12].

3 The Match Tuning Problem

We describe our model of a matching system, then use the model to define the match tuning problem. The vast majority of current schema matching systems consider only 1-1 matches, such as contact-info = phone [37]. Hence, in this paper we focus on the problem of tuning such systems, leaving those that finds complex matches (e.g., address = concat(city, state) [15]) as future work. We handle only relational schemas, but the ideas we offer here carry over to other data representations (e.g., XML schemas).

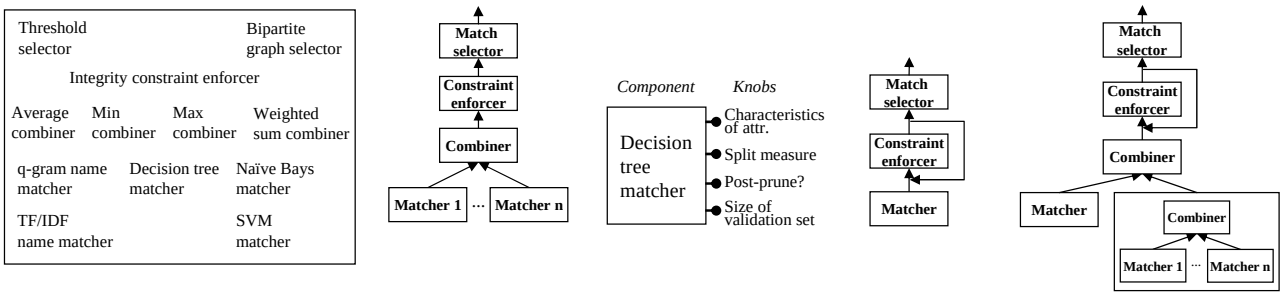
3.1 Modeling 1-1 Matching Systems

We define an 1-1 matching system $\mathcal{M}$ to be a triple (L, G, K), where L is a library of matching components, G is a directed graph that specifies the flow of execution among the components of $\mathcal{M}$, and K is a collection of *control variables* (henceforth *knobs*) that the user (or a tuning system such as eTuner) can set. (A component description includes K_c, the set of knobs available for that component.) In what follow we elaborate on the above concepts, using the LSD system in Figures 2.a-c as a running example. LSD is a learning-based multi-component matching system, and is described in detail in [19].

3.1.1 Library of Matching Components

Such a library contains the following four types of components, variants of which have often been proposed in the literature [37, 21]:

- *Matcher (schemas → similarity matrix):* A matcher takes two schemas S and T and outputs a *similarity matrix*, which assigns to each attribute pair s_i of S and t_j of T a similarity score between 0 and 1. Library L in Figure 2.a has five matchers. The first two compare the names of two attributes (using q-gram and TF/IDF techniques, respectively) to compute their similarity score [18, 19]. The remaining three matchers exploit data instances [19].



(a) Sample library for LSD (b) Execution graph of LSD (c) Sample knobs of a matcher (d) Execution graph of SimFlood (e) Execution graph of LSD-SF

Figure 2: The LSD (a-c), SimFlood (b), and LSD-SF (c) systems.

- *Combiner (matrix $\times \ldots \times$ matrix $\rightarrow$ matrix):* A combiner merges multiple similarity matrices into a single one. Combiners can take the average, minimum, maximum, or a weighted sum of the similarity scores (Figure 2.a) [18, 22, 19]. More complex types of combiner include decision tree [22], and elaborate (often hand-crafted) scripts [8].
- *Constraint Enforcer (matrix $\times$ constraints $\rightarrow$ matrix):* Such an enforcer exploits pre-specified domain constraints or heuristics to transform a similarity matrix (often coming from a combiner) into another one that better reflects the true similarities. Library L in Figure 2.a has a single constraint enforcer, which exploits integrity constraint such as "lot-area cannot be smaller than house-area" [19].
- *Match Selector (matrix $\rightarrow$ matches):* This component selects matches from a given similarity matrix. The simplest selection strategy is *thresholding*: all pairs of attributes with similarity score exceeding a given threshold are returned as matches [18]. More complex strategies include formulating the selection as an optimization problem over a weighted bipartite graph [30] (Figure 2.a).

3.1.2 Execution Graph

This is a directed graph whose nodes specify the components of $\mathcal{M}$ and whose edges specify the flow of execution among the components. The graph has multiple levels, and must be *well-formed* in that (a) the lowest-level components must be matchers that take as input the schemas to be matched, (b) the highest-level component must be a match selector that outputs matches, and (c) all components must get their input. In the following we describe the execution graphs of four matching systems that we experimented with in Section 6.

LSD: The execution graph of LSD [19] is shown in Figure 2.b and has four levels. It states that LSD first applies the n matchers, then combines their output similarity matrices using a combiner. Next, LSD applies a constraint enforcer, followed finally by a match selector. (We omit displaying domain constraints as an input to the enforcer, to avoid clutter.)

COMA & SimFlood: Figure 1.a shows the execution graph of the COMA system [18], which was the first to clearly articulate and embody the multi-component architecture. Figure 2.d shows the execution graph of the SimFlood matching system [30]. SimFlood employs a single matcher (a name matcher [30]), then iteratively applies a constraint enforcer. The enforcer exploits the heuristic "two attributes are likely to match if their neighbors (as defined by the schema structure) match" in a sophisticated manner to improve the similarity scores. Finally, SimFlood applies a match selector (called *filter* in [30]).

LSD-SF: We can combine LSD and SimFlood to build a system called LSD-SF, whose execution graph is shown in Figure 2.e. Here, the LSD system (without the match selector) is treated as another matcher, and is combined with the name matcher of SimFlood, before the constraint enforcer of SimFlood.

User Interaction: Current matching systems usually offer two execution modes: *automatic* and *interactive* [18, 19, 37]. The first mode is as described above: the system takes two schemas, runs without any user intervention, and produces matches. In the second mode users can provide feedback *during* execution, and the system can selectively rerun certain components, based on the feedback (e.g., see [18, 19]). Since our current focus is on automating the entire tuning process (allowing optional user feedback only in creating the synthetic workload, but not *during* the staged tuning, see Section 4.2), we leave the problem of tuning for the interactive mode as future work. Put another way, we tune to optimize the matching provided when user interaction begins.

