The VLDB Journal (2007) 16:97–122
DOI 10.1007/s00778-006-0024-z

SPECIAL ISSUE PAPER

eTuner: tuning schema matching software using synthetic scenarios

Yoonkyong Lee · Mayssam Sayyadian · AnHai Doan · Arnon S. Rosenthal

Received: 15 January 2006 / Accepted: 11 June 2006 / Published online: 14 September 2006

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. The results show that eTuner produced tuned matching systems that achieve higher accuracy than using the systems with currently possible tuning methods.

Keywords Schema matching · Tuning · Synthetic schemas · Machine learning · Compositional approach

Y. Lee (✉) · M. Sayyadian · A. Doan
University of Illinois,
Urbana, IL 61801, USA
e-mail: ylee11@cs.uiuc.edu

M. Sayyadian
e-mail: sayyadia@cs.uiuc.edu

A. Doan
e-mail: anhai@cs.uiuc.edu

A. S. Rosenthal
The MITRE Corporation,
Bedford, MA 01730, USA
e-mail: arnie@mitre.org

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 [62].

Manually finding the matches is labor intensive, thus numerous automatic matching techniques have been developed (see [5,24,29,55,62] for recent surveys). Each individual matching technique has its own strength and weakness [23–25,62]. Hence, increasingly, matching tools are being assembled from *multiple components*, each employing a particular matching technique [23–25,29,62].

The multi-component nature is powerful in that it makes matching systems highly extensible and (with sufficient skills) customizable to a particular application domain [10,63]. 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 Sect. 5 we show that the untuned versions of several off-the-shelf matching systems achieve only 14–62% F-1 accuracy on four real-world domains. (The accuracy measure F-1 combines precision and recall, and is commonly used in recent schema matching work [22,23,41,46,62]; see Sect. 5 for more details.)

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 [73] to elaborate matches into semantic mappings (e.g., in form of SQL queries [73] which specify exact relationships between elements of different schemas, see a more detailed description in [19,67,73]). 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. The work [66] for example develops Kite, a system that enables keyword search over multiple heterogeneous relational databases. Kite first automatically finds semantic matches across the schemas of the databases, then leverages these matches to return the best ranked list of answers to the human user. In such scenarios, 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 [20], a task that is difficult even for learning experts [20]. 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.

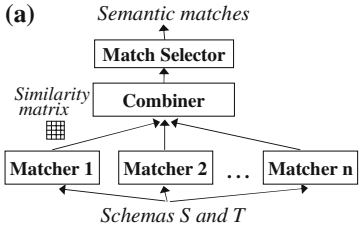
In this paper we describe eTuner, an approach to automatically tune schema matching systems. In developing eTuner, we address the following challenges:

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 1a shows a matching system which has $(n + 2)$ components: n matchers, one combiner, and one selector (Sect. 2 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 and only 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, an instance of S (i.e., data tuples that conform to S), and a schema matching system $\mathcal{M}$, how to tune $\mathcal{M}$ 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 [25,62].

Synthesize workload with known ground truth Tuning the system $\mathcal{M}$ mentioned above amounts to searching



(b)

EMPLOYEES

id	first-name	last-name	salary
1	Mike	Brown	40,000
6	Jean	Laup	60,000
3	Bill	Jones	70,000
4	Kevin	Bush	30,000

Fig. 1 An example of multi-component matching systems

for the "best" knob configuration for matches to S. The *quality* of a particular knob configuration of $\mathcal{M}$ 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 [22]). 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" schema S to generate a collection of synthetic schemas $S_1, S_2, \ldots, S_n$. For example, we can apply the rule "abbreviating a table name to the first three letters" to change the name EMPLOYEES of the table in Fig. 1b 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 or infinite, making exhaustive search impractical. Hence we implement a sequential, greedy approach, denoted *staged tuning*. Consider the matching system $\mathcal{M}$ in Fig. 1a. Here, we first tune each of the matchers $1, \ldots, n$ in isolation, then tune the combination of the combiner and the matchers, assuming that 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. Many different types of knob exist (e.g., discrete, continuous, set valued, ordered, etc.), each raising a different tuning challenge. We describe in detail how to address these challenges in Sect. 4.

In summary, we make the following concrete contributions:

- Establish that it is feasible to tune a matching system, automatically.
- Describe how to synthesize matching problems for which ground truth is known. Leverage such synthetic workload to estimate the quality of a matching system's result. For potential applications beyond the tuning context, see Sects. 6.4 and 7.
- Establish that staged tuning is a workable optimization solution for the problem of finding the "best" knob configuration without doing an exhaustive search. 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. 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 key contribution of this paper, we believe, is the demonstration that leveraging synthetic workload can provide a principled approach to tuning schema matching systems, a long-standing problem. However, this is just a first step. Significant works remain to fully realize the potentials of the approach, and are discussed in Sect. 7.

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

2 The match tuning problem

In this section, 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 [62]. 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) [19,35,72]) as future work. In this paper, we handle only relational schemas, and defer handling other data representations (e.g., XML schemas) to the future work.

2.1 Modeling 1–1 matching systems

We define a 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. The description of each component in L lists the set of knobs available for that component.

In what follows we elaborate on the above concepts, using the matching system in Fig. 2 as a running example. This system is a version of LSD, a learning-based multi-component matching system described in [24–26].

2.1.1 Library of matching components

Such a library contains the following four types of components, variants of which have often been proposed in the literature [29,62]:

- *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. (In the rest of the paper, we will use "matrix" as a shorthand for "similarity matrix".) Library L in Fig. 2a 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 [23,25]. The remaining three matchers exploit data instances [25].
- *Combiner* (*matrix* × ··· × *matrix* → *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 (Fig. 2a) [23,25,30]. More complex types of combiner include stacking (an ensemble learning method, employed for example in LSD [25]), decision tree [30], and elaborate (often hand-crafted) scripts (e.g., in Protoplasm [10]).
- *Constraint enforcer* (*matrix* × *constraints* → *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. The constraints can refer to those over the relational representation (e.g., $R.X \leq 10$), or over the domain of discourse. For example, library L in Fig. 2a has a single constraint enforcer, which exploits integrity constraints such as "lot-area cannot be smaller than house-area" [25].
- *Match Selector* (*matrix* → *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 [23]. More complex strategies include formulating the selection as an optimization problem over a weighted bipartite graph [46] (Fig. 2a).

2.1.2 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 this paper. (Section 5 gives a complete description of the four systems.)

LSD The execution graph of LSD [25] is shown in Fig. 2b 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.) The following example illustrates the working of LSD:

Example 1 Consider matching the schemas of data sources *realtor.com* and *homes.com* in Fig. 4. Suppose LSD consists of two matchers: name matcher and Naive Bayes matcher. Then it starts by applying these two matchers to compute similarity scores between the attributes of the schemas. Consider the two attributes agent-name of schema *realtor.com* and contact-agent of *homes.com*. The name matcher examines the similarity of their names ("contact agent" vs. "agent name") and outputs a similarity score, say 0.5. The similarity scores of all such attribute pairs are stored in a similarity matrix, shown in the upper half of Fig. 4b.

The Naive Bayes matcher examines the similarity of the attributes based on their data instances (e.g., "James Smith" and "Mike Doan" vs. "(206) 634 9435" and "(617) 335 4243", see Fig. 4a), and outputs another similarity matrix (see the lower half of Fig. 4b). In this particular example, notice that the Naive Bayes matcher assigns the low score of 0.1 to the two attributes agent-name and contact-agent, because their data instances are not similar to one another.

The combiner then merges the two similarity matrices. Suppose that the combiner simply takes the average of the corresponding scores, then it outputs a similarity matrix as shown in Fig. 4c. Notice that the combined

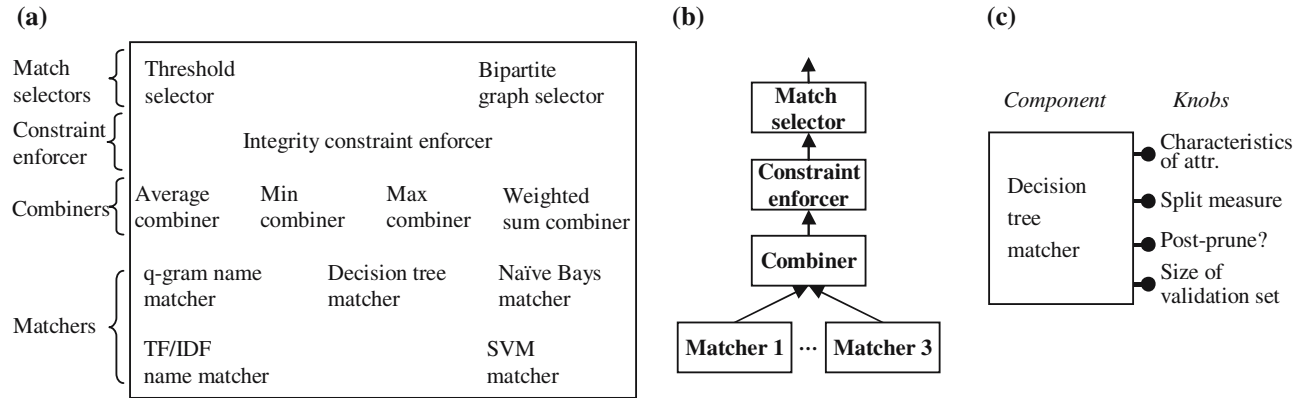


Fig. 2 The LSD system: **a** library of matching components, **b** execution graph, and **c** sample knobs

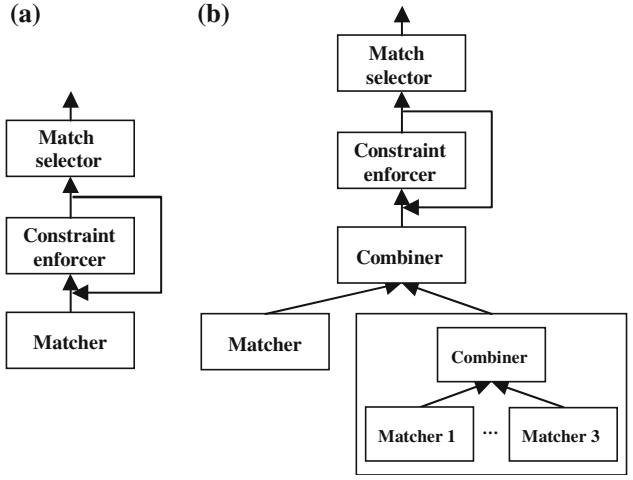


Fig. 3 Execution graphs of **a** the SimFlood matching system, and **b** the LSD–SF matching system

score of agent-name and contact-agent is now (0.5+0.1)/2 = 0.3. This matrix can be "unfolded" into a set of match predictions as shown in Fig. 4d. The first prediction (in the first row of the figure) for example states that area matches address with score 0.7, and description with score 0.3.