3.1.3 Tuning Knobs

Knobs of the Components: Matching components are treated as *black boxes*, but we assume that each of them has a set of knobs that are "exposed" and can be adjusted. Each knob is either (I) unordered discrete, (II) ordered discrete or continuous, or (III) set valued.

For example, Figure 2.c shows a decision tree matcher that has four knobs. The first knob, characteristics-of-attr, is set-valued. The matcher has defined a broad set of *salient characteristics* of schema attributes, such as the type of attribute (integer, string, etc.), the min, max, average value of the attribute , and so on (see [26, 22] for more examples). The user (or eTuner) must assign to this knob a *subset*

of these characteristics, so that the matcher can use the selected characteristics to compare attributes. If no subset is assigned, then a default one is used. In learning terminology, this is known as *feature selection*, a well-known and difficult problem [16].

The second knob, split-measure, is unordered discrete (with values "information gain" or "gini index"), and so is the third knob, post-prune? (with values "yes" or "no"). The last knob, size-of-validation-set, is ordered discrete (e.g., 40 or 100). These knobs allow the user to control several decisions made by the matcher during the training process.

Knobs of the Execution Graph: For each node of the execution graph, we assume the user (or eTuner) can plug in one of the several components from the library. Consider for example node Matcher 1 of the execution graph in Figure 2.b. The system $\mathcal{M}$ may specify that this node can be assigned either the q-gram name matcher or TF/IDF name matcher from the library (Figure 2.a).

Consequently, each node of an execution graph can be viewed as a unordered discrete knob. Note that it is conceptually possible to define "data flow" knobs, e.g., to change the topology of the execution graph. However, most current matching systems (with the possible exception of [8]) do not provide such flexibility, and it is not examined here.

Finally, we note that the model described above covers a broad range of current matching systems, including LSD, COMA, and SimFlood, as discussed earlier, but also AutoMatch, Autoplex, GLUE, PromptDiff [7, 20, 34] and those in [22, 27, 33], and Protoplasm, an industrial-strength matching system under development at Microsoft Research [8].

3.2 Tuning of Matching Systems

We are now in a position to define the general tuning problem. Given

- matching system $\mathcal{M} = (L, G, K)$, as defined above;
- workload $\mathcal{W}$ consisting of schema pairs $(S_1, T_1), (S_2, T_2), ..., (S_n, T_n)$ (often the range of schemas will be described qualitatively, e.g., "future schemas to be integrated with our warehouse"); and
- utility function $\mathcal{U}$ defined over the process of matching a schema pair using a matching system; $\mathcal{U}$ can take into account performance factors such as matching accuracy, execution time, etc;

the *match tuning problem* is to find a combination of knob values (called a *knob configuration*) k^* that maximizes the average utility over all schema pairs in the workload. Formally, let $\mathcal{M}(k)$ be the matching system $\mathcal{M}$ using the knob configuration k, and let $\mathcal{K}$ be the space of all knob configurations, as defined by $\mathcal{M}$, then

$$k^* = argmax_{k \in \mathcal{K}} [\sum_{i=1}^{n} \mathcal{U}(\mathcal{M}(k); (S_i, T_i))]/n \qquad (1)$$

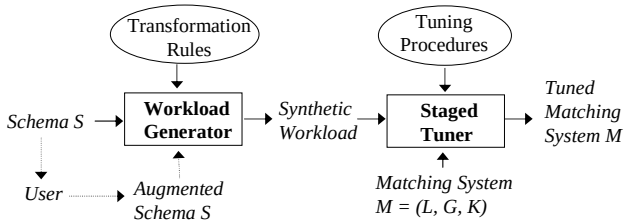


Figure 3: The eTuner architecture.

where $\mathcal{U}(\mathcal{M}(k); (S_i, T_i))$ is the utility of applying $\mathcal{M}(k)$ to the schema pair (S_i, T_i).

Problem Definition: In this paper we restrict the general problem. First, we use just one utility function $\mathcal{U}$ *accuracy*, a combination of precision and recall formalized in Section 6. The rationale for using this measure appear in [17, 37, 27]. Second, we tune $\mathcal{M}$ for the workload of matching a single schema S with all future schemas T_i. This scenario arises in numerous contexts, including data integration and warehousing [19, 37]. In the next two sections we describe the eTuner solution to this problem.

4 The eTuner Approach

The eTuner architecture (see Figure 3) consists of two main modules: workload generator and staged tuner. Given a schema S, the *workload generator* applies a set of transformation rules to generate a synthetic workload. The *staged tuner* then tunes a matching system $\mathcal{M}$ using the synthetic workload and tuning procedures stored in an eTuner repository. The *tuned* system $\mathcal{M}$ can now be applied to match schema S with any subsequent schema. It is important to note that the transformation rules and the tuning procedures are created only *once*, independently of any application domain, when implementing eTuner.

While the tuning process is completely automatic, eTuner can also exploit user assistance to generate an even higher quality synthetic workload. Specifically, the user can "augment" schema S with information on the relationships among attributes (see the dotted arrows in Figure 3).

The rest of this section describes the workload generator, in both automatic and user-assisted modes, while the next section describes the staged tuner.

4.1 Automatic Workload Creation

Given a schema S and a parameter n, the workload generator proceeds in three steps. (1) It uses S to create two schemas U and V, which are identical to S but are associated with different data tuples. (2) It perturbs V to generate n schemas $V_1, V_2, \ldots, V_n$. (3) For each schema $V_i, i \in [1, n]$, it traces the derivation process to create the set of correct semantic matches Ω_i between U and V_i, then outputs the set of triples $\{(U, V_i, \Omega_i)\}_{i=1}^{n}$ as the synthetic workload. We now describe the three steps in detail.

Input: schema S, data tuples D, transformation functions T, workload size n
Output: synthetic workload W (n schema pairs and their correct matches)
Let T_t, T_c, T_n, T_v, T_f be table-, column-, name-, value- and format-transformation rules in repository T
1. Split schema S into U, V
  1.1. Let $U = S$ and $V = S$
  1.2. Create data set D_u, D_v such that $D_u \cap D_v = \emptyset$, $D_u \cup D_v = D$, $|D_u| = |D_v|$
2. Generate n schemas $V_1, V_2, \ldots V_n$ from V
  2.1. Let $V_i = V$
  2.2. Perturb number of tables in V_i using rules in T_t
  2.3. Perturb the structure of each table in V_i using rules in T_c
  2.4. Foreach name n_k in schema V_i do change n_k using rules in T_n
  2.5. Foreach column c_j in V_i do
    Let d_{cj} = data associated with c_j in D_v
    Let σ_{cj}^2 = variance(d_{cj}) and μ_{cj} = mean(d_{cj})
    Perturb σ_{cj}^2 and μ_{cj} using rules in T_v
    Generate $|d_{cj}|$ data values using a Gaussian distribution generator with perturbed σ_{cj}^2 and μ_{cj}
    Perturb format of each generated data value using rules in T_f
3. Foreach (U, V_i) do generate its correct match set Ω_i
  3.1. Let $\Omega_i = \emptyset$
  3.2. Foreach column c in V_i do
    Foreach column c' in V do if c is generated from c' then add (c, c') to Ω_i
4. Return $W = \{(U, V_1, \Omega_1), (U, V_2, \Omega_2), \ldots (U, V_n, \Omega_n)\}$

Figure 4: High-level description of the workload generator.

4.1.1 Create Schemas U and V from Schema S

The workload generator begins by creating two schemas U and V which are identical to S. Next, it partitions data tuples D associated with S (if any) into two equal, but *disjoint* sets D_u and D_v, then assign them to U and V, respectively. This is to ensure that once V has been perturbed into V_i, we can pair U and V_i to form a matching scenario where *the schemas do not share any data tuple*. Using schemas that share data tuples would make matching easier [15, 9] and thus may significantly bias the tuning process.

The above step is illustrated in Figure 5.a, which shows a schema S with three tables. The schemas V and U generated from S also have three tables with identical structures. However, table 3 of S, which we show in detail as table EMPLOYEES in Figure 5.a, has in effect being partitioned into two halves. Its first two tuples go to the corresponding table of schema V, while the remaining two tuples go to schema U. We experimented, and found that the above simple strategy of randomizing, then halving tuples in each table worked as well as more complex strategies.

4.1.2 Create Schemas $V_1, \ldots, V_n$ by Perturbing V

To create a schema, say, V_1, the workload generator perturbs schema V in several steps, using a set of prespecified, domain-independent rules stored in eTuner.

• **Perturbing Number of Tables:** The generator randomly selects a *perturb-number-of-tables* rule to apply to the tables of schema V. This is repeated up to α_t times (currently set to two in our experiments). eTuner currently has two such rules. The first one randomly selects two joinable tables, and *merges* them based on a join path to create a new table. The second rule randomly selects and *splits* a table into two (that can be joined to recover the original table).

As an example, after applying the rules, schema V at the top of Figure 5.a, which has three tables 1, 2, 3, has been transformed into schema V_1, which has only two tables 12 and 3. The tables 1 and 2 of V have been merged into table 12 of V_1.

• **Perturbing the Structure of Each Table:** For each table of schema V_1, the generator now perturbs its structure. It randomly selects *column-transformation* rules to apply to the columns of the table, up to α_c times (currently set to four). eTuner has three such rules. The first one merges two columns. Currently, two columns can be merged only if (a) they are neighboring columns, and (b) they share a prefix or suffix (e.g., first-name and last-name). The second rule randomly removes a column from the table. The third rule swaps two columns.

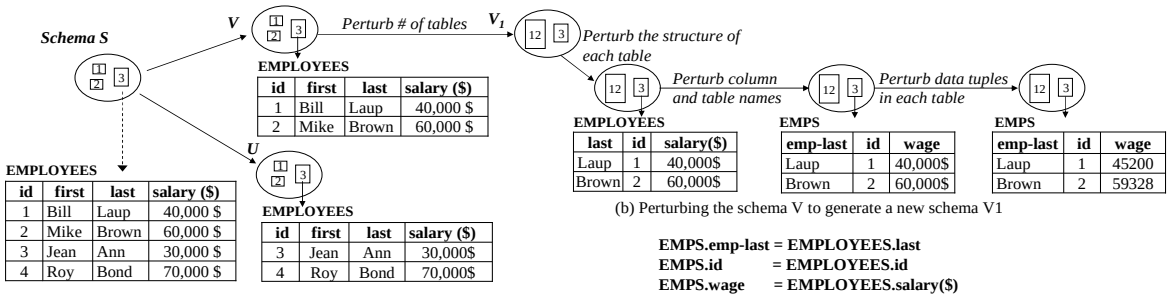
Continuing with our example, in Figure 5.b, for table EMPLOYEES, column first is dropped and two columns last and id are swapped.

• **Perturbing Table and Column Names:** In the next step, the name of each table and its columns in schema V_1 are perturbed. eTuner has implemented a set of rules that capture common name transformations [28, 15, 37]. Examples include abbreviation to the first three or four characters, dropping all vowels, replacing the name with a synonym (currently obtained from Merriam-Webster's online thesaurus), and dropping prefixes (e.g., changing ACTIVE-EMPS to EMPS). Rules that perturb a column name also consider adding a perturbed version of the table name as prefix, or borrowing prefixes from neighboring columns. We also add a rule that changes a column name into a random sequence of characters, to model cases where column names are not intelligible to anyone other than the data creator. For each name, the rules are called up to α_n times (currently set to two).

In Figure 5.b, the name of table EMPLOYEES has been abbreviated to EMPS (the first three letters plus "S" for plurality). The name of column last has been added the new table name as a prefix, to become emplast. Finally, the name of column salary($) has been replaced with the synonym wage.

• **Perturbing Data:** In the final step, the generator perturbs the data of each table column in V_1, by perturbing the format, then values of the data. eTuner has a set of rules that capture common transformation of data formats (and is extensible to adding more rules). Examples include "dropping or adding $ sign", "changing dates from 12/4 to Dec 4",etc. For each column, the generator applies such rules up to α_d times (currently set to two).