Notice that both area and comments are predicted to best match address, a wrong outcome. The constraint enforcer can address such situation. Given a domain integrity constraint, such as "only one attribute can match address" (see Fig. 4e), the enforcer will adjust the similarity scores to best reflect this constraint (see [25] for more detail). Finally, the match selector returns the matches with the highest score, as shown in Fig. 4f.

COMA and SimFlood Figure 1a shows the execution graph of the COMA system [23], which was the first to clearly articulate and embody the multi-component architecture (recently a more advanced version of COMA has become publicly available as COMA++ at http://dbs.uni-leipzig.de/Research/coma.html). Figure 3a shows the execution graph of the SimFlood matching system [46]. SimFlood employs a single matcher (a name matcher [46]), 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 [46]).

LSD–SF We can combine LSD and SimFlood to build a system called LSD–SF, whose execution graph is shown in Fig. 3b. Here, the LSD system (without the constraint enforcer and 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* [23,25,62]. 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 [23,25]). 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 Sect. 3.2), 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.

2.1.3 Tuning knobs

Knobs of the components We treat matching components as *black boxes*, but assume that each of them has a set of knobs that are "exposed" and can be adjusted.

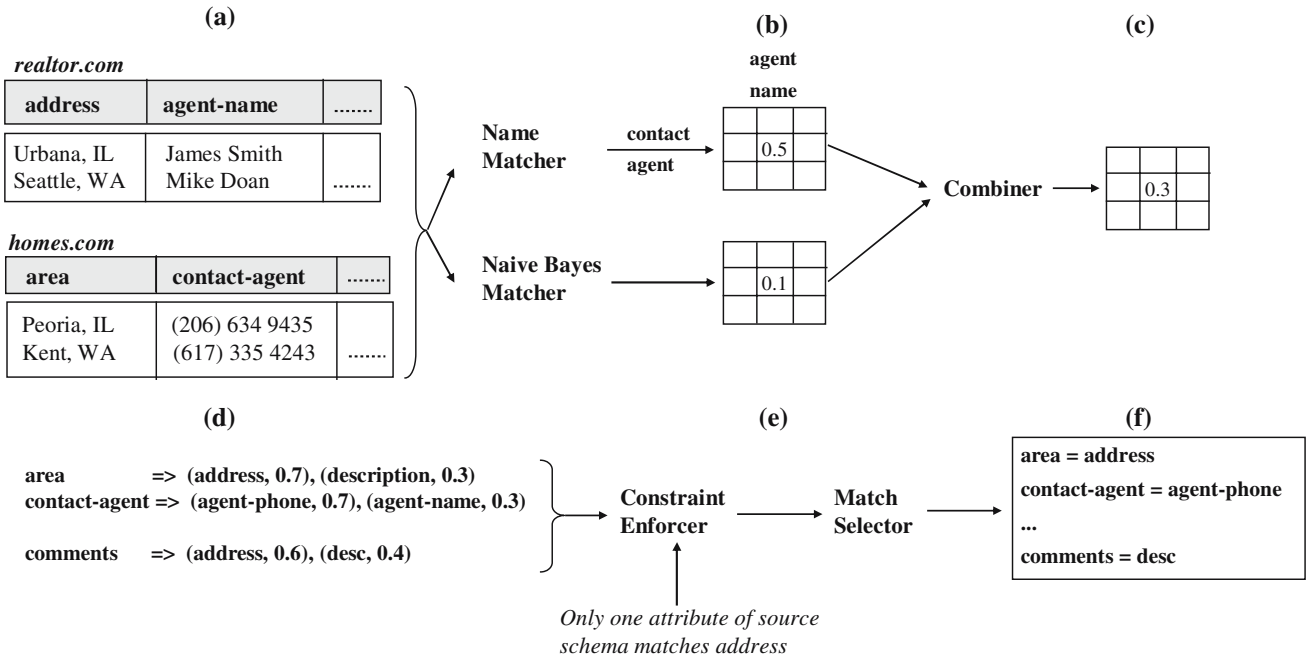


Fig. 4 An illustration of the working of the LSD system

Each knob is either (I) unordered discrete, (II) ordered discrete or continuous, or (III) set valued.

For example, Fig. 2c 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 the attribute (integer, string, etc.), the min, max, average value of the attribute , and so on (see [30,40] 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 [20]. Figure 8 lists a sample of features that matching systems commonly use (and hence are encoded in eTuner for tuning purposes).

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 decision tree matcher during the training process.

As another example, consider a combiner over n matchers $m_1, \ldots, m_n$, which merges the matchers' similarity matrices by computing weighted sums of the scores (e.g., [25]). Specifically, the combiner assigns to each matcher m_k a weight w_k, then compute the combined score:

$$\text{score}(a_i, a_j) = \sum_{k=1}^{n} w_k * \text{score}(m_k, a_i, a_j),$$

where $\text{score}(m_k, a_i, a_j)$ is the similarity score between attributes a_i and a_j as produced by matcher m_k. In this case, the combiner has n knobs, each of which must be set to reflect the weight w of the corresponding matcher.

Knobs of the execution graph For each node of the execution graph, we assume that the user (or eTuner) can plug in one of the several components from the library. Consider for example the node Matcher 1 of the execution graph in Fig. 2b. 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 (Fig. 2a).

Consequently, each node of an execution graph can be viewed as an 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 [10]) 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 [9,28,53], those in [30,41,52], and COMA++ and Protoplasm, industrial-strength matching systems under

development at the University of Leipzig[1] and Microsoft Research [10], respectively.

2.2 Tuning of matching systems

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

Definition 1 (Match 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*)* C^* *that maximizes the average utility over all schema pairs in the workload. Formally, let* $\mathcal{M}(C)$ *be the matching system* $\mathcal{M}$ *using the knob configuration* C*, and let* $\mathcal{C}$ *be the space of all knob configurations, as defined by* $\mathcal{M}$*, then*

$$C^* = argmax_{C \in \mathcal{C}} \left[\sum_{i=1}^{n} \mathcal{U}(\mathcal{M}(C), (S_i, T_i)) \right] \Big/ n, \qquad (1)$$

where $\mathcal{U}(\mathcal{M}(C), (S_i, T_i))$ *is the utility of applying* $\mathcal{M}(C)$ *to the schema pair* (S_i, T_i)*, and function* $argmax_a\ E(a)$ *returns the argument* a *that maximizes* $E(a)$*.*

In this paper, we restrict the above general problem. First, we use just one utility function $\mathcal{U}$ *accuracy*. Specifically we use F-1, a combination of precision and recall formalized in Sect. 5. F-1 is an accuracy measure commonly used in the field of Information Retrieval (IR). Since the problem of schema matching can be viewed as a variant of the IR problem (e.g., retrieve all and only matching attribute pairs vs. retrieve all and only relevant documents), the measure F-1 has also been often used in recent schema matching work [22,23,41,46,62].

As a second restriction, we tune $\mathcal{M}$ for the workload of matching a single schema S with all future schemas T_i (e.g., "future schemas" to be integrated with our warehouse, as mentioned earlier). This scenario arises in numerous contexts, including data integration and warehousing [25,62]. In the next two sections, we describe the eTuner solution to this problem.

[1] http://dbs.uni-leipzig.de/en/Research/coma.html.

3 The eTuner Approach

The eTuner architecture (see Fig. 5) 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 Fig. 5).

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

3.1 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 perturbation 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.

3.1.1 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 in

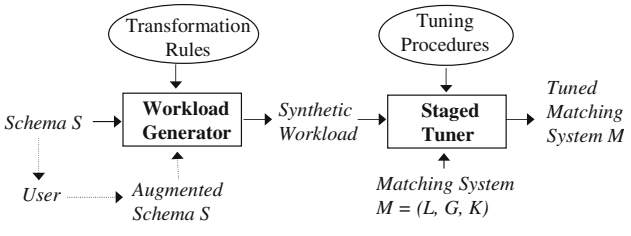


Fig. 5 The eTuner architecture

size, 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 [11,19] and thus may significantly bias the tuning process.