Once the format of a column c has been perturbed, the generator perturbs the data values. If the values are numeric (e.g., price, age, etc), then they are assumed to have been generated from a normal distribution with mean μ_c and variance ${\sigma_c}^2$. Thus, the generator estimates μ_c and ${\sigma_c}^2$ from current data values in column c. It then randomly decides whether to perturb the mean and variance by a random amount in



(a) Splitting S to create two identical schemas V and U, with disjoint data tuples

(b) Perturbing the schema V to generate a new schema V1

(c) Sample matches created between V1 and U

Figure 5: Perturbing schema S to generate two schemas U and V_1 and the correct matches between them.

the range +/-[10,100]%. Let the new mean and variance be μ'_c and variance ${\sigma'_c}^2$. Then each value x is now generated according to the above normal distribution. If the values are textual (e.g., house description), then the generator randomly adds or remove text tokens. More detail can be found in the full paper.

For example, consider column wage of Table EMPS in Figure 5.b (the rightmost table). Its format has been perturbed so that the signs "$" and "," are dropped, and its values have been changed, so that "40,000$" is now "45200".

4.1.3 Create Semantic Matches between V_i and U

In the final step, the generator retraces the perturbation history to create correct semantic matches between V_1 and U. Briefly, if attribute a of V_1 is derived from attributes $b_1, \ldots, b_k$ of schema V, then (since schemas U and V are identical) we create $a = b_1, \ldots, a = b_n$ as correct matches between V_1 and U. Figure 5.c lists the correct matches between table EMPS of V_1 and table EMPLOYEES of U. As another example, suppose attributes first-name and last-name of V are merged to create attribute name of V_1, then the generator derives the matches name = first-name and name = last-name.

Let Ω_i be the set of derived semantic matches between V_i and U. The workload generator then returns the set of triples $\{(U, V_i, \Omega_i)\}_{i=1}^{n}$ as the synthetic workload on which to tune matching system $\mathcal{M}$.

4.2 User-Assisted Workload Creation

The generator can exploit user assistance whenever available, to build a better workload, which in turn improves tuning performance.

To illustrate the benefits of user assistance, suppose each employee can be contacted via two phone numbers, phone-1 and phone-2 (as attributes of schema U). Suppose while generating schema V_1 attribute phone-1 is renamed emp-phone and phone-2 is dropped. Then the generator will declare the match emp-phone = phone-1 correct (between V_1 and U), but will not recognize emp-phone = phone-2 as also correct (since emp-phone is *not* derived from phone-2, see Section 4.1.3). This is counter-intuitive, since both numbers are the employee's phone numbers. Furthermore, it will force the tuning algorithm to look for "artificial" ways to distinguish the two phone numbers, thereby overfitting the tuning process.

To address this issue, we say a group of attributes $G = \{a_{i1}, \ldots, a_{in}\}$ of schema S are *match-equivalent* if and only if whenever a match $b = a_{ij}, 1 \leq j \leq n$ is judged correct, then all other matches $b = a_{ik}, 1 \leq k \leq n, k \neq j$, are also judged correct. In the above example, phone-1 and phone-2 are match equivalent. We ask the user to identify match equivalent attributes of schema S. The generator then refines the set of correct semantic matches, so that if $G = \{a_{i1}, \ldots, a_{in}\}$ is match equivalent, and $b = a_{ij}, 1 \leq j \leq n$ is correct, then $b = a_{ik}, 1 \leq k \leq n, k \neq j$ are also correct.

The user does not have to specify *all* match-equivalent attribute groups, only as much as he/she can afford. Further, such grouping is a relatively low-level effort, since it involves examining only schema S, and judging if attributes are semantically close enough to be deemed match equivalent. Such attributes are often neighbors of one another, facilitating the examination. Section 6 shows that such user assistance can significantly improve the tuning performance. The user can also assist in many other ways, e.g., by suggesting domain-specific perturbation rules; but such possibilities are outside the scope of this paper.

5 Tuning with the Synthetic Workload

We now describe how to tune a $\mathcal{M}$ with a synthetic workload $\mathcal{W}$ as created in the previous section.

5.1 Staged Tuning

Our goal is to find a knob configuration of $\mathcal{M}$ that maximizes the average accuracy over $\mathcal{W}$. The configuration space is usually huge, making exhaustive search impractical.

Consequently, we propose a *staged, greedy* tuning approach. Assume the execution graph of $\mathcal{M}$ has k levels. We first tune each match component at the bottom, k-th level in isolation. Next, we tune subsystems that consist of components at the $(k-1)$th and k-th levels. While tuning such subsystems, we assume the components at the k-th level have been tuned, so their knob values are fixed, and we only need to tune knobs at level $(k-1)$th. If there is a loop of m components, then the loop is treated as a single component when being considered for addition to a subsystem.

This staged tuning repeats until we have reached the first level and hence have tuned the entire system.

Consider for example tuning the LSD system in Figure 2.b. We first tune each of the matchers $1 \ldots n$. Next, we tune the subsystem consisting of the combiner and the matchers, but assuming that the matchers have been tuned. Then we tune the subsystem consisting of the constraint enforcer, combiner, and matchers, assuming that the combiner and matchers have been tuned, and so on. Suppose the execution graph has k levels, m nodes per level, and each node can be assigned one of the n components in the library. Assume each component has p knobs, and each knob has q values. Then staged tuning examines only a total of $k \times (m \times (n \times p \times q))$ out of $(n \times p \times q)^{k \times m}$ knob configurations, a drastic reduction. Section 6 shows that while not guaranteeing to find the optimal knob configuration, staged tuning still outperforms currently possible tuning methods.

5.2 Tuning Subsystems of $\mathcal{M}$

We now describe in detail how to tune a subsystem $\mathcal{S}$ of the original matching system $\mathcal{M}$. First, if $\mathcal{S}$ does not produce matches as output (e.g., producing similarity matrix instead), we add the match selector of $\mathcal{M}$ as the top component of $\mathcal{S}$. This is to enable the evaluation of $\mathcal{S}$'s accuracy on the synthetic workload.

We then tune the knobs of $\mathcal{S}$ as follows. Recall from Section 3.1.3 that there are three types of knobs: (I) unordered discrete, (II) ordered discrete or continuous, and (III) set valued. Type-I knobs usually have few values (e.g., "yes"/"no"), while Type-II knobs usually have a large number of values. Hence, we first convert each type-II knob into a type-I knob, by selecting q equally-spaced values (currently set to six). For example, for value range [0,1], we select 0, 0.2, etc., for value range [0,500], we select 0, 100, 200, etc.

We now only have type-I and type-III knobs. In fact, in practice we often have just one type-III (set-valued) knob: selecting features for a matcher (e.g., [22, 19]). Hence, we assume that there is just one type-III knob for subsystem $\mathcal{S}$, which handles feature selection. In the next step, we form the Cartesian space of all type-I knobs. This space is usually small, since each type-I knob has few values, and $\mathcal{S}$ does not have many knobs (due to the staged tuning assumption). For each knob setting in this Cartesian space, we can then tune for the lone type-III knob, as described in detailed in Section 5.3 below, then select the setting with the highest accuracy.

At this moment, we have selected a value for all type-I and type-III knobs of $\mathcal{S}$. Recall that some type-I knobs are actually converted from type-II ones, which are ordered discrete or continuous. We can now focus on these type-II knobs, and perform hill climbing to obtain a potentially better knob configuration.

Tuning Interrelated Knobs: We may know of fast procedures to tune a set of interrelated knobs. For example, a weighted sum combiner has n knobs that specify matcher weights [19]. They can be tuned using linear or logistic regression (over the synthetic workload) [19]. However, such tuning often requires that all other knobs of $\mathcal{S}$ have been set (otherwise $\mathcal{S}$ cannot be run). For this reason, in Step 1 we run the tuning process as described earlier, to obtain reasonable values for the knobs of $\mathcal{S}$. Then in Step 2 we run procedures to tune interrelated knobs (if any, these procedures are stored in eTuner). If this tuning results in a better knob configuration, then we take it; otherwise we use the knob configuration found in Step 1.

Feature	Descriptions
IsNumeric	If numeric, *YES*; else *NO*
# of @	Number of the "@" symbol
# of $	Number of the "$" symbol
# of token	Number of tokens
# of digit	Number of digits
Type	Type of attributes
Min/nbMin	Minimum length/non-blanks of character attributes Minimum value of numeric attributes
Max/nbMax	Maximum length/non-blanks of character attributes Maximum value of numeric attributes
Avg/nbAvg	Average length/non-blanks of character attributes Average value of numeric attributes
CV/nbCV	CV of length/non-blanks of character attributes CV of numeric attributes
SD/nbSD	SD of length/non-blanks of character attributes SD of numeric attributes

Figure 6: Sample features that eTuner uses in selecting a best set of features for the schema attributes.

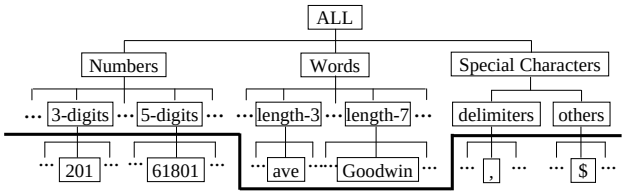


Figure 7: An example taxonomy for the Naive Bayes matcher.

5.3 Tuning to Select Features

We now describe how to tune the type-III knob that selects features for subsystem $\mathcal{S}$. Without loss of generality, assume $\mathcal{S}$ is a matcher.

Recall from Section 3.1.3 that a matcher often transforms each schema attribute into a feature vector, then uses the vectors to compare attributes. In eTuner we have enumerated a set of features judged to be salient characteristics of schema attributes, based on our matching experience and the literature (e.g., [26, 18, 30, 22, 7, 28, 19, 20]). Figure 6 shows 16 sample features. The goal of tuning is then to select from the set F of all enumerated features a subset F^* that best assist the matching process.

The simplest solution to find F^* is to enumerate all subsets of F, run $\mathcal{S}$ with each of the subsets over the synthetic workload, then select the subset with the highest matching accuracy. This solution is clearly impractical. Hence, we consider a well-known selection method called *wrapper* [16], which starts with a set of features (e.g., the empty set), then considers adding or deleting a single feature. The possible changes to the feature set are evaluated by running $\mathcal{S}$ over the synthetic workload, and the best change is made. Then a new set of changes is considered. However, even this

Domains

Domain	# schemas	# tables per schema	# attributes per schema	# tuples per table
Real Estate	5	2	30	1000
Courses	5	3	13	50
Inventory	10	4	10	20
Product	2	2	50	120

Matching systems

LSD:	6 Matchers, 6 Combiners, 1 Constraint enforcer, 2 Match selectors, 21 Knobs
iCOMA:	10 Matchers, 4 Combiners, 2 Match selectors, 20 Knobs
SimFlood:	3 Matchers, 1 Constraint enforcer, 2 Match selectors, 8 Knobs
LSD-SF:	7 Matchers, 7 Combiners, 1 Constraint enforcer, 2 Match selectors, 10 Knobs

Figure 8: (a) Real world domains and (b) matching systems for our experiments

greedy algorithm is too expensive. Even just for 20 features, it would run $\mathcal{S}$ over the synthetic workload 210 times.

To reduce the runtime complexity, given the feature set F, we first apply another selection method called *Relief-F* (described in detail in [16]) to select a small subset F'. *Relief-F* detects relevant features well, and runs very fast, as it examines only the synthetic workload, not running any matching algorithm [16]. We then apply the above greedy *wrapper* algorithm to the much smaller set F' to select the final set of features F^*.

Selecting Features for Text-Based Matchers: Features as described above are commonly used by learning methods such as decision tree, neural network [26, 22, 19, 20] and also by many rule-based methods (e.g., [18, 28, 30]). However, many learning-based (e.g., Naive Bayes, SVM) as well as IR-based matching methods (e.g., [14, 19]) view data instances as *text fragments*, and as such operate on a different space of features. We now consider generating such feature spaces and the associated feature selection problem.

We can treat each distinct word, number, or special characters in the data instances as a feature. Thus, the address 201 Goodwin ave. Urbana, IL 61801 is represented with eight features: four words, two numbers, and two special characters "," and ".". However, for zip codes, specific values such as "61801" are not important; what we really need (to match attributes accurately) is knowing that they are 5-digit numbers. Hence, we should consider abstracted features, such as 5-digits, in addition to word-level features.

Figure 7 shows a sample taxonomy of features over text for eTuner (adapted from [10]). A line cutting across this taxonomy represents a selected feature set. Consider for example the thick line in the figure. It states that all numbers are abstracted into 1-digit, 2-digits, etc, all words can be treated as features, and so on. Given this, the above address is now represented as the set {3-digits, Goodwin, ave, delimiters,Urbana, delimiters, IL, 5-digits}. To find the best feature set, we employ a method similar to the *wrapper* method described earlier, starting from the feature set at the bottom of the taxonomy.