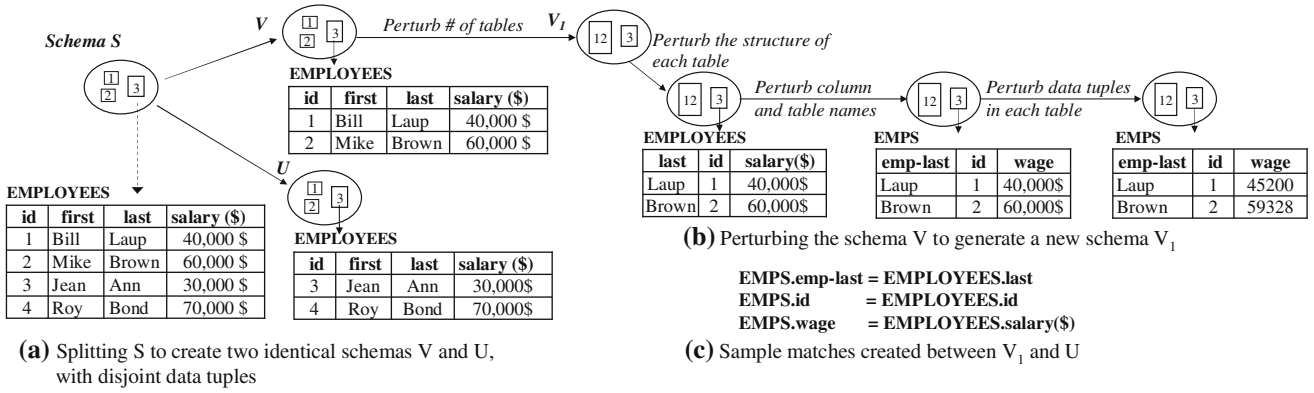
The above step is illustrated in Fig. 6a, which shows a schema S with three tables. The schemas V and U generated from S also have three tables with identical structures. However, table 3 of S, which we show in detail as table EMPLOYEES in Fig. 6a, has in effect being partitioned into two halves. Its first two tuples go to the corresponding table 3 of schema V, while the remaining two tuples go to table 3 of schema U.

More complex partitioning strategies are possible. For instance, we can try to partition table 3 of schema S in a way that preserves "joinability". Specifically, we try to partition so that the number of tuples in the equijoin between tables 2 and 3 of schema V will be roughly equal to the number of tuples in the equijoin between tables 2 and 3 of schema U, and so on. However, we experimented, and found that the above simple strategy of randomizing, then halving tuples in each table worked as well as these more complex strategies.

3.1.2 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 α_t times (currently set to two in our experiments). eTuner currently has three such rules. The first one randomly selects two tables that are joinable via a key-foreign key constraint, and *merges* them based on that join path to create a new table. The second rule randomly selects and *splits* a table into two. When splitting the table, it adds to each half a column id and populates these columns with values such that the two halves can be joined via these id columns to recover the original table. The third rule does nothing (i.e., leaves the tables "as is").
  As an example, after applying the rules, schema V at the top of Fig. 6a, 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, exactly α_c times (currently set to four). eTuner has four such rules. The first one merges two columns. Currently, two columns can be merged 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. The fourth rule does nothing.
  Continuing with our example, in Fig. 6b, 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 [19,42,62]. Examples include doing nothing, abbreviating to the first three or four characters, dropping all vowels, replacing a 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 α_n times (currently set to two).
  In Fig. 6b, 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 emp-last. 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", "adding two more fractional digits to make numbers precise", "converting the unit of numbers"(e.g., from meters to feet), "changing the format of area codes of phone numbers", "inserting hyphens into phone numbers", and "changing the format of dates"(e.g., from 12/4 to Dec 4). For each column, the generator applies such rules α_f times (currently set to two).



(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 V_1

(c) Sample matches created between V_1 and U

Fig. 6 Perturbing schema S to generate two schemas U and V_1 and the correct matches between them

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 the range $\pm$[10,100]%. Let the new mean and variance be μ'_c and ${\sigma'_c}^2$, respectively. Then each value x is now generated according to the normal distribution:

$$\text{prob}_c(x) = \frac{1}{\sigma'_c\sqrt{2\pi}} \exp\left(\frac{-(x-\mu'_c)^2}{2{\sigma'_c}^2}\right) \quad (2)$$

If the data instances of column c are textual (e.g., house description), they are perturbed in the same way, with some minor differences. Specifically, first the generator tokenizes all instances of column c, then compiles the vocabulary Q of all tokens. Second, it computes the length (i.e., the number of tokens) of each data instance. Assuming that the length is generated according to a normal distribution with mean μ_c and variance ${\sigma_c}^2$, the generator perturbs μ_c and ${\sigma_c}^2$ by random amounts in the range $\pm$[10,100]% to generate new mean μ'_c and variance ${\sigma'_c}^2$. The each new textual data instance x is now generated as follows: first, the length of x is generated according to the normal distribution with mean μ'_c and variance ${\sigma'_c}^2$; second, tokens for x are taken randomly from the vocabulary Q.

We note that while the above data perturbation methods appear to work well in our experiments, more sophisticated perturbation methods are possible, and finding a (near) optimal one is an interesting research problem.

Continuing with our example, consider column wage of Table EMPS in Fig. 6b (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".

3.1.3 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 6c 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 $\mathcal{M}$.

Figure 7 gives the pseudo code of the workload generator.

3.2 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

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_f, T_v be table-, column-, name-, format- and value-transformation rules in repository T
1. Split schema S into U, V
  a. Let $U = S$; Let $V = S$
  b. 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, … V_n from V
  a. Let $V_i = V$
  b. Perturb number of tables in V_i using rules in T_t
  c. Perturb the structure of each table in V_i using rules in T_c
  d. Foreach name n_k in schema V_i do change n_k using random rules in T_n
  e. Foreach column c_j in V_i do
    Perturb format of c_j with rules in T_f
    Let d_{cj} = data associated to c_j in D_v
    Let σ_{cj}^2 = variance(d_{cj}); Let 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 pertuebed σ_{cj}^2 and μ_{cj}
    Perturb format of each generated data values using rules in T_f
3. Foreach (U, V_i) do generate its correct match set Ω_i
  a. Let $\Omega_i = \emptyset$
  b. 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 $W = \{(U, V_1, \Omega_1), (U, V_2, \Omega_2), \ldots (U, V_n, \Omega_n)\}$

Fig. 7 High-level description of the workload generator

emp-phone = phone-2 as also correct (since emp-phone is *not* derived from phone-2, see Sect. 3.1.3). 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. As another example, (depending on application) a user may also judge first-name and last-name 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 match $b = a_{ij}$ is correct for some j in the range $[1, n]$, then for all k in the range $[1, n]$ such that $k \neq j$, match $b = a_{ik}$ is 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 5 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.

4 Tuning with the synthetic workload

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

4.1 Staged tuning

Our objective is to find a knob configuration of $\mathcal{M}$ that maximizes the average accuracy over $\mathcal{W}$ (see Definition 1). We can view this problem as a *search* in the space of possible knob configurations. However, exhaustive search is impractical, since the configuration space is usually huge. For example, the LSD system described in Sect. 5 has 21 knobs, with at least two possible values per knob, resulting in at least 2^{21} configurations.

To address this problem, we propose a *staged, greedy* tuning approach. Assume that the execution graph of $\mathcal{M}$ has k levels. We first tune each node at the bottom, i.e., at the kth level, in isolation. Next, we tune subsystems that consist of nodes at the $(k-1)$th and kth levels. While tuning such subsystems (described in detail in the following subsection), we assume that the nodes at the

kth level have been tuned, so their knob values are fixed, and we only need to tune knobs at level $(k-1)$. 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 Fig. 2b. 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 that 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 that 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 5 shows that while not guaranteeing to find the optimal knob configuration, staged tuning still outperforms currently possible tuning methods.

4.2 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 Sect. 2.1.3 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., [25,30]). 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 detail in Sect. 4.3 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 [25]. They can be tuned using linear or logistic regression (over the synthetic workload) [25]. 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.

4.3 Tuning to select features

The only thing that remains is to 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 Sect. 2.1.3 that a matcher often transforms each schema attribute into a feature vector, then uses these 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., [9,23,25,28,30,40,42,46]). Figure 8 shows 16 sample features. The objective of tuning is then to select from the set F of all enumerated features a subset F^* that best assists 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 greedy selection method called *wrapper* [20], 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. Figure 9 describes the wrapper method [20], as adapted to our context.

However, the wrapper method can still be very expensive. For example, 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 [20] and shown in Fig. 12) 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 algorithms [20]. 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 [25,28,30,40] and also by many rule-based methods (e.g., [23,42,46]). However, many learning-based (e.g., Naive Bayes, SVM) as well as IR-based matching methods (e.g., [18,25]) 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.

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

Fig. 8 Sixteen sample features that eTuner uses in selecting a best set of features for the schema attributes. CV stands for "coefficient of variation" and SD for "standard deviation"

Input: set of features F, subsystem S
Output: the best set of features
Let $P = \emptyset$ be the current set of selected features and
$Q = F$ be the set of unseen features
1. Foreach feature f in Q do
   a. Let $P' = \{f\} \cup P$
   b. Run S using P' and
      let A_f be the matching accuracy
2. Let f^* be the f having the highest A_f,
   $P = \{f^*\} \cup P$ and $Q = Q \setminus \{f^*\}$
3. If A_f is not converged, go to Step 1
   Else, return P

Fig. 9 High-level description of the wrapper feature selection method (called step-wise selection in [20])

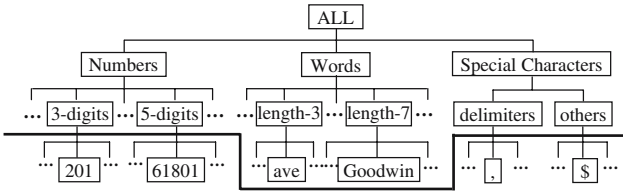


Fig. 10 An example taxonomy for the Naive Bayes matcher

Figure 10 shows a sample taxonomy of features over text for eTuner (adapted from [12]). A line cutting across this taxonomy represents a selected feature set. Consider for example the thick line in the figure. It states that (a) all numbers are abstracted into 1-digit, 2-digits, etc, (b) all words can be treated as features, and (c) all special characters are abstracted to delimiters and others. 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 (see Fig. 11), starting from the feature set at the bottom of the taxonomy (the one with no abstracted features). In each iteration we add a new abstraction (at a higher level of the taxonomy) if it leads to increased accuracy, as measured by applying the matcher to the synthetic workload. Since the number of abstraction is relatively small, the feature selection step is fast.