6 Empirical Evaluation

We now present experimental results over four real-world domains and four matching systems, to demonstrate the need for tuning and the utility of eTuner.

Domains: We obtained publicly available schemas in four domains. The schemas have been used in recent schema matching experiments [19, 15, 27]. The domains have varying numbers of schemas (2-10) and diverse schema sizes (10-50 attributes per schema, see Figure 8.a). Real Estate lists houses for sale. Courses contains time schedules for several universities. Inventory describes business product inventories, and Product stores product descriptions of groceries.

Matching Systems: Figure 8.b summarizes the four matching systems in our experiments. We began by obtaining three multi-component systems that were proposed recently. The LSD system was originally developed by one of us [19] to match XML DTDs. We adapted it to relations. The SimFlood system [30] was downloaded from the Web. The COMA system was described in [18]. Since we did not have access to COMA, we implemented a version of it called iCOMA. The iCOMA library includes all components described in [18], except the hybrid and reuse matchers. We also added the decision tree matcher to the library, to exploit data. Finally, we combined LSD and SimFlood (as described in Section 3), to obtain LSD-SF, the fourth matching system. Figure 8.b shows that the systems have 4-18 components, with 7-25 knobs (the full paper will give a complete description).

Experimental Methodology: For each domain, we randomly selected a schema to be the source schema S. Next we applied the above four matching systems (tuned in several ways, as described below) to match S and the remaining schemas in the domain (treated as future target schemas). This was repeated four times except for Product, which contains only 2 sources. We then report the average accuracy per domain. For eTuner, we set the size of the synthetic workload at 30, and the number of tuples per schema table at 50.

Performance Measure: Following recent schema matching practice [18, 17, 30, 27, 37], we use the F_1 score to evaluate matching accuracy. Given a set of candidate matches for S and T, we have $F_1 = (2PR)/(P + R)$, where *precision* P is the percentage of candidate matches that are correct, and *recall* R is the fraction of all correct matches discovered. The goal of tuning is to find the knob configuration that maximizes F_1 score.

6.1 The Need for Tuning

We begin by demonstrating the need for tuning, using Figures 9.a-d. The figures show the results for LSD, iCOMA, SimFlood, and LSD-SF, respectively. Each figure shows the results over four domains: Real Estate, Product, Inventory, and Course. Thus we have a total

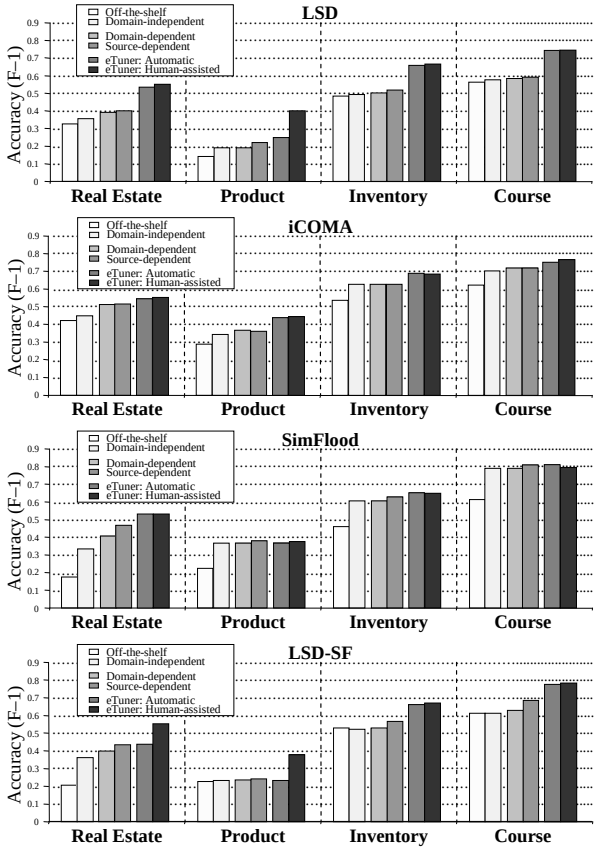


Figure 9: Matching accuracy for (a) LSD, (b) iCOMA, (c) SimFlood, and (d) LSD-SF.

of 16 groups: one for each pair of system and domain, separated by dotted vertical lines on the figures.

We first applied the matching systems "as is" to the domains, and reported the accuracy as the first bar in each group. For instance, for LSD and Real Estate (the first group of Figure 9.a), the first bar is 33%. The "as is" accuracy is 14-62% across all 16 cases, demonstrating that "off-the-shelf" matching systems are quite brittle.

Next, we did our best to tune each system independently of any domain, in effect imitating a vendor tuning a system before release. (We found graduate student volunteers not suitable for this task, suggesting that administrators will also have difficulty tuning. See below for details). We examined literature about each matching system, leveraged our knowledge of machine learning and schema matching, and tweaked the systems on pairs of schemas not otherwise used in the experiments. The *second bar* in each group reports the accuracy of applying the tuned systems, scattered in the range 19-78% across all 16 cases. This accuracy suggests that tuning matching systems once and for all does not work well, implying the need for more context dependent settings.

6.2 "Quick and Dirty" Tuning

Next, we examined the following. Whenever we need to match two schemas S and T, does it seem possible to provide a simple interactive tuning wizard? Perhaps one might carry out "quick and dirty" tuning, by just tweaking a few knobs, examining the output of the matching system, then adjusting the knobs again? If this works, then there is no compelling need for automated tuning.

We asked a few graduate students to perform such tuning on six pairs of schemas, and found two major problems. First, it turned out to be very difficult to explain the matching systems in sufficient details so that the volunteers feel they can tune effectively. Consider for example the decision tree matcher described in Section 3.1.3. We found that the tuned version of this matcher improves accuracy significantly, so tuning it is necessary. However, it was very difficult to explain the meaning of its knobs (see Section 3.1.3) to a volunteer who lacked knowledge of machine learning. Second, even after much explanation, we found that we could perform "quick and dirty" tuning better than volunteers. Similar difficulties arose when we asked volunteers to tune systems in a domain independent manner (as described earlier).

Thus, we carried out tuning ourselves, allotting one hour per matching task. The measured accuracy over the six matching tasks is 21-65%, suggesting that "quick and dirty" tuning is not robust. The key difficulty was that despite our expertise, we still were unable to predict the effects of tuning certain (combinations of) knobs. Lacking the ground truth matches (during the tuning process), we were also unable to estimate the quality of each knob configuration with high accuracy.

6.3 Domain- & Source-Dependent Tuning

Next, we examined if it is possible to tune just once per domain, or once per a source S (before matching S with future schemas).

We tuned each matching system for each domain, in a manner similar to domain-independent tuning, but taking into account the characteristics of the domain sources. (For example, if a domain has many textual attributes, then we assigned more weight to the Naive Bayes text classifier [19].) The third bar in each group (Figures 9.a-d) shows accuracy 19-78%.

We then explored source-dependent tuning. Given a source S, we assume that we already know matches between S and two other sources $S_1 - S_2$ in the same domain. We used staged tuning of eTuner over these known matches to obtain a tuned version of the matching system. Next, we manually tweaked the system, trying to further improve its accuracy over matching S with $S_1 - S_2$. The fourth bar in each group (Figures 9.a-d) shows accuracy 22-81%.

The results show that source-dependent (most labor consuming) tuning beats domain-dependent tuning (less labor consuming, as carried out only once per domain) by 1-7%, which in turns beats domain-independent tuning (least costly) by 0-6%.

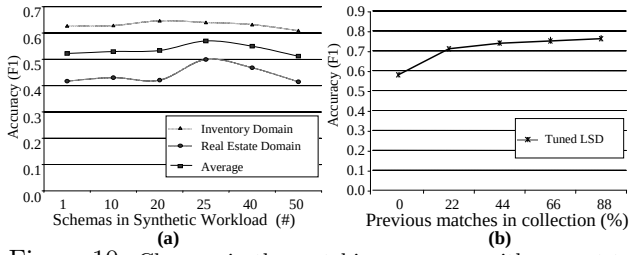


Figure 10: Changes in the matching accuracy with respect to (a) size of the synthetic workload, and (b) the number of prior matched schema pairs in the workload.

6.4 Tuning with eTuner

The fifth bar (second bar from the right) of each group (Figures 9.a-d) then shows the accuracy of matching systems tuned automatically with eTuner. The results show accuracy 23-82% across all 16 groups. eTuner is better than source-dependent tuning (the best tuning method so far) in 14 out of 16 cases, by 1-15%, and is slightly worse in 2 cases, by 2%. The cost of using eTuner consists mainly of "hooking" it up with the knobs of a matching system, and would presumably be born by vendors and amortized over all uses. The above analysis demonstrates the promise of eTuner over previous tuning alternatives.

Zooming into the experiments shows that tuning improves all levels of matching systems. For example, the accuracy of matchers improves by 6% and of combiner by 13% for LSD.

User-Assisted Tuning: The last bar of each group (Figures 9.a-d) shows the accuracy of eTuner with user-assisted workload creation (Section 4.2), with users being volunteer graduate students. The results show accuracy 38-79% across all 16 groups, improving 1-14% over automatic tuning (except in three cases there is no improvement, and one case of decreased accuracy by 1%). The results show the benefits of user assistance in tuning.

6.5 Sensitivity Analysis

Synthetic Workload: Figure 10.a shows the accuracies of automatic eTuner, as we vary the size (i.e., number of schemas generated) of the synthetic workload. The accuracies are for LSD over Real Estate and Inventory, though we observed similar trends in other cases. As the workload size increases, the number of schema/data perturbation rules that it captures increases. This improves accuracy. After size 25-30, however, accuracy starts decreasing. This is because at this point, all perturbation rules have been captured in the workload. As the workload's size increases, its "distance" from real workloads increases, and so tuning overfits the matching system. Thus, for the current set of perturbation rules (as detailed in Section 4.1), we set the optimal workload size at 30. The results also show no abrupt degradation of accuracy, thus demonstrating that the tuning performance is robust for small changes in the workload size.

Adding Perturbation Rules to Matching Systems: It is interesting to note that even if a schema matching system captures *all* perturbation templates of eTuner, it still does not necessarily do well, due to the difficulty of "reverse engineering". For example, the iMAP complex matching system [15] contains a far richer set of perturbation rules than eTuner. Nevertheless, its accuracy on 1-1 matching (as reported in [15] on a different domain) is only 62-71%.

Exploiting Prior Match Results: Figure 10.b shows the accuracy of LSD over Inventory, as we replaced 0%, 22%, etc. of the synthetic workload with real schema pairs that have been matched in the same domain. The results show that exploiting previously matched schema pairs indeed improves the quality of the synthetic workload, thereby matching accuracy. This is important because such prior match results are sometimes available [19, 18]. However, while such match results can complement the synthetic matching scenarios, exploiting them alone does not work as well, as we demonstrated with source-dependent tuning described in Section 6.3.

Runtime Complexity: Our unoptimized version of eTuner took under 30 minutes to tune a schema S, spending the vast majority of time in the staged tuning step. We expect that tuning matching systems will often be carried out offline, e.g., overnight, or as a background task. In general, the scalability of tuning techniques such as eTuner will benefit from scaling techniques developed for matching very large schemas [38] as well as optimization within the tuning module, such as reusing results across matching steps and more efficient, specialized procedures for knob tuning.

7 Conclusion and Future Work

We have demonstrated that tuning is important for fully realizing the potentials of multi-component matching systems. Current tuning methods are *ad hoc*, labor intensive, or brittle. Hence, we have developed eTuner, an approach to automatically tune schema matching systems. Given a schema S and a matching system $\mathcal{M}$, our key idea is to synthesize a collection of matching scenarios involving S, for which we already know the ground-truth matches, and then use the collection to tune system $\mathcal{M}$. This way, tuning can be automated, and can be tailored to the particular schema S. We evaluated eTuner on four matching systems over four real-world domains. The results show that matching systems tuned with eTuner achieve higher accuracy than with current tuning methods, at little cost to the user.