Input: set of types of columns F, taxonomy T,
subsystem S
Output: set of abstraction levels of F
Let $I = \{i_{10}, \ldots, i_{|F|0}\}$, where i_{jk} represents the k^{th} abstraction level of type j
1. Run S using I and let A be the matching accuracy
2. For $l := 1$ to L, where L is the height of T
   Foreach type f in F do
      Let $I' = \{i_{fl}\} \cup I$
      Run S using I' and let A' be the matching accuracy
      If $A' > A$ then let $I = \{i_{fl}\} \cup I$
3. Return I

Fig. 11 High-level description of the wrapper method, as adapted to feature selection for text-based matchers

Input: set of features F, schema S, number of examples N,
number of iteration L, number of near neighbor examples to compute B,
threshold for filtering W
Output: set of relevant features
1. Preprocessing
a. Let $E = \varnothing$ be the set of examples, $<f_1, \ldots, f_{|F|}, c>$,
where $f_1, \ldots, f_{|F|}$ are values for features and c is a column name in S
b. For $n:=1$ to N
Randomly pick R data instances
Foreach column c in S do
Compute a feature vector, $<f_1, .., f_{|F|}>$ using R
$E = \{<f_1, \ldots, f_{|F|}, c>\} \cup E$
2. Foreach feature f in F do
Let $w_f = 0.0$ be the weight for feature f
3. Foreach column c in S do
Let p_c be the fraction of examples belonging to c in E
4. For $i:=1$ to L do
a. Randomly pick an example e
b. Let H be a set of B examples, e', nearest to e, where $e'_c = e_c$
c. Foreach class $c' (\neq e_c)$ in S do
Let M_c be a set of examples, e'', nearest to e, where $e''_c = c'$
d. Foreach feature f in F do
Let $w_f = w_f - 1/LB\ [\Sigma_{[e' \text{ in } H]} \delta(e'_f, e_f)] + \Sigma_{[c \neq e_c]}\ [[P_{ec} / ((1-p_c)LB)][\Sigma_{[e'' \text{ in } M_c]} \delta(e''_f, e_f)]]$,
where δ is the dot product of two vectors
5. Let $F^* = \varnothing$ be the set of relevant features
Foreach feature f in F do
If $w_f \geq W$, then $F^* = \{f\} \cup F^*$
6. Return F^*

Fig. 12 High-level description of the Relief-F algorithm [20], as adapted to feature selection in eTuner

5 Empirical evaluation

We have evaluated eTuner over four matching systems applied to four real-world domains. In this section, we first describe the domains, each of which consists of a set of data sources, the matching systems, and our experimental settings.

Next, we examine five manual and semi-automatic tuning methods that can be used instead of eTuner. Specifically, we consider the following methods (described in detail in Sects. 5.2– 5.4): (1) Applying the off-the-shelf matching systems "as is", that is, no tuning. (2) Tuning each system independently of any domain, in effect imitating a vendor tuning a system before release. (3) "Quick and dirty" tuning, by tweaking a few knobs, examining the output of a matching system, then adjusting the knobs again. (4) Tuning a matching system once for each domain, taking into account the characteristics of data sources in the domain. (5) Tuning a matching system once for each data source, by leveraging known matches from the schema of that data source to several other schemas. Our results show that source-dependent tuning [method (5)] is most labor consuming, but also yields the highest average matching accuracy.

We then examine tuning with eTuner. Our results show that when using matching systems tuned with eTuner, we improve matching accuracy in 14 out of 16 matching scenarios (described in Sect. 5.2), by 1–15%, compared to using matching systems tuned with the source-dependent tuning method. eTuner yields lower matching accuracy in only two cases, by 2%. In addition to higher accuracy in most cases, eTuner also incurs relative little user effort, which consists mainly of "hooking" eTuner up with the knobs of a matching system. In contrast, source-dependent tuning is far more labor intensive. Finally, we show that eTuner is robust to changes in the synthetic workload, that it can exploit prior match results whenever available, and that the synthetic workload and the staged tuner perform well compared to the "ideal workload" and to exhaustive search. Overall, the experimental results demonstrate the promise of the eTuner approach.

5.1 Experimental settings

Domains We obtained publicly available schemas in four domains. The schemas have been used in recent

(a)

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

(b)

Agent-Details

id	first_name	last_name	phone_number	…
1	Janet	Simpson	693-3579	…
2	Renee	Thomas	598-2553	…
3	Emily	Prowell	596-9446	…
…	…	…	…	…

House-Details

city	state	price	#bedroom	agent_id	…
Bay	TX	$54,500	1	1	…
Bay	TX	$194,500	3	2	…
Weatherford	TX	$76,500	3	3	…
…	…	…	…	…	…

(c)

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

(d)

Books

Title	Author	ISBN	Category	…
Great Expections	Charles Dickens	12345678	Fiction	…
…	…	…	…	…

Music

AlubumName	Artist	Price	Genre	…
Nirvana	Nirvana	19	Rock	…
…	…	…	…	…

Warehouse

ID	Location	Manager	…
2	321 Phony St. FakeVille, WA 98235	Jack Daniel	…
…	…	…	…

Availability

WarehouseID	Title	ID	Quantity
1	Adore	17589724	475
…	…	…	…

Fig. 13 **a** Real world domains, **b** matching systems for our experiments, **c** a sample schema from Real Estate, and **d** a sample schema from Inventory

schema matching experiments [19,25,41]. The domains have varying numbers of schemas (2–10) and diverse schema sizes (10–50 attributes per schema, see Fig. 13a). 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. Figure 13c,d shows sample schemas in Real Estate and Inventory.

Matching systems Figure 13b 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 [25] to match XML DTDs. We adapted it to relational schemas. The SimFlood system [46] was downloaded from the Web.[2] The COMA system was described in [23]. Since we did not have access to COMA, we implemented a version of it called iCOMA. The iCOMA library includes all components described in [23], except the hybrid and reuse matchers. Virtually all matchers of COMA exploit only schema related information. We added the decision tree matcher to the library, to also exploit data instances. Finally, we combined LSD and SimFlood (as described in Sect. 2), to obtain LSD–SF, the fourth matching system. Figure 13b shows that the systems have 6–17 components, with 8–21 knobs. We now describe each matching system in detail.

- LSD: This system has six matchers, six combiners, one constraint enforcer, and two match selectors (Fig. 14). Both the decision tree matcher [30] and the Naive Bayes matcher [25,50] exploit data instances. All knobs for the decision tree matcher have been discussed in Sect. 2.1.3 (and see [50] for further detail). The Naive Bayes matcher has one knob, abstraction-level, for choosing the abstraction level of text tokens, as described in Sect. 4.3.
  The name matcher, the edit distance matcher, and the q-gram matcher exploit names of attributes and tables. The name matcher is similar to the name-based evaluator in [19]. It has one knob for choosing a tokenizing rule. Currently, there are five rules: use each word as a token (*no-stemming*), use Porter's stemmer (*stemming*), and generate q-gram tokens after using Porter's stemmer (*stemming-2gram*, *stemming-3gram*, *stemming-4gram*). The edit distance matcher computes the number of edit operations necessary to transform one name into another. The q-gram matcher compares names based on their associated sets of q-grams, i.e., sequences of q characters. The q-gram matcher has a knob, gram-size, to select the value of q among 2, 3, and 4.

[2] http://www-db.stanford.edu/~melnik/mm/sfa.

Fig. 14 The library of matching components of LSD

Component type	Component name	Knob name	Knob Type
Matcher	Decision tree matcher	characteristics-of-attr	type III
		split-measure	type I
		post-prune?	type I
		size-of-validation-set	type II
	Naïve Bayes matcher	abstraction-level	type I
	Name matcher	stemming-algorithm	type I
	Edit distance matcher		
	Q-gram matcher	gram-size	type II
	Common instance matcher		
Combiner	Average combiner		
	Min combiner		
	Max combiner		
	Linear regression combiner		
	Decision tree combiner	split-measure	type I
		post-prune?	type I
		size-of-validation-set	type II
	Column-based decision tree combiner	split-measure	type I
		post-prune?	type I
		size-of-validation-set	type II
Constraint Enforcer	Integrity constraint enforcer		
Match Selector	Threshold-based selector	threshold	type II
	Window-based selector	window-size	type II

The last matcher, common instance matcher, compares two attributes based on the number of instances that they share.

The average, min, and max combiners take the (respectively) average, min, and max of the similarity scores. The linear regression combiner learns a weight for each matcher and combines the similarity matrices using the learned weights. The decision tree combiner is similar to the decision tree matcher, except that for each pair of attributes the feature set is a set of similarity scores generated by matchers. Since the feature set is fixed, we only need to tune three knobs: split-measure, post-prune?, and size-of-validation-set. The column-based decision tree combiner is just like the decision tree combiner, but it constructs a decision tree for each target attribute.

The integrity constraint enforcer and the threshold-based selector are described in Sect. 2.1.1. The threshold-based selector has a knob for setting the threshold value. LSD also has a window-based selector. For each target attribute, it selects a pair of attributes having the highest similarity score and pairs of attributes whose scores are within the boundary of the window size with best score. Its knob is called window-size. The total number of knobs for LSD is 21 (counting also knobs of the execution graph, such as whether a certain matcher should be used).

- iCOMA: Fig. 15 lists components of iCOMA: ten matchers, four combiners, and two match selectors. Most components are the same as those of LSD, except for five new matchers and one new combiner. The affix, soundex, and synonym matcher exploit attribute names. The data type matcher exploits data types and the user feedback matcher exploits user-specified matches. These matchers are fully described in [23].

  The weighted sum combiner computes the weighted sum of similarity matrices using the weight for each matcher. Since iCOMA has five matcher nodes in its execution graph, the weighted sum combiner has five knobs, each of which must be set to reflect the weight of the corresponding matcher. Thus, iCOMA has a total of 20 knobs.
- SimFlood: Figure 16 lists the components of SimFlood: three matchers, one constraint enforcer, and two match selectors. The three new components are the exact string matcher, the SF-join constraint

Fig. 15 The library of matching components of iCOMA

Component type	Component name	Knob name	Knob Type
Matcher	Decision tree matcher	characteristics-of-attr	type III
		split-measure	type I
		post-prune?	type I
		size-of-validation-set	type II
	Name matcher	stemming-algorithm	type I
	Edit distance matcher		
	Q-gram matcher	gram-size	type II
	Common instance matcher		
	Affix matcher		
	Soundex matcher		
	Synonym matcher		
	Data type matcher		
	User feedback matcher		
Combiner	Average combiner		
	Min combiner		
	Max combiner		
	Weighted sum combiner	A knob for each matcher	type II
Match Selector	Threshold-based selector	threshold	type II
	Window-based selector	window-size	type II

Fig. 16 The library of matching components of SimFlood

Component type	Component name	Knob name	Knob Type
Matcher	Exact string matcher		
	Edit distance matcher		
	Q-gram matcher	gram-size	type II
Constraint Enforcer	SF-join constraint enforcer	PropagationCoefficient	type I
		FixpointFormula	type I
Match Selector	Threshold-based selector	threshold	type II
	Type&threshold-based selector	threshold	type II

enforcer, and the type and threshold-based selector. The exact string matcher returns 1 if both attribute names are same. Otherwise, it returns 0 as a similarity score.

The SF-join constraint enforcer exploits the heuristic "two attributes are likely to match if their neighbors match". As described in [46], it has two knobs – PropagationCoefficient and FixpointFormula. The PropagationCoefficient knob chooses a rule for computing the propagation coefficients and the FixpointFormula selects one variation of the fixpoint formula (and see [46] for further detail).

The type and threshold-based selector is similar to the threshold-based selector discussed earlier, but it also considers the type of an attribute; if attributes in a candidate match have different types, the selector discards this candidate match. This selector has one knob, and SimFlood has a total of eight knobs.

- LSD–SF: Figure 17 lists the components of LSD–SF seven matchers, seven combiners, one constraint enforcer, and two match selectors. There is only one new matching component: the LSD–SF combiner. This combiner merges the similarity matrices from the matcher that originally comes from SF (the left matcher in Figure 3b), and LSD (the big box at the lower-right corner in Fig. 3b). It has one knob called which-matcher?, for selecting one of these two matchers. Since we assume that the "LSD" matcher is tuned already, LSD–SF has only 10 knobs.

Fig. 17 The library of matching components of LSD–SF

Component type	Component name	Knob name	Knob Type
Matcher	Decision tree matcher	characteristics-of-attr	type III
		split-measure	type I
		post-prune?	type I
		size-of-validation-set	type II
	Naïve Bayes matcher	abstraction-level	type I
	Name matcher	stemming-algorithm	type I
	Edit distance matcher		
	Q-gram matcher	gram-size	type II
	Common instance matcher		
	Exact string matcher		
Combiner	Average combiner		
	Min combiner		
	Max combiner		
	Linear regression combiner		
	Decision tree combiner	split-measure	type I
		post-prune?	type I
		size-of-validation-set	type II
	Column-based decision tree combiner	split-measure	type I
		post-prune?	type I
		size-of-validation-set	type II
	LSD-SF combiner	which-matcher?	Type I
Constraint Enforcer	SF-join constraint enforcer	PropagationCoefficient	type I
		FixpointFormula	type I
Match Selector	Threshold-based selector	threshold	type II
	Type&threshold-based selector	threshold	type II

Experimental methodology For each of the four domains described in Fig. 13, 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 two 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 [22,23,41,46,62], 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 objective of tuning is to find the knob configuration that maximizes F_1 score.

5.2 The need for tuning

We begin by demonstrating the need for tuning, using Fig. 18a–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 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 Fig. 18a), 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

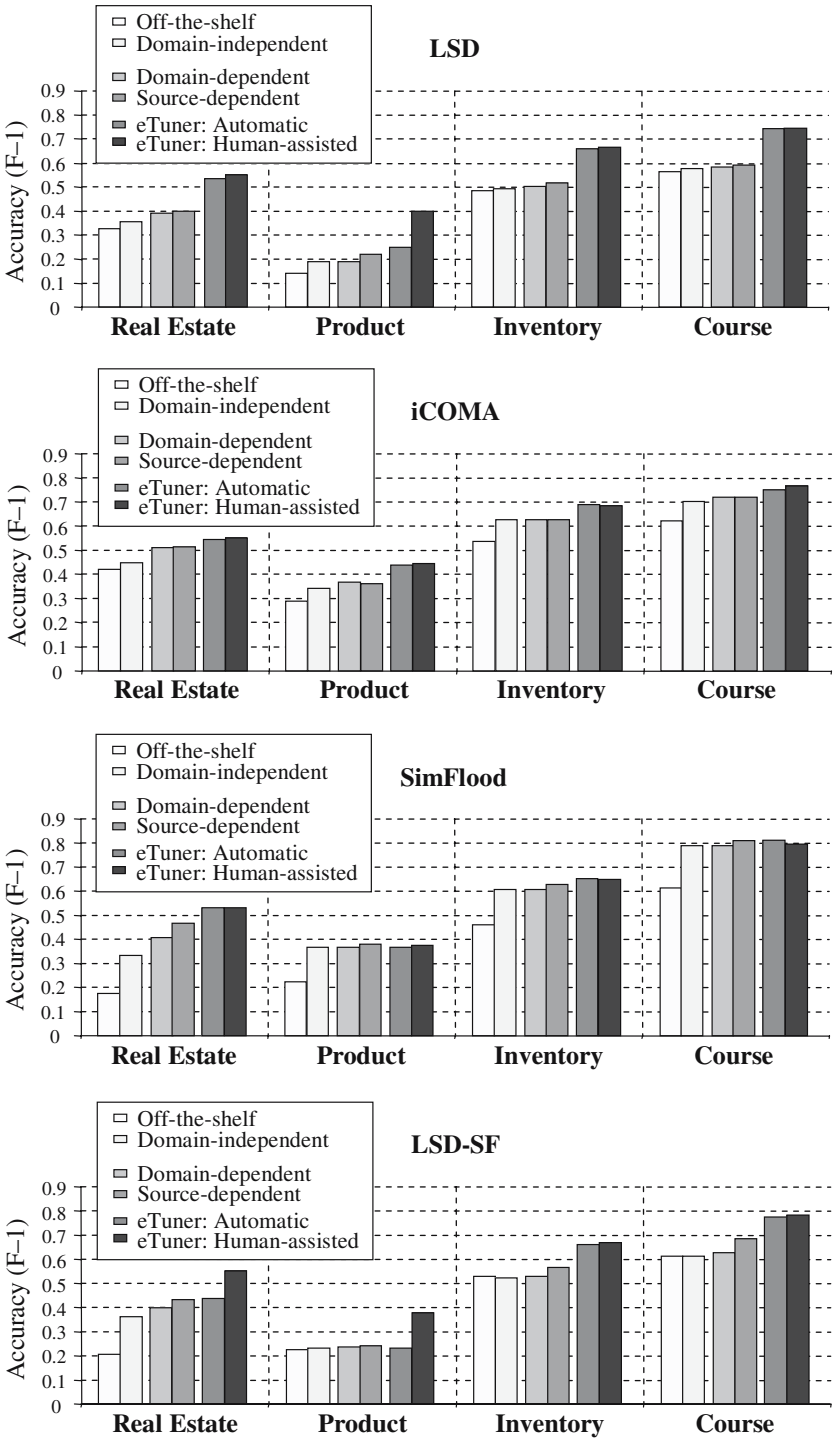


Fig. 18 Matching accuracy for **a** LSD, **b** iCOMA, **c** SimFlood, and **d** LSD–SF

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.

5.3 "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 that they can tune effectively. Consider for example the decision tree matcher described in Sect. 2.1.3. 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 Sect. 2.1.3) 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%. 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.

5.4 Domain- and 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 [25].) The third bar in each group (Fig. 18a–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 (Fig. 18a–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%.

5.5 Tuning with eTuner

The fifth bar (second bar from the right) of each group (Fig. 18a–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 two 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, which achieve lower accuracy and incur a significantly higher labor cost.

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 (Figs. 18a–d) shows the accuracy of eTuner with user-assisted workload creation (Sect. 3.2), with users being volunteer graduate students. The average number of groupings specified by a user for product, real estate, inventory and course domain is respectively 9, 3.5, 2, and 2. 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 potential benefits of user assistance in tuning.

5.6 Sensitivity analysis

Synthetic workload Figure 19a 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 Sect. 3.1), 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 [19] contains a far richer set of perturbation rules than eTuner. Nevertheless, its accuracy on 1–1 matching (as reported in [19] on a different domain) is only 62–71%.

Exploiting prior match results Figure 19b 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 [23, 25]. 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 Sect. 5.4.

Runtime complexity Our unoptimized version of eTuner took under 30 min 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 [63] as well as optimization within the tuning module, such as reusing results across matching steps and more efficient, specialized procedures for knob tuning.