For future work, we are exploring better search methods, and more extensive evaluation of eTuner. The current work also hints at some possible resemblances between match tuning and query optimization: given problem (query answering vs. schema matching) and a set of operators (e.g., hash join, index join

vs. matchers, combiners), how to quickly assemble an execution tree that performs optimally in some sense (time vs. accuracy). It might be interesting to further explore this connection. We also consider applying the above idea of using synthetic input/output pairs to make a system robust to other contexts. We have successfully adapted it to mapping maintenance [29], and are adapting it to record linkage systems (e.g., [23, 3]).

Acknowledgments: We thank the reviewers for invaluable comments. This work was supported by NSF grants CAREER IIS-0347903 and ITR 0428168.

References

[1] Karl Aberer. Special issue on peer to peer data management. *SIGMOD Record*, 32(3), September 2003.

[2] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollr, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database tuning advisor for microsoft sql server 2005. In *VLDB*, 2004.

[3] P. Andritsos, R. J. Miller, and P. Tsaparas. Information-theoretic tools for mining database structure from large data sets. In *Proc. of SIGMOD-2004*.

[4] Goksel Aslan and Dennis McLeod. Semantic heterogeneity resolution in federated databases by metadata implantation and stepwise evolution. *VLDB Journal*, 8(2):120–132, 1999.

[5] C. Batini, M. Lenzerini, and SB. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Survey*, 18(4):323–364, 1986.

[6] S. Bergamaschi, S. Castano, M. Vincini, and D. Beneventano. Semantic integration of heterogeneous information sources. *Data and Knowledge Engineering*, 36(3):215–249, 2001.

[7] J. Berlin and A. Motro. Database schema matching using machine learning with feature selection. In *CAiSE*, 2002.

[8] P. A. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. Industrial-strength schema matching. *SIGMOD Record, Special Issue in Semantic Integration*, December 2004.

[9] Alexander Bilke and Felix Naumann. Schema matching using duplicates. In *Proc. of ICDE-05*.

[10] V. Borkar, K. Deshmukh, and S. Sarawagi. Automatic text segmentation for extracting structured records. In *Proc. of SIGMOD-01*.

[11] S. Castano and V. De Antonellis. A schema analysis and reconciliation tool environment. In *IDEAS*, 1999.

[12] S. Chaudhuri, B. Dageville, and G. Lohman. Self-managing technology in database management systems (tutorial). In *Proc. of VLDB-04*.

[13] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB*, 2000.

[14] C. Clifton, E. Housman, and A. Rosenthal. Experience with a combined approach to attribute-matching across heterogeneous databases. In *Proc. of the IFIP Working Conference on Data Semantics (DS-7)*, 1997.

[15] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering complex matches between database schemas. In *Proc. of SIGMOD*, 2004.

[16] T. G. Dietterich. Machine learning research: Four current directions. *AI Magazine*, 18(4):97–136, 1997.

[17] H. Do, S. Melnik, and E. Rahm. Comparison of schema matching evaluations. In *Proc. of the 2nd Int. Workshop on Web Databases (German Informatics Society)*, 2002.

[18] H. Do and E. Rahm. Coma: A system for flexible combination of schema matching approaches. In *VLDB*, 2002.

[19] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: A machine learning approach. In *Proceedings of the ACM SIGMOD Conference*, 2001.

[20] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to map ontologies on the semantic web. In *Proc. of the World-Wide Web Conference (WWW-02)*, 2002.

[21] A. Doan, N. Noy, and A. Halevy. Introduction to the special issue on semantic integration. *SIGMOD Record*, 33(4):11–13, 2004.

[22] D. Embley, D. Jackman, and L. Xu. Multifaceted exploitation of metadata for attribute match discovery in information integration. In *Proc. of the WIIW-01*, 2001.

[23] Venkatesh Ganti, Surajit Chaudhuri, and Rajeev Motwani. Robust identification of fuzzy duplicates. In *ICDE*, 2005.

[24] B. He and K. C.-C. Chang. Statistical schema matching across web query interfaces. In *Proc. of SIGMOD-03*, 2003.

[25] J. Kang and J. Naughton. On schema matching with opaque column names and data values. In *SIGMOD*, 2003.

[26] W. Li, C. Clifton, and S. Liu. Database integration using neural network: implementation and experience. *Knowledge and Information Systems*, 2(1):73–96, 2000.

[27] J. Madhavan, P. Bernstein, A. Doan, and A. Halevy. Corpus-based schema matching. In *Proc. of ICDE-05*.

[28] J. Madhavan, P.A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *Proc. of VLDB-01*.

[29] R. McCann, B. Alshebli, Q. Le, H. Nguyen, L. Vu, and A. Doan. Mapping maintenance for data integration systems. In *Proc. of VLDB-05*.

[30] S. Melnik, H. Molina-Garcia, and E. Rahm. Similarity flooding: a versatile graph matching algorithm. In *Proc. of ICDE-02*.

[31] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *Proc. of VLDB-98*.

[32] P. Mitra, G. Wiederhold, and J. Jannink. Semi-automatic integration of knowledge sources. In *Proc. of Fusion-1999*.

[33] F. Neumann, CT. Ho, X. Tian, L. Haas, and N. Meggido. Attribute classification using feature analysis. In *Proceedings of the Int. Conf. on Data Engineering (ICDE)*, 2002.

[34] N.F. Noy and M.A. Musen. PROMPT: Algorithm and tool for automated ontology merging and alignment. In *Proc. of the National Conference on Artificial Intelligence*, 2000.

[35] A. Ouksel and A. P. Seth. Special issue on semantic interoperability in global information systems. *SIGMOD Record*, 28(1), 1999.

[36] L. Palopoli, D. Sacca, and D. Ursino. Semi-automatic, semantic discovery of properties from database schemes. In *Proc. of IDEAS-98*, pages 244–253.

[37] E. Rahm and P.A. Bernstein. On matching schemas automatically. *VLDB Journal*, 10(4), 2001.

[38] E. Rahm, H. Do, and S. Massmann. Matching large XML schemas. *SIGMOD Record, Special Issue in Semantic Integration*, December 2004.

[39] L. Seligman and A. Rosenthal. The impact of xml in databases and data sharing. *IEEE Computer*, 2001.

[40] W. Wu, C. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the Deep Web. In *Proc. of SIGMOD*, 2004.

[41] L.L. Yan, R.J. Miller, L.M. Haas, and R. Fagin. Data driven understanding and refinement of schema mappings. In *Proceedings of the ACM SIGMOD*, 2001.