5.7 Additional experiments

Finally, we examine the comparative accuracy of the synthetic workload and the staged tuner. Clearly, the *ideal workload* on which to tune a matching system would be the *actual workload*, that is, the set of all future schemas, together with the correct matches from these schemas to the schema S. In practice, of course, this workload is not available for tuning purposes. Still, we want to know how "good" the current synthetic workload is, that is, how the matching accuracy based on it would compare to that based on the actual workload, which forms a kind of "ceiling" on matching accuracy.

Figure 20a–d show the results for the four matching systems, respectively. Each figure comprises four groups, showing the results for the four domains, respectively. The first two bars in each group are reproduced from Fig. 18. They show the accuracies of the matching system, as tuned with eTuner automatically and via human assistance. The last bar in each group shows the accuracy of the matching system, as tuned automatically with eTuner *on the actual workload*.

The results show that the accuracy with current synthetic workloads is already within 10% of that with the actual workload (except for two cases, LSD/Inventory and LSD-SF/Inventory, where it is within 19%). These results suggest that the current synthetic workloads already perform quite reasonably, though there is still some room for improvement.

In the next experiment, we tuned LSD and iCOMA on synthetic workloads but instead of doing a staged search as discussed in Sect. 4, we conducted a search as exhaustively as possible. Specifically, if the search space is finite, then we did carry out an exhaustive search. If the search space is infinite, due to continuous-value knobs, then we discretized all such knobs, to obtain a finite search space. Our objective is to examine how close is the knob configuration found by staged tuning to the optimal one (as found by exhaustive search).

Table 1 shows the results of this experiment, for LSD and iCOMA on Inventory and Course. Each numeric cell of this table lists the accuracy obtained in the corresponding context. The accuracies for the knob configurations obtained via exhaustive search are listed in bold font, under "Ceiling 2". Interestingly, the accuracy with the staged tuner is within 3% of that with exhaustive search for all cases. The results thus suggest that for these experimental settings staged tuning finds close-to-optimal knob configurations.

6 Related work

In this section, we first discuss related work in schema matching, then the implications of the current research beyond the schema matching context.

6.1 Schema matching techniques

Schema matching has received increasing attention over the past two decades (see [4,5,21,24,29,55,62] for recent surveys). A wealth of matching techniques has been developed. The techniques fall roughly into two groups: rule-based and learning-based solutions (though several techniques that leverage ideas from the fields of infor-

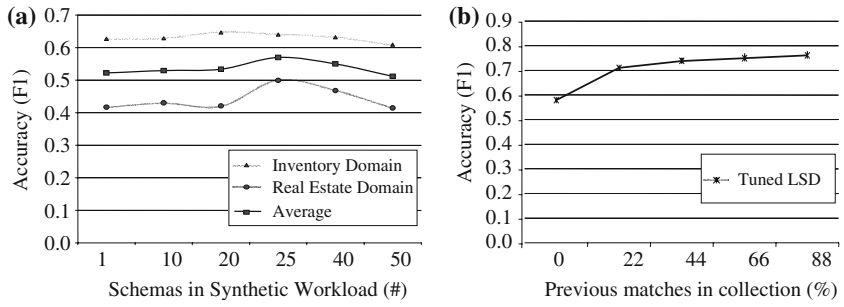


Fig. 19 Changes in the matching accuracy with respect to **a** size of the synthetic workload, and **b** the number of prior matched schema pairs in the workload

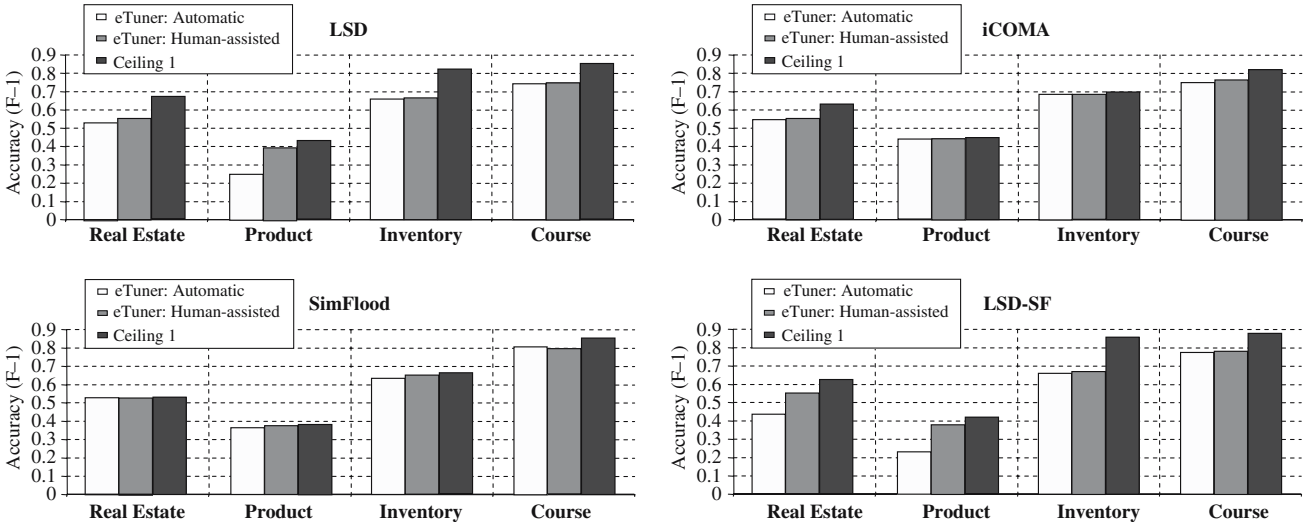


Fig. 20 The performance of the workload generator

Table 1 The performance of the staged tuner

	LSD			iCOMA		
	eTuner: automatic	eTuner: human-assisted	Ceiling 2 automatic	eTuner: human-assisted	eTuner:	Ceiling 2
Inventory	0.66	0.664	**0.669**	0.688	0.685	**0.699**
Course	0.743	0.746	**0.766**	0.752	0.766	**0.767**

mation retrieval and information theory have also been developed [18,36]).

Rule-based solutions Many of the early as well as current matching solutions employ hand-crafted rules to match schemas [14,42,46,49,51,57].

In general, hand-crafted rules exploit *schema information* such as element names, data types, structures, number of subelements, and integrity constraints. A broad variety of rules have been considered. For example, the TranScm system [49] employs rules such as "two elements match if they have the same name (allowing synonyms) and the same number of subelements". The DIKE system [56–58] computes the similarity between two schema elements based on the similarity of the characteristics of the elements and the similarity of related elements. The ARTEMIS and the related MOMIS [7,14] system compute the similarity of schema elements as a weighted sum of the similarities of names, data types, and substructures. The CUPID system [42] employs rules that categorize elements based on names, data types, and domains. Rules therefore tend to be domain-independent, but can be tailored to fit a certain domain. Domain-specific rules can also be crafted.

Rule-based techniques provide several benefits. First, they are relatively inexpensive and do not require training as in learning-based techniques. Second, they typically operate only on schemas (not on data instances), and hence are fairly fast. Third, they can work very well in certain types of applications and for domain representations that are amenable to rules [53]. Finally, rules can provide a quick and concise method to capture

valuable user knowledge about the domain. For example, the user can write regular expressions that encode times or phone numbers, or quickly compile a collection of county names or zip codes that help recognize those types of entities.

The main drawback of rule-based techniques is that they cannot exploit data instances effectively, even though the instances can encode a wealth of information (e.g., value format, distribution, frequently occurring words in the attribute values, and so on) that would greatly aid the matching process. In many cases, effective matching rules are simply too difficult to hand craft. For example, it is not clear how to hand craft rules that distinguish between "movie description" and "user comments on the movies", both being long textual paragraphs. In contrast, learning methods such as Naive Bayes can easily construct "probabilistic rules" that distinguish the two with high accuracy, based on the frequency of words in the paragraphs.

Another drawback is that rule-based methods cannot exploit previous matching efforts to assist in the current ones. Thus, in a sense, systems that rely solely on rule-based techniques have difficulties learning from the past, to improve over time. The above reasons have motivated the development of learning based matching solutions.

Learning-based solutions Many such solutions have been proposed in the past decade, e.g., [8,9,11,18,19,25,30,40,52]. The solutions have considered a variety of learning techniques and exploited both schema and data information. For example, the SemInt system [40] uses a neural-network learning approach. It matches schema elements based on attribute specifications (e.g, data types, scales, the existence of constraints) and statistics of data content (e.g., maximum, minimum, average, and variance). The LSD system [25] employs Naive Bayes over data instances, and develops a novel learning solution to exploit the hierarchical nature of XML data. The iMAP system [19] (and also the ILA and HICAL systems developed in the AI community [60,65]) matches the schemas of two sources by analyzing the description of objects that are found in both sources. The Autoplex and Automatch systems [8,9] use a Naive Bayes learning approach that exploits data instances to match elements.

In the past 5 years, there is also a growing realization that schema- and data-related evidence in two schemas being matched often is inadequate for the matching process. Hence, several works have advocated learning from the *external evidence* beyond the two current schemas. Several types of external evidence have been considered. Some recent works advocate exploiting *past matches* [9,10,23,25,30,62]. The key idea is that a matching tool must be able to learn from the past matches, to predict successfully matches for subsequent, unseen matching scenarios.

The work [41] goes further and describes how to exploit a *corpus of schemas and matches* in the domain. This scenario arises, for example, when we try to exploit the schemas of numerous real-estate sources on the Web, to help in matching two specific real-estate source schemas. In a related direction, the works [34,71] describe settings where one must *match multiple schemas* all at once. Here the knowledge gleaned from each matching pair can help match other pairs, as a result we can obtain better accuracy than just matching a pair in isolation. The work [44,45] discusses how to learn from a *corpus of users* to assist schema matching in data integration contexts. The basic idea is to ask the *users* of a data integration system to "pay" for using it by answering relatively simple questions, then use those answers to further build the system, including matching the schemas of the data sources in the system. This way, an enormous burden of schema matching is lifted from the system builder and spreads "thinly" over a mass of users.

6.2 Multi-component matching solutions

The synergistic nature of matching techniques (e.g., based on rules, learning, information retrieval, information theory, graph algorithms, etc.) suggests that an effective matching solution should employ many of the techniques, each on the types of information that it can effectively exploit. To this end, several recent works [10,19,23,25,30,63] have described a system architecture that employs multiple modules called *matchers*, each of which exploits well a certain type of information to predict matches. The system then combines the predictions of the matchers to arrive at a final prediction for matches. Each matcher can employ one or a set of matching techniques as described earlier (e.g., hand-crafted rules, learning methods, IR-based ones). Combining the predictions of matchers can be manually specified [10,23] or automated to some extent using learning techniques [25].

Besides being able to exploit multiple types of information, the multi-matcher architecture has the advantage of being highly modular and can be easily customized to a new application domain. It is also extensible in that new, more efficient matchers could be easily added when they become available. A recent work [19] also shows that the above solution architecture can be extended successfully to handle complex matches.

Incorporating domain constraints It was recognized early on that domain integrity constraints and heuristics provide valuable information for matching purposes. Hence, almost all matching solutions exploit some forms of this type of knowledge.

Most works exploit integrity constraints in matching schema elements *locally*. For example, many works match two elements if they participate in similar constraints. The main problem with this scheme is that it cannot exploit "global" constraints and heuristics that relate the matching of *multiple* elements (e.g., "at most one element matches house-address"). To address this problem, several recent works [25,27,42,46] have advocated moving the handling of constraints to *after* the matchers. This way, the constraint handling framework can exploit "global" constraints and is highly extensible to new types of constraints.

While integrity constraints constitute *domain-specific* information (e.g., house-id is a key for house listings), heuristic knowledge makes *general* statements about how the matching of elements relate to each other. A well-known example of a heuristic is "two nodes match if their neighbors also match", variations of which have been exploited in many systems (e.g., [42,46,49,54]). The common scheme is to *iteratively* change the matching of a node based on those of its neighbors. The iteration is carried out one or twice, or until some convergence criteria are reached.

Current developments An important current research direction is to evaluate the above multi-component architecture in real-world settings. The works [10,63] take some initial steps in this direction. The work [10] builds Protoplasm, an industrial-strength schema matching system, while the work [63] examines the scalability of matching systems to very large XML schemas.

A related direction focuses on creating robust and widely useful matcher operators and developing techniques to quickly and efficiently combine the operators for a particular matching task. A next logical direction is to make the frameworks easy to customize for a particular set of matching tasks. Our current eTuner work aims at automating the customization.

Finally, our work can also be seen as a part of the trend toward self-tuning databases, to reduce the high total cost of ownership [2,15,16].

6.3 Leveraging synthetic workloads

As far as we know, our work is the first to employ synthetic workloads in schema matching. Synthetic workloads and inputs have also recently been exploited in several other contexts. The recovery-oriented computing (ROC) project [59] focuses on building distributed systems (e.g., Internet services, computer networks) that are robust to failures. Toward this end, it generates and injects artificial faults into the target systems, to evaluate their robustness [13]. The work [31] constructs synthetic text documents that exhibit certain properties. It then examines how various information retrieval methods work with respect to these documents. The objective is to examine formal properties of these information retrieval methods. Finally, several learning approaches have also exploited artificial inputs. The work [47] for example shows that artificial training examples can improve the accuracy of certain learning methods.

Two common observations cut across the above scenarios. First, in certain settings knowledge about the application domain can be distilled into a relatively concise "generative model" that can then be used to generate synthetic data. For example, in schema matching one schema can be modeled as being generated by perturbation of another schema (in the same domain). The set of common perturbations is relatively small, and can be captured with a set of rules, as described in Sect. 3.1. Second, synthetic data can help significantly in improving the robustness or examining properties of application systems.

We are applying this general idea of using synthetic input/output pairs to make a system robust to additional contexts. Recently, we have successfully adapted it to the problem of maintaining semantic matches and the closely related problem of maintaining wrappers (e.g., [17,38,39,43,48,69]), as the data at the sources evolves (see [43] for more detail). We plan to adapt the same idea to improving record linkage systems (e.g., [3,33,70]).

Exploiting previously matched schemas Several recent works exploit previously matched schema pairs to improve matching accuracy (e.g., [9,23,25,41]). 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 Sect. 5.6 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.

6.4 Compositional approaches

Arguably, the success of relational data management derives partly from the following three factors. (1) It

is possible to define a small set of core operators (e.g., select, project, join) such that most common queries can be expressed as a composition of these operators. (2) Effective optimization techniques exist to select a good composition (i.e., execution tree). (3) Everything is made as "declarative" as possible, to enable effective user interaction, customization, and rapid development of applications.

It can be argued that the development of schema matching solutions has been following a similar *compositional* approach. First, monolithic solutions were developed. Next, they were "broken down", and multi-component solutions have been developed. Our current eTuner work suggests that these solutions can be "distilled" to extract a core set of operators, and that solutions that compose of these operators can be tuned, that is, partially optimized. There are then two interesting future directions: can we develop better tuning/optimization techniques, and can we make schema matching solutions as "declarative" as possible?

Compositional solutions have also been considered, or are currently under development, in several other contexts, most recently in record linkage [6], data integration [64], and text data management [68]. Other contexts include information extraction (e.g., [32]), solving crossword puzzles [37], and identifying phrase structure in NLP [61].

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 research, interesting directions include:

Finding optimal synthetic workload Given a schema S and a matching system M, how to find an optimal or good synthetic workload for tuning M? A crucial parameter that we must decide will be the size of the workload (i.e., the number of synthetic schemas). Another issue is the best strategy to partition the data during the synthetic workload generation process (would the strategy of dividing data into two equal in size but disjoint halves always work best?).

Better search strategies The current staged search strategy is adequate for offline tuning of schemas of small to moderate size. Better search strategies will enable scaling up to schemas of large size, incorporating user interaction into the tuning process (not just at the start of the tuning process, as in the current work), and finding potentially better knob configuration.

Considering other data representations We have considered only matching systems that handle relational representations. An important future direction is to tune systems that match other types of data representations, such as XML schemas. A key issue is to develop the set of "perturbations" that can be used to generate a synthetic workload. To tune accurately, the "perturbations" must reflect common logical and conceptual changes of the data representation as accurately as possible. Hence, given a data representation, it is important to discover what types of logical and conceptual changes it often undergoes.

Tuning additional kinds of matching scenarios So far we have tuned a matching system M for maximizing accuracy of matching a schema S with future schemas. Since the scenario of matching two schemas S and T is also common in practice, tuning M specifically for such scenarios is also important. Other important kinds of scenarios include tuning for complex matching [19] and tuning also for other performance factors beyond accuracy (e.g., execution time).

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

References

1. Aberer, K.: Special issue on peer to peer data management. SIGMOD Rec. **32**(3), 138–140 (2003)
2. Agrawal, S., Chaudhuri, S., Kollr, L., Marathe, A.P., Narasayya, V.R., Syamala, M.: Database tuning advisor for microsoft sql server 2005. In: VLDB, 2004
3. Andritsos, P., Miller, R.J., Tsaparas, P.: Information-theoretic tools for mining database structure from large data sets. In: Proceedings of SIGMOD, 2004
4. Aslan, G., McLeod, D.: Semantic heterogeneity resolution in federated databases by metadata implantation and stepwise evolution. VLDB J. **8**(2), 120–132 (1999)
5. Batini, C., Lenzerini, M., Navathe, S.B.: A comparative analysis of methodologies for database schema integration. ACM Comput. Surv. **18**(4), 323–364 (1986)

6. Benjelloun, O., Garcia-Molina, H., Jonas, J., Su, Q., Widom, J.: Swoosh: a generic approach to entity resolution. Technical report, Stanford University (2005)
7. Bergamaschi, S., Castano, S., Vincini, M., Beneventano, D.: Semantic integration of heterogeneous information sources. Data Knowl. Eng. **36**(3), 215–249 (2001)
8. Berlin, J., Motro, A.: Autoplex: automated discovery of content for virtual databases. In: Proceedings of the Conference on Cooperative Information Systems (CoopIS), 2001
9. Berlin, J., Motro, A.: Database schema matching using machine learning with feature selection. In: Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE), 2002
10. Bernstein, P.A., Melnik, S., Petropoulos, M., Quix, C.: Industrial-strength schema matching. SIGMOD Record, Special Issue in Semantic Integration, December 2004
11. Bilke, A., Naumann, F.: Schema matching using duplicates. In: Proceedings of the International Conference on Data Engineering (ICDE), 2005
12. Borkar, V., Deshmukh, K., Sarawagi, S.: Automatic text segmentation for extracting structured records. In: Proceedings of SIGMOD-01
13. Brown, A., Kar, G., Keller, A.: An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In: Proceedings of the Seventh IFIP/IEEE International Symposium on Integrated Network Management (IM), 2001
14. Castano, S., De Antonellis, V.: A schema analysis and reconciliation tool environment. In: Proceedings of the International Database Engineering and Applications Symposium (IDEAS), 1999
15. Chaudhuri, S., Dageville, B., Lohman, G.: Self-managing technology in database management systems (tutorial). In: Proceedings of VLDB, 2004
16. Chaudhuri, S., Weikum, G.: Rethinking database system architecture: towards a self-tuning risc-style database system. In: VLDB, 2000
17. Chidlovskii, B.: Automatic repairing of web wrappers. In: Third International Workshop on Web Information and Data Management, 2001
18. Clifton, C., Housman, E., Rosenthal, A.: Experience with a combined approach to attribute-matching across heterogeneous databases. In: Proceedings of the IFIP Working Conference on Data Semantics (DS-7), 1997
19. Dhamankar, R., Lee, Y., Doan, A., Halevy, A., Domingos P.: iMAP: discovering complex matches between database schemas. In: Proceedings of SIGMOD, 2004
20. Dietterich, T.G.: Machine learning research: four current directions. AI Mag. **18**(4), 97–136 (1997)
21. Do, H.: Schema matching and Mapping-based Data Integration. PhD Thesis, University of Leipzig, 2006
22. Do, H., Melnik, S., Rahm, E.: Comparison of schema matching evaluations. In: Proceedings of the 2nd International Workshop on Web Databases (German Informatics Society), 2002
23. Do, H., Rahm, E.: Coma: a system for flexible combination of schema matching approaches. In: Proceedings of the 28th Conference on Very Large Databases (VLDB), 2002
24. Doan, A.: Learning to Map between Structured Representations of Data. PhD Thesis, University of Washington, 2003
25. Doan, A., Domingos, P., Halevy, A.: Reconciling schemas of disparate data sources: A machine learning approach. In: Proceedings of the ACM SIGMOD Conference, 2001
26. Doan, A., Domingos, P., Halevy, A.: Learning to match the database schemas: a multistrategy approach. Mach. Learn. **50**(3), 279–301 (2003)
27. Doan, A., Madhavan, Dhamankar, R., Domingos, P., Halevy, A.: Learning to match ontologies on the Semantic Web. VLDB J. **12**, 303–319 (2003)
28. Doan, A., Madhavan, J., Domingos, P., Halevy, A.: Learning to map ontologies on the semantic web. In: Proceedings of the World-Wide Web Conference (WWW-02), 2002
29. Doan, A., Noy, N., Halevy, A.: Introduction to the special issue on semantic integration. SIGMOD Rec. **33**(4), 11–13 (2004)
30. Embley, D., Jackman, D., Xu, L.: Multifaceted exploitation of metadata for attribute match discovery in information integration. In: Proceedings of the WIIW-01, 2001
31. Fang, H., Tao, T., Zhai, C.: A formal study of information retrieval heuristics. In: Proceedings of the ACM SIGIR Conference, 2004
32. Freitag, D.: Machine learning for information extraction in informal domains. PhD. Thesis, Deptartment of Computer Science, Carnegie Mellon University, 1998
33. Ganti, V., Chaudhuri, S., Motwani, R.: Robust identification of fuzzy duplicates. In: ICDE, 2005
34. He, B., Chang, K.: Statistical schema matching across web query interfaces. In: Proceedings of the ACM SIGMOD Conference (SIGMOD), 2003
35. He, B., Chang, K.C.C., Han, J.: Discovering complex matchings across Web query interfaces: a correlation mining approach. In: Proceedings of the ACM SIGKDD Conference (KDD), 2004
36. Kang, J., Naughton, J.: On schema matching with opaque column names and data values. In: Proceedings of the ACM SIGMOD International Conference on Management of Data SIGMOD-03), 2003
37. Keim, G., Shazeer, N., Littman, M., Agarwal, S.: Cheves, C., Fitzgerald, J., Grosland, J., Jiang, F., Pollard, S., Weinmeister, K.: PROVERB: the probabilistic cruciverbalist. In: Proceeeedings of the 6th National Conference on Artificial Intelligence (AAAI-99), pp. 710–717 (1999)
38. Kushmerick, N.: Wrapper verification. World Wide Web J. **3**(2), 79–94 (2000)
39. Lerman, K., Minton, S., Knoblock, C.: Wrapper maintenance: a machine learning approach. J. Artif. Intell. Res. **18**:149–187 (2003)
40. Li, W., Clifton, C., Liu, S.: Database integration using neural network: implementation and experience. Knowl. Inf. Syst. **2**(1), 73–96 (2000)
41. Madhavan, J., Bernstein, P., Doan, A., Halevy, A.: Corpus-based schema matching. In: Proceedings of the 18th IEEE International Conf. on Data Engineering (ICDE), 2005
42. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic schema matching with cupid. In: Proceedings of VLDB, 2001
43. McCann, R., Alshebli, B., Le, Q., Nguyen, H., Vu, L., Doan, A.: Mapping maintenance for data integration systems. In: Proceedings of VLDB 2005
44. McCann, R., Doan, A., Kramnik, A.: Varadarajan, V.: Building data integration systems via mass collaboration. In: Proceedings of the SIGMOD-03 Workshop on the Web and Databases (WebDB-03), 2003
45. McCann, R., Kramnik, A., Shen, W., Varadarajan, V., Sobulo, O., Doan, A.: Integrating data from disparate sources: a mass collaboration approach. In: Proceedings of the International Conference on Data Engineering (ICDE), 2005
46. Melnik, S., Molina-Garcia, H., Rahm, E.: Similarity flooding: a versatile graph matching algorithm. In: Proceedings of the International Conference on Data Engineering (ICDE), 2002
47. Melville, P., Mooney, R.: Creating diversity in ensembles using artificial data. J. Inf. Fusion Spec. Issue Divers. Mult. Classifier Syst. 6(1):99–111 (2004)

48. Meng, X., Hu, D., Li, C.: Schema-guided wrapper maintenance for web-data extraction. In: Fifth International Workshop on Web Information and Data Management, 2003
49. Milo, T., Zohar, S.: Using schema matching to simplify heterogeneous data translation. In: Proceedings of the International Conference on Very Large Databases (VLDB), 1998
50. Mitchell, T.: Machine Learning. McGraw-Hill, NY (1997)
51. Mitra, P., Wiederhold, G., Jannink, J.: Semi-automatic integration of knowledge sources. In: Proceedings of Fusion, 1999
52. Neumann, F., Ho, C.T., Tian, X., Haas, L., Meggido, N.: Attribute classification using feature analysis. In: Proceedings of the International Conference on Data Engineering (ICDE), 2002
53. Noy, N.F., Musen, M.A.: PROMPT: algorithm and tool for automated ontology merging and alignment. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), 2000
54. Noy, N.F., Musen, M.A.: Anchor-PROMPT: using non-local context for semantic Matching. In: Proceedings of the Workshop on Ontologies and Information Sharing at the International Joint Conference on Artificial Intelligence (IJCAI), 2001
55. Ouksel, A., Seth, A.P.: Special issue on semantic interoperability in global information systems. SIGMOD Re. **28**(1) 5–12 (1999)
56. Palopoli, L., Sacca, D., Terracina, G., Ursino, D.: A unified graph-based framework for deriving nominal interscheme properties, type conflicts, and object cluster similarities. In: Proceedings of the Conf. on Cooperative Information Systems (CoopIS), 1999
57. Palopoli, L., Sacca, D., Ursino, D.: Semi-automatic, semantic discovery of properties from database schemes. In: Proceedings of the International Database Engineering and Applications Symposium (IDEAS-98), pp. 244–253 (1998)
58. Palopoli, L., Terracina, G., Ursino, D.: The system DIKE: towards the semi-automatic synthesis of cooperative information systems and data warehouses. In: Proceedings of the ADBIS-DASFAA Conference, 2000
59. Patterson, D.A., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., Treuhaft, N.: Recovery-oriented computing (ROC): motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, University of California, 2002
60. Perkowitz, M., Etzioni, O.: Category translation: Learning to understand information on the Internet. In: Proceedigns of Internatinal Joint Conference on AI (IJCAI), 1995
61. Punyakanok, V., Roth, D.: The use of classifiers in sequential inference. In: Proceedings of the Conference on Neural Information Processing Systems (NIPS-00), 2000
62. Rahm, E., Bernstein, P.A.: On matching schemas automatically. VLDB J. **10**(4) 334–350 (2001)
63. Rahm, E. Do, H., Massmann, S.: Matching large XML schemas. SIGMOD Record, Special Issue in Semantic Integration, December 2004
64. Rahm, E., Thor, A., Aumueller, D., Do, H., Golovin, N., Kirsten, T.: iFuice — Information fusion utilizing instance correspondences and peer mappings. In: Proceedings of the Eighth International Workshop on the Web and Databases (WebDB), 2005
65. Ryutaro, I., Hideaki, T., Shinichi, H.: Rule induction for concept hierarchy alignment. In: Proceedings of the 2nd Workshop on Ontology Learning at the 17th International Joint Conference on AI (IJCAI), 2001
66. Sayyadian, M., LeKhac, H., Doan, A., Gravano, L.: Keyword search across heterogeneous relational databases. Technical report, Department of Computer Science, Universtiy of Illinois (2006)
67. Seligman, L., Rosenthal, A.: The impact of xml in databases and data sharing. IEEE Computer, 2001
68. UIMA: Unstructured information management architecture. http://www.research.ibm.com/UIMA/
69. Velegrakis, Y., Miller, R., Popa, L., Mylopoulos, J.: Tomas: a system for adapting mappings while schemas evolve. In: Proceedings of the Twentieth International Conference on Data Engineering, 2004
70. Weis, M., Naumann, F.: Dogmatix tracks down duplicates in xml. In: Proceedings of the ACM Conference on Management of Data (SIGMOD), 2005
71. Wu, W., Yu, C., Doan, A., Meng, W.: An interactive clustering-based approach to integrating source query interfaces on the Deep Web. In: Proceedings of SIGMOD, 2004
72. Xu, L., Embley, D.: Using domain ontologies to discover direct and indirect matches for schema elements. In: Proceedigns of the Semantic Integration Workshop at ISWC-03. http://smi.stanford.edu/si2003, 2003
73. Yan, L.L., Miller, R.J., Haas, L.M., Fagin, R.: Data driven understanding and refinement of schema mappings. In: Proceedings of the ACM SIGMOD, 2